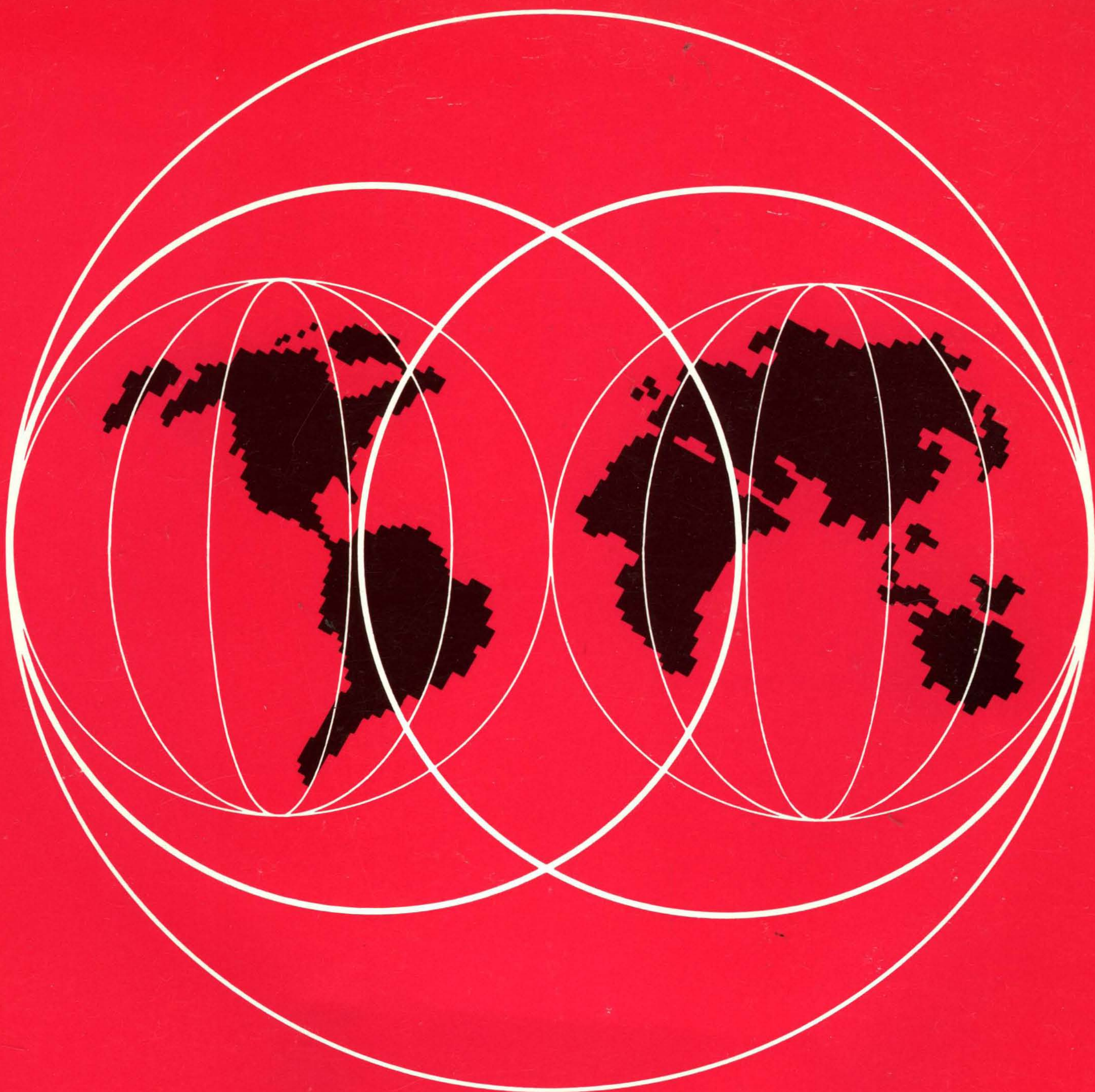


IBM

International Technical Support Centers

WRITING A DEVICE DRIVER FOR AIX VERSION 3

GG24-3629-00

Writing a Device Driver for AIX Version 3

Document Number GG24-3629

May, 1991

International Technical Support Center
Austin, Texas

Take Note

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xv.

First Edition (May, 1991)

This edition applies to Version 3, Release Number 1 of Advanced Interactive Executive, Program Number 5756-030 (the AIX 3.1 Operating System).

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, International Technical Support Center
11400 Burnet Road
Dept. 948, Building 983
Austin, TX 78758 USA

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1991. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Abstract

This document describes the writing and debugging of device drivers for the RISC/6000 running under the AIX Version 3.1 operating system.

This document is intended for programmers and software support personnel who need to know detailed information on writing device drivers. A knowledge of device drivers, device hardware, and the C programming language is assumed.

AIX (435 pages)

© Copyright IBM Corp. 1991

Contents

Chapter 1. Introduction . . . 1-1
1.1 Device Driver Concepts . . . 1-1
1.1.1 Special Files in AIX . . . 1-1
1.1.2 Major and Minor Numbers . . . 1-2
1.2 Device Driver Types . . . 1-3
1.2.1 Block Mode Device Drivers . . . 1-3
1.2.2 Character Mode Device Drivers . . . 1-3
1.3 Device Driver Roles . . . 1-4
1.3.1 Device Head Role . . . 1-4
1.3.2 Device Handler Role . . . 1-4
1.3.3 Combining the Roles . . . 1-4
1.4 Device Driver Structure . . . 1-6
1.4.1 Device Driver Top Half Routines . . . 1-7
1.4.2 The Device Driver Bottom Half . . . 1-7

Chapter 2. Programmer's model of the hardware . . . 2-1
2.1 Micro Channel Overview . . . 2-1
2.2 System I/O Structure . . . 2-3
2.2.1 Overview . . . 2-3
2.2.2 RISC System/6000 Addressing Model . . . 2-3
2.3 I/O Data Transfer Protocols . . . 2-9
2.3.1 Programmed I/O Mode . . . 2-9
2.3.2 DMA Transfers . . . 2-9
2.4 Interrupt Processing . . . 2-11
2.4.1 Priority Assignment . . . 2-12
2.4.2 Off-Level Interrupts . . . 2-13
2.5 Addressing Micro Channel Adapters . . . 2-13
2.5.1 Identifying an Adapter . . . 2-13
2.5.2 Setting Adapter Attributes . . . 2-14
2.5.3 Enabling an Adapter . . . 2-14
2.5.4 I/O Macros . . . 2-14
2.5.5 Sample I/O on the RISC System/6000 . . . 2-18
2.5.6 Byte Reversal from the System Bus to the Micro Channel Bus . . 2-19
2.5.7 Additional PIO Macro Information . . . 2-21

Chapter 3. Interface to Device Drivers . . . 3-1
3.1 Aix Version 3.1 Structure . . . 3-1
3.1.1 AIX and the Interrupt and Process Environments . . . 3-1
3.1.2 The AIX Interrupt Handler Environment . . . 3-3
3.1.3 The AIX Process Environment . . . 3-4
3.1.4 Preemption in the AIX Operating System . . . 3-5
3.2 Kernel Interface . . . 3-6
3.2.1 The Device Switch Table . . . 3-7
3.2.2 Entry Points Common to Character and Block Device Drivers . . 3-8
3.2.3 Entry Points for Character and Raw Access to Block Device Driver 3-9
3.2.4 Entry Points Unique to Character Device Drivers . . . 3-9
3.2.5 Entry Points Unique to Block Device Drivers . . . 3-9
3.2.6 Entry Points for Trusted Computing Path Device Drivers . . . 3-9
3.2.7 Miscellaneous Entry Points NOT Found in the Device Switch Table 3-9
3.3 Kernel Services . . . 3-10
3.3.1 I/O Services . . . 3-10

© Copyright IBM Corp. 1991

3.3.2 Memory Services 3-13
3.3.3 Other Services 3-15

Chapter 4. Overview of a Character Device Driver 4-1
4.1 Implementation 4-1
4.1.1 ddconfig Device Driver Entry Point 4-1
4.1.2 ddmpx Device Driver Entry Point 4-12
4.1.3 ddopen Device Driver Entry Point 4-19
4.1.4 ddclose Device Driver Entry Point 4-24
4.1.5 ddread Device Driver Entry Point 4-29
4.1.6 ddwrite Device Driver Entry Point 4-34
4.1.7 ddioctl Device Driver Entry Point 4-42
4.1.8 ddselect Device Driver Entry Point 4-46
4.1.9 dddump Device Driver Entry Point 4-52

Chapter 5. Overview of a Block Device Driver 5-1
5.1 Introduction 5-1
5.1.1 Block I/O Device Driver Entry Points 5-1
5.1.2 Character Access to Block Device Drivers 5-6
5.1.3 Block I/O Device Device Summary 5-7

Chapter 6. Device Drivers Configuration 6-1
6.1 Introduction 6-1
6.1.1 General Structure of the Device Configuration Subsystem 6-1
6.1.2 Device Configuration Database Overview 6-5
6.1.3 Device Configuration Procedure Overview 6-7
6.2 Configuring an Unsupported Device to the System 6-13
6.2.1 Modifying the Predefined Database 6-13
6.2.2 Writing Device Methods 6-14

Chapter 7. SMIT Interface 7-1
7.1 Introduction 7-1
7.2 SMIT Screens 7-3
7.2.1 Menu Screens 7-3
7.2.2 Selector Screens 7-3
7.2.3 Dialog Screens 7-4
7.3 SMIT Database 7-5
7.4 Command Building and Running 7-12
7.4.1 Task Building 7-12
7.4.2 Command Execution 7-12
7.5 Dialogs Example 7-14
7.5.1 List All Defined Ric Ports 7-14
7.5.2 Add a Ric Port 7-15
7.6 Additions to the SMIT Database 7-20
7.6.1 Database Creation 7-20
7.6.2 SMIT Extensions Debugging 7-20
7.6.3 Task Additions 7-21

Chapter 8. Device Drivers Packaging 8-1
8.1 Introduction 8-1
8.2 Design Guidelines 8-1
8.3 The installp Command 8-1
8.4 Ensuring installp Command Compatibility 8-2
8.5 Files for installp Operation 8-2
8.5.1 LPP Option List File: lpp_name 8-3

8.5.2 Instal Script . . . 8-4
8.5.3 al (Option.al) . . . 8-4
8.5.4 size (Option.size) . . . 8-4
8.5.5 copyright . . . 8-4
8.5.6 lpp.cleanup . . . 8-4
8.5.7 Prereq (Option.prereq) . . . 8-4
8.5.8 config (Option.config) . . . 8-5
8.5.9 lpp.deinst . . . 8-5
8.5.10 inventory (Option.inventory) . . . 8-5
8.5.11 productid . . . 8-5
8.5.12 lpp.acf . . . 8-5
8.6 Installp Example . . . 8-5
8.6.1 Introduction . . . 8-6
8.6.2 How to Use the Makefile . . . 8-6
8.6.3 Root/lpp_name File . . . 8-8
8.6.4 Apply List Files . . . 8-8

Chapter 9. Tools for Debugging Device Drivers . . . 9-1
9.1 Debugging Overview . . . 9-1
9.2 System Dump . . . 9-1
9.2.1 Initiating a System Dump . . . 9-1
9.2.2 Including Device Driver Information in a System Dump . . . 9-2
9.2.3 Formatting a System Dump . . . 9-5
9.3 The crash Command . . . 9-6
9.3.1 crash Subcommands . . . 9-7
9.4 The Kernel Debugger . . . 9-18
9.4.1 The Kernel Debug Program Commands . . . 9-20
9.5 Using the Kernel Debugger to Debug Device Drivers . . . 9-29
9.5.1 Setting Breakpoints in Device Driver Routines . . . 9-29
9.5.2 Setting Breakpoints in System Routines . . . 9-30
9.5.3 Displaying Registers on a Micro Channel Adapter . . . 9-30
9.5.4 How to read/write Data Variables in your Device Driver . . . 9-31
9.6 Error Logging . . . 9-35
9.6.1 Pre-Coding Steps to Consider . . . 9-37
9.6.2 Coding Steps . . . 9-38
9.6.3 What Really Happens in /dev/error . . . 9-44
9.7 Performance Tracing for AIX . . . 9-44
9.7.1 Introduction . . . 9-44
9.7.2 Use of the trace Facility . . . 9-48
9.7.3 Controlling trace . . . 9-50
9.7.4 Producing a trace Report . . . 9-53
9.7.5 Defining trace Events . . . 9-56
9.7.6 Usage Hints . . . 9-74

Chapter 10. Hints and Tips . . . 10-1
10.1 Crash and Kernel Debugging Addresses . . . 10-1
10.2 Pinning Device Driver Code . . . 10-1
10.3 Compiling Device Drivers . . . 10-3
10.4 Working with Kernel Processes . . . 10-3
10.4.1 Writing a Kernel Process . . . 10-3
10.4.2 Compiling a Kernel Process . . . 10-3
10.4.3 Linking a Kernel Process . . . 10-4
10.4.4 Loading a Kernel Process . . . 10-4
10.4.5 Starting a Kernel Process . . . 10-4

Appendix A. AIX Devices A-1
A.1 Device Classes, Subclasses, and Types Overview A-1
A.2 Device Dependencies and Child Devices A-1
A.3 The Run Time Configuration Commands A-3
A.3.1 The mkdev Command A-3
A.3.2 The chdev Command A-3
A.3.3 The rmdev Command A-3
A.3.4 The cfgmgr Command A-3
A.4 Devices Location Codes A-3
A.4.1 Printer and Plotter Devices A-4
A.4.2 TTY Devices A-4
A.4.3 Direct-Attached Disks and SCSI Devices A-5
A.4.4 Diskette Drive Devices A-5
A.4.5 Adapter Devices A-5
A.4.6 Multiprotocol Port Devices A-5

Appendix B. ODM B-1
B.1 ODM Object Classes B-1
B.1.1 Predefined Devices (PdDv) B-1
B.1.2 Predefined Attribute (PdAt) B-6
B.1.3 Predefined Connection (PdCn) B-9
B.1.4 Customized Devices (CuDv) B-10
B.1.5 Customized Attribute (CuAt) B-13
B.1.6 Customized Dependency (CuDep) B-15
B.1.7 Customized Device Driver (CuDvDr) B-15
B.1.8 Customized VPD (CuVPD) B-16
B.1.9 Configuration Rules (Config_Rules) B-17
B.2 ODM Commands B-19
B.2.1 ODM Commands That Handle Objects B-19
B.2.2 ODM Commands That Handle Object Classes B-19
B.3 ODM Routines B-20
B.3.1 ODM Subroutines That Handle Objects B-20
B.3.2 ODM Subroutines That Handle Object Classes B-20
B.3.3 ODM Subroutines That Handle Other ODM Functions B-20
B.4 Device Configuration Library Routines B-21
B.5 Real Time Interface Co-Processor Adapter Configuration Files B-22
B.5.1 ODM Stanzas (ric.add) B-22
B.5.2 Message Catalog for Ric Adapter and Ports B-27
B.5.3 Adapter Configuration Method (cfgrica.c) B-28
B.5.4 Ric Port Configuration Method (cfgricp.c) B-37
B.5.5 Header File for Configuration Methods (debug.h) B-52
B.5.6 Makefile for Configuration Methods and Message Catalog B-53

Appendix C. SMIT C-1
C.1 Object Classes C-1
C.1.1 Menu Object Class (sm_menu_opt) C-1
C.1.2 Menu Object Class (sm_menu_opt) Used for Aliases C-2
C.1.3 Selector Header Object Class (sm_name_hdr) C-3
C.1.4 Dialog Header Object Class (sm_cmd_hdr) C-5
C.1.5 Dialog/Selector Command Option Object Class (sm_cmd_opt) . . C-7
C.2 Additional Information C-12
C.2.1 Information Retrieval C-12
C.2.2 Default Values Setting C-12
C.2.3 Flags and Parameters Setting C-13
C.2.4 Aliases and Fast Paths C-14

C.3 Examples C-15
C.3.1 ODM Stanzas for Ric Dialogs (sm_ric.add file) C-15
C.3.2 SMIT Log File C-35

Appendix D. Installp/Updatep Files D-1
D.1 Required Files for Creating Compatible Application Programs D-1
D.1.1 The lpp_name File D-1
D.1.2 The liblpp.a File D-2
D.1.3 The instal Script File D-2
D.1.4 The al File D-4
D.1.5 The copyright File D-4
D.1.6 The size File D-5
D.1.7 The lpp.cleanup File D-5
D.1.8 The special File D-6
D.1.9 The service_num File D-7
D.1.10 The arp File D-7
D.1.11 The update Script File D-7
D.1.12 The al_Level File D-9
D.2 Optional Files for Creating Compatible Application Programs D-10
D.2.1 The config File and Option.config File D-10
D.2.2 The prereq and Option.prereq File D-10
D.2.3 The lpp.doc File D-12
D.2.4 The Filename.err File D-12
D.2.5 The Filename.trc File D-12
D.2.6 The Filename.evt File D-13
D.2.7 The lpp.acf File D-13
D.2.8 The productid File D-14
D.2.9 The inventory File D-14
D.2.10 The lpp.deinst File D-14
D.2.11 The rename File D-14
D.2.12 The lpp.instr File D-15
D.3 Real Time Interface Co-Processor Device Driver Package D-16
D.3.1 Makefile D-16
D.3.2 Instal D-19
D.3.3 ricdd.driver.config D-23
D.3.4 Lpp.cleanup D-24
D.3.5 Ricc.src.cleanup D-25
D.3.6 Ricc.driver.cleanup D-26
D.3.7 Lpp.deinst D-28

Appendix E. Sample Character Device Driver E-1
E.1 Device Driver Main Body E-1
E.2 Device Driver Header Files E-21
E.2.1 ric.h E-21
E.2.2 ricstruct.h E-25
E.2.3 ricsmisc.h E-32
E.3 Device Driver Makefile E-32

Appendix F. Device Driver Miscellaneous F-1
F.1 The busresolve system call F-1

Index X-1

Figures

1-1. Extract from the Result of ls -l /dev. 1-1
1-2. Major and Minor Numbers Typical Example 1-3
1-3. The HFT Subsystem 1-5
1-4. The SCSI Subsystem 1-6
2-1. System Block Diagram 2-3
2-2. RISC System/6000 Addressing for Bus-Attached and System Memory 2-5
2-3. Address Translation - Bus Memory, I/O, and I/O Control 2-6
2-4. I/O Segment Register 2-7
2-5. RISC System/6000 Addressing Model 2-8
2-6. Data Transfer to a 16-Bit Micro Channel Device 2-20
2-7. Data Transfer to a 32-Bit Micro Channel Device 2-21
3-1. AIX Interrupt Handlers and Processes 3-2
4-1. Device Driver ddconfig Entry Point 4-2
4-2. Code Sample of the ricconfig Routine 4-5
4-3. Example of a DDS 4-11
4-4. Device Driver ddmpx Entry Point for an open 4-12
4-5. Relationship of Major Numbers, Minor Numbers and Channels . 4-14
4-6. ddmpx for open and create 4-15
4-7. ddmpx for close 4-16
4-8. Code Sample of the ricmpx Routine 4-18
4-9. Device Driver ddopen Entry Point 4-19
4-10. Code Sample of the ricopen Routine 4-22
4-11. Device Driver ddclose Entry Point 4-25
4-12. Device Driver ddclose Program Flow 4-26
4-13. Code Sample of the ricclose Routine 4-28
4-14. Device Driver ddread Entry Point 4-30
4-15. Code Sample of the ricread Routine 4-32
4-16. Device Driver ddwrite Entry Point 4-35
4-17. Code Sample of the ricwrite Routine 4-37
4-18. Device Driver ddioctl Entry Point 4-43
4-19. Code Sample of the ricioctl Routine 4-45
4-20. Device Driver ddselect Entry Point 4-47
4-21. Code Sample of the ricselect Routine 4-50
5-1. Entry Points for a Block Device Driver 5-2
5-2. The mbuf structure 5-4
6-1. Structure of the Configuration Subsytem 6-2
6-2. Device States 6-4
6-3. Example of Devices Graph 6-9
6-4. How cfgmgr Executes Config_Rules 6-12
6-5. How Device Methods Get Invoked 6-16
7-1. Some Relationships among SMIT Menus, Selectors and Dialogs 7-2
7-2. Hierarchy of sm_menu_opt Objects 7-9
7-3. SMIT Dialogs 7-10
7-4. SMIT Screen Example 7-14
9-1. System Dump Flow 9-3
9-2. Kernel Dump Image 9-5
9-3. Flow of the Error Logging Facility 9-36
9-4. dderr.h 9-42
9-5. errlog_ex 9-43
9-6. Flow Involved in Starting/Stopping Trace 9-45

© Copyright IBM Corp. 1991

9-7. Trace Formatting 9-46
9-8. Trace Hook Summary 9-47
9-9. Trace Example Using Subcommands 9-51
9-10. Sample Code - trace Triggers 9-53
9-11. Format of a trace Event Record 9-57
9-12. Syntax of Stanza in Format File 9-61
9-13. Trace Format File Syntax 9-65
9-14. Sample C Code---Trace Program Loop 9-72
9-15. Sample Trace Event Stanza 9-73
9-16. Formatted Trace Results 9-74

Tables

7-1. SMIT Screen Types 7-1
7-2. Object Used to Create Screens 7-5
7-3. sm_cmd_hdr Relationships 7-11
9-1. Kernel Debugger Commands List 9-20
B-1. PdDv Object Class Descriptors B-2
B-2. PdAt Object Class Descriptors B-7
B-3. PdCn Object Class Descriptors B-10
B-4. CuDv Object Class Descriptors B-11
B-5. CuAt Object Class Descriptors B-14
B-6. CuDep Object Class Descriptors B-15
B-7. CuDvDr Object Class Descriptors B-15
B-8. Contents of Value1, Value2, and Value3 Descriptors B-16
B-9. CuVPD Object Class Descriptors B-16
B-10. Config_Rules Object Class Descriptors B-17
B-11. Rule Values B-18

© Copyright IBM Corp. 1991

Special Notices

This publication is intended to help the programmer to write device drivers for the AIX 3.1 operating system that runs on the IBM RISC/6000 hardware.

The information in this publication is not intended as the specification of the programming interfaces that are provided by the AIX 3.1 operating system. This information is to serve as an example of a device driver. For information relating to the actual programming interfaces for AIX 3.1, please see the PUBLICATIONS SECTION of the IBM Programming Announcement for AIX 3.1.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX
AIXWindows
IBM
InfoExplorer
Information Retrieval Symbol
InfoTrainer
InfoWindow
PS/2
RISC System/6000
RT
RT PC
RT Personal Computer

© Copyright IBM Corp. 1991

The following terms, which are denoted by a double asterisk (* *) in this publication, are trademarks of other companies.

UNIX is a trademark of UNIX System Laboratories, Inc.
Goofy is a trademark of the Walt Disney Corporation

Preface

This document is intended to assist the programmer in the development of a device driver by presenting detailed information and examples.

Topics presented include:

- Introductory device driver concepts
- Overview of RISC/6000 hardware as it relates to device drivers with adapters residing on the Micro Channel bus
- Examples of a character device driver
- A menu driven interface for device driver configuration (SMIT)
- Device driver packaging concepts
- Tools for device driver debugging
- Information on the Object Data Manager (ODM)

The document is organized as follows:

- "Introduction"

  This provides the programmer with a brief overview of device driver concepts.

- "Programmer's model of the hardware"

  This provides the programmer with the understanding of how addressing is used to access memory and I/O in tthe RISC/6000.

- "Interface to Device Drivers"

  This chapter describes AIX 3.1 kernel concepts and how device drivers are integrated into it. It also describes the main kernel services used to accomplish this.

- "Overview of a Character Device Driver"

  This documents the organization of a character device driver, complete with an example.

- "Overview of a Block Device Driver"

  This documents the organization of a block device driver.

- "Device Drivers Configuration"

  This describes how the device configuration is accomplished with AIX 3.1.

- "SMIT Interface"

  This describes how menus can be added to the System Management Interface Tool (SMIT) to provide a high level user interface for the addition and deletion of a new device.

- "Device Drivers Packaging"

  This describes how you would create an install diskette for the packaging of a device driver.

© Copyright IBM Corp. 1991

- "Tools for Debugging Device Drivers"

  This describes the tools for debugging device drivers including the kernel debugger, crash, trace and error logging.

- "Hints and Tips"

  This describes miscellaneous hints and tips on compiling and linking device drivers.

Related Publications

The following publications are considered particularly suitable for a more detailed discussion of the topics covered in this document.

Prerequisite Publications

RISC System/6000 CD-ROM Hypertext Base Library, SC23-2163

Kernel Extensions and Device Support Programming Concepts, SC23-2207

Calls and Subroutines Reference, SC23-2198

RISC System/6000 General Programming Concepts, SC23-2205

RISC System/6000 Hardware Technical Reference General Information SC23-2643

RISC System/6000 Performance Monitoring and Tuning Guide SC23-2365

Additional Publications

Micro Channel Architecture Bus Master R1.1, GG24-3477

Plain Man's View of Micro Channel Architecture, GG24-3584

Micro Channel Architecture, IBM Personal Systems Technical Journal G325-5004

© Copyright IBM Corp. 1991

Acknowledgments

The advisor for this project was:

Ron Smetana
International Technical Support Center, Austin

The authors of this document are:

Luc Smolders, IBM Belgium
Susan C. Williams, IBM UK
Ron Smetana, IBM Austin

This publication is the result of a residency conducted at the International Technical Support Center, Austin.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Virgil Albaugh, IBM Austin
Greg Birgen, IBM Austin
Tim Boyce, IBM Austin
Jan Brown, IBM Austin
Mikey Buratti, IBM Austin
Sam Drake, IBM Almaden
Elizabeth Francis, IBM Austin
Bob Kovacs, IBM Austin
Dr. Rhys Lewis, IBM, UK
Sergio Perrone, IBM Italy
Cheryl Pervier, IBM Austin

A special thanks for our friends at ACSC who developed a device driver class for AIX Version 3 and provided indirect influence for our book.

Editor's Note

It is virtually impossible to thank everyone who has contributed to this publication. The authors hope that this document will pull together information that was originally shipped with AIX version 3. In this spirit, we would like to thank everyone who has contributed to the original AIX documentation. We have gathered much information from the shipped manuals and clarified, rewrote and added as we saw fit. A special thanks to the developers and contractors who reviewed the material.

© Copyright IBM Corp. 1991

Chapter 1. Introduction

1.1 Device Driver Concepts

A device driver is a section of code that provides support for a device. Device drivers run in a privileged state, as AIX kernel extensions, and have access to a number of functions that are unavailable to normal application programs. They shield the user from device-specific details and provide a common I/O model for accessing the devices for which they provide support.

In general, user application programs do not wish to manipulate the Micro Channel bus or the I/O capabilities of the RISC System/6000 directly.[1] As such, device drivers are often written to handle specialized or otherwise unsupported equipment.

1.1.1 Special Files in AIX

The system interface to devices, which is supported by device drivers, is through the file system. Each device that is accessible to a user-mode application has a file name and can be accessed as if it was an ordinary file. By convention this device file name is found in the **/dev** directory in the root file system. This device name along with the associated inode is known as a device **special file**. A special file appears to applications to be a standard AIX disk file; it has a name, resides in a directory, and can be opened, read, written and be manipulated by other standard system calls. But a special file is not a disk file; rather, it is a method by which an application program can cause a device driver to be invoked. Application programs open the special files and use read, write and ioctl calls to communicate with the device driver.

Special files are created by the AIX **mknod** command and by the **mknod()** system call. These create a special file with a given name, in a given directory. Instead of having a standard disk file associated with the file name, three pieces of information are kept:

- Major device number
- Minor device number
- Type of special file: character or block

```
brw-rw-rw-   1 root    system    14,  0 Aug 11 20:17 fd0
brw-rw----   1 root    system    10,  9 Jul 17 08:50 hd1
crw-rw-rw-   1 root    system    15,  0 Jul 12 15:26 rmt0
crw-rw-rwT   1 root    system    20,  0 Jul 12 15:26 tok0
```

Figure 1-1. Extract from the Result of ls -l /dev.

[1] The exception is via the *machine device driver*, called **/dev/bus0**.

© Copyright IBM Corp. 1991

The major device number is the fifth field from the result of **ls -l** (see Figure 1-1), the minor is the sixth field, and c or b in the first column indicates a character or a block device.

1.1.2 Major and Minor Numbers

Devices are generally identified in the kernel through major and minor numbers. Usually, a major number identifies a particular device driver. Minor numbers identify various device instances known to the device driver. However, a device driver may be assigned multiple major numbers. Also, minor numbers can be used to identify different modes of operation for a device as well as different device instances.

Programs do not need to understand these major and minor numbers to access devices. A program accesses a device as though it were a file by opening the device's corresponding special file located in the **/dev** directory. The special file's inode contains a particular major and minor number combination specified when the special file was created. This relationship remains constant until the special file is deleted.

The major number uniquely identifies the relevant device driver and thus is used to uniquely identify the device to the kernel. The interpretation of the minor number is entirely dependent on the particular device driver. Most frequently, the minor number is used to select one of multiple subdevices supported by the device driver. As a minor device number, it usually serves as an index into a device driver-maintained array of information about each of several devices or subdevices supported by the device driver.

To see a typical use of major and minor device numbers, let's assume a Micro Channel adapter that can drive up to three printers. Since all the printers are driven by the same device driver, the special files for the printers all share the same major device number. Since the printers are all separate entities, they would each have their own minor device number. If each printer had multiple personalities (if a single printer could print files using two different datastreams ... PostScript and HPGL, perhaps), then each physical printer might be represented by more than one special file, with multiple minor device numbers being assigned to each physical printer. This is shown in Figure 1-2 on page 1-3. Figure 9-1 on page 9-3 shows the flow of a system dump.

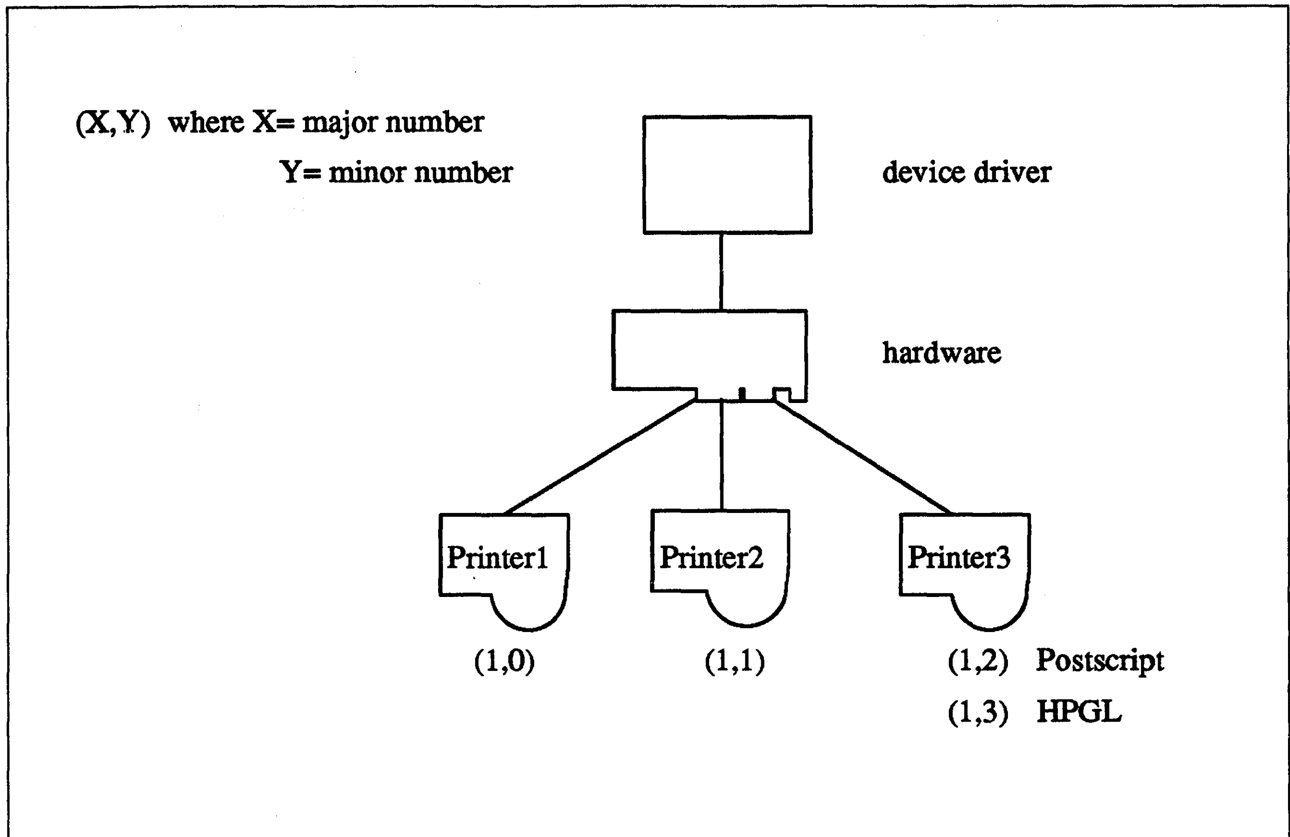


Figure 1-2. Major and Minor Numbers Typical Example

1.2 Device Driver Types

There are two major families of device drivers in AIX, known as character mode drivers and block mode drivers. These classes of device driver are distinguished by the type of devices they support and the interfaces that are presented to the kernel.

1.2.1 Block Mode Device Drivers

The block device interface is suitable for random access storage devices with fixed-size addressable data blocks. Devices supported by block device drivers can also potentially support a mounted file sytem. Block device drivers can provide character device interfaces and access to their block devices by providing a character special file, as well as the block special file. For example *fd0* is the floppy disk device, and *rfd0* gives a character interface access to the same device. Character device interfaces to block devices are called *raw I/O*, and the corresponding device is called a **raw** device (rfd0).

1.2.2 Character Mode Device Drivers

The character device interface is more suitable to non-random access devices, such as terminals, printers and networks. These devices cannot directly support mounted file systems.

1.3 Device Driver Roles

Device drivers can play two *roles* in the AIX operating system: the *device head* role and the *device handler* role. Most simple device drivers will in fact act in both roles, but other configurations are possible. An entry point for a device driver is either in a device head or in a device handler.

1.3.1 Device Head Role

A *device head* is a device driver or a portion thereof that provides interfaces to application programs via the standard **open**, **close**, **read**, **write**, and related system calls A device driver acting in this role takes I/O requests from application programs and communicates them to a *device handler*.

The interface between application programs and a device head is rigidly defined by the AIX kernel itself[2].

Device head routines are responsible for the following functions:

- They convert the request from the form of the file I/O function call to a form that the routines acting in the corresponding device handler role understand.
- They perform the appropriate data blocking and buffering.
- They manage the device. This includes such actions as maintaining queues of I/O requests and handling error recovery and error logging.

1.3.2 Device Handler Role

A *device handler* is the portion of a device driver that communicates with the actual device or adapter. The device handler takes requests from a device head and implements the requests on real hardware.

The interface between a device head and a device handler is essentially undefined by AIX, though a large number of primitive functions are provided by AIX to assist in constructing an interface. The details, however, are left to the driver author. The interface between the device handler and the device itself is naturally dependent on the hardware being manipulated, though AIX again provides a set of functions which assist in performing the hardware interfacing.

1.3.3 Combining the Roles

A simple device driver will provide both a device head and device handler, and will be a self-contained entity. But more complex scenarios are possible.

For example, it is possible that a large number of different adapters and devices could provide the same interface to application programs. For example, it might be desirable to have a large number of different types of plotters appear identically to user programs, while in reality they might attach to the system in very different ways (and might even take different datastreams, which are to be hidden from the application programs).

[2] Routines providing the device head role must conform to the programming model for system calls described in Chapter 4 of *Kernel Extensions and Device Support Programming Concepts*.

One way to implement this is to have a single driver operating as a device head, with multiple drivers acting as device handlers (one handler for each real adapter). This is how the **hft** subsystem on the RISC System/6000 operates; for example, many different adapters can all be configured to appear as hfts, and a single device head communicates to the appropriate device handlers for the adapter in use. Separate keyboard, mouse, and display drivers all interact with the hft device head under the covers. These low level drivers *only* interface to hardware and to the hft device head; there is no way for applications to interface with the keyboard driver except via the hft interface. See Figure 1-3 for a diagram of this.

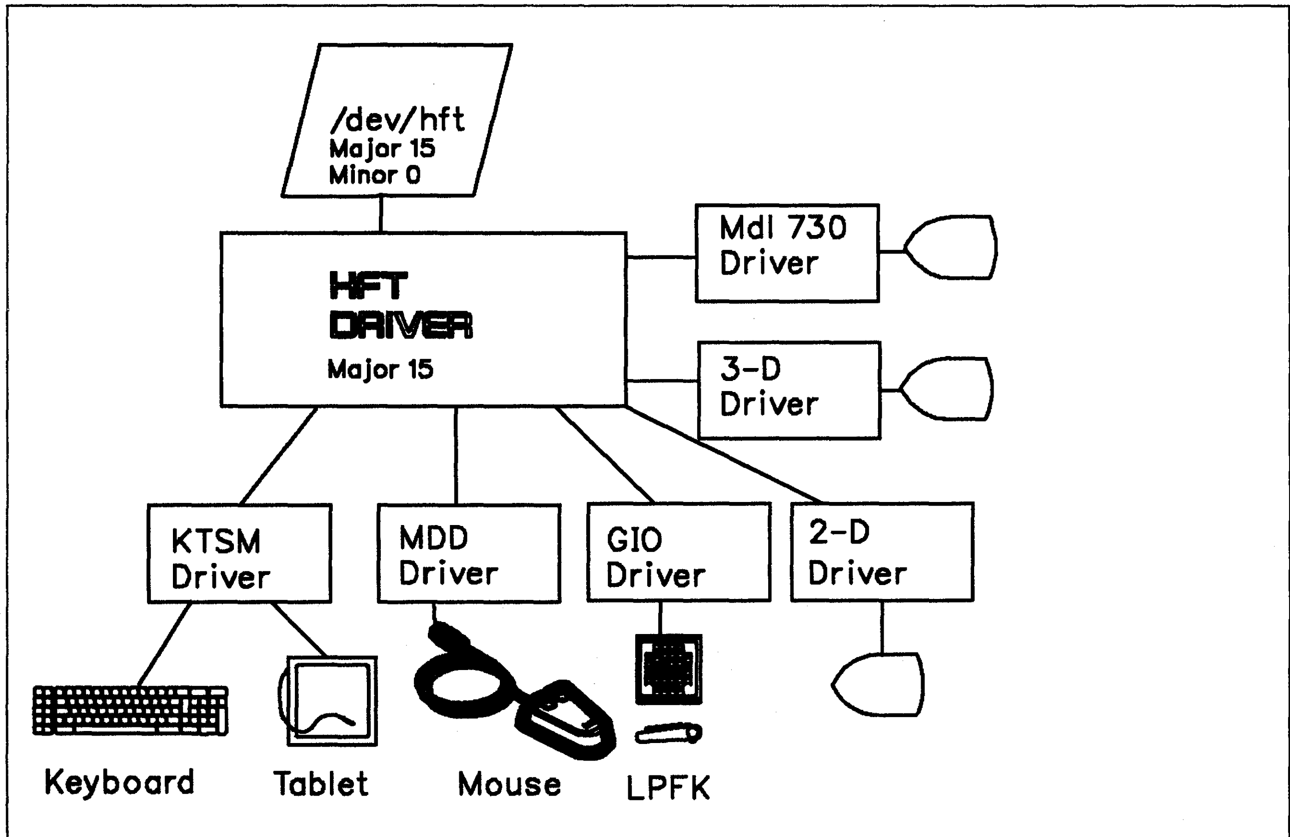


Figure 1-3. The HFT Subsystem. This is an example of a complex system of device drivers. The HFT driver provides only the device head role, while the others act only as device handlers.

A somewhat opposite example is the support of the SCSI bus on the RISC System/6000 (see Figure 1-4 on page 1-6). One Micro Channel adapter interfaces the SCSI bus to the Micro Channel; a generic SCSI device driver provides the device handler for this adapter. This generic SCSI driver handles sending commands on the SCSI bus, but does not know what commands to send. Numerous different device drivers operate in the device head role interface between applications and the SCSI device handler; these drivers support specific devices such as disk, tape, and CD-ROM, which exist on the SCSI bus itself. In addition, the SCSI device driver contains a "generic" device head, which allows applications to manipulate other unsupported devices on the SCSI bus directly. Figure 1-4 on page 1-6 shows the SCSI subsystem.

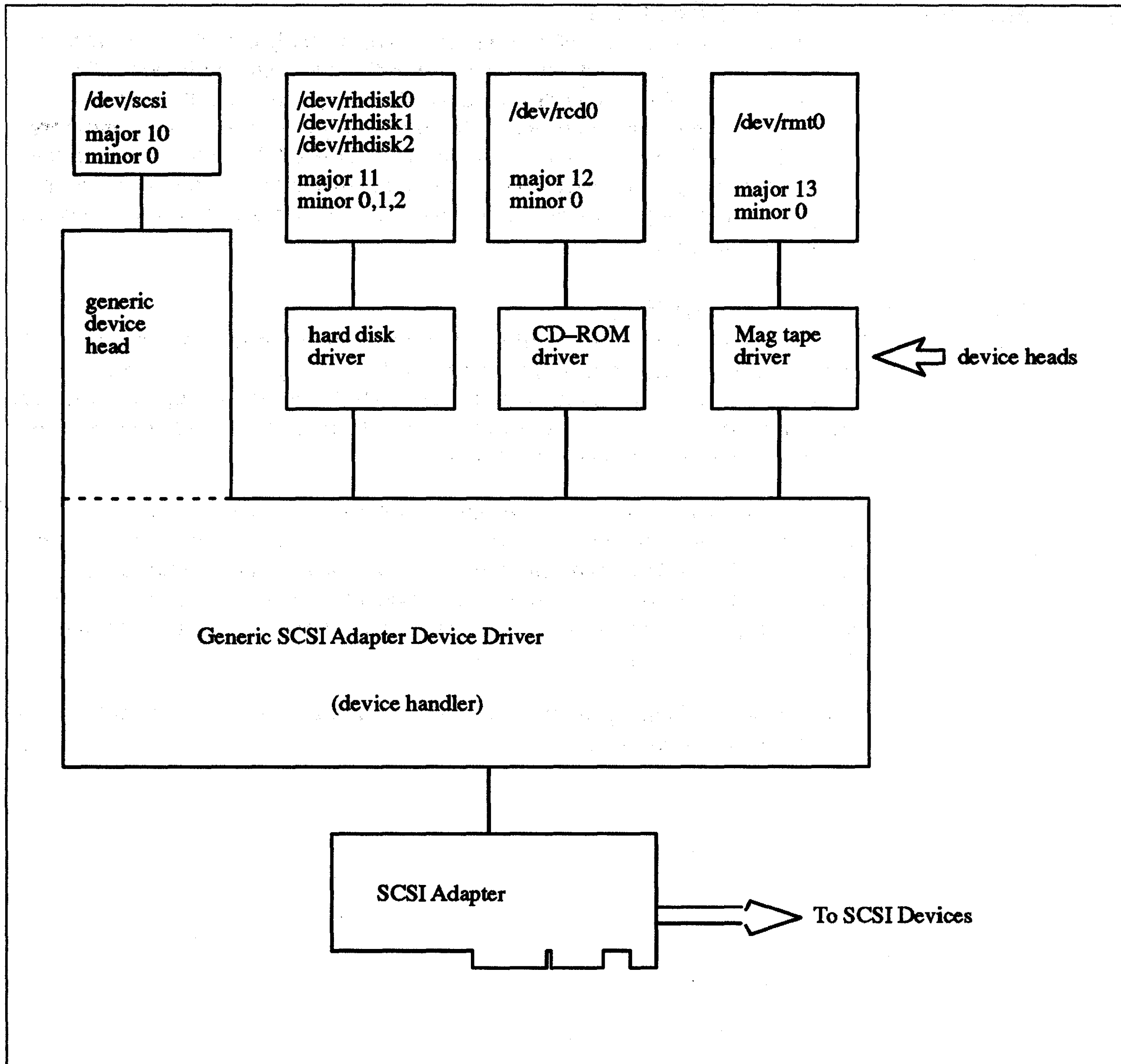


Figure 1-4. The SCSI Subsystem. This is an example of a complex system of device drivers. In this case, the generic SCSI driver provides both a device head and a device handler; the other drivers only provide the device head function. They call the generic SCSI driver for device handler operations.

1.4 Device Driver Structure

Device driver routines providing support for physical devices typically execute in two different types of environment, thus leading to a two-part structure. One part, referred to as the **top half** of the device driver, **always** executes in the process environment. Routines in this part typically provide the device head role, since they are always executed in the environment of the calling process.

The other part, referred to as the **bottom half** of the device driver, executes in the process or interrupt environment. Routines in this part normally provide the device handling role, since they deal with actual device I/O typically driven by hardware interrupts.

1.4.1 Device Driver Top Half Routines

Since routines in the top half of a device driver are only called in the process environment, the code and data accessed in this environment is normally **pageable**. The AIX kernel is designed to allow large portions of kernel code and data to be pageable in order to decrease the amount of physical memory required by the kernel. This is very important for the AIX kernel, because the design philosophy is to create fairly large data structures in pageable virtual memory. These large data structures can then support a wide range of system loads and configurations.

1.4.2 The Device Driver Bottom Half

The second half of the device driver structure is referred to as the bottom half. This half of the device driver typically consists of a routine that starts I/O operations, an interrupt handler, and (optionally) off-level interrupt handling and device time-out routines. The device driver's strategy and dump routines are also considered part of the bottom half.

This part of the device driver executes in both the interrupt handler environment and in the environment of the calling process. Both the code for this part of the device driver and the data it accesses must be **pinned** so that page faults are not taken in the interrupt execution environment. In addition, routines in the bottom half can utilize only kernel services that are specified as callable in the interrupt environment.

Please refer to "Pinning Device Driver Code" on page 10-1 for a discussion on how to pin device drivers.

Chapter 2. Programmer's model of the hardware

2.1 Micro Channel Overview

The Micro Channel architecture provides a standardized hardware interface for adding I/O devices to a computer. Implementations of this architecture provide a number of *slots*, which are electrical connectors into which circuit cards (commonly called *adapters*) may be inserted.

Each Micro Channel adapter provides addressable resources that reside in either the I/O address space, Bus Memory address space or the IOCC[1] address space.

1. An adapter may provide a set of *registers* (also called *ports*). These ports are each identified by an *address*, which is a 16-bit number. No two ports in a computer system can have the same address. These port addresses are said to make up the machine's **I/O address space**. As we will see, many adapters can be configured with the adapter's ports at a number of different addresses.

   Ports may be read and written. They are one byte wide. Some ports on some adapters may be read but not written; some ports may be written but not read; others may be both read and written. Each adapter's designer determines what ports the adapter will provide and what functions it will provide.

   In a simple adapter, writing a byte to a certain port might result in the byte being written to some media controlled by the adapter ... a printer, perhaps. Most adapters also provide ports that control the operation of the attached devices; the "baud rate" of a communications line, for example, might be changeable by software by writing to a port on the machine's serial adapter.

   In many computers, ports are manipulated using special I/O instructions. In the RISC System/6000 the I/O address space appears to be main memory to the central processor, and is accessed in that fashion. Certain actions must be taken by software to map the I/O address space into virtual memory; these will be described in further detail later.

2. Adapters may also provide *memory* resources to the computer system. In the PS/***n*** families of computers, these memory resources may be used by the system as primary main memory for the main processor. Adapter memory is configured to be present in a given address range; this range may frequently be assigned by software.

   Memory resources are often provided by adapters other than "memory cards." For example, display adapters often provide access to each pel displayed on the screen via memory resources.

---

[1] The IOCC is the RISC/6000 interface between the high-speed processor bus and the Micro Channel bus. The IOCC is responsible for data buffering and protection.

© Copyright IBM Corp. 1991

In the RISC System/6000, the Micro Channel is not used to provide main memory for the processor. Memory resources provided by Micro Channel adapters may be mapped into virtual memory using mechanisms similar to those used for the I/O address space. Note that memory provided by Micro Channel adapters resides neither in the processor's **real memory address space,** nor in I/O space; rather, it resides in a third memory address space, called **Bus Memory address space**. This address space uses 32-bit addresses, and has therefore a potential size of 4 GB.

3. **POS** : Programmable Option Select

   One major design goal of the Micro Channel is to allow devices to be configured entirely through software; no *DIP switches* or *jumpers* may be manipulated on Micro Channel adapters to set interrupt levels, memory addresses or other options. Additionally, host software must be able to determine which adapters are installed in a particular machine without manual configuration. These goals are accomplished through the Programmable Option Select (POS) features of the Micro Channel.

   Each Micro Channel adapter provides a set of *POS Registers* which may be read (and sometimes written) to query and set various options. A separate set of POS registers is provided for each Micro Channel slot provided in the machine in question. There are, in general, a maximum of eight POS registers for each adapter.

   POS registers are in a fourth address space called **IOCC address space**.

Refer to the *RISC/6000 Technical Reference Manual* for additional information on the Micro Channel.

2.2 System I/O Structure

2.2.1 Overview

General I/O bus support functions for load and store instructions, interrupt, and channel control are provided by the **I/O Channel Controller (IOCC)**. The Micro Channel is attached to the IOCC. Also attached to the IOCC and to the Micro Channel is the Standard I/O. Figure 2-1 describes the logical view of the IOCC in the Risc System/6000 machine.

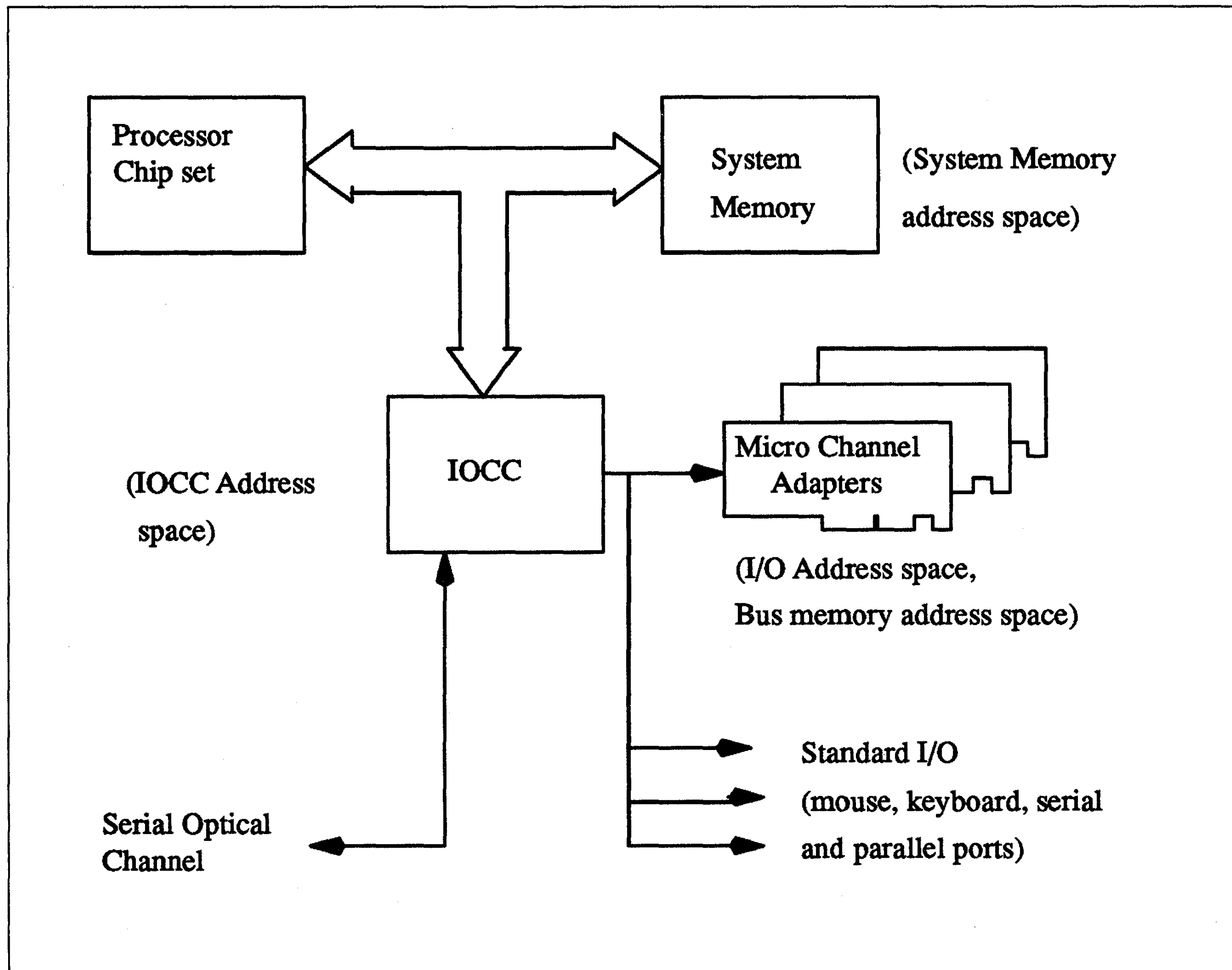


Figure 2-1. System Block Diagram

The IOCC architecture also includes support for DMA slave transfers, DMA master address translation and data buffering, load and store data buffering.

2.2.2 RISC System/6000 Addressing Model

What we have seen in "Micro Channel Overview" on page 2-1 can be summarized, and completed as follows :

- The RISC System/6000 can access up to 4 GB of **system** memory (32-bit physical addresses), grouped in 16 segments of 256 MB each.

- Next to that, the RISC System/6000 can also access 16 segments of 256 MB of memory on the Micro Channel bus (**bus-attached** memory).
- One other address space is also available: the IOCC address space.

The RISC/6000 addressing for system memory and for the (Micro Channel) bus attached memory is shown in Figure 2-2 on page 2-5.

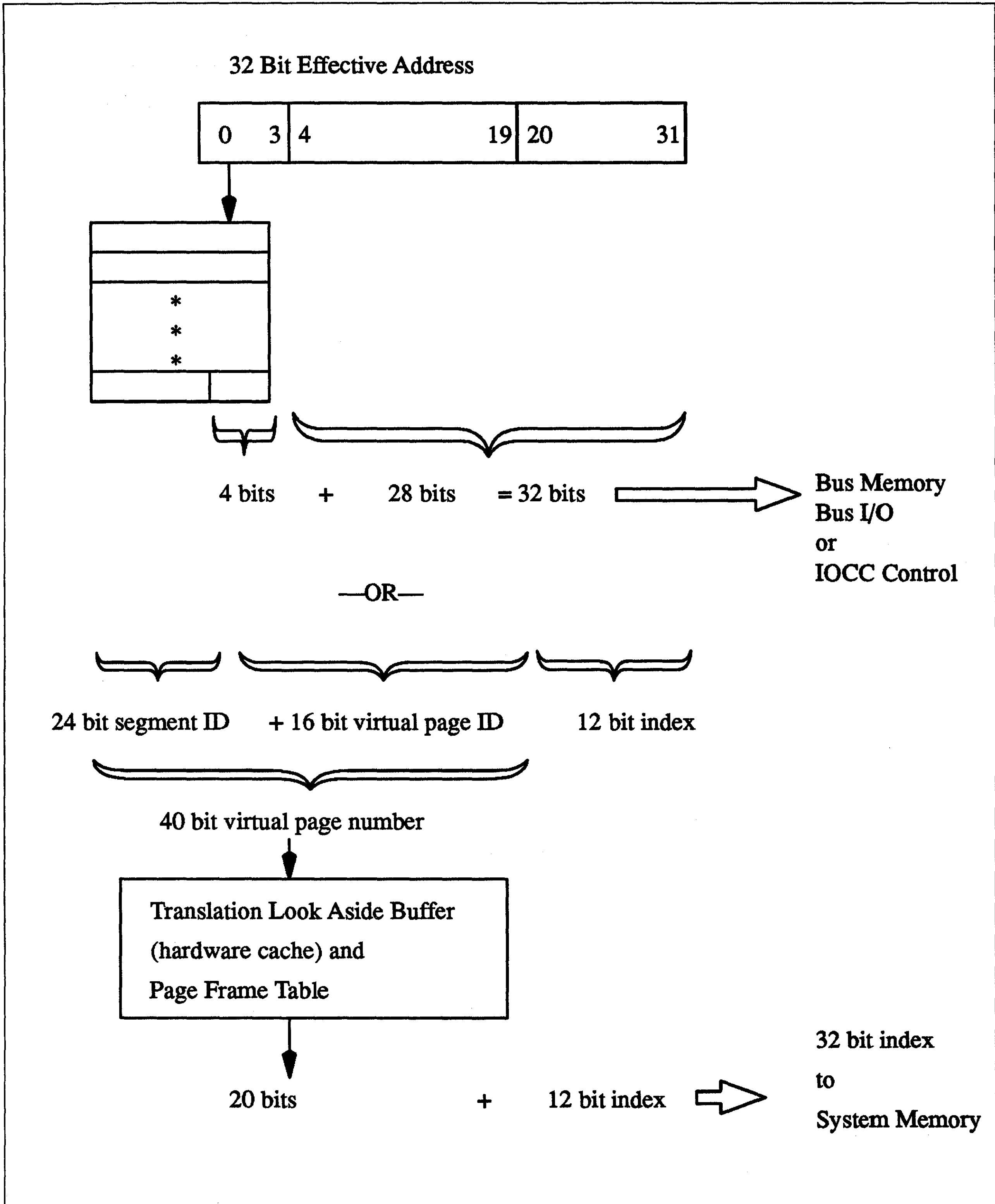


Figure 2-2. RISC System/6000 Addressing for Bus-Attached and System Memory

A 32-bit effective address is used to translate into a 32-bit real address in both cases.

For Micro Channel bus memory, I/O, or IOCC control, a 4-bit extention from a segment register is used to form this 32-bit address. This, in addition to proper

control bits in the segment register, is used to decode the IOCC control mode from the bus memory and I/O modes.

For system memory, a 24-bit extention from a segment register is appended to 16 bits from the effective address to form a 40-bit virtual page address. This 40-bit address is then mapped by either hardware (TLB) or software (page frame table) to form a 20-bit real page address. This 20-bit page address is then concatenated with a 12-bit index from the effective address to get the 32-bit real address.

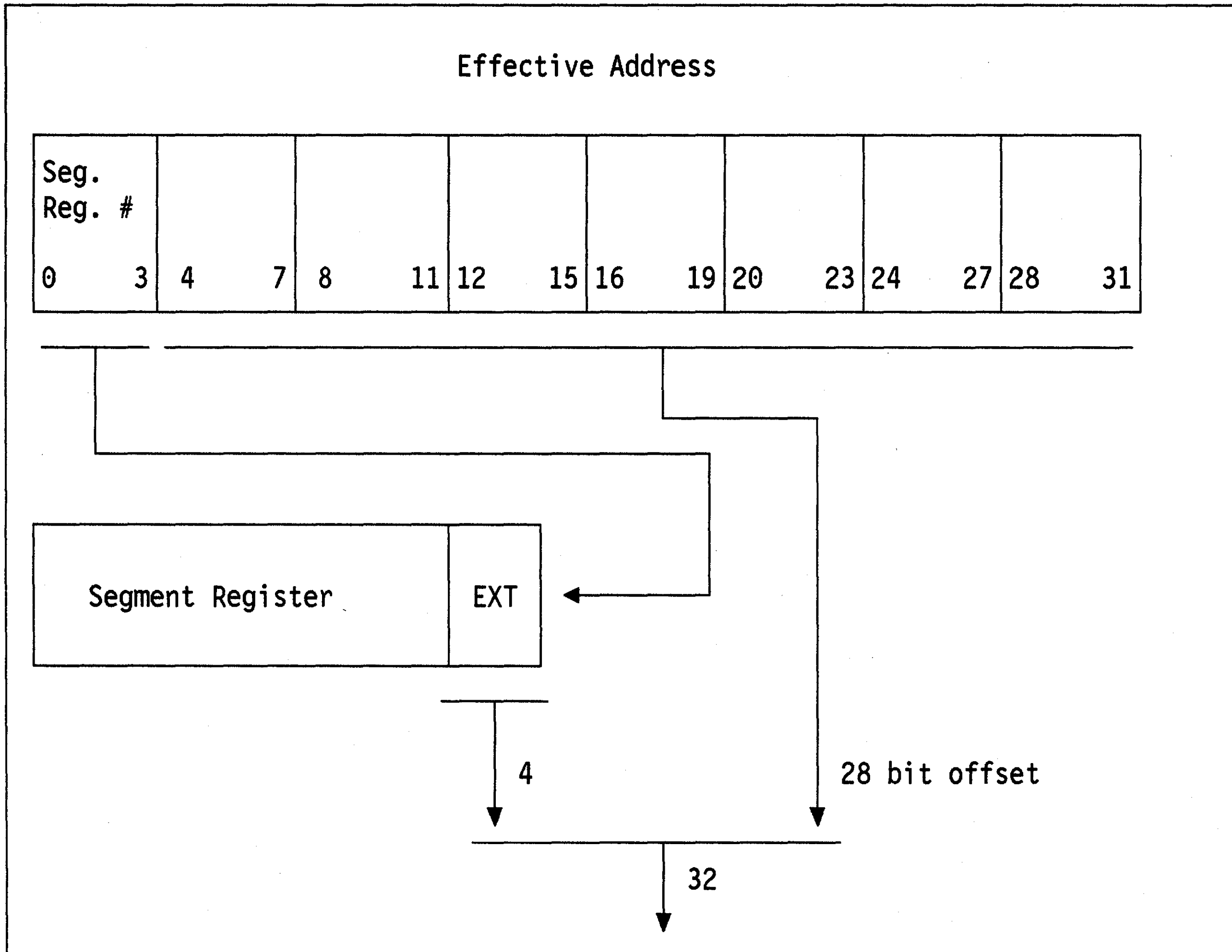


Figure 2-3. Address Translation - Bus Memory, I/O, and I/O Control

Load and Store instructions can be issued to devices on the I/O bus in a similar manner to those issued to system memory. The programmer specifies a segment register identifying a specific address space and supplies an offset into that space. The offset is obtained from the effective address and is not translated prior to being applied as a bus address. Figure 2-3 shows that address translation mechanism.

The bit meaning of the segment register (see Figure 2-4 on page 2-7) when using IOCC and I/O bus addresses (i.e. when T=1--NOT system memory address space) are:

Bits	Description
0	This bit defines if the Load or Store instruction is targeted to system memory or I/O address spaces (T=0 means system memory access).
1	Privilege key: this bit is generally set to 0 when the operating system is in control, and 1 when in the user mode.
2 - 3	These bits are reserved and should be set to 0.
4 - 11	Bus Unit Identification **(BUID)**: the field is decoded to select the IOCC. Addresses between x'20-23' are assigned to the IOCC, but the present version of the IOCC is cabled for a BUID of x'20. This field should thus be set to x'20.
12 - 13	Address check and increment: should be set to 1.
14 - 23	These bits are reserved and must be set to 0.
24	IOCC Select: this bit selects the IOCC control mode when set to 1.
25	RT Compatibility Select: this bit selects the RT compatibility mode when set to 1 and when the IOCC Select bit equals 0.
26	Bypass: this bit should be set to 1.
27	This bit is reserved and must be set to 0.
28 - 31	Bus Memory Extent: this field is concatenated with effective address bits 4 to 31 to form a 32-bit I/O bus address when working in bus memory mode.

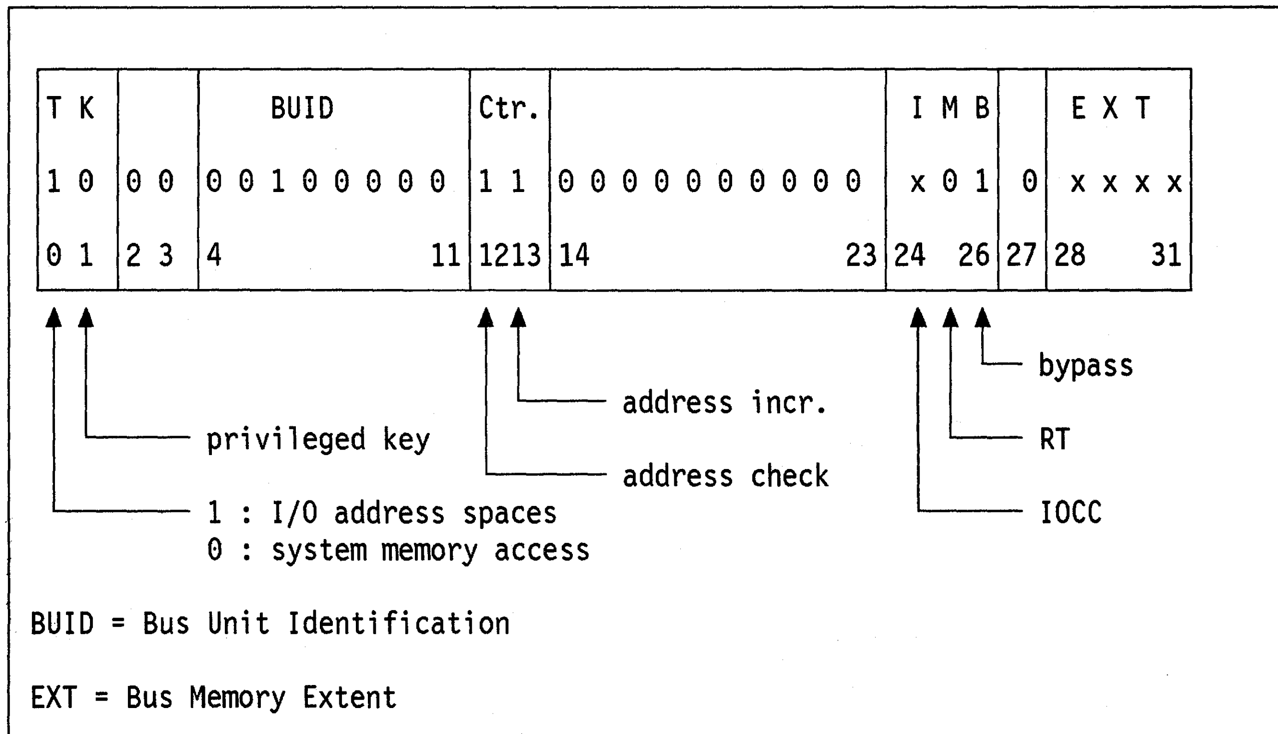


Figure 2-4. I/O Segment Register

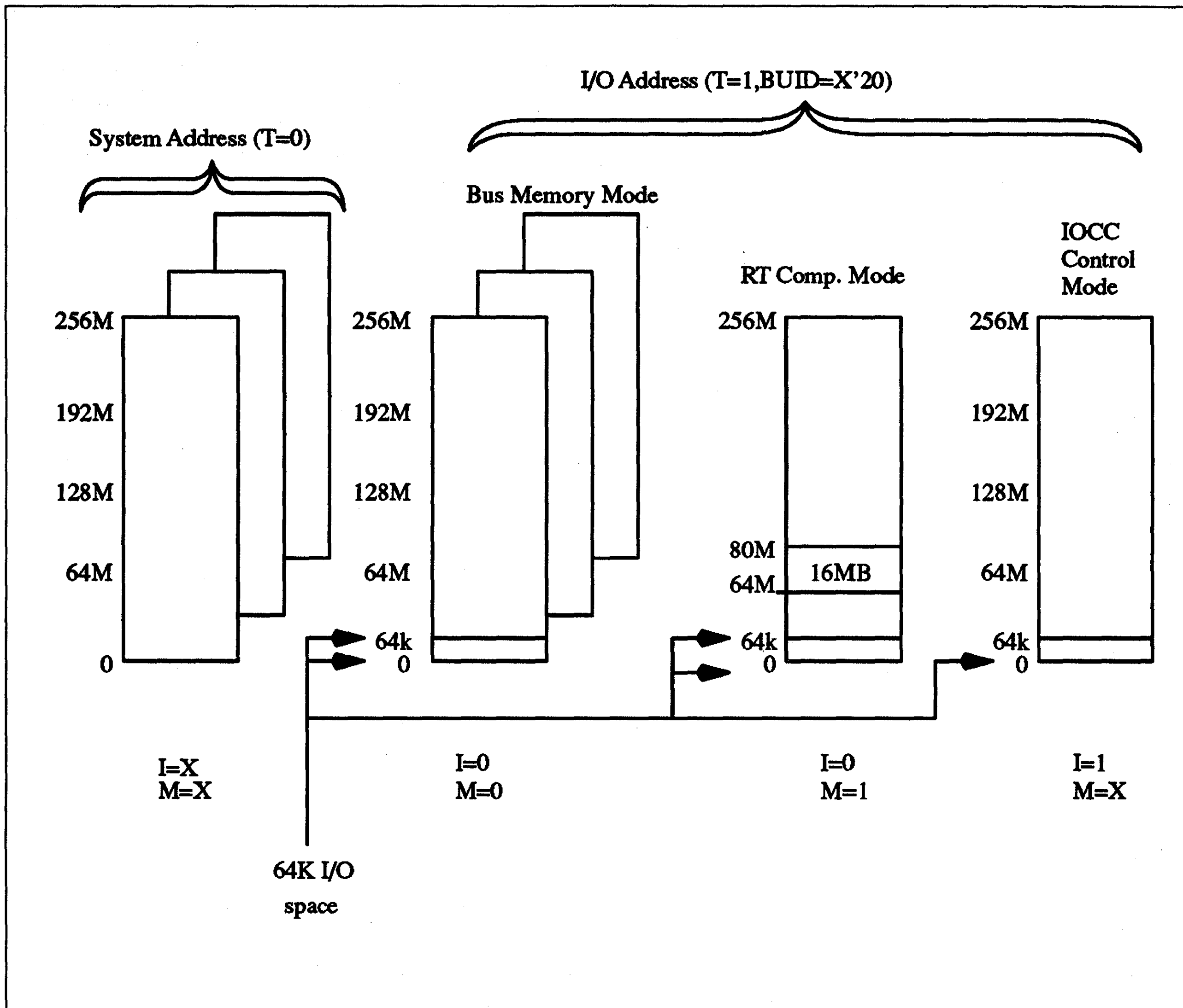


Figure 2-5. RISC System/6000 Addressing Model

Figure 2-5 gives a complete view of the different possible addressing modes. These address spaces are selected by way of control bits in the Segment register resulting in five different effective address operating modes as follows :

1. **System address mode**: when T=0 in the segment register, the system memory is accessed.

2. **Bus memory mode**: when T=1, I=0, and M=0, the memory on the Micro Channel is accessed. The 32-bit bus memory address is formed by concatenating 28 bits of the effective address with the 4 extent bits from the segment register. This partitions the bus memory address into 16 segments of 256 MB, for a total of 4 GB. Separate segment registers must be used for addressing different segments.

3. **I/O devices mode**: the 16-bit device address is taken directly from the lower 16 bits of the effective address. To address a device within the 64 KB Micro Channel I/O space, effective address bits 4 through 15, and segment register bits 28 through 31 must all be set to 0. Effective addresses are not translated, but are used as real addresses into the I/O space. Note that these 64 KB can be accessed when utilizing bus memory mode, RT compatibility mode, and IOCC control mode as illustrated in Figure 2-5.

4. **RT compatibility mode**: this addressing mode assists in the simulation of the RT system allowing for 24-bit addressing. In this mode, the segment register control bits are in the following state, T=1, I=0, M=1. In this mode, the 16 MB of bus memory are mapped into the effective address range going from 64 MB to 80 MB. Please refer to the *RISC/6000 Hardware Technical Reference Manual* for more information on the RT compatibility mode.

5. **IOCC Control mode**: when T=1 and I=1. Included in this address space are IOCC registers, address translation tables for Bus Master and DMA Slave transfer (TCW and TAG tables), and Non-Volatile Random Access Memory (NVRAM). Access to the adapters' **POS registers** is also achieved by using this address space.

2.3 I/O Data Transfer Protocols

2.3.1 Programmed I/O Mode

This mode provides data transfer capability between processor General Purpose Registers and memory (system or bus), I/O space or IOCC space. Up to 128 bytes can be transferred with one command. The IOCC contains a 128-byte PIO buffer to handle the data width mismatch between the 8-byte System Bus and 1-byte, 2-byte or 4-byte target device.

See "I/O Macros" on page 2-14 for the commands that perform the programmed I/O operations.

2.3.2 DMA Transfers

2.3.2.1 Types of DMA Adapters

The Micro Channel supports two types of DMA adapters. These are DMA slaves and DMA masters.

A DMA slave adapter is the simpler form of adapter. It requires extensive system support to generate addresses and control the transfer length. The system hardware limits a DMA slave adapter to performing only one sequential transfer at a time.

A DMA master generates its own bus address and controls its own transfer length. A DMA master adapter is therefore only limited by its own hardware in the number and type of transfers that it can perform. For example, a DMA master disk adapter can support one or more concurrent DMA transfers for each disk connected to it. A DMA master LAN adapter can support having the header at one location in system memory and the data at another location.

2.3.2.2 Block DMA Transfers

A block DMA transfer consists of transferring data between sequential locations on the adapter and sequential locations in memory. All DMA slaves are essentially limited to this type of transfer.

A DMA slave can have only one contiguous block transfer in progress at any one time. The maximum size of this transfer is currently x'100000 bytes, as defined in the *<sys/sysdma.h>* header file.

A DMA master can have one or more block transfers in progress at any one time. Each transfer must be assigned part of that DMA master's fixed-size window into system memory. This window is assigned to the adapter during system configuration.

A device driver must call either the **d_slave** service to set up a DMA slave transfer or the **d_master** service to set up a DMA master transfer. The device driver should then set up the device to perform the DMA transfer. The device transfers data when it is available and interrupts the processor upon completion of the DMA transfer. The device driver then calls the **d_complete** service to clean up after the DMA transfer. These steps are typically repeated each time a DMA transfer is to occur. See "DMA Management" on page 3-11 for more information on using the DMA kernel services.

2.3.2.3 DMA Processing

Direct memory access (DMA) allows a device to access memory without going through the processor. Using DMA consists of the following steps:

1. Allocating a DMA channel.
2. Initializing the DMA channel.
3. Enabling the DMA channel.
4. Performing one or more DMA transfers.
5. Disabling the DMA channel.
6. Freeing the DMA channel.

The DMA transfer itself, in Step 4 above, consists of the following steps:

1. Arbitrating for the bus.
2. Generating an address.
3. Performing the data transfer.

The AIX kernel provides a set of services that assist in performing DMA operations. "DMA Management" on page 3-11 will provide more information on how to use these services.

2.3.2.4 DMA Channels and How They Are Assigned

A DMA channel is the means by which DMA transfers for different adapters are distinguished from each other. A DMA channel is a resource that cannot be shared simultaneously by two adapters.

The Micro Channel allows for assignment of DMA channels at system configuration time. Each time the RISC System/6000 is booted, the Micro Channel bus configuration method scans the bus and creates a list of all adapter cards plugged into the slots. For each adapter plugged into a slot, the method uses the adapter ID (sensed from the POS registers) to look up the adapter's assignable resources in the device's database. (This is known as the "predefined attributes" (PdAt) of the ODM database.)

If the adapter uses a DMA channel, the database describes all possible DMA channels to which the adapter can be programmed and a default or preferred choice. The bus configuration method then selects a unique DMA channel for each adapter requiring DMA in the system. The assigned DMA channel

numbers are written into the Customized Attributes (CuAt) database object for each adapter if the adapter's default value is not used. (If the default value is used, the CuAt database does not get modified.)

The **busresolve** system call is used from the device's configuration method. This system call is used to resolve the resource (DMA and interrupt) conflicts on the Micro Channel bus. If the resources cannot be resolved at this time, the configuration method should return a failure return code. See "The busresolve system call" on page F-1 for more details.

When the adapter's specific configuration method is called later in the configuration process, it reads the assigned DMA channel or channels from the database for the specific adapter being configured. The adapter's configuration method then puts these channels in a device-dependent structure used to initialize the device driver supporting the adapter.

When the device driver for the adapter in the specified slot is initialized, the information in the device-dependent structure is written to the adapter's POS registers. This is done by the configuration routine of the device driver. This action properly configures the adapter.

Please see "Device Drivers Configuration" on page 6-1 for more information on device driver configuration.

2.4 Interrupt Processing

Adapters on the Micro Channel can generate interrupts to the host CPU. Each interrupt is associated with a particular "Interrupt Request Level" (IRQ), which is a small positive integer. One adapter might generate interrupts on IRQ2, while another might generate interrupts on IRQ3. Assignment of IRQ levels to adapters is done by setting POS registers on the adapter during device initialization. Adapter interrupt levels are assigned in the same manner as the DMA levels.

CAUTION

Do not change the Predefined Attributes (PdAt) of the RISC/6000 by removing information that was shipped with the base AIX product.

IRQ levels may be *shared*; i.e. more than one adapter may generate interrupts at the same IRQ level. Each adapter will provide a register which may be interrogated to determine if the current interrupt is due to this particular adapter; this allows software to determine which adapter actually generated the interrupt.

Note that interrupt sharing on a particular level gives you worse performance than having each adapter on a separate interrupt level.

Each adapter also provides a procedure, which software must follow, that will *reset* an interrupt indication; this must be done before any more interrupts will be generated by the adapter.

A device driver may provide an *interrupt handler*. This is a C function that will be called by the AIX kernel whenever an interrupt occurs on a given IRQ level. The interrupt handler must first determine if the interrupt was indeed caused by the adapter this driver is managing; if not, the handler exits immediately. The interrupt handler then performs whatever processing is needed to deal with the interrupt; it then resets the interrupt both in the adapter and in the Micro Channel and returns to its caller.

Interrupt handlers, like most of the AIX kernel, are *preemptable*. They run with some interrupts enabled. When a device driver configures itself, it specifies the priority of interrupts from its associated adapter. When an interrupt occurs, only interrupts from devices at that priority level and below are disabled; higher priority interrupts may still occur. This allows interrupt handlers for low priority devices (a printer, for example) to be preempted if an interrupt occurs on a high priority device (an unbuffered high-speed communications link, perhaps).

Routines are provided by AIX that allow interrupts to be disabled and enabled at higher levels during the operation of the interrupt handler. This should be kept to a minimum and used with caution so that the higher priority interrupts can be serviced.

2.4.1 Priority Assignment

A device's interrupt priority is selected based on two criteria: its maximum interrupt latency requirements and the device driver's interrupt execution time. The interrupt latency requirement is the maximum time within which an interrupt must be serviced. (If it is not serviced in this time, some event is lost or performance is degraded seriously.) The interrupt execution time is the number of machine cycles required by the device driver to service the interrupt. A device with a short interrupt latency time must have a short interrupt service time. In other words, a device that will "lose" data if not serviced quickly must have a higher priority interrupt level. This in turn requires that it spends less time in the interrupt handler. The general rule for interrupt service time is based on the following interrupt priority table:

Interrupt Priority Versus Interrupt Service Times

Priority	Service Time (machine cycles)
INTCLASS0	less than 200 cycles
INTCLASS1	greater than 200 but less than 400 cycles
INTCLASS2	greater than 400 but less than 600 cycles
INTCLASS3	greater than 600 but less than 800 cycles
INTOFFL0	less than 1500 cycles (off-level priority)
INTOFFL1	greater than 1500 but less than 2500 cycles (off-level priority)
INTOFFL2	greater than 2500 but less than 5000 cycles (off-level priority)
INTOFFL3	5000 cycles or greater. (off-level priority)

Two other predefined priorities are available: **INTMAX** which corresponds to all interrupts disabled, and **INTBASE** which corresponds to all interrupts enabled.

See <sys/m_intr.h> for the associated priority levels of these interrupts.

HINT

A method for finding out your interrupt class could be as follows. Put a trace hook with a time stamp into both the entry and the exit points of your interrupt handler. (Please see "Performance Tracing for AIX" on page 9-44 for trace hook information.) The resulting trace will tell you the cumulative time that was spent in the handler. Dividing this time by the cycle speed of your RISC System/6000 system will give you the number of cycles used.

2.4.2 Off-Level Interrupts

The INTOFFLn interrupt priorities are for off-level interrupt processing. Typically, they are used when the interrupt service time for an operation exceeds the time allowed at that interrupt priority. For example, if a device interrupts and you know that it will take more than 800 cycles to service, your interrupt handler should reschedule it so that it will use one of the off-level routines.

The **i_sched** service is used to schedule off-level processing. The operation is then set up to be performed at an off-level interrupt priority. This allows other device interrupts to preempt the operation of the off-level handler at a small cost of additional system overhead.

Operations that do not meet the off-level service time requirements must be scheduled to be performed under a kernel process in order to maintain adequate system real-time performance. Device driver routines providing the device handler role often include an off-level processing routine. The kernel calls the off-level routine to perform device-specific processing after the following events have taken place:

- The interrupt handler has completed its processing.
- The interrupt has been reset.

The processing associated with a device interrupt can be time consuming. The off-level routine allows a device to perform this processing at a less favored priority. This in turn enables interrupt handlers to run as fast as possible by avoiding interrupt-processing delays and device overrun conditions.

This routine must be part of the bottom half of the device driver when present.

2.5 Addressing Micro Channel Adapters

2.5.1 Identifying an Adapter

The first two POS registers for each adapter define what kind of adapter it is. These two bytes contain a read-only number which uniquely identifies the functions and capabilities of the adapter. A particular make and model of Ethernet adapter might be identified by number 0x1234, while a particular variety of SCSI disk controller might be identified by number 0x9876. IBM maintains a central registry for these numbers; even adapters produced by other companies are guaranteed to be uniquely identified by the value of their first two POS registers.

The AIX Version 3 boot process uses the adapter ID to determine what type of Micro Channel adapter resides in each of the machine's expansion slots; this determines which device drivers are loaded as part of the system.

Note:

The unique instance of an adapter (if there are more than one of that same type of adapter) is by the POS ID and SLOT NUMBER.

Also, the POS ID identifies only the base adapter and NOT a "daughter board" that may be plugged into that adapter. Therefore, an adapter with more that one kind of daughter board cannot be differentiated unless the POS IDs are different.

2.5.2 Setting Adapter Attributes

The first two POS registers for each Micro Channel slot in the system (POS register 0 and POS register 1) contain the adapter ID, described above. The other POS registers control various attributes of the adapter. These attributes vary from adapter to adapter, but some typical attributes are:

- The *Interrupt ReQuest level (IRQ)* on which the adapter will generate interrupts (discussed later).
- The beginning I/O address of the adapter's ports.
- The beginning "memory" address of the adapter's RAM and/or ROM (if any).

Other specialized attributes may also be set using the POS registers. The exact bit values used to set various options is dependent on the adapter in question; see the technical information provided with the adapter for specific bit assignments.

2.5.3 Enabling an Adapter

The low order bit of POS register 2 (the third register) indicates whether the adapter is enabled or not. If this bit is set to zero (as it is when the system is booted or powered up) then the adapter is disabled; it may not generate interrupts or otherwise make use of the system's resources, nor may the system make use of the adapter's resources. After the adapter has been correctly configured by the device driver and once the device driver is ready to process interrupts from the device, the device driver must modify POS register 2 to set this bit. Once the bit is set, the adapter may generate interrupts and may be the target of input and output requests from the host CPU.

(Note that you should disable your adapter when you are unconfiguring it as well as for detected adapter error conditions that result in the shutdown of your adapter.)

2.5.4 I/O Macros

As discussed before ("Micro Channel Overview" on page 2-1), in the RISC System/6000, POS registers appear to be virtual memory. They are maintained in a memory segment which is managed by the IOCC hardware component (see "Overview" on page 2-3).

Kernel services exist to manipulate virtual memory, but they should never be used because very convenient C language macros can also be used to do the same job (examples will be given in "Setting POS Registers from within a Device Driver" on page 2-18 and "Simple I/O" on page 2-19). For instance, **IOCC_ATT** will map the IOCC segment into the device driver's virtual address space and **IOCC_DET** will unmap (delete) the IOCC segment from that address space. Another macro, **POSREG(** ***n,m*****)**, will return the address of the ***n***th POS register of the adapter in slot ***m***. This address can be used quite easily as a pointer to the POS register. Fetching the byte at that address, using the **BUSIO_GETC** macro, will give the current value of the register, while setting the byte (with the **BUSIO_PUTC** macro) at that address will modify the POS register.

See <sys/mdio.h> for POSREG and <sys/ioacc.h> for BUSIO_GETC and BUSIO_PUTC.

As discussed above, most Micro Channel adapters provide a number of ports through which they can communicate with host software. Like POS registers, these registers are also memory mapped in the RISC System/6000. The **BUSIO_ATT** macro will map the I/O segment into a device driver's virtual address space; the address of the port in question (set by the POS process) is used as a displacement within that segment to determine the memory mapped address of the port. Once the address is determined by reading the dds, the port may be accessed through a number of convenient macros provided by AIX Version 3. When the accesses are complete, the **BUSIO_DET** macro will remove the I/O segment from the virtual address space.

Memory provided on Micro Channel adapters can also be manipulated using the **BUSMEM_ATT** and **BUSMEM_DET** macros.

Note that for code running **in user mode**, like a configuration method (see "Device Drivers Configuration" on page 6-1), the *machine device driver* is used to access the IOCC and I/O address spaces. With that device driver, you can directly read or write a POS register, or any adapter port by using the system call IOCTL on /dev/bus0 with different commands:

- MIOCCGET and MIOCCPUT allow you to read or write in the IOCC address space
- MIOBUSGET and MIOBUSPUT allow you to read or write in the I/O bus address space.

We will see later an example (see "Querying POS Registers from a Configuration Method" on page 2-18) of how to use this device driver.

2.5.4.1 Attach/Detach Macros

In order to access I/O resources with the **BUSMEM_ATT**, **BUSIO_ATT**, or **IOCC_ATT** macros you must supply what is unfortunately called a *bus ID*. The bus ID is in fact the value you need to load to a segment register (i.e. the whole word shown in Figure 2-4 on page 2-7). This word should not be confused with the *Bus Unit IDentification* which is the ID of the IOCC you want to talk to (should always be x'20), and which is part of the bus ID!

The bus ID value to use when manipulating IOCC resources (POS registers, etc.) is defined in */usr/include/sys/iocc.h* by the name **IOCC_BID**. The bus ID to use when accessing resources on the Micro Channel is never defined in any

documentation; but if you choose the recommended values (see again Figure 2-4 on page 2-7), you will find that:

```
#define BUS_ID 0x820C0020
```

will work for bus memory and bus I/O access.

The following list describes all the Attach/Detach macros :

- BUSIO_ATT(bid,io_addr)
- BUSIO_DET(io_addr)
- BUSMEM_ATT(bid,mem_addr)
- BUSMEM_DET(mem_addr)
- IOCC_ATT(bid,iocc_addr)
- IOCC_DET(iocc_addr)

The above macros are defined in */usr/include/sys/adspace.h*.

The purpose of the Attach macro is to give you an index (i.e. segment register content) to use to access a certain address space (bus memory, bus I/O, or IOCC).
They are all basically doing the same job:

1. Allocate a segment register.
2. Load the segment register with the specified value (bid).
3. Modify the specified address so that it will select the proper segment register at address translation time (see Figure 2-3 on page 2-6).
4. Return the modified address.

This is done by calling the same kernel service (*vm_att*) in all macros. The difference comes from the "protection" (by use of masks[2] before Step 2) that is used between the bid you give and the one passed to the vm_att service.

The Detach macros are all the same (they simply call the *vm_det* service to deallocate the segment register), but are convenient for writing *readable* code: *something*_**attach** should always be followed by *something*_**detach**.

IMPORTANT

Although segment registers are saved whenever another process or interrupt handler executes, it is always important to use the proper Detach macro (BUSIO_DET, BUSMEM_DET, or IOCC_DET) when your access is complete. This will free the segment register that you have used. The system will crash if you issue a BUSIO_ATT, BUSMEM_ATT, or IOCC_ATT macro and there are not any available segment registers.

[2] For instance, the IOCC_ATT macro masks every bit except the buid (the "IOCC number") part. If you consider the fact that so far only x'20 is valid as a buid, and the fact that the predefined IOCC_BID has a value of x'820C00E0, you will see that the macro is really secure!

2.5.4.2 Data Transfer Macros

These macros are defined in */usr/include/sys/ioacc.h*.

1. Read or write the specified data to or from bus memory

BUS_PUTL(p,v)	Write the specified unsigned long value (v) to the supplied bus memory address (p).
BUS_PUTS(p,v)	Same, but v is a short.
BUS_PUTC(p,v)	Same, but v is a char.
BUS_GETL(p)	Read an unsigned long value from the supplied bus memory address (p).
BUS_GETS(p)	Read a short.
BUS_GETC(p)	Read a char.

2. Read or write the specified data to or from bus I/O including the IOCC

BUSIO_PUTL(p,v)	Write the specified unsigned long value (v) to the supplied bus I/O address (p).
BUSIO_PUTS(p,v)	Same, but v is a short.
BUSIO_PUTC(p,v)	Same, but v is a char.
BUSIO_GETL(p)	Read an unsigned long value from the supplied bus I/O address (p).
BUSIO_GETS(p)	Read a short.
BUSIO_GETC(p)	Read a char.

3. Multi-byte (string) macros

BUS_PUTSTR(d,s,l)	Write the specified number of bytes (l) to the destination bus memory address (d) from the source system memory address (s).
BUSIO_PUTSTR(d,s,l)	Same to I/O address.
BUS_GETSTR(d,s,l)	Read the specified number of bytes (l) from the destination bus memory address (d) from the source system memory address (s).
BUSIO_GETSTR(d,s,l)	Same from I/O address.

Look at /usr/include/sys/ioacc.h

To use the data transfer macros, it is helpful to look at their definition in /usr/include/sys/ioacc.h. Included in this header file are some additional declarations of macros for bus memory and bus I/O that also perform byte reversal operations.

2.5.5 Sample I/O on the RISC System/6000

2.5.5.1 Querying POS Registers from a Configuration Method

From a configuration method the *machine device driver* is used to examine the POS registers. The following code, taken from the samples shipped with AIX Version 3, returns in *cardid* the adapter ID from the adapter in the specified slot.

```
#include <sys/mdio.h>
int slot;
ushort cardid;
MACH_DD_IO mddRecord;
uchar pos[2];
int fd;

pos[0] = 0xff;
pos[1] = 0xff;

fd = open("/dev/bus0", O_RDWR)

mddRecord.md_size = 2;
mddRecord.md_incr = MV_BYTE;
mddRecord.md_data = pos;
mddRecord.md_addr = POSREG(0, slot);

ioctl(fd, MIOCCGET, &mddRecord)

close(fd);

cardid == ((pos[0] << 8) | pos[1]))
```

2.5.5.2 Setting POS Registers from within a Device Driver

To manipulate POS registers from within a device driver a different method is used. Here is some code from a device driver that sets **POS register 2** to **0x32**.

```
#include <sys/mdio.h>
#include <sys/adspace.h>
#include <sys/iocc.h>
#include <sys/ioacc.h>
int bus_val;
char pos2;

pos2 = 0x32;

/* Gain access to the bus */
bus_val = IOCC_ATT(IOCC_BID, 0);  /* bus_val is now a pointer to
                                     the beginning of the iocc address space */

pptr = bus_val + IO_IOCC + POSREG(2, slot_number);
               /* IO_IOCC is an index to the start of all POS registers
               in the IOCC address space */

BUSIO_PUTC(pptr, pos2);    /* Load the contents into POS2 */
IOCC_DET(bus_val);
```

Note that the POS registers are not at the beginning of the IOCC address space. They begin at the **IO_IOCC** [3] address. Therefore, you have to add that value to the address returned by IOCC_ATT.

2.5.5.3 Simple I/O

Writing the value **0x12** to the I/O port at I/O address **0x9876** could be done via the following code:

```
#define BUS_ID 0x820c0020   /* Micro Channel Bus ID */

unsigned char* p;
ulong bus_val;

bus_val = BUSIO_ATT(BUS_ID, 0);
p = (unsigned char*) (bus_val + 0x9876);
BUSIO_PUTC(p, 0x12);
BUSIO_DET(bus_val);
```

2.5.6 Byte Reversal from the System Bus to the Micro Channel Bus

As Figure 2-1 on page 2-3 shows, data from the processor chip set or the system memory to (or from) the Micro Channel adapters passes through the IOCC. The IOCC translates this data from the IBM system bus to the Micro Channel bus by performing byte swapping.

When programs write (or read) data to the hardware residing on the Micro Channel, they need not be concerned with this byte swapping. This is done automatically by the IOCC. Care must be taken when passing data structures to/from the hardware adapters that have vendor microprocessors resident. In this case, it may be necessary to swap bytes so that the bytes will reside where you expect them to.

Figure 2-6 on page 2-20 shows how the IOCC swaps bytes for a transfer to a 16-bit Micro Channel device.

Figure 2-7 on page 2-21 shows how a transfer is done to a 32-bit Micro Channel device.

[3] **IO_IOCC** is defined in */usr/include/sys/iocc.h*.

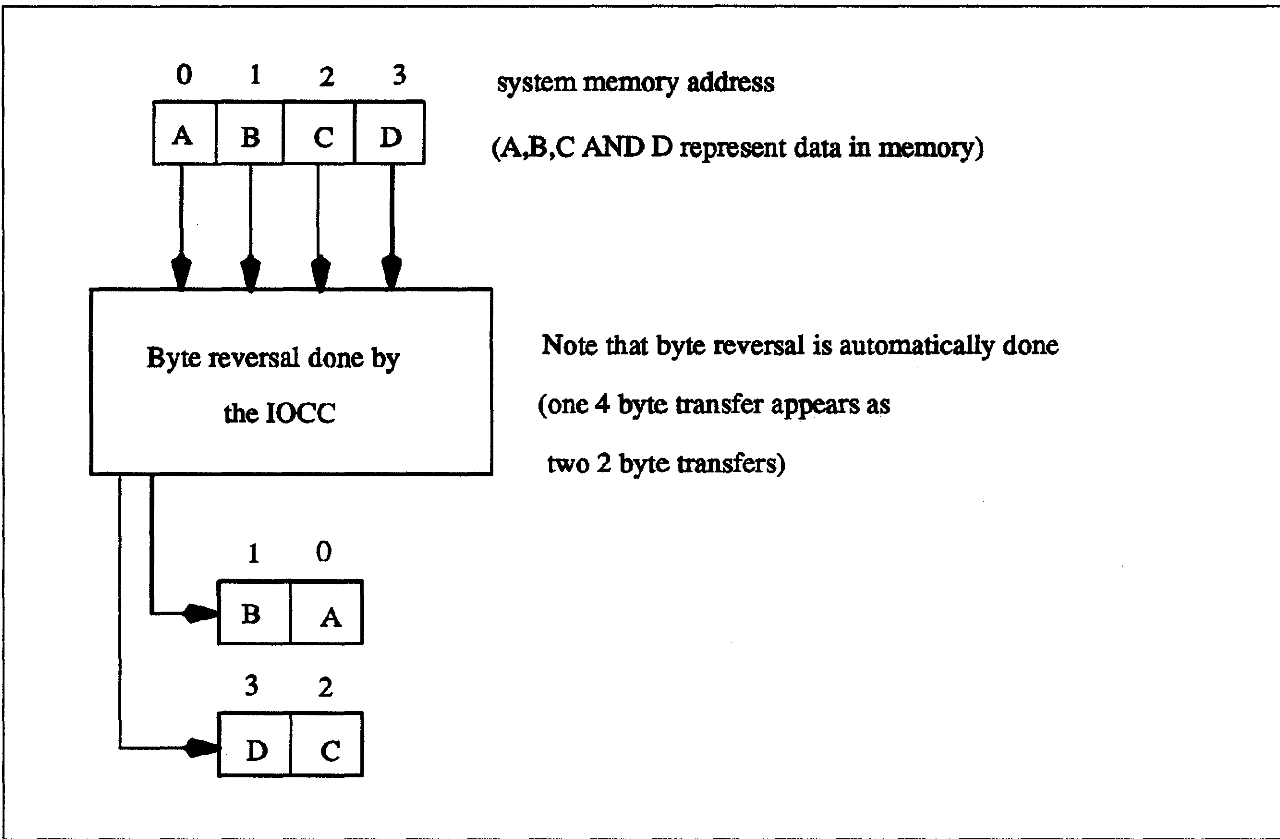


Figure 2-6. Data Transfer to a 16-Bit Micro Channel Device

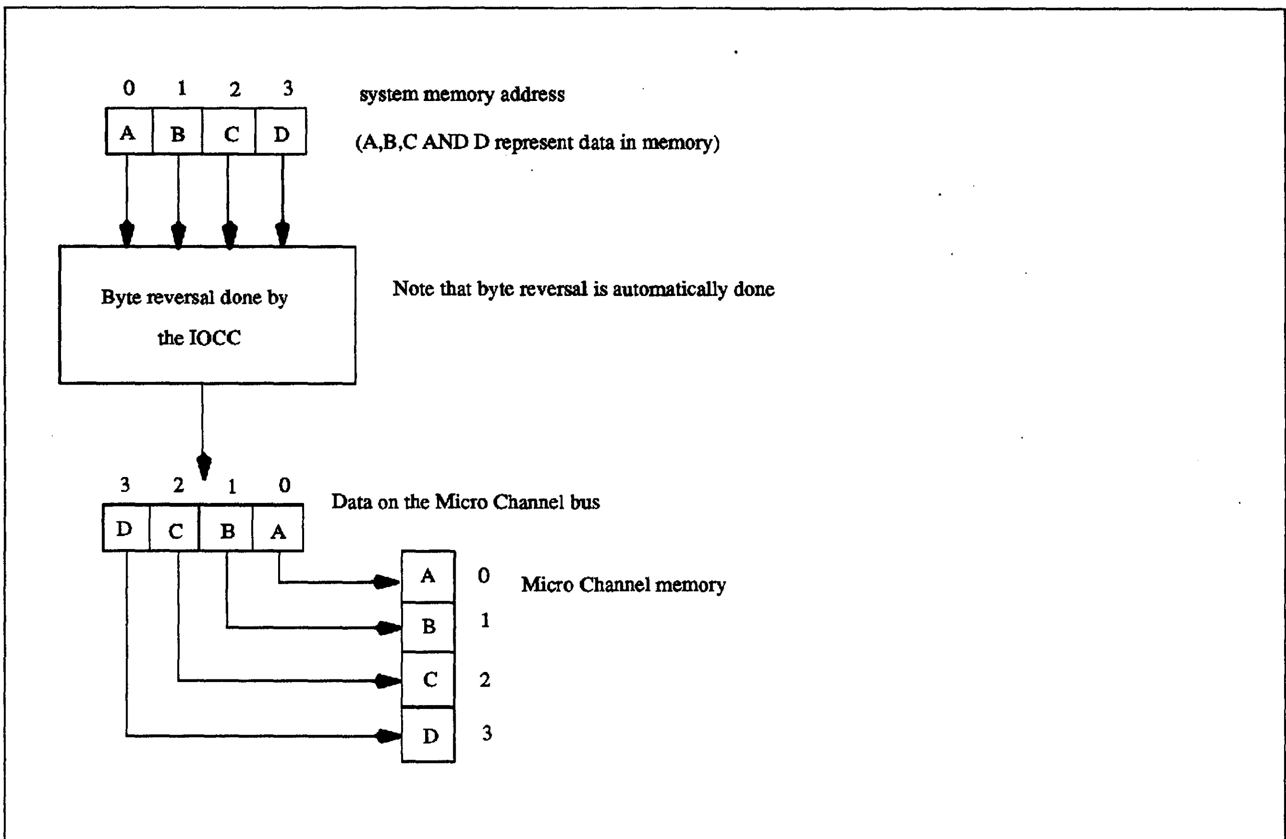


Figure 2-7. Data Transfer to a 32-Bit Micro Channel Device

2.5.7 Additional PIO Macro Information

In AIX version 3.1.5 and later, TWO versions of macros that provide access to the I/O bus are provided in the <sys/ioacc.h> header file. A version of the macros without an X as the last character of their name has no error handling built into it and should be used in conjunction with the **pioassist** or **setjmpx / clrjmpx** services for handling I/O errors. (This version also exists in the earlier levels of AIX version 3.) A second version of I/O macros (which was added in the AIX 3.1.5 release) is denoted by an X in the last character of their name, has a low overhead error catching mechanism built into it. These macros provide an alternate mechanism for performing programmed I/O by utilizing a very fast kernel mechanism for catching I/O errors that occur during the programmed I/O transfer. These macros return a 0 return value if no error occurred, or return a non-zero value if an error occurred during the I/O transfer. These macros utilize kernel routines that provide much faster exception handler setup than the **setjmpx**, **clrjmpx** exception catching services utilized by **pioassist**. In many cases where a small amount of I/O is performed at one time, this method will result in a much faster execution time due to the low exception catching setup mechanism utilized by each macro invocation. In other cases where there are large number of programmed I/O macros used in a block of code, the pioassist mechanism used in conjunction with the macros without built-in error catching may result in better performance due to the utilization of a single exception handler for all programmed I/O instructions contained within the block of code.

2.5.7.1 Macros performing programmed I/O writes

The first set of programmed I/O macros write the specified data to bus memory or bus I/O address space. The BUS_PUTC (put character) macro may also be used to write POS registers.

BUS_PUTL(long *ioaddr, long data)

Write the specified long value (data) to the supplied bus memory or bus I/O address (ioaddr) without error handling.

int BUS_PUTLX(long *ioaddr, long data)

Write the specified long value (data) to the supplied bus memory or bus I/O address (ioaddr) with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

BUS_PUTS(short *ioaddr, short data)

Write the specified short value (data) to the supplied bus memory or bus I/O address (ioaddr) without error handling.

int BUS_PUTSX(short *ioaddr, ushort data)

Write the specified short value (data) to the supplied bus memory or bus I/O address (ioaddr) with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

BUS_PUTC(char *ioaddr, char data)

Write the specified character value (data) to the supplied bus memory, bus I/O or POS address (ioaddr) without error handling.

int BUS_PUTCX(char *ioaddr, char data)

Write the specified character value (data) to the supplied bus memory, bus I/O or POS address (ioaddr) with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

int BUS_PUTSTRX(char *ioaddr, char *saddr, int count)

Copy count bytes from memory specified by saddr to bus memory or bus I/O address starting at the address specified by ioaddr with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

2.5.7.2 Macros performing byte reversed I/O writes

int BUS_PUTLRX(long *ioaddr, long data)

Write the specified long value (data) to the supplied bus memory or bus I/O address (ioaddr) in byte reversed format with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer. Note: The IOCC (I/O controller) on the RISC System / 6000 automatically converts 32 bit transters from big endian format on the system into little endian format as seen by the device, therefore this macro will undo this conversion. This macro should be used when the device or the data on the

device is stored in BIG ENDIAN format instead of the ususal LITTLE ENDIAN format found on most microchannel adapters.

int BUS_PUTSRX(short *ioaddr, short data)

Write the specified short value (data) to the supplied bus memory or bus I/O address (ioaddr) in byte reversed format with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer. Note: The IOCC (I/O controller) on the RISC System / 6000 automatically converts 16 bit transters from big endian format on the system into little endian format as seen by the device, therefore this macro will undo this conversion. This macro should be used when the device or the data on the device is stored in BIG ENDIAN format instead of the ususal LITTLE ENDIAN format found on most microchannel adapters.

2.5.7.3 Macros performing programmed I/O reads

The following macros read the specified data from bus memory or bus I/O address space. The BUS_GETC (get character) macro may also be used to read POS registers.

long BUS_GETL(long *ioaddr)

Read the specified long value from the supplied bus memory or I/O address (ioaddr). This macro is an expression.

int BUS_GETLX(long *ioaddr, long *data)

Reads the specified long value into the variable data from the supplied bus memory or I/O address (ioaddr) with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

short BUS_GETS(short *ioaddr)

Reads the specified short value from the supplied bus memory or I/O address (ioaddr) without error handling. This macro is an expression.

int BUS_GETSX(short *ioaddr, short *data)

Reads the specified short value (data) from the supplied bus memory or I/O address (ioaddr) with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

char BUS_GETC(char *ioaddr)

Reads the specified character value (data) from the supplied bus memory, bus I/O or POS address (ioaddr) without error handling. This macro is an expression.

int BUS_GETCX(char *ioaddr, char *data)

Reads the specified character value (data) from the supplied bus memory, bus I/O or POS address (ioaddr) with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

int BUS_GETSTRX(char *ioaddr, char *daddr, int count)

Copy count bytes from bus memory or bus I/O address starting at the address specified by ioaddr to memory starting at daddr with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer.

2.5.7.4 Macros performing byte reversed I/O reads

int BUS_GETLRX(long *ioaddr, long *data)

Read the specified long value (data) from the supplied bus memory or bus I/O address (ioaddr) in byte reversed format with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer. Note: The IOCC (I/O controller) on the RISC System / 6000 automatically converts 32 bit transters from little endian format on the device into big endian format, therefore this macro will undo this conversion. This macro should be used when the device or the data on the device is stored in BIG ENDIAN format instead of the ususal LITTLE ENDIAN format found on most microchannel adapters.

int BUS_GETSRX(short *ioaddr, short data)

Write the specified short value (data) to the supplied bus memory or bus I/O address (ioaddr) in byte reversed format with built-in exception catching. The return value will be 0 for success, otherwise an error (exception) occurred during the transfer. Note: The IOCC (I/O controller) on the RISC System / 6000 automatically converts 16 bit transters from little endian format on the device into big endian format, therefore this macro will undo this conversion. This macro should be used when the device or the data on the device is stored in BIG ENDIAN format instead of the ususal LITTLE ENDIAN format found on most microchannel adapters.

2.5.7.5 PIO Error Recovery Considerations

In general, all I/O operations must include provisions to handle detectible errors. Except for Programmable Option Select (POS) accesses, it is possible for all PIO operations to have a variety of synchronous errors. These generally require the device driver to support synchronous I/O error exception handling.

When utilizing the programmed I/O macros with built in exception catching, a non-zero return code indicates that an exception or error occurred during the operation. If the operation was a read, the contents of the destination data area are invalid. If the return code is non-zero it will have one of the exception values defined in <sys/except.h>. If the value is EXCEPT_IO then the exception is an error caused by programmed I/O and should be handled. If the return code is non-zero and not EXCEPT_IO some other error (usually a programming error) has occurred and an assert should be coded to stop the system and provide a dump. If the operation is to be retried, it should be retried up to PIO_RETRY_COUNT times (see <sys/except.h>). If the operation still fails, it should be considered a permanent error and handled accordingly, usually by returning EIO to the caller of the device driver.

POS operations, unlike other PIO operations, cannot detect synchronous I/O errors. It is suggested that device driver programmers implement an algorithm which reads back the data after a read or write and compares it to see if a simple data error occurred. If the data miscompares, the POS operation should

be retried, and the data read back again to see if the miscompare recurred. This should be repeated up to PIO_RETRY_COUNT (see <sys/except.h>) times. If the operation still fails, it should be considered a permanent error and handled accordingly.

2.5.7.6 Programmed I/O Examples w/o error catching

Examples of typical PIO operations follow. To simplify the examples, error catching has been omitted, however must never be omitted in operational code. When using these macros, **pioassist** or **setjmpx** kernel services should be used to catch and handle I/O errors. For more information on error handling of this type refer to Device Handler Error Recovery. I/O errors that are not handled by device drivers will cause a kernel exception to occur, resulting in a system crash. While error handling is required, error recovery is optional but strongly advised. Error recovery involves retrying the failing operations one or more times before shutting down the device.

Perform a 32-bit read of system bus I/O space at address 0x3F0.

```
caddr_t addr;
ulong   io_addr, buid, bid, data_read_from_device;
 /* note that buid and base io_address is typically known to the
device driver via its configuration parameters */
buid = 0x82000000; /* hard coded only for example */
io_addr = 0x3F0; /* hard coded only for example */
bid = buid | 0x000C0020; /* enable bus access modes */
addr = BUSIO_ATT( bid, io_addr );
data_read_from_device = BUS_GETL( addr );
BUSIO_DET( addr );
```

Perform a 16-bit write of 0xE010 to system bus memory space at address 0xE0082.

```
caddr_t addr;
ulong   mem_addr, buid, bid;
ushort  data_to_write_to_device;
 /* note that buid and base bus memory address is typically known
to the device driver via its configuration parameters */
buid = 0x82000000; /* hard coded only for example */
mem_addr = 0xE0082; /* hard coded only for example */
bid = buid | 0x000C0020; /* enable bus access modes */
data_to_write_to_device = 0xE010;
addr = BUSMEM_ATT( bus_id, mem_addr );
BUS_PUTS( addr, data_to_write_to_device );
BUSMEM_DET( addr );
```

Perform a read of both POS address 0 and 1 registers for the Micro Channel Adapter in slot 3. Note: refer to the RISC System/6000 Hardware Technical Reference for information regarding generating POS effective addresses.

```
caddr_t bus_addr;
ulong   bus_id, slot;
uchar   value0, value1;
/* note that bus_id is typically known to the
   device driver via its configuration parameters */
slot = 3;
bus_addr = IOCC_ATT( bus_id, 0 );
value0 = BUS_GETC( bus_addr + ((slot - 1) << 16) + 0x400000 );
value1 = BUS_GETC( bus_addr + ((slot - 1) << 16) + 0x400001 );
IOCC_DET( bus_addr );
```

Perform a 32 byte string write to bus memory address 0x3000 from system memory.

```
caddr_t addr;
sys_string charffl32";
ulong   mem_addr, buid, bid;
int rc, length;
/* note that buid and base bus memory address is typically known
to the device driver via its configuration parameters */
buid = 0x82000000; /* hard coded only for example */
mem_addr = 0x3000; /* hard coded only for example */
bid = buid | 0x000C0020; /* enable bus access modes */
length = 32;
addr = BUSMEM_ATT( bus_id, mem_addr );
rc = BUS_PUTSTRX( addr, sys_string, length );
BUSMEM_DET( addr );
```

2.5.7.7 Programmed I/O Example with error catching

An example of a typical PIO operation follows. This example uses an error catching macro to catch any errors and calls a common error handling routine in the device driver to retry the operation, perform the error logging and return with either a permanent error status, or a successfully recovered status if an error occurs. Note that error handling is performed by a separate routine instead of using inline code so that instruction caching is optimized for the normal path.

Perform a 32-bit read of system bus I/O space at address 0x3F0.

```
    enum pio_func { GETC, GETS, GETSR, GETL, GETLR,
                        PUTC, PUTS, PUTSR, PUTL, PUTLR
    };
    struct dev_regs {
      long status_reg;
      short cmd_reg;
       .
       .
       .
      };
    struct dev_regs *devaddr;
    long   io_addr, bid, buid, status;
    buid = 0x82000000; /* hard coded only for example */
    io_addr = 0x3F0; /* hard coded only for example */
    bid = buid | 0x000C0020;
    /* note that buid and io_addr are typically known to the
       device driver via its configuration parameters (dds) */
    devaddr = (struct dev_regs *) BUSIO_ATT( bid, io_addr );
    if (rc = BUS_GETLX( &devaddr->status_reg, &status ))
      rc=pio_recov(GETL, &dds, rc, &devaddr->status_reg , (long) &status);
    BUSIO_DET( devaddr );
    return (rc);
}

/* pio_recov - general pio error handling and recovery routine example */
int pio_recov ( enum pio_func iofunc, struct dds *ddsp, int exception,
                void *ioaddr, void *ioparam )
{
   int retry_count= PIO_RETRY_COUNT;
    while (TRUE)
     {
```

```
        assert(exception == EXCEPT_IO);
         if (retry_count <= 0)
           /* log permanent error here - out of retries */
        /* dds pointer is used for error logging to determine which device the
           error is reported against */
           return(EIO);
        else
        {
           /* log temporary error here  */
           retry_count--;
        }
         /* retry the PIO function and return if successful */
        switch (iofunc)
        {
         case GETL:
           exception = BUS_GETLX ((long *)ioaddr, (long *)ioparam);
            break;
         case GETS:
            exception = BUS_GETSX ((short *)ioaddr, (short *)ioparam);
             break;
          .
          . /* case entries for all enumerated I/O functions */
          .
        } /* end of switch */
         if (exception == 0)
          return (0);
         } /* end of while TRUE*/
}
```

Chapter 3. Interface to Device Drivers

3.1 Aix Version 3.1 Structure

3.1.1 AIX and the Interrupt and Process Environments

The AIX 3.1 kernel is a collection of processes, interrupt handlers, device drivers, and system calls that provide an environment in which application programs can execute. There are two types of execution states, two types of processes and two types of execution environments. There are also other characteristics, such as whether or not the process is pageable or preemptible. Figure 3-1 on page 3-2 shows a overview of processes and their characteristics for AIX in Version 3.

© Copyright IBM Corp. 1991

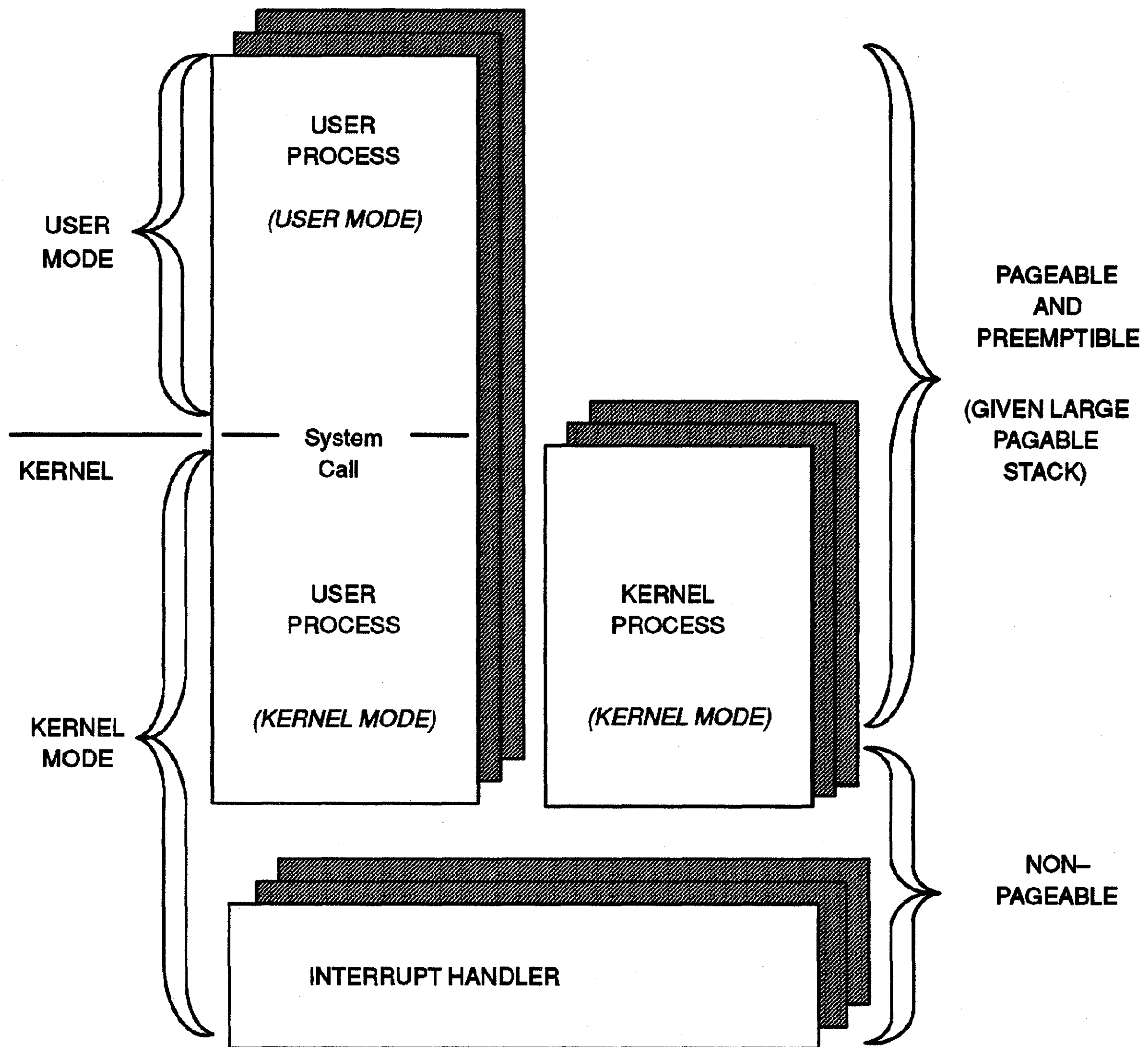


Figure 3-1. AIX Interrupt Handlers and Processes

At the hardware level, the processor can be in one of two execution states: privileged or unprivileged. The execution state determines what instructions the processor allows.

Programs execute in one of two modes, similar to the processor execution states; they are user and kernel mode. A program running in user mode can only affect its own execution environment and runs in the processor unprivileged state. Programs running in kernel mode can affect the execution environments of all programs because they can call kernel services and run in privileged state. The execution mode changes when a system call is made.

In order to provide facilities for a real-time environment and to improve system efficiency, there are two types of execution environments. First, there is the process environment. This environment is independent of external events and must share the processor in a way that is fair to all. Processes that run in this environment are time sliced and can be preempted by a higher priority process. This includes both user and kernel processes. In the process environment, a process is pageable and preemptible. That means it can be interrupted by another process. Secondly, there is the interrupt handler environment. In this environment, a process must respond to external events quickly. It is not time sliced, and runs to completion. It can be preempted by a higher priority interrupt handler. It cannot be paged out.

A user process is usually connected with an end user and is usually running some application or command. It usually runs in user execution mode, where it can only affect its own environment. If it has the proper authority, it may execute a system call, which temporarily changes to kernel or privileged mode, to access some restricted system resource. A user process is a process created primarily to execute a user program and usually doesn't need privileged operations. A user process is the environment that a user program executes in. Most processes are user processes. When the process requires a function performed by the system, it uses the system subroutines, called system calls or kernel calls. When the user process issues a system call, the execution environment switches from user to kernel mode.

A kernel process is primarily created to execute a kernel program. It is always running in kernel mode and may affect the environments of other processes. A kernel process in one that executes in kernel mode all the times.

3.1.2 The AIX Interrupt Handler Environment

An interrupt handler executes in a restrictive type of program environment. They must not page fault or wait and can only use a restricted set of kernel services. All data access is typically in global memory to avoid paging. The path length must be short - 100 instructions is often the guideline for those with high priority, 1000 cycles for those with relatively low priority. By having these restrictions, the context switch time (the time required to dispatch a process) is small.

An interrupt handler may change its interrupt priority by enabling/disabling interrupts, masking interrupts, or scheduling to run code at an off-level priority. It can only be preempted by a higher priority interrupt handler. There are services (**i_disable** and **i_enable**) to enable and disable interrupts. The **i_mask** and **i_umask** will mask off interrupts below a specified priority.

For an interrupt handler to schedule additional interrupt handler code to be run at a lower interrupt priority use the **i_sched** kernel service. An example of off-level scheduling would be an interrupt handler for asynchronous terminals. They can receive keystroke data from the terminal at a very high priority to ensure that overrun of the device does not occur. Once the keystroke data is buffered in the operating system, the rest of the character processing can be done at a lower level.

Interrupt handlers are not time sliced, so they run to completion unless interrupted by a higher priority interrupt. They cannot wait on events or dispatch processes. The path length has to be short because they are not time

sliced. They can change their interrupt priorities and can disable other interrupts. The stack is always small and pinned to increase the efficiency.

In AIX Version 3, there are 12 interrupt priorities defined for the RISC/6000. See /usr/include/sys/m_intr.h for their definition.

Please see "Priority Assignment" on page 2-12 for a more detailed discussion of interrupt handler priorities.

Interrupt Handler Characteristics

Interrupt handlers have the following characteristics:

- Preemptible by higher priority interrupts only
- Cannot wait or dispatch processes
- Must have short path length
- May change interrupt priority
- Interrupt handlers are pinned in memory
- Must only access pinned data (page faults are not allowed).

3.1.3 The AIX Process Environment

This section describes the process type of environment. There are 128 process priorities in Version 3. Process priorities are always of lower priority than interrupt handler priorities. The priority may be changed with either the **setprio** system call or the **nice** command.

The scheduler in AIX Version 3 is similar to the one in AIX 2.2.1, (which was AT&T based) except for one primary item: most processes are now preemptible, including kernel processes. Also, a process can make itself exempt from being preempted, such as a real-time process. The scheduler is exempt along with some other critical pieces of the kernel.

User process priorities are assigned by the standard Unix algorithm based on the ratio of the amount of compute time to real-time recently used by the process. At every tick of the system timer, the p_cpu field (processor usage) in the process table for the running process is incremented. The compute time to real-time ratio is updated every second. Using a negative exponential distribution, the kernel decreases p_cpu by half its value for every process at or above the base user level and recalculates the priority of the processes. Processes that accumulated a lot of execution time are less favored than processes with very litte execution time. A user process can execute the **nice** system call to induce a bias in the calculation. A **setprio** system call can also be executed to bypass the calculation.

The priority of a process is determined by the following formula:

min ((real-time-flag ? 0 : cpu-usage) + nice-value, wake-up-priority, lock-priority, 127)

The real-time-flag means that this process executes at a fixed priority and is exempt from having its priority recomputed due to CPU usage.

The wake-up-priority is the priority at which the process should be dispatched once its sleep ends.

The lock-priority is set for a process which holds a lock by those requesting the lock to ensure that the lock holder executes at a higher priority equal to that of the most favored lock requester until it relinquishes the lock.

This means the priority is the lowest of the following numbers: the cpu-usage (or 0 if it's a real-time program), the nice-value, the wake-up priority, lock priority, or 127.

Process Characteristics

Processes have the following characteristics:

- Preemptible by higher priority process or ANY interrupt
- Can wait and/or dispatch processes
- Has large pagable stack
- Page faults are allowed
- Has one of 128 process priorities (0 is highest, 127 is lowest)
- Process priority may be fixed (exempt from scheduling) or normal.

3.1.4 Preemption in the AIX Operating System

The AIX kernel is designed to allow preemption by other processes while executing in kernel mode. This change to allow preemption was made in order to enhance support for real-time processes and large multiuser systems.

Most existing Unix device drivers do not expect to be preempted. The effects of preemption on existing Unix device drivers are from redispatching another process before the preempted process is allowed to finish a request. Instead, a higher priority process may start running. This higher priority process may update the same data areas as the preempted process which causes data inconsistencies.

Kernel mode processes that update global kernel data are not considered reentrant. Reentrant means the routine does not modify itself or its data. Routines that are reentrant are not a problem. However, if you are modifying global kernel data structures or private structures that are shared between processes then some form of serialization is required. A serially reusable routine must provide serialization using one of two methods available in the kernel. One way is to priortize processes accessing the data. The highest priority process is guaranteed to finish its work before a lower level process.

This is difficult to achive on systems that provide paging unless the system does not dispatch another process until the page for the waiting process comes in allowing the higher priority process to complete. In a paging system it is better to be able to dispatch another process during the page in process. In AIX, locks are used for this reason. Serialization is accomplished by using the lockl kernel service with the **kernel_lock** parameter. Therefore a process with the **kernel_lock** will not be preempted by another kernel mode process (unless the original process waits on I/O). Locks provide a serialization mechanism for

processes executing in kernel mode. A lock is not an enforceable policy. A lock is a request to the system to serialize your use of a resource with other processes. If all processes do not adhere to this policy, data inconsistencies will occur.

Programming Hint

The lock word is nothing but a unique token. All processes which share a data structure typically use the address of the data structure as their unique token when calling **lockl**.

The process currently owning the lock may update the information agreed among the other processes. For example, process A wishes to insert a new structure into a doubly linked list. Process A obtains the lock head_pointer which is the address of the start of the list and proceeds to insert element x onto the list. During this time most Unix systems would not allow another process in the kernel to be dispatched, because the pointers may not be completely updated. With AIX, however, page faulting or handling of external interrupts may halt the execution of process A and allow another more favored process to execute. If the new process also alters the head_pointer list it may find the data structure inconsistent and result in a data exception in the kernel. Locks avoid this scenario and allow kernel mode processes to page fault and support real-time.

Kernel extensions release a lock using unlockl. If you call unlockl your extension believes it owns a lock. If you do not own the lock and call unlockl chances are your process was not serialized during some critical update. The kernel will then crash the system.

Please see "Process Management Kernel Services" on page 3-19 for a discussion on the **lockl/unlockl** system calls.

3.2 Kernel Interface

A device driver at the user application level supports the same user interface as the file system, namely; open, close, read, and write routines. A device driver implements this concept through the use of kernel services.

In its simplest form, a device driver moves data between hardware devices and user applications where the user applications supply and consume information. It may also be involved in the translation of information. Presenting information to the application that is translated from the tty services is an example of this.

In addition to supplying a user with information, a computer employs numerous types of devices for storing and collecting data. Device drivers provide a transparent method for managing information storage and retrieval.

For example, a device driver moves keystrokes to a program and moves characters to the screen for display. In this simple view a device driver is trivial. However, because there is typically more time spent waiting for devices to input and output information on Unix machines, the design is to allow multiple users to input/output data. This complicates the system dramatically.

The kernel services provide services for moving data, serializing the use of data, data integrity and notification of data delivery. With these mechanisms a device driver can satisfy the need for information among many users while optimizing the resources of the machine.

3.2.1 The Device Switch Table

The device switch table binds the device driver to the system and extends the kernel. The kernel however needs to advertise that the service is available. Advertisement is accomplished through the file system and the **mknod** command. The special file provides a name which is translated to a device number. The device number is then used to index the switch table.

The order of operations in your device driver configuration routine should make sure that the mknod is issued as the last command, allowing the driver to be completely set up before any user is able to access the device driver. See "Adapter Configuration Method (cfgrica.c)" on page B-28 for sample configuration code.

The AIX device switch table provides services for collecting and storing device driver location information. In addition, kernel services are provided for querying the entries.

devswqry query to see if a device and associated routines and configuration information are available.

devswadd adds information for a device driver to the device switch table.

devswdel deletes information for a device driver from the device switch table.

The kernel services listed above are the only method for accessing the device switch table. This is different than other Unix systems that provide direct access to the switch table. Direct access to the switch table on AIX would be potentially a problem because it is a global data structure. Any number of users could be adding a device driver and so the problems of serialization exist.

There are other global kernel data structures that are also implemented using a service to alter and read information instead of providing a global variable. That is why some kernel services exist - to serialize the access and changing of global data structures.

The following entry points exist in the device switch table. These entry points (routines) will be discussed in subsequent sections:

- open routine entry point
- close routine entry point
- read routine entry point
- write routine entry point
- ioctl routine entry point
- strategy routine entry point
- select routine entry point
- config routine entry point

- print routine entry point
- dump routine entry point
- mpx routine entry point
- revoke routine entry point.

Note that not all routines are required. If a routine is not required, the entry in the device switch table can call one of two special kernel routines **nodev** or **nulldev**. An entry point can be assigned a null value. This will result in a call to a dummy kernel routine called **nulldev** which will always return a successful return code. An entry point can also be assigned a value of **nodev**. This will result in a return code value of ENODEV.

3.2.2 Entry Points Common to Character and Block Device Drivers

Please refer to "Overview of a Character Device Driver" on page 4-1 for detailed information on character device driver entry points and to "Overview of a Block Device Driver" on page 5-1 for detailed information on block device driver entry points.

The names of entry points (like ddconfig, ddopen, ddread, etc.) do not mean anything in terms of symbols exported by the kernel. They are place holder names purely for description. Your driver is ONLY bound to the kernel through the switch table. The device switch table expects an entry in each field whether or not your driver supports such a routine. See nodev or nulldev if you do not have a routine. Your driver may export symbols but please see "Pinning Device Driver Code" on page 10-1 for a description of possible problems that can be caused by referencing code that is not pinned.

The following entry points are common for both character and block device drivers:

- **ddconfig**

  The ddconfig routine is an AIX innovation and is not part of standard Unix. This routine supports run time and IPL time configuration of the device driver. Typically, it is the first routine in your driver but may be placed elsewhere. ddconfig is responsible for reading the configuration information requested by the user/administrator, setting the driver up according to this information and binding the routines in the driver to the switch table so that the user level configuration routine (configuration method) can complete. This makes the user interface available (open, close, read, write) for use.

- **ddopen**

  The ddopen routine binds a user to the device driver. Some drivers wait until open to finish configuring additional device parameters. Also, some drivers support multiple users per device (for example, the token-ring and Ethernet device drivers do this). Others handle multiplexing through user-written routines such as the printer backend services (piobe). The open routine determines whether one or more users may be allowed to access the device concurrently.

- **ddclose**

  The ddclose routine is responsible for deallocating resources associated with a particular user.

- **ddioctl**

  The ddioctl routine provides control commands and parameters to the device.

- **dddump**

  The dddump routine allows the device driver to use the device as the target of a system dump. Usually hard disks or tape devices will have this capability.

3.2.3 Entry Points for Character and Raw Access to Block Device Driver

- **ddread**

  Allows data to be read from a device. This is sequential data specified as one or more bytes.

- **ddwrite**

  Allows sequential data to be written to a device.

3.2.4 Entry Points Unique to Character Device Drivers

- **ddselect**

  Allows a user to poll a hardware device to determine whether specific events should have occurred.

- **ddmpx**

  Allows for multiple users to share a resource on a hardware adapter. For example, this could be a port on a communications adapter.

3.2.5 Entry Points Unique to Block Device Drivers

- **ddstrategy**

  Allows block-oriented reads or writes to be performed on a block-oriented device (like a hard disk).

3.2.6 Entry Points for Trusted Computing Path Device Drivers

- **ddrevoke**

  Allows a character device driver to disable a device for all users. This results in a particular user having exclusive access to a device. This entry point is only required for devices that are supported in the trusted computing path.

3.2.7 Miscellaneous Entry Points NOT Found in the Device Switch Table

The following routines do not have entries in the device switch table. They are registered using kernel services.

The start I/O routine is typically known only to other routines within the device driver, such as the strategy and interrupt-handling routines.

Interrupt handling routines are also registered using kernel services. Note that some character device drivers, particularly pseudo-device drivers, may not have a bottom half if they have no need to execute in the interrupt environment.

The entry point for the component dump routine is registered with the kernel at initialization time. This is registered by using the **dmp_add** kernel service. The purpose of the component dump routine is to allow your device driver to save tables and information if something causes a system dump to happen. See "Including Device Driver Information in a System Dump" on page 9-2 for more information.

HINT

Writing and registering a component dump routine for your device driver can be very useful for tracing the cause of abnormal system dumps related to your driver.

3.3 Kernel Services

This section contains a non-exhaustive list of AIX V3 kernel services. The complete list can be found in in Chapter 6 of *Kernel Extensions and Device Support Programming Concepts*.

3.3.1 I/O Services

3.3.1.1 Programmed I/O Macros

See "I/O Macros" on page 2-14

3.3.1.2 Interrupt Management

The eight Interrupt Management services are:

i_clear Removes an interrupt handler from the system.

i_disable Disables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a less-favored interrupt priority.

i_enable Enables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a more-favored interrupt priority.

i_init Defines an interrupt handler to the system, connects it to an interrupt level, and assigns an interrupt priority to the level.

i_mask Disables a bus interrupt level.

i_reset Resets a bus interrupt level.

i_sched Schedules off-level processing.

i_unmask Enables a bus interrupt level.

Additional considerations for interrupt handlers:

- Code and data referenced should be pinned in memory.
- Code should be prepared to handle preemption by equal or higher priority interrupts.
- Code should execute on a small stack.

3.3.1.3 DMA Management

The AIX operating system kernel provides 10 services for managing DMA channels and performing DMA operations:

d_init Initializes a DMA channel.
Flags to be used:

- For bus master: MICRO_CHANNEL_DMA
- For bus slave: MICRO_CHANNEL_DMA and DMA_SLAVE

d_clear Frees a DMA channel.

d_master Initializes a block-mode DMA transfer for a DMA master.
Flags to be used:

- For system memory to device: DMA_WRITE_ONLY
- For device to system memory: DMA_READ
- For bus memory to device: BUS_DMA
- For device to bus memory: DMA_READ and BUS_DMA

d_slave Initializes a block-mode DMA transfer for a DMA slave.
The flags to be used are the same as for DMA master transfer.

d_complete Cleans up after a DMA transfer.

d_mask Disables a DMA channel.

d_unmask Enables a DMA channel.

d_move Provides consistent access to system memory that is accessed asynchronously by a device and by the processor on a RISC System/6000.

d_align Assists in allocation of DMA buffers.

d_roundup Assists in allocation of DMA buffers.

A device driver must call the **d_slave** service to set up a DMA slave transfer or call the **d_master** service to set up a DMA master transfer. The device driver then sets up the device to perform the DMA transfer. The device transfers data when it is available and interrupts the processor upon completion of the DMA transfer. The device driver then calls the **d_complete** service to clean up after the DMA transfer. This process is typically repeated each time a DMA transfer is to occur.

The **d_align** service returns the alignment value required for starting a buffer on a processor cache line boundary. The **d_roundup** service can be used to round the desired DMA buffer length up to a value that is an integer number of cache lines. These two services allow buffers to be used for DMA to be aligned on a cache line boundary and allocated in whole multiples of the cache line size so that the buffer is not split across processor cache lines. This reduces the possibility of consistency problems because of DMA and also minimizes the number of cache lines that must be flushed or invalidated when used for DMA. For example, these services can be used to provide alignment as follows:

```
align = d_align();
buffer_length = d_roundup(required_length);
buf_ptr = xmalloc(buffer_length, align, pinned_heap);
```

Data must be carefully accessed when a DMA operation is in progress. The **d_move** service provides a means of accessing the data while DMA transfer is being performed on it. This service uses the same I/O controller data buffers that the DMA master does when accessing data from the shared data area in system memory. Using the same buffer keeps the processor data accesses and device data access consistent. On the RISC System/6000 platform, this is necessary since the I/O controller provides buffer caching of data accessed by bus master devices.

3.3.1.4 Block I/O

The three Block I/O kernel services are:

iodone Performs block I/O completion processing.

iowait Waits for block I/O completion.

uphysio Performs character I/O for a block device using a **uio** structure.

3.3.1.5 Buffer Cache

The 14 Buffer Cache kernel services are:

bawrite Writes the specified buffer's data without waiting for I/O to complete.

bdwrite Releases the specified buffer after marking it for delayed write.

bflush Flushes all write-behind blocks on the specified device from the buffer cache.

binval Invalidates all of the specified device's blocks in the buffer cache.

blkflush Flushes the specified block if it is in the buffer cache.

bread Reads the specified block's data into a buffer.

breada Reads in the specified block and then starts I/O on the read-ahead block.

brelse Frees the specified buffer.

bwrite Writes the specified buffer's data.

clrbuf Sets the memory for the specified buffer structure's buffer to all zeros.

getblk Assigns a buffer to the specified block.

geteblk Allocates a free buffer.

geterror Determines the completion status of the buffer.

purblk Purges the specified block from the buffer cache.

3.3.1.6 Character I/O

The 13 Character I/O kernel services are:

getc Retrieves a character from a character list.

getcb Removes the first buffer from a character list and returns the address of the removed buffer.

getcbp Retrieves multiple characters from a character buffer and places them at a designated address.

getcf Retrieves a free character buffer.

getcx Returns the character at the end of a designated list.

pincf Manages the list of free character buffers.

putc Places a character at the end of a character list.

putcb Places a character buffer at the end of a character list.

putcbp Places several characters at the end of a character list.

putcf Frees a specified buffer.

putcfl Frees the specified list of buffers.

putcx Places a character on a character list.

waitcfree Checks the availability of a free character buffer.

3.3.2 Memory Services

The Memory kernel services provide kernel extensions with the ability to:

- Dynamically allocate and free memory
- Pin and unpin code and data
- Access user memory and transfer data between user and kernel memory
- Create, reference, and change virtual memory objects.

3.3.2.1 Memory Management Kernel Services

The three Memory Management services are:

init_heap Initializes a new heap to be used with kernel Memory Management services.

xmalloc Allocates memory. Two heaps are provided in the kernel segment for use by kernel extensions: **kernel_heap** which is not pinned, and **pinned_heap** which is pinned.

xmfree Frees allocated memory.

3.3.2.2 Memory Pinning

The six Memory Pinning services are:

pin Pins the address range in the system (kernel) space.

pincode Pins the code and data associated with an object file.

pinu Pins the specified address range in user or system memory.

unpin Unpins the address range in system (kernel) address space.

unpincode Unpins the code and data associated with an object file.

unpinu Unpins the specified address range in user or system memory.

3.3.2.3 User Memory Access

In a system call or kernel extension running under a user process, data in the user process can be moved in or out of the kernel using the **copyin** or **copyout** services. The **uimove** service is used for scatter/gather operations. If user data is to be referenced asynchronously, such as from an interrupt handler or a kernel process, the cross memory services must be used.

The 10 user Memory Access services are:

copyin Copies data between user and kernel memory.

copyinstr Copies a character string (including the terminating null character) from user to kernel space.

copyout Copies data between user and kernel memory.

fubyte Fetches, or retrieves, a byte of data from user memory.

fuword Fetches, or retrieves, a word of data from user memory.

subyte Stores a byte of data in user memory.

suword Stores a word of data in user memory.

uiomove Moves a block of data between kernel space and a space defined by a uio structure.

ureadc Writes a character to a buffer described by a uio structure.

uwritec Retrieves a character from a buffer described by a uio structure.

3.3.2.4 Cross Memory Kernel Services

The Cross Memory services allow data to be moved between the kernel and an address space other than the current process address space. A data area within one region of an address space is attached by calling the **xmattach** service. As a result, the virtual memory object cannot be deleted while data is being moved in or out of pages belonging to it. A cross memory descriptor is filled out by the xmattach service. The attach operation must be done while under a process. When the data movement is completed, the **xmdetach** service can be called. The detach operation can be done from an interrupt handler.

The **xmemin** service can be used to transfer data from an address space to kernel space. The **xmemout** service can be used to transfer data from kernel space to an address space. These routines may be called from interrupt handler level routines if the referenced buffers are in memory.

Cross memory services provide the **xmemdma** service to prepare a page for DMA processing. The xmemdma service flushes any data from cache into memory and hides the page. A page is hidden by invalidating processor access to the page. Any processor references to the page result in page faults with the referencing process waiting on the page to be unhidden. The xmemdma service returns the real address of the page for use in preparing DMA address lists. When the DMA transfer is completed, the xmemdma service must be called again to unhide the page.

Data movement by DMA or an interrupt handler requires that the pages remain in memory. This is ensured by pinning the data areas using the **pinu** service. This can only be done under a process, since the memory pinning services page fault on pages not present in memory.

The **unpinu** service unpins pinned pages. This can be done by an interrupt handler if the data area is the global kernel address space. It must be done under the process if the data area is in user process space.

The five Cross-Memory services are:

xmattach Attaches to a user buffer for cross-memory operations.

xmdetach Detaches from a user buffer used for cross-memory operations.

xmemin Performs a cross-memory move by copying data from the specified address space to kernel global memory.

xmemout Performs a cross-memory move by copying data from kernel global memory to a specified address space.

xmemdma Prepares a page for DMA I/O or processes a page after DMA I/O is complete.

3.3.3 Other Services

3.3.3.1 Device Driver Management

The AIX kernel provides a relatively complete set of program and device driver management services. These services include general kernel extension loading and binding services and device driver binding services. Also provided are services that allow kernel extensions to be notified of base kernel configuration changes, user mode exceptions, and system wide process state changes.

The **kmod_load**, **kmod_entrypt**, **kmod_unload** services provide kernel extension loading and binding services. The **sysconfig subroutine** makes these services available to user mode programs. However, kernel-mode callers executing in a kernel process environment can also use them. These services provide the same kernel object-file load, unload, and query functions provided by the sysconfig subroutine as well as the capability to obtain a module's entry point with the kernel module ID assigned to the module.

The kmod_load, kmod_entrypt, kmod_unload services can be used to dynamically alter the set of routines loaded into the kernel based on system configuration and application demand. Subsystems and device drivers can use these services to load large, seldom-used routines on demand. Device driver binding services include the **devswadd**, **devswdel**, **devswqry** services, which are used to add or remove a device driver entry from the dynamically managed device switch table. They also query for information concerning a specific entry in the device switch table.

Some kernel extensions may be sensitive to the settings of base kernel run time configurable parameters that are found in the var structure defined in the *sys/var.h* header file. These parameters can be set during system boot or run time by a privileged user performing system configuration commands that use the sysconfig subroutine to alter values in the var structure. Kernel extensions may register or remove a configuration notification routine with the **cfgnadd** and **cfgndel** kernel services. This routine is called each time the sysconfig subroutine is used to change base kernel tunable parameters found in the var structure. In addition, the **prochadd** and **prochdel** kernel services allow kernel extensions to be notified when any process in the system has a state transition, such as being created, exiting, being swapped in or swapped out.

The **uexadd** and **uexdel** kernel services give kernel extensions the capability to intercept user mode exceptions.[1] The default action when an exception occurs is, in *user mode*, to send a signal, and in *kernel mode* to halt the system. These

---

[1] An exception is a synchronous event directly associated with the instruction that is executing when the exception occurs.

user mode exception handlers may use this capability to dynamically reassign access to single-use resources or to clean up after some particular user mode error. Note that on user mode exceptions, all registered handlers are invoked until one claims the exception. Therefore, handlers must return either **EXCEPT_HANDLED** or **EXCEPT_NON_HANDLED**. The associated **uexblock** and **uexclear** services can be used by these handlers to block and resume process execution when handling these exceptions. Section "Process Management Kernel Services" on page 3-19 talks about dealing with kernel mode exceptions.

The **pioassist** and **getexcept** kernel services are typically used by device drivers to obtain detailed information about exceptions that occur during I/O bus access. The getexcept service can also be used by any exception handler requiring more information about an exception that has occurred. The **selnotify** kernel service replaces the traditional Unix *selwakeup* kernel function and is used by device drivers supporting the poll or select functions when asynchronous event notification is requested. The **iostadd** and **iostdel** services are used by tty and disk device drivers to register device activity reporting structures to be used by the *iostat* and *vmstat* commands.

Finally, the **getuerror** and **setuerror** services can be used by kernel extensions that provide or use system calls to access the *u.u_error* field for the current process. This is typically used by kernel extensions providing system calls to return error codes, and is used by other kernel extensions to check error codes upon return from a system call (since there is no errno global variable in the kernel).

NOTE

Return values from kernel entry points get put into u.u_error. Therefore, when you return from your device driver routines (ddopen, ddclose,ddread, etc.) you should always specify a return value.

The 23 Kernel Program/Device Driver Management kernel services are:

cfgnadd Registers a notification routine to be called when system-configurable variables are changed.

cfgndel Removes a notification routine for receiving broadcasts of changes to system configurable variables.

devdump Calls a device driver dump-to-device routine.

devstrat Calls a block device driver's strategy routine.

devswadd Adds a device entry to the device switch table.

devswdel Deletes a device driver entry from the device switch table.

devswqry Checks the status of a device switch entry in the device switch table.

getexcept Allows kernel exception handlers to retrieve additional exception information.

getuerror Allows kernel extensions to retrieve the current value of the u_error field.

iostadd Registers an I/O statistics structure used for updating I/O statistics reported by the iostat subroutine.

iostdel Removes the registration of an I/O statistics structure used for maintaining I/O statistics on a particular device.

kmod_entrypt Returns a function pointer to a kernel module's entry point.

kmod_load Loads an object file into the kernel or queries for an object file already loaded.

kmod_unload Unloads a kernel object file.

pio_assist Provides a standardized programmed I/O exception handling mechanism for all routines performing programmed I/O.

prochadd Adds a system wide process state-change notification routine.

prochdel Deletes a process state change notification routine.

selnotify Wakes up processes waiting in a poll or select subroutine or the fp_poll kernel service.

setuerror Allows kernel extensions to set the u_error field in the u area.

uexadd Adds a system wide exception handler for catching user mode process exceptions.

uexblock Makes a process non-runnable when called from a user mode exception handler.

uexclear Makes a process blocked by the uexblock service runnable again.

uexdel Deletes a previously added system wide user mode exception handler.

3.3.3.2 Logical File System Kernel Services

The Logical File System services (also known as the *fp_services*) allow processes running in kernel mode to open and manipulate files in the same way that user mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

The Logical File System services are one component of the logical file system, which provides the functions required to map system call requests to virtual file system requests. The logical file system is responsible for resolution of file names and file descriptors. It tracks all open files in the system using the file table. The Logical File System services are lower level entry points into the system call support within the logical file system.

Routines in the kernel that must access data stored in files or that must set up paths to devices are the primary users of these services. This occurs most commonly in device drivers, where a lower level device driver must be accessed or where the device requires microcode to be downloaded. Use of the Logical File System services is not, however, restricted to these cases.

A process can use the Logical File System services to establish access to a file or device by calling:

- The **fp_open** service with a path name to the file or device it must access.
- The **fp_opendev** service with the device number of a device it must access.

- The **fp_getf** service with a file descriptor for the file or device.

If the process wants to retain access past the duration of the system call, it must then call the **fp_hold** service to acquire a private file pointer.

These three services return a file pointer that is needed to call the other Logical File System services. The other services provide the functions that are provided by the corresponding system calls.

Other Considerations: The Logical File System services are available only in the process environment. In addition, calling the **fp_open** service at certain times can cause a deadlock. The lookup on the file name must acquire file system locks. If the process is already holding any lock on a component of the path, the process will be deadlocked. Therefore, do not use the fp_open service when the process is already executing an operation that holds file system locks on the requested path. The operations most likely to cause this condition are those that create files.

There are 17 Logical File System kernel services.

fp_access Checks for access permission to an open file.

fp_close Closes a file.

fp_fstat Gets the attributes of an open file.

fp_getdevno Gets the device number and/or channel number for a device.

fp_getf Retrieves a pointer to a file structure.

fp_hold Increments the open count for a specified file pointer.

fp_ioctl Issues a control command to an open device or file.

fp_lseek Changes the current offset in an open file.

fp_open Opens a regular file or directory.

fp_opendev Opens a device special file.

fp_poll Checks the I/O status of multiple file pointers/descriptors and message queues.

fp_read Performs a read on an open file with arguments passed.

fp_readv Performs a read operation on an open file with arguments passed in iovec elements.

fp_rwuio Performs read and write on an open file with arguments passed in a uio structure.

fp_select Provides for cascaded, or redirected, support of the select or poll request.

fp_write Performs a write operation on an open file with arguments passed.

fp_writev Performs a write operation on an open file with arguments passed in iovec elements.

3.3.3.3 Process Management Kernel Services

The Process and Exception Management kernel services provided by the base AIX kernel provide the capability to:

- Create kernel processes
- Register exception handlers
- Provide process serialization
- Generate and handle signals
- Support event waiting and notification.

Kernel extensions can use the **creatp** and **initp** services to create and intialize a kernel process. Kernel processes are sheduled like user mode processes, but execute only within the kernel protection domain and have all security privileges. They can use the **sig_chk** service to poll for signals that have been sent to the kernel process.[2]

Kernel processes are usually created as follows:

- A user mode process loads the kernel extension containing the process code (using **sysconfig**).
- A user mode process invokes the kernel extension's config entry point (using **sysconfig** again). The config entry point uses *creatp* and *initp* to make the process ready to run.
- The user process becomes the parent.

The **setpinit** kernel service allows a kernel process to change its parent process from the one that created it to the init process, so that the creating process does not receive the death-of-child signal upon kernel process termination.

The **setjmpx**, **clrjmpx**, and **longjmpx** kernel services allow a kernel extension to register an exception handler by:

- Saving the exception handler's context with the setjmpx kernel service
- Removing its saved context with the clrjmpx kernel service if no exception occurred
- Invoking the next registered exception handler with the longjmpx kernel service if it was unable to handle the exception.

The typical sequence of operation should be:

- Extension uses *setjmpx* to save state.
- *setjmpx* returns zero.
- The exception occurs.
- The kernel first-level exception handler arranges for *longjmpx* to be called after the retrun to the interrupted code.
- *longjmpx* executes in the environment of the interrupted code.

---

[2] Kernel processes are not preemptable by signals.

- Execution continues at the return from *setjmpx*, but **with a non-zero return code**.

The **lockl** and **unlockl** kernel services allow kernel extensions executing in the process environment to acquire or release locks that are typically used to serialize access to a resource. This provides a method to access and update global memory. The AIX kernel processes are preemptable. This means that even though you own a lock on a data structure in global memory, a kernel process of higher priority may preempt your process. That higher priority process will not preempt your process IF it tries to lock the data structure that you have locked. In that case, your process will execute at this higher priority until you release the lock (and then the other process will preempt you.)

CAUTION: TO AVOID DEADLOCK

No deadlock detection is available.

To avoid deadlock, you must obey the following rules:

- Your system call (kernel service) should never return with locks still being held. (If you write a kernel process and use locks, do not return back to the calling process until you have released all locks.)
- Nesting locks (using more than one at a time) is permitted only if the nested lock has a finer granularity.
- Order of locks:
  - The **kernel_lock** has the coarsest granularity.
  - The file system locks (private to the filesystem).
  - Device driver locks (private to a device driver).
  - Private fine granularity locks.
- Locks must be unlocked in the reverse order of obtaining them.

The **getpid** kernel service can be used by a kernel extension in either the process or interrupt environment to determine the current execution environment and obtain the process ID of the current process if in the process environment.

The event notification services provide support for primitive interprocess communications where there can be only one process waiting on the event or shared event interprocess communications where there can be multiple processes waiting on the event. The traditional *sleep* and *wakeup* kernel services are also provided for code that is being ported from other Unix operating systems or previous versions of the AIX operating system. These compatibility services require that the caller have the global *kernel_lock* (defined in /usr/include/sys/lockl.h), which is released before waiting in the sleep routine and re-acquired upon wakeup.

NOTE

For performance reasons, use the e_sleep and e_wakeup kernel services instead of the traditional sleep and wakeup calls from the kernel.

The **e_wait** and **e_post** kernel services support single waiter event notification by using mutually agreed upon event control bits for the process being posted. There are a limited number of control bits available for use by kernel extensions. If the *kernel_lock* is owned by the caller of the e_wait service, it is released and re-acquired upon wakeup.

The **e_wakeup**, **e_sleep** and **e_sleepl** kernel services support a shared event notification mechanism that allows for multiple processes to be waiting on the shared event. These services support an unlimited number of shared events (by using caller-supplied event words). All processes waiting on the shared event are awakened by the e_wakeup service. If the caller of the e_sleep service owns the kernel lock, it is released before waiting and is reacquired upon wakeup. The e_sleepl service provides the same function as the e_sleep service except that a caller-specified lock is released and reacquired instead of the kernel_lock.

There are 19 Process and Exception Management kernel services:

clrjmpx Removes a saved context by popping the most recently saved jump buffer from the list of saved contexts.

creatp Creates a new kernel process.

e_post Notifies a process of the occurrence of one or more events.

e_sleep Forces a process to wait for the occurrence of a shared event.

e_sleepl Forces a process to wait for the occurrence of a shared event.

e_wait Forces a process to wait for the occurrence of an event.

e_wakeup Notifies processes waiting on a shared event of the event's occurrence.

getpid Gets the process ID of the current process.

initp Changes the state of a kernel process from idle to ready.

lockl Locks a conventional process lock.

longjmpx Allows exception handling by causing execution to resume at the most recently saved context.

pdsignal Sends a signal to a process group.

pidsig Sends a signal to a process.

setjmpx Allows saving the current execution state or context.

setpinit Sets the parent of the current kernel process to the init process.

sig_chk Provides a kernel process the ability to poll for receipt of signals.

sleep Forces the calling process to wait on a specified channel.

unlockl Unlocks a conventional process lock.

wakeup Activates processes sleeping on the specified channel.

Chapter 4. Overview of a Character Device Driver

The purpose of this chapter is to describe in detail the components and makeup of a character device driver. The way this will be done is that a topic will be discussed, then it will be illustrated with an example of working code. The code is from the sample device driver found in "Sample Character Device Driver" on page E-1. This device driver will drive one port on the Real Time Interface Co-Processor with an optional interface board containing eight RS232 serial communications ports. This example is to demonstrate a method of writing a device driver. The device driver itself has not been extensively tested.

The name of the driver is **ric** (ffrom Real Time Interface Co-Processor), therefore the various entry points will be prefixed with this name. However, in the discussion of the general features of an entry point, the prefix **dd** will be used.

4.1 Implementation

4.1.1 ddconfig Device Driver Entry Point

Figure 4-1 on page 4-2 shows the device driver entry point.

© Copyright IBM Corp. 1991

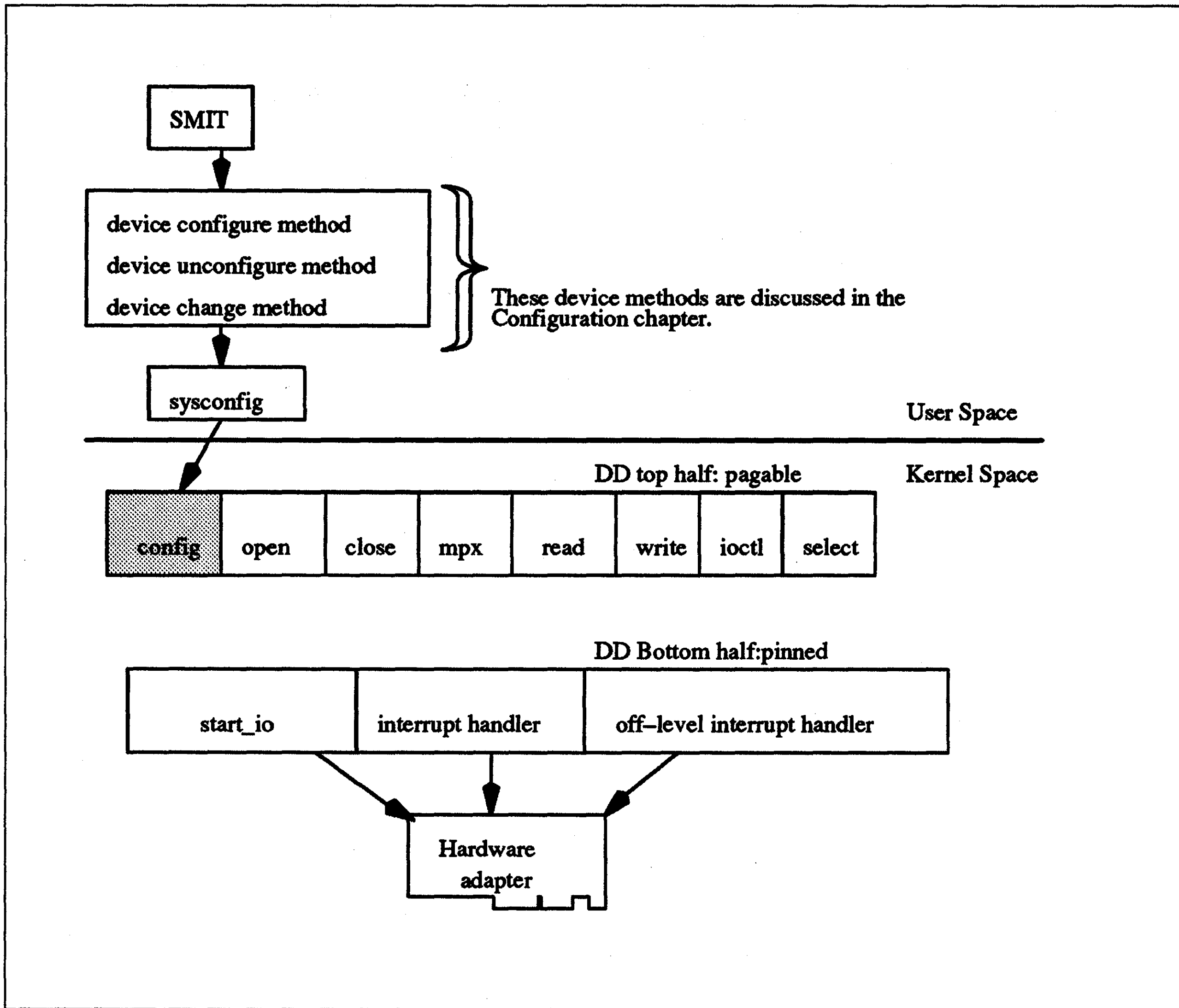


Figure 4-1. Device Driver ddconfig Entry Point

The **ddconfig** entry point is used to configure the device driver. It is called to do the following tasks:

- Initialize the device driver
- Terminate the device driver
- Request configuration data for the supported device
- Perform other device-specific configuration functions.

The **ddconfig** routine and its operations are called in the process environment only. Refer to Figure 4-2 on page 4-5 for the **ricconfig** code sample.

The **ddconfig** routine is called by the device's **Configure**, **Unconfigure**, or **Change** method. Typically, it is called once for each minor device number which is supported. This is determined by the specific device method. (A device method is a process that goes about configuring/unconfiguring or changing a device. It does this via a call to sysconfig---see the example in /usr/lpp/bos/samples/cfgyyy.c.)

Additional device-specific functions relating to configuration could also be provided by **ddconfig**, such as returning device vital product data (VPD)[1]. **ddconfig** is invoked through the **sysconfig** subroutine by the **Configure** method. (sysconfig actually calls an SVC which in turn calls ddconfig so that the supervisor state is entered.) A discussion of device methods is included in "Device Drivers Configuration" on page 6-1.

Three parameters are expected by **ddconfig.** These are **devno**, **cmd**, and **uiop**, where:

devno specifies the major and minor device numbers

cmd specifies the function to be performed

uiop points to a **uio** structure describing the relevant data area for configuration information.

The following example shows the syntax of a **ddconfig** entry point.

```
#include <sys/device.h>
#include <sys/types.h>

int ddconfig(devno, cmd, uiop)
dev_t devno;
int cmd;
struct uio *uiop;
```

The values for the **cmd** parameter which are usually supported by device drivers and their associated methods are:

CFG_INIT Initialize the device driver and internal data areas.

CFG_TERM Terminate the device driver associated with **devno**.

CFG_QVPD Query device-specific vital product data (VPD).

The data area pointed at by the **uiop** parameter has two different purposes, depending on the **cmd** function. If the **CFG_INIT** command has been requested, the **uiop structure** describes the location and length of the device-dependent data structure, **DDS**, from which to read the information. If the **CFG_QVPD** command has been requested, the **uiop** structure describes the area in which to write vital product data information. The content and format of this information is established by the specific device methods in conjunction with the device driver.

The **uiomove** kernel service may be used to facilitate the copying of information into or out of this data area. The format of the **uio** structure is defined in the **<sys/uio.h>** header file.

The **ddconfig** routine sets the return code to **0** if no errors are detected for the operation specified. If an error is to be returned to the caller, a nonzero return

[1] (Vital Product Data is device specific information that is specific to an adapter and can include information such as the engineering change level of the adapter, etc.)

code should be provided. The return code used should be one of the values defined in the **<sys/errno.h>** header file. If this routine was invoked by a **sysconfig** subroutine call, the return code is passed to its caller (typically a device method). It is passed by presenting the error code in the **errno** external variable and providing a **-1** return code to the subroutine.

```
 1
 2  /****************************************************************************
 3
 4       ricconfig: performs operations necessary for the intitialisation
 5            of an individual port on the adapter.  ricconfig will be
 6            called for each valid port during the bus/device config
 7            phase of the boot procedure.
 8
 9  ****************************************************************************/
10  int ricconfig(devno, cmd, uiop)
11  dev_t devno;
12  int cmd;
13  struct uio *uiop;
14  {
15          int port_num;           /* port number  */
16          int adapt_num;          /* adapter number */
17          int minor_num;          /* minor device number  */
18          t_ric_dds       *dds_ptr;/* pointer to DDS      */
19          t_acb           *acb_ptr;/* pointer to ACB      */
20          int ret;                /* return values        */
21          unsigned long   bus_sr; /* IO Seg Reg number mask */
22          unsigned long   iob;    /* io base address      */
23          unsigned long   memb;   /* bus memory base      */
24
25       /* get minor number. macro defined in /usr/include/sys/sysmacros.h */
26          minor_num = minor(devno);
27
28       /* if the minor number is bad, return */
29          if (minor_num >= (MAX_ADAP*NUM_PORTS))
30          {
31             return(EINVAL);
32          }
33
34        /* get a DDS pointer    */
35          dds_ptr = dds_dir[minor_num];
36
37     switch(cmd)              /* switch on command type */
38     {
39       /* initialise device driver and internal data areas        */
40          case CFG_INIT:
41          {
42
43              /* first check whether dds exists      */
44                  if (dds_ptr != (t_ric_dds *)NULL)
45                  {
46                          return(EINVAL);
47                  }
48
49          /* now, if this is the first time through CFG_INIT, certain
50           * things must be done. no active adapters means first time
51           */
52                  if (act_adap == 0)
53                  {
54                       /* pin ric into memory  */
55                          if (( ret = pincode(ricconfig)) !=0)
56                          {
57                               /* return if pin fails     */
58                                  return(ret);
59                          }
60                       /* ok, so now it is pinned      */
```

Figure 4-2 (Part 1 of 5). Code Sample of the ricconfig Routine

```
 61
 62                  /* add entry points to the devsw table  */
 63
 64                         ricsw.d_open     = ricopen;
 65                         ricsw.d_close    = ricclose;
 66                         ricsw.d_read     = ricread;
 67                         ricsw.d_write    = ricwrite;
 68                         ricsw.d_ioctl    = ricioctl;
 69                         ricsw.d_strategy = nodev;
 70                         ricsw.d_ttys     = NULL;
 71                         ricsw.d_select   = ricselect;
 72                         ricsw.d_config   = ricconfig;
 73                         ricsw.d_print    = nodev;
 74                         ricsw.d_dump     = nodev;
 75                         ricsw.d_mpx      = ricmpx;
 76                         ricsw.d_revoke   = nodev;
 77                         ricsw.d_dsdptr   = NULL;
 78                         ricsw.d_selptr   = NULL;
 79                         ricsw.d_opts     = 0;
 80
 81          /* if adding the entry points to devsw fails, return    */
 82                         if((ret = devswadd(devno, &ricsw)) != 0)
 83                         {
 84                                 unpincode(ricconfig);
 85                                 return(ret);
 86                         }
 87                  } /* end first time through */
 88           /* For this example we are allocating pinned space and  */
 89           /* then we will copy the dds data structure */
 90           /* allocate space for dds */
 91                 dds_ptr = (t_ric_dds *)xmalloc (sizeof(t_ric_dds),
 92                            2, pinned_heap);
 93
 94           /* if the xmalloc fails, return */
 95                 if(dds_ptr == (t_ric_dds *)NULL)
 96                 {
 97                         free_it_up(act_adap, devno, NULL, NULL);
 98                         return(ENOMEM);
 99                 }
100
101           /* zero out dds */
102                 bzero((char *)dds_ptr, sizeof(t_ric_dds));
103
104           /* copy input struct into dds */
105                 ret = uiomove(dds_ptr, sizeof(t_ric_dds), UIO_WRITE,
106                                 uiop);
107
108           /* if uiomove is bad */
109                 if(ret)
110                 {
111                         free_it_up(act_adap, devno, dds_ptr, NULL);
112                         return(ret);
113                 }
114
115           /* set port number from dds */
116                 port_num = dds_ptr->dds_dvc.port_num;
117
118           /* adapter number is slot number */
119                 adapt_num = dds_ptr->dds_hdw.slot_num;
120                 acb_ptr = acb_dir[adapt_num];
```

Figure 4-2 (Part 2 of 5). Code Sample of the ricconfig Routine

```
121
122                /* if no ACB for this device */
123                    if(acb_ptr == (t_acb *)NULL)
124                    {
125                            /* allocate memory for the acb */
126                            acb_ptr = (t_acb *)xmalloc(sizeof(t_acb),
127                                       2, pinned_heap);
128
129                            /* if the allocation fails */
130                            if(acb_ptr == (t_acb *)NULL)
131                            {
132                                    free_it_up(act_adap, devno, dds_ptr,
133                                                  NULL);
134                                    return(ENOMEM);
135                            }
136
137                            /* zero out acb */
138                            bzero((char *)acb_ptr, sizeof(t_acb));
139
140                            /* now fill it in */
141                            acb_ptr->p_port_dds[port_num] = dds_ptr;
142
143                    /* now set up the POS register settings */
144                            acb_ptr->int_lvl = dds_ptr->dds_hdw.bus_intr_lvl;
145                            acb_ptr->slot_num = (unsigned
146                                                char)(dds_ptr->dds_hdw.slot_num);
147                            acb_ptr->arb_lvl = dds_ptr->dds_hdw.dma_lvl;
148                            acb_ptr->io_base = dds_ptr->dds_hdw.bus_io_addr;
149                            acb_ptr->mem_base = dds_ptr->dds_hdw.bus_mem_addr;
150                            acb_ptr->dma_base = dds_ptr->dds_hdw.tcw_bus_mem_addr;
151                            acb_ptr->io_segreg_val = IO_SEG_REG;
152                            acb_ptr->adapter_state = 0;
153                            acb_ptr->cpu_page = 0xFF;
154
155                    /* invoke set_POS to set POS registers */
156                    set_POS( acb_ptr );
157
158                    /* set up segment register for next phase */
159                    bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val, 0);
160
161                    /* set up the busio and bus memory base address for the card */
162                    iob = acb_ptr->io_base + bus_sr;
163                    memb = acb_ptr->mem_base + bus_sr;
164                    ret = reset_card ( acb_ptr, bus_sr, iob, memb);
165
166                    /* free up segment register */
167                    BUSIO_DET(bus_sr);
168
169                    if /* reset failed... */
170                       ( ret )
171                    {
172                        free_it_up(act_adap, devno, dds_ptr, acb_ptr);
173                        return(EIO);
174                    }
175
176                    acb_ptr->c_intr_rcvd = 0;     /* zero interrupt count  */
177
178                    /* now we set up our DMA channel by calling d_init */
179                    acb_ptr->dma_channel_id =
180                            d_init((int)acb_ptr->arb_lvl, MICRO_CHANNEL_DMA,
181                                    acb_ptr->io_segreg_val);
182
```

Figure 4-2 (Part 3 of 5). Code Sample of the ricconfig Routine

```
183                 /* free up resources if d_init failed */
184                 if (acb_ptr->dma_channel_id == DMA_FAIL)
185                 {
186                     free_it_up(act_adap, devno, dds_ptr, acb_ptr);
187                     return(EIO);
188                 }
189
190                 /* enable DMA channel */
191                 d_unmask( acb_ptr->dma_channel_id);
192
193                 act_adap++;                     /* adapter is now active */
194                 acb_dir[adapt_num] = acb_ptr;
195
196             } /* end of no existing acb if */
197
198             acb_ptr->n_cfg_ports++;
199             acb_ptr->p_port_dds[port_num] = dds_ptr;
200             dds_dir[minor_num] = dds_ptr;
201             break;
202
203         } /* end case CFG_INIT  */
204
205      /* terminate the device driver associated with the specified devno */
206         case CFG_TERM:
207         {
208             if (dds_ptr == NULL)
209                     return(EACCES);
210
211             if (dds_ptr->dds_dvc.port_state != CLOSED)
212                     return(EBUSY);
213
214             port_num = dds_ptr->dds_dvc.port_num;
215             adapt_num = dds_ptr->dds_hdw.slot_num;
216             acb_ptr = acb_dir[dds_ptr->dds_hdw.slot_num];
217
218             /* decrement number of configured ports on this adapter */
219             acb_ptr->n_cfg_ports--;
220
221             /* if last configured port on adapter, free adapter resources */
222             if (acb_ptr->n_cfg_ports == 0)
223             {
224                     /* Release the dma_channel */
225                     d_mask(acb_ptr->dma_channel_id);
226                     d_clear(acb_ptr->dma_channel_id);
227
228                     /* decrement number of active adapters */
229                     act_adap--;
230
231                     free_it_up(act_adap, devno, dds_ptr, acb_ptr);
232                     acb_dir[adapt_num] = (t_acb *)NULL;
233             }
234             else
235             {
236                     /* free up allocated resources. If number         */
237                     /* of active adapters now zero,                   */
238                     /* delete switch table entry and unpin the driver */
239                     free_it_up(act_adap, devno, dds_ptr, NULL);
240                     acb_ptr->p_port_dds[port_num] = NULL;
241             }
242
243             dds_dir[minor_num] = NULL;
244             break;
245         } /* end case CFG_TERM */
```

Figure 4-2 (Part 4 of 5). Code Sample of the ricconfig Routine

```
246
247       /*  query device specific VPD         */
248          case CFG_QVPD:
249                  break;
250
251          default:
252                  return(EINVAL);
253      } /* end switch statement */
254      return(0);
255   } /* end ricconfig */
256
```

Figure 4-2 (Part 5 of 5). Code Sample of the ricconfig Routine

4.1.1.1 A Brief Discussion of the DDS

The Device Dependent Structure, or **DDS**, contains information that describes a device instance to the device driver. It typically contains information about device-dependent attributes as well as any other information the driver needs to communicate with the device. The device driver writer defines what information is to go into the DDS. In many cases, information about a device's parent is included. For instance, a driver needs information about the adapter, and the bus the adapter is plugged into, to communicate with a device connected to an adapter.

A device's **DDS** is built each time the device is configured. The **Configure** method can fill in the **DDS** with fixed values, computed values, and information from the Configuration database. Most of the information from the Configuration database usually comes from the attributes for the device in the Customized Attribute, or **CuAt** object class, but it can come from any of the object classes. Information from the database for the device's parent device or parent's parent device can also be included. The **DDS** is passed to the device driver with the **SYS_CFGDD** option of the **sysconfig** subroutine, which calls the device driver's **ddconfig** routine with the **CFG_INIT** command.

The **Change** method is invoked when changing the configuration of a device. The **Change** method must ensure consistency between the Configuration database and the view that any device driver may have of the device. This is accomplished by:

1. Not allowing the configuration to be changed if the device has configured children, that is, children in either the Available or Stopped states. This ensures that a **DDS** that has been built using information in the database about a parent device will remain valid because the parent cannot be changed.

2. If a device has a device driver and the device is in either the Available or Stopped states, the **Change** method must communicate to the device driver any changes that would affect the **DDS**. This may be accomplished with ioctl operations, if the device driver provides the support to do so. It can also be accomplished by taking the following steps:

   a. Terminating the device instance by calling the **sysconfig** subroutine with the **SYS_CFGDD** option. The **SYS_CFGDD** operation calls the device driver's **ddconfig** routine with the **CFG_TERM** command.

   b. Rebuilding the **DDS** using the changed information.

c. Passing the new **DDS** to the device driver by calling the **sysconfig SYS_CFGDD** operation. This operation then calls the **ddconfig** routine with the **CFG_INIT** command.

Many **Change** methods simply invoke the device's **Unconfigure** method, apply changes to the database, then invoke the device's Configure method. This process ensures the two stipulated conditions since the **Unconfigure** method, and thus the change, will fail, if the device has Available or Stopped children. Also, if the device has a device driver, its **Unconfigure** method terminates the device instance. Its **Configure** method also rebuilds the **DDS** and passes it to the driver.

There is no single defined **DDS** format. Writers of device drivers and device methods must agree upon a
particular device's **DDS** format. When obtaining information about a parent device, you may want to group that information together in the **DDS**.

When building a **DDS** for a device connected to an adapter card, you will typically need to pick up the following adapter information:

slot number Obtained from the connwhere descriptor of the adapter's Customized Device, or **CuDv**, object.

bus resources Obtained from attributes for the adapter in the Customized Attribute, or **CuAt**, or Predefined Attribute, or **PdAt** object classes. These include attributes for bus interrupt levels, interrupt priorities, bus memory addressed, bus I/O addresses, and DMA arbitration levels.

These two attributes must be obtained for the adapter's parent bus device:

bus_id Identifies the I/O bus. This field is needed by the device driver to access the I/O bus.

bus_type Identifies the type of bus, such as a Micro Channel bus, or a PC AT bus.

Note

The **getattr** device configuration subroutine should be used whenever attributes are obtained from the Configuration database. This routine returns the Customized attribute value if the attribute is represented in the Customized Attribute (CuAt) object class. Otherwise, it returns the default value from the Predefined Attribute (PdAt) object class.

Finally, a **DDS** generally includes the device's logical name. This is used by the device driver to identify the device when logging an error for the device.

Figure 4-3 on page 4-11 shows an example of a **ric DDS**.

```
 1
 2  /*************************************************************************
 3   *      Define Device Structure                                         *
 4   *************************************************************************/
 5  typedef struct   RICDDS
 6  {
 7      struct DDS_HDW
 8      {
 9          unsigned int    slot_num;       /* slot number of adapter */
10
11          unsigned int    bus_intr_lvl;   /* interrupt level */
12
13          unsigned short  intr_priority;  /* interrupt priority */
14
15          unsigned short  dma_lvl;        /* this is the bus arbitration level */
16                                          /* for this adapter */
17
18          unsigned int    bus_io_addr;    /* base of Bus I/O area for this */
19                                          /* adapter */
20
21          unsigned int    bus_mem_addr;   /* base of Bus Memory */
22                                          /* addressability for this adapter */
23
24          unsigned int    tcw_bus_mem_addr; /* base of Bus Memory DMA */
25                                            /* addressability for this adapter */
26
27      } dds_hdw;
28
29      struct DDS_DVC
30      {
31          unsigned char   port_num;       /* Port Number for this port */
32
33          unsigned char   port_state;     /* Port State */
34
35          unsigned short  rdto;           /* Receive Data Transfer Offset */
36
37          int             net_id;         /* Network ID */
38
39      } dds_dvc;
40
41      struct DDS_RAS
42      {
43          t_cio_stats cio_stats;          /* number of receives for port  */
44          t_err_threshold err_thresh;     /* number of transmits for port */
45      } dds_ras;
46
47      struct DDS_VPD
48      {
49          unsigned short  card_id;        /* Card ID...POS0 & POS1 */
50          unsigned short  ver_num;        /* Version Number */
51          char            devname[16];      /* logical device name */
52          char            adpt_name[16];  /* logical adapter name */
53      } dds_vpd;
54
```

Figure 4-3 (Part 1 of 2). Example of a DDS

```
55      struct DDS_WRK
56      {
57          unsigned short  field1;    /* put whatever you want here */
58          unsigned short  field2;
59          unsigned char   field3;
60      } dds_wrk;
61
62  } t_ric_dds;
63
```

Figure 4-3 (Part 2 of 2). Example of a DDS

4.1.2 ddmpx Device Driver Entry Point

Figure 4-4 shows the device driver ddmpx entry point for an open (or create) system call.

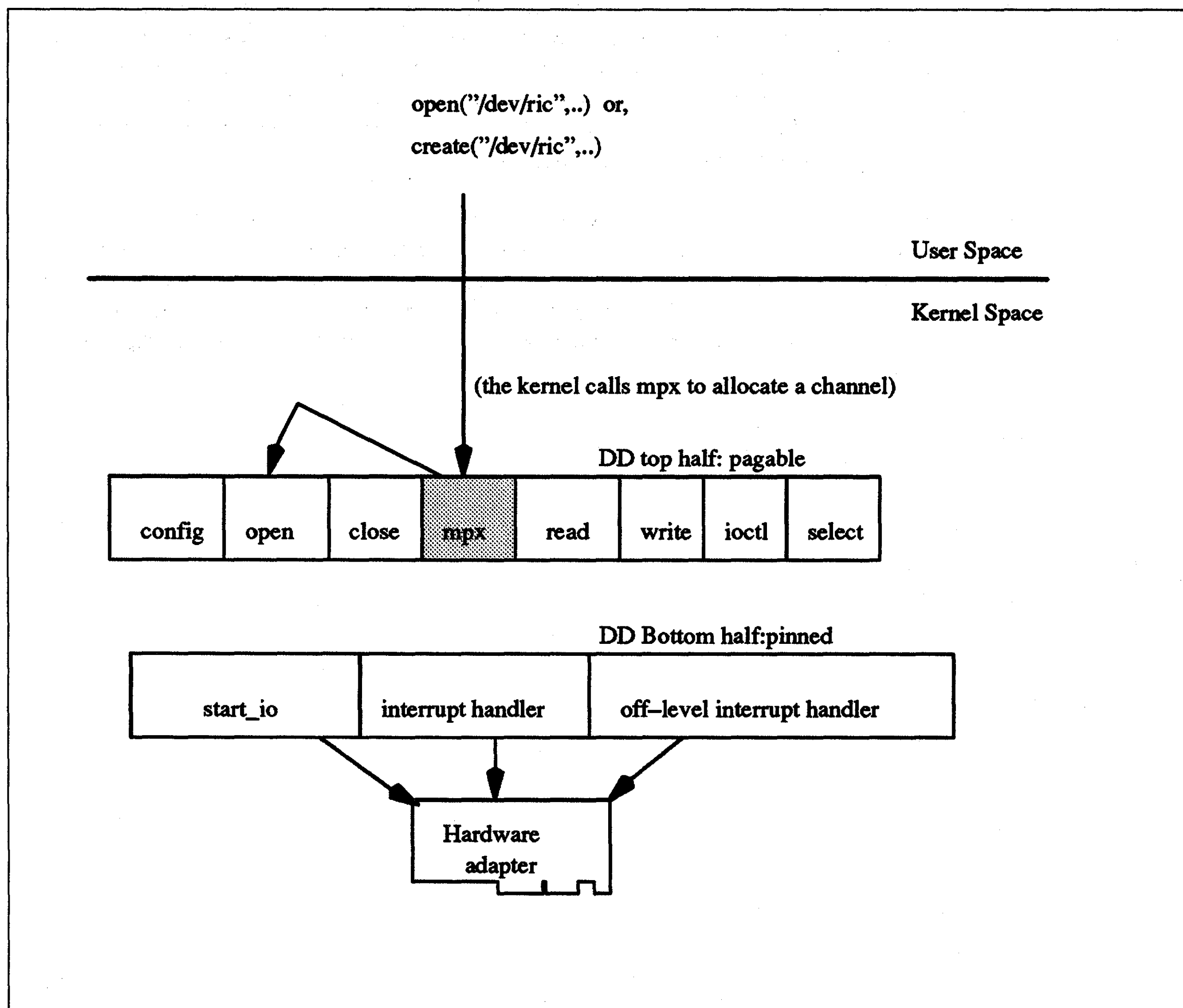


Figure 4-4. Device Driver ddmpx Entry Point for an open

The **ddmpx** device driver entry point allocates or deallocates a channel for a multiplexed device driver. This routine is called in the process environment only. See Figure 4-8 on page 4-18 for the **ricmpx** code sample.

The following example shows the syntax of a **ddmpx** entry point.

```
#include <sys/device.h>
#include <sys/types.h>

int ddmpx (devno, chanp, channame)
dev_t devno;
chan_t *chanp;
char *channame;
```

The values passed to the **ddmpx** entry point are:

devno Specifies the major and minor device numbers.

channame Points to the path name extension for the channel to be allocated.

chanp Address of the channel ID, passed by reference. The channel ID will be allocated by the ddmpx.

A multiplexed device driver is a character class device driver that supports the assignment of channels to provide finer access control to a device or virtual subdevice. This type of device driver has the capability to decode special channel-related information appended to the end of the path name of the special file for the device. This path name extension is used to identify a logical or virtual subdevice or channel.

Figure 4-5 on page 4-14 shows the relationship that exists between major numbers, minor numbers, and multiplexed channels. A major number can be used to indicate a certain port on an adapter. A channel can be used to indicate a certain process that has access to a port (minor number) on an adapter. Multiple processes can therefore share a port - hence the term "Multiplexed" is used.

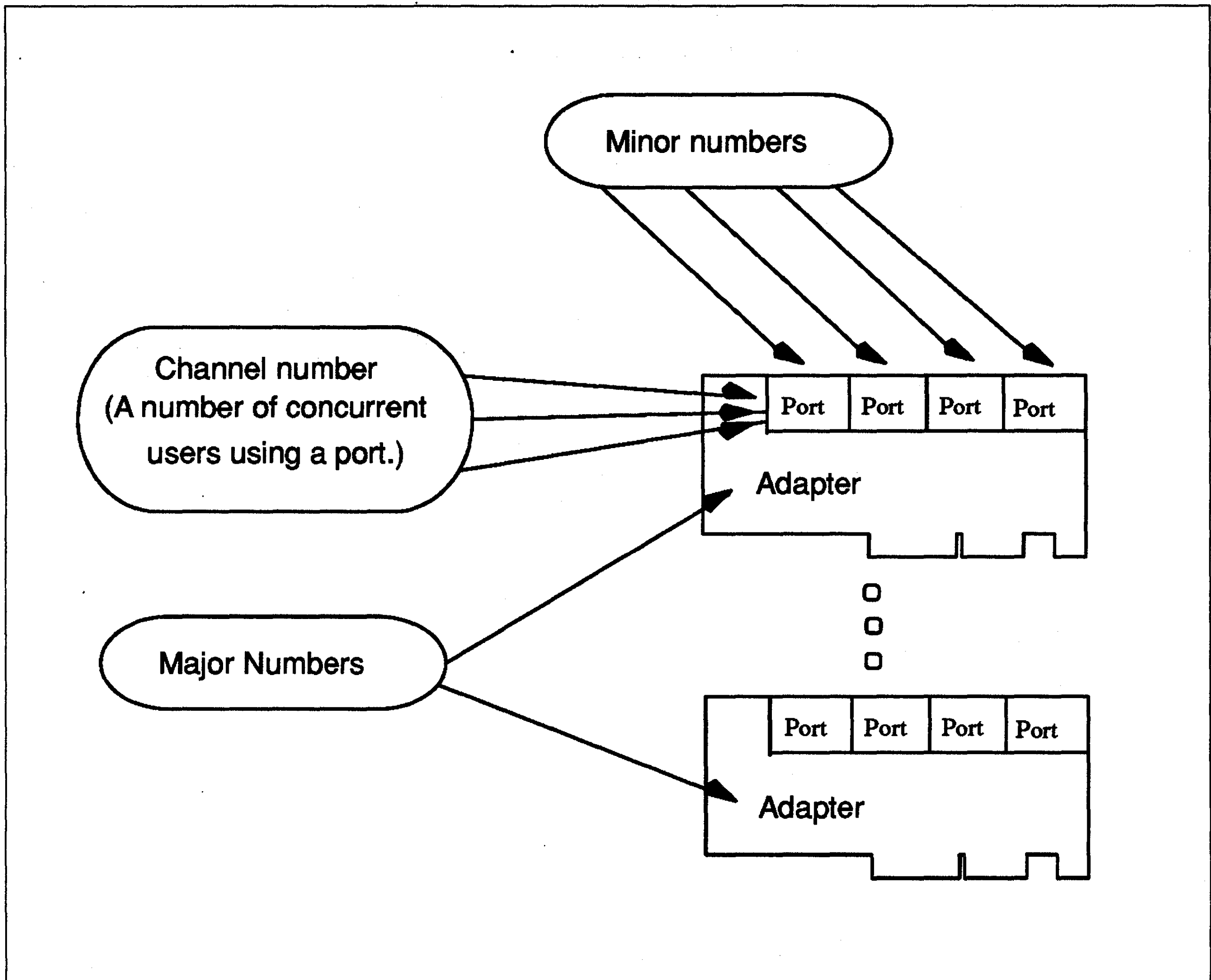


Figure 4-5. Relationship of Major Numbers, Minor Numbers and Channels

Only multiplexed character class device drivers may provide the **ddmpx** routine, and every multiplexed driver must do so. The **ddmpx** routine may not be provided by block device drivers even when providing raw read/write access. A multiplexed device driver is a character class device driver that supports the assignment of channels to provide finer access control to a device or virtual subdevice. This type of device driver has the capability to decode special channel-related information appended to the end of the path name of the special file for the device. This path name extension is used to identify a logical or virtual subdevice or channel.

When an **open** or **creat** subroutine call is issued to a device instance supported by a multiplexed device driver, the kernel calls the device driver's **ddmpx** routine to allocate a channel. Upon allocation, the kernel dynamically creates in-core inodes, or **gnodes**, for channels on a multiplexed device to allow the protection attributes to be different for various channels.

To allocate a channel, the **ddmpx** routine is called with a **channame** pointer to the path name extension. The path name extension starts after the first **/** character that follows the special file name in the path name. The **ddmpx** routine should perform the following actions:

- Parse this path name extension.
- Allocate the corresponding channel.
- Return the channel ID through the chanp parameter.

If no **path name extension** exists, the **channame** pointer points to a null character string. In this case, an available channel should be allocated and its **channel ID** returned through the **chanp** parameter.

If no error is returned from the **ddmpx** routine, the returned **channel ID** is used to determine if the channel was already allocated. If already allocated, the **gnode** for the associated channel has its reference count incremented. If the channel was not already allocated, a new **gnode** is created for the channel. In either case, the device driver's **ddopen** routine is called with the channel number assigned by the **ddmpx** routine. If a nonzero return code is returned by the **ddmpx** routine, the channel is assumed not to have been allocated, and the device driver's **ddopen** routine is not called. Refer to Figure 4-6.

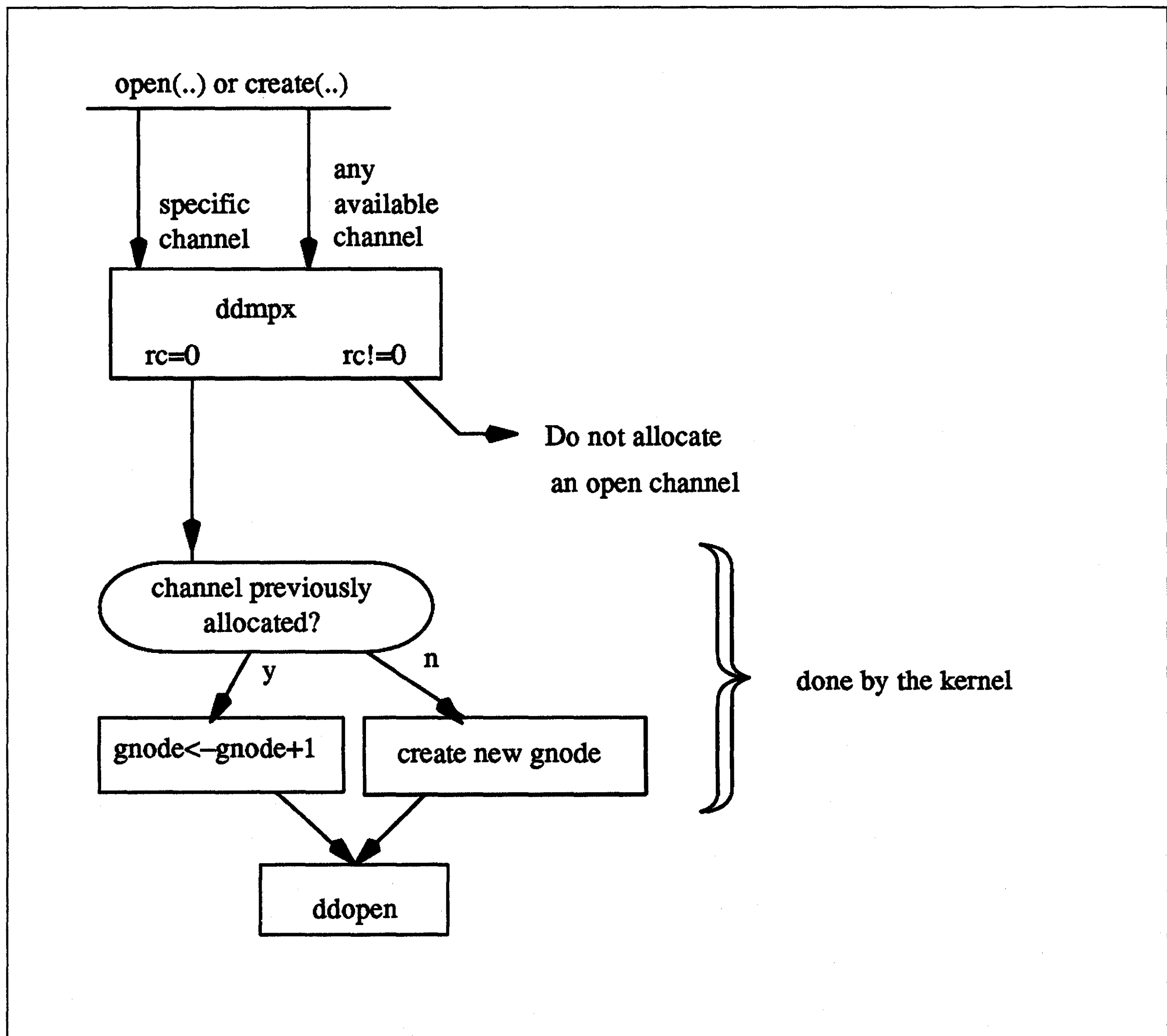


Figure 4-6. ddmpx for open and create

When a **close** subroutine call is issued to a device instance supported by a multiplexed device driver, the kernel decrements the channel's **gnode** reference count and if this count is now equal to zero, it calls the **ddmpx** routine (i.e. ddmpx is called when the channel is no longer used). The **ddmpx** routine deallocates the channel after the **ddclose** routine was called to close the last use of the channel. If a nonzero return code is returned by the **ddclose** routine, the **ddmpx** routine is still called to deallocate the channel. The **ddclose** routine's return code is saved, to be returned to the caller. If the **ddclose** routine returned no error, but a nonzero return code was returned by the **ddmpx** routine, the channel is assumed to be deallocated, although the return code is returned to the caller. Refer to Figure 4-7.

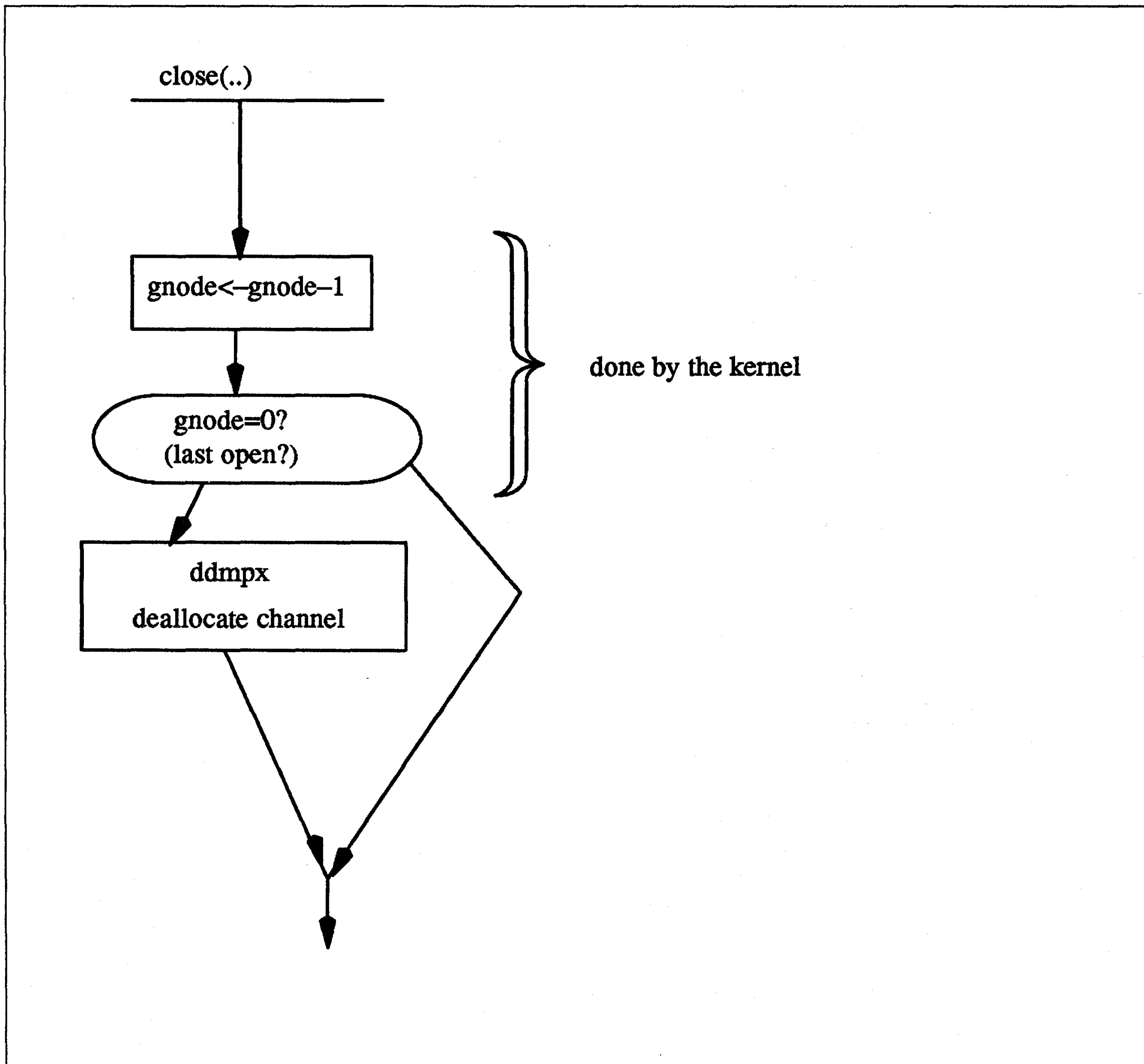


Figure 4-7. ddmpx for close

To deallocate a channel, the **ddmpx** routine is called with a null **channame** pointer and the channel ID passed by reference in the **chanp** parameter. If the channel **gnode** reference count has gone to **0**, the kernel calls the **ddmpx**

routine to deallocate the channel after invoking the **ddclose** routine to close it. The **ddclose** routine should not itself deallocate the channel.

If the allocation or deallocation of a channel is successful, the **ddmpx** routine should return a return code of **0**. If an error occurs on allocation or deallocation, a nonzero return code should be returned. The return code should conform to the return codes described for the **open** and **close** subroutines in the *POSIX 1003.1* standard, where applicable. Otherwise, the return code should be one defined in the **<sys/errno.h>** header file.

REMEMBER

The ddmpx routine should allocate an unused channel if:

```
channame = (char *) NULL;.
```

The ddmpx routine should close a channel if:

```
channame = NULL;
```

```
 1
 2   /**************************************************************************
 3    *
 4    *      ricmpx is the mpx entry point to allocate or deallocate a
 5    *      channel.
 6    *
 7    **************************************************************************/
 8   ricmpx(devno, chanp, channame)
 9   dev_t devno;
10   int *chanp;
11   char *channame;
12   {
13       t_acb               *acb_ptr;         /* pointer to ACB */
14    /* ACB is the adapter control block. There is one ACB for each  */
15    /* adapter in the system */
16       t_ric_dds           *dds_ptr;         /* pointer to DDS */
17       int                 tmp_chan;       /* local chan storage */
18
19
20     /* if minor number is bad, return  */
21       if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
22       {
23           return(EINVAL);
24       }
25     /* Note: in our sample program, a port on the RIC will be allocated if */
26     /* the minor device number that is passed in has not been previously   */
27     /* allocated a port. (port 0 is always allocated here) Whatever process */
28     /* opens the port totally owns the port until a ricmpx call is made to */
29     /* deallocate that port. */
30
31     /* set up DDS pointer */
32       dds_ptr = dds_dir[minor(devno)];
33
34     /* if dds pointer is null, return error */
35       if (dds_ptr == NULL)
36           return(EINVAL);
37
38     /* get the acb pointer */
39       acb_ptr = acb_dir[dds_ptr->dds_hdw.slot_num];
40
41     /* see if we've been called to deallocate the channel */
42       if ( channame == (char *)NULL )
43       {
44         /* Deallocate the channel */
45           dds_ptr->dds_wrk.cur_chan_num = 0;
46
47         /* on a deallocate, always set diag flag to 0 */
48           acb_ptr->diag_flag = 0;
49       }
50       else
51       {
52         /* get channel allocated indicator */
53           tmp_chan = (int)dds_ptr->dds_wrk.cur_chan_num;
54
55         /* if channel number already allocated, return error */
56           if (tmp_chan > 0)
57           {
58               return(ENXIO);
59           }
```

Figure 4-8 (Part 1 of 2). Code Sample of the ricmpx Routine

```
60
61            /* not diagnostics open */
62              acb_ptr->diag_flag = 0;
63
64          dds_ptr->dds_wrk.cur_chan_num = 1;        /* allocate channel 0 */
65          *chanp = 0;                              /* channel returned is 0 */
66      }
67      return(0);
68  } /* end ricmpx */
69
```

Figure 4-8 (Part 2 of 2). Code Sample of the ricmpx Routine

4.1.3 ddopen Device Driver Entry Point

Figure 4-9 shows the device driver ddopen entry point.

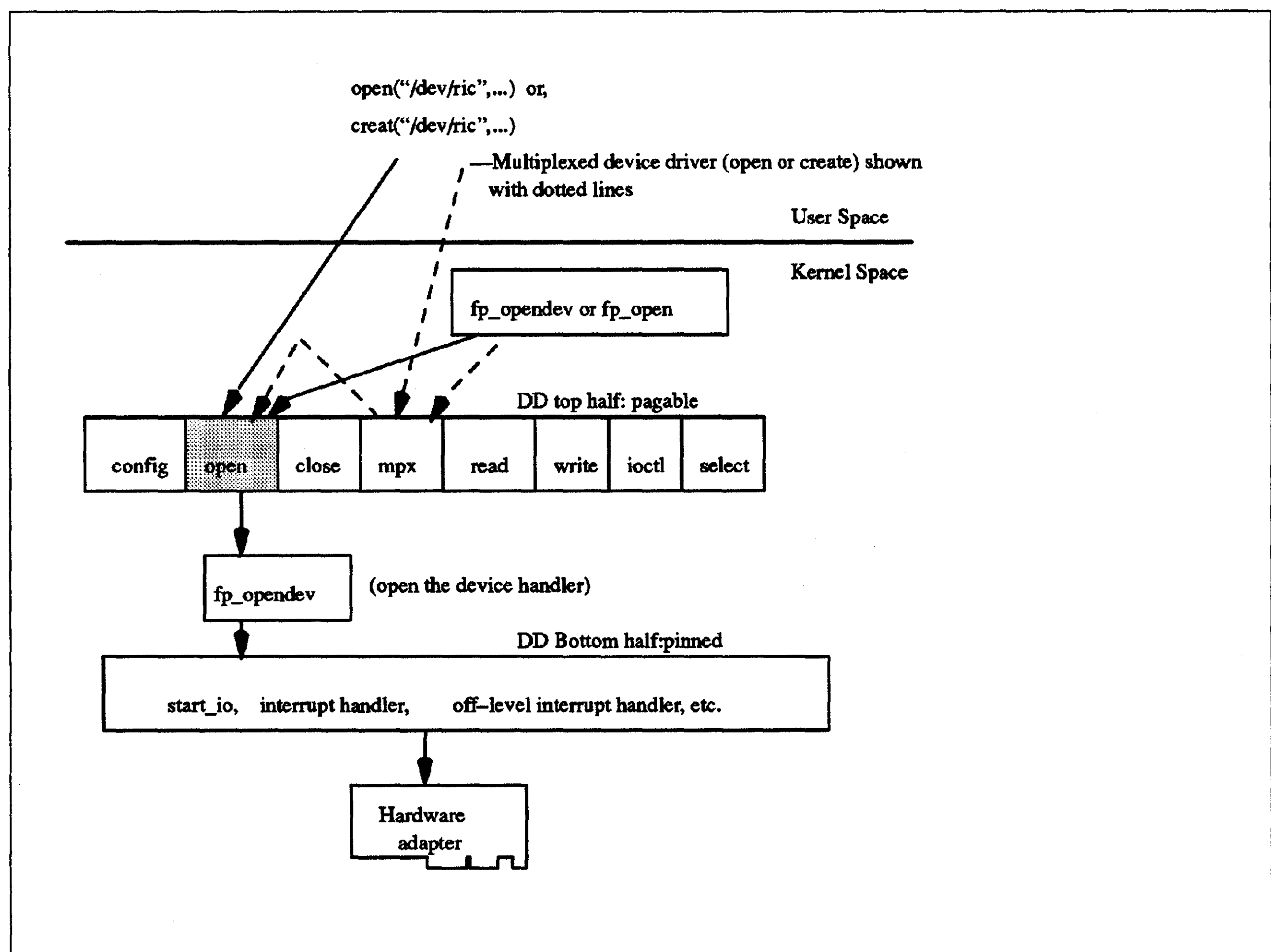


Figure 4-9. Device Driver ddopen Entry Point

The **ddopen** device driver entry point prepares a device for reading, writing, or control functions. The **ddopen** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Locking Device Driver Data Structures

Please refer to page 3-20 for a discussion on the lockl and unlockl kernel services and why you need to use them for global data structures. (Remember, the AIX kernel is preemptable.)

See Figure 4-10 on page 4-22 for the **ricopen** sample code.

The **ddopen** routine expects four parameters. These are **devno**, **devflag**, **chan**, and **ext**, where:

devno Indicates major and minor device numbers.

devflag Specifies open file control flags.

chan Specifies the channel number.

ext Specifies the extension parameter.

The following example shows the syntax of a **ddopen** entry point.

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

int ddopen (devno, devflag, mpxchan, ext)
dev_t devno;
ulong devflag;
chan_t mpxchan;
int ext;
```

The kernel calls the **ddopen** routine of a device driver when a program issues an **open** or **creat** subroutine call. It can also be called when a system call, kernel process, or other device driver uses the **fp_opendev** or **fp_open** kernel service to use the device.

The **ddopen** routine must first ensure exclusive access to the device, if necessary. Many character devices, such as printers and plotters, should be opened by only one process at a time. The **ddopen** routine can enforce this by maintaining a static flag variable, which is set to **1** if the device is open and **0** if not. Each time the **ddopen** routine is called, it checks the value of the flag. If the value is other than zero, the **ddopen** routine returns with a return code of **EBUSY** to indicate that the device is already open. Otherwise, the routine sets the flag and returns normally. The **ddclose** entry point later clears the flag when the device is closed. Since most block devices can be used by several processes at once, a block driver should not try to enforce opening by a single user.

The **ddopen** routine must initialize the device if this is the first open that has occurred. Initialization involves the following steps:

- The **ddopen** routine should allocate the required system resources to the device, such as DMA channels, interrupt levels, and priorities. It should, if necessary, register its device interrupt handler for the interrupt level required to support the target device. The **i_init** and **d_init** kernel services are available for initializing these resources.
- If this device driver is providing the *head role* for a device and another device driver is providing the *handler role*, the **ddopen** routine should open the device handler by using the **fp_opendev** kernel service.

Note: The **fp_opendev** kernel service requires a **devno** parameter to identify which device handler to open. This **devno** value, taken from the appropriate DDS, should have been stored in a special save area when the device driver's **ddconfig** routine was called.

The flag word **devflag** has the following flags, as defined in the **<sys/device.h>** header file:

DKERNEL Entry point called by kernel routine using the **fp_opendev** or **fp_open** kernel service.

DREAD Open for reading.

DWRITE Open for writing.

DAPPEND Open for appending.

DNDELAY Device open in non-blocking mode.

The **ddopen** entry point can indicate an error condition to the user mode application program by returning a nonzero return code. Returning a nonzero return code causes the **open** or **creat** subroutines to return a value of **-1** and makes the return code available to the usermode application in the **errno** external variable. The return code used should be one of the values defined in the **<sys/errno.h>** header file.

If a nonzero return code is returned by the **ddopen** routine, the **open** request is considered to have failed. No access to the device instance is available to the caller as a result. In addition, for non-multiplexed drivers, if the failed open was the first open of the device instance, the kernel calls the driver's **ddclose** entry point to allow resources and device driver state to be cleaned up. If the driver was multiplexed, the kernel does not call the **ddclose** entry point on an open failure. When applicable, the return values defined in the *POSIX 1003.1* standard for the **open** subroutine should be used.

```
 1
 2  /*************************************************************************
 3   *
 4   *      ricopen sets up the interrupt and dma services, as well as
 5   *      checking that everything is in order for an open to occur
 6   *
 7   *************************************************************************/
 8  ricopen(devno, devflag, mpxchan, ext_ptr)
 9  dev_t devno;
10  ulong devflag;
11  int mpxchan;
12  struct kopen_ext *ext_ptr;
13  {
14          int     ricintr();      /* interrupt handler    */
15          int     ricoffl();      /* offlevel     */
16          int port_num;           /* port number  */
17          int adapt_num;          /* adapter number */
18          int ilev;               /* adapter interrupt level      */
19          int old_pri;            /* interrupt level      */
20          int counter;            /* loop control counter */
21          struct intr     *intr_ptr;/* interrupt pointer   */
22          t_sel_que       *sqelm1_ptr;/* select queue element pointer     */
23          t_sel_que       *sqelm2_ptr;/* select queue element pointer     */
24          t_chan_info     *tmp_chnptr;/* temp channel info pointer        */
25          t_ric_dds       *dds_ptr;/* pointer to DDS      */
26          t_acb           *acb_ptr;/* pointer to ACB      */
27          int ret;                /* return values        */
28          unsigned long   bus_sr; /* IO Seg Reg number mask */
29          unsigned char   io_ptr; /* io base pointer      */
30          unsigned char   comreg; /* COMREG on Portmaster */
31
32        /* if minor number is bad, return  */
33          if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
34          {
35                  return(EINVAL);
36          }
37
38        /* if the channel number out of range, return */
39        /* Note that we are not really a multiplexed device  */
40          if ( mpxchan != 0 )
41          {
42                  return(ECHRNG);
43          }
44
45        /* get dds pointer from dds directory */
46          dds_ptr = dds_dir[minor(devno)];
47
48        /* if port not configured, return error */
49          if (dds_ptr == NULL)
50          {
51                  return(EINVAL);
52          }
53
54          adapt_num = dds_ptr->dds_hdw.slot_num;
55          acb_ptr = acb_dir[adapt_num];
56
57          port_num = dds_ptr->dds_dvc.port_num;
```

Figure 4-10 (Part 1 of 3). Code Sample of the ricopen Routine

```
 58
 59        /* check to see whether any ports have been opened on
 60         * the indicated adapter.  If not, register the
 61         * interrupt handler and fill in the off level
 62         * interrupt structures.
 63         */
 64        /* no registration has occured for this adapter */
 65           if(acb_ptr->n_open_ports == 0)
 66           {
 67
 68                /* first initialise the offlevel intr structures */
 69                   acb_ptr->arq_sched = FALSE;
 70                   acb_ptr->offl.p_acb_intr = (struct t_acb *)acb_ptr;
 71                   intr_ptr = &(acb_ptr->offl.offl_intr);
 72                   INIT_OFFL3(intr_ptr, ricoffl, IO_SEG_REG);
 73
 74                   acb_ptr->slih_intr.next = NULL;
 75                   acb_ptr->slih_intr.handler = ricintr;
 76                   acb_ptr->slih_intr.bus_type = BUS_MICRO_CHANNEL;
 77                   acb_ptr->slih_intr.flags = 0;
 78                   acb_ptr->slih_intr.level = acb_ptr->int_lvl;
 79                   acb_ptr->slih_intr.priority = INTCLASS1;
 80                   acb_ptr->slih_intr.bid = IO_SEG_REG;
 81
 82                   acb_ptr->cmd_queue_lock = LOCK_AVAIL;
 83
 84                 /* registration of interrupt handler fails */
 85                   if((ret = i_init(&acb_ptr->slih_intr)) != 0)
 86                   {
 87                           return(ENXIO);
 88                   }
 89
 90
 91               /* enable interrupts on the adapter */
 92                   bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val, 0);
 93
 94                   io_ptr = (unsigned char *)( acb_ptr->io_base + bus_sr );
 95
 96                   comreg = PIO_GETC( io_ptr + COMREG );
 97
 98                   PIO_PUTC( io_ptr + COMREG, comreg | COM_IE );
 99
100                   BUSIO_DET( bus_sr );
101
102           } /* end of no open ports loop  */
103
104         /* first time through successfully, allocate channel structure */
105           if(dds_ptr->dds_wrk.p_chan_info[mpxchan] == NULL)
106           {
107                 /* allocate memory for channel related structures */
108                   dds_ptr->dds_wrk.p_chan_info[mpxchan] = tmp_chnptr =
109                       (t_chan_info *)xmalloc((uint)sizeof(t_chan_info),(uint)2,
110                                             pinned_heap);
111
112                 /* memory allocation failed, return */
113                   if( tmp_chnptr == NULL)
114                   {
115                           return(ENOMEM);
116                   }
117
118                   bzero((void *)tmp_chnptr, (uint)sizeof(t_chan_info));
```

Figure 4-10 (Part 2 of 3). Code Sample of the ricopen Routine

```
119
120                 /* set major/minor device number */
121                    tmp_chnptr->devno = devno;
122                    tmp_chnptr->rcv_event_lst = EVENT_NULL;
123                    tmp_chnptr->xmt_event_lst = EVENT_NULL;
124                    acb_ptr->txfl_event_lst = EVENT_NULL;
125
126            }
127
128         /* now fetch the temporary channel info pointer */
129            tmp_chnptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
130
131         /* set common values for user and kernel llc calls */
132            tmp_chnptr->devflag = devflag; /* device flags opened with */
133
134        /* set port state variable to open */
135            dds_ptr->dds_dvc.port_state = OPEN;
136
137        /* increment number of open ports */
138            acb_ptr->n_open_ports++;
139
140            return(0);
141    } /* end ricopen */
142
```

Figure 4-10 (Part 3 of 3). Code Sample of the ricopen Routine

4.1.4 ddclose Device Driver Entry Point

Figure 4-11 on page 4-25 shows the device driver ddclose entry point.

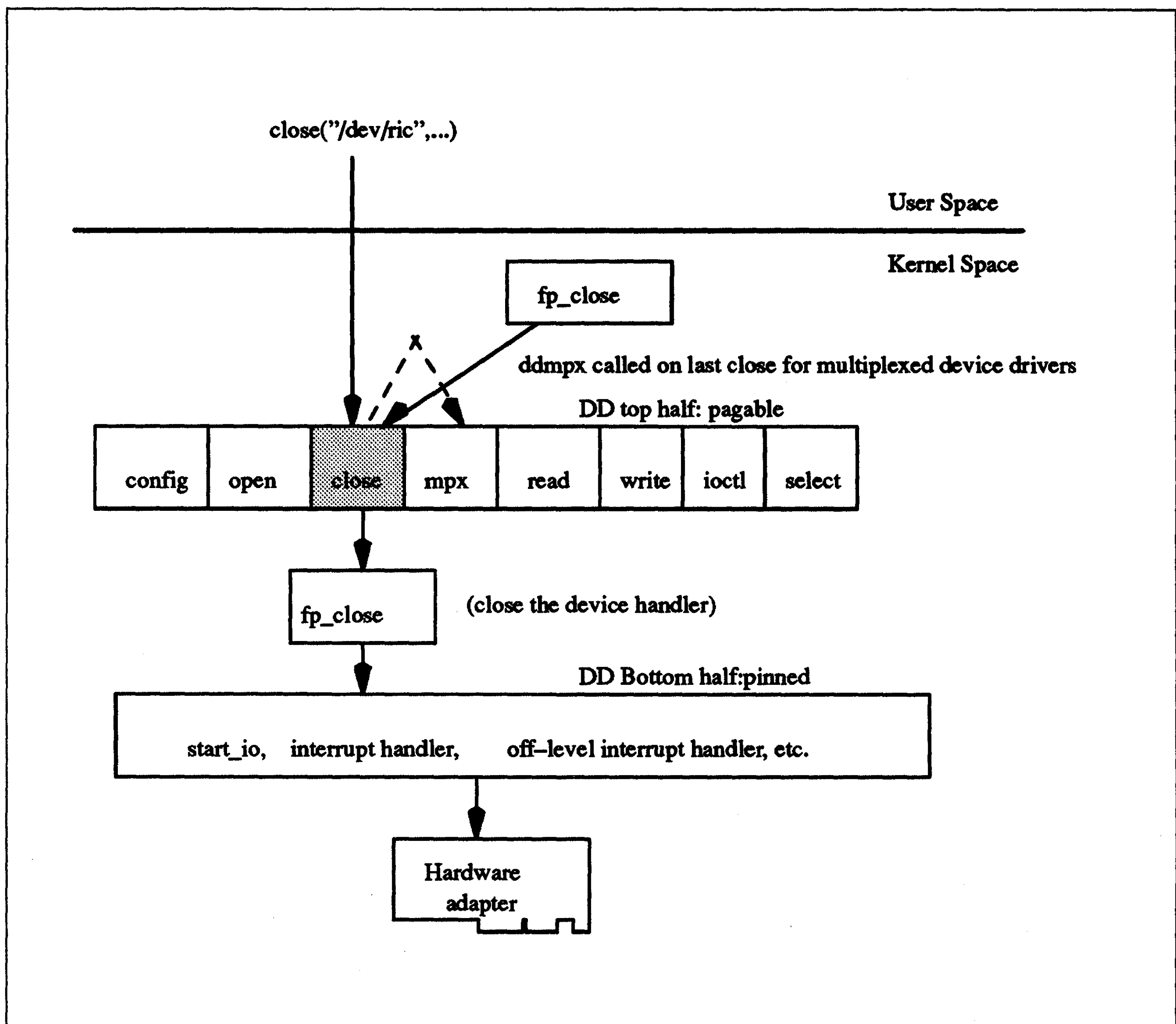


Figure 4-11. Device Driver ddclose Entry Point

The **ddclose** device driver entry routine closes a previously open device instance. (A device instance is a specific port on a communications adapter or a hard disk on a SCSI adapter, etc.) The **ddclose** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver. Refer to Figure 4-13 on page 4-28 for the **ricclose** sample code.

The **ddclose** routine expects two parameters. These are **devno** and **chan**, where:

devno Specifies the major and minor device numbers of the device instance to close.

chan Specifies the channel number (for multiplexed devices only).

The following example shows the syntax of a **ddclose** entry point.

```
#include <sys/device.h>
#include <sys/types.h>

int ddclose (devno, chan)
dev_t devno;
chan_t chan;
```

Please refer to Figure 4-12 for ddclose program flow for multiplexed and non-multiplexed device drivers.

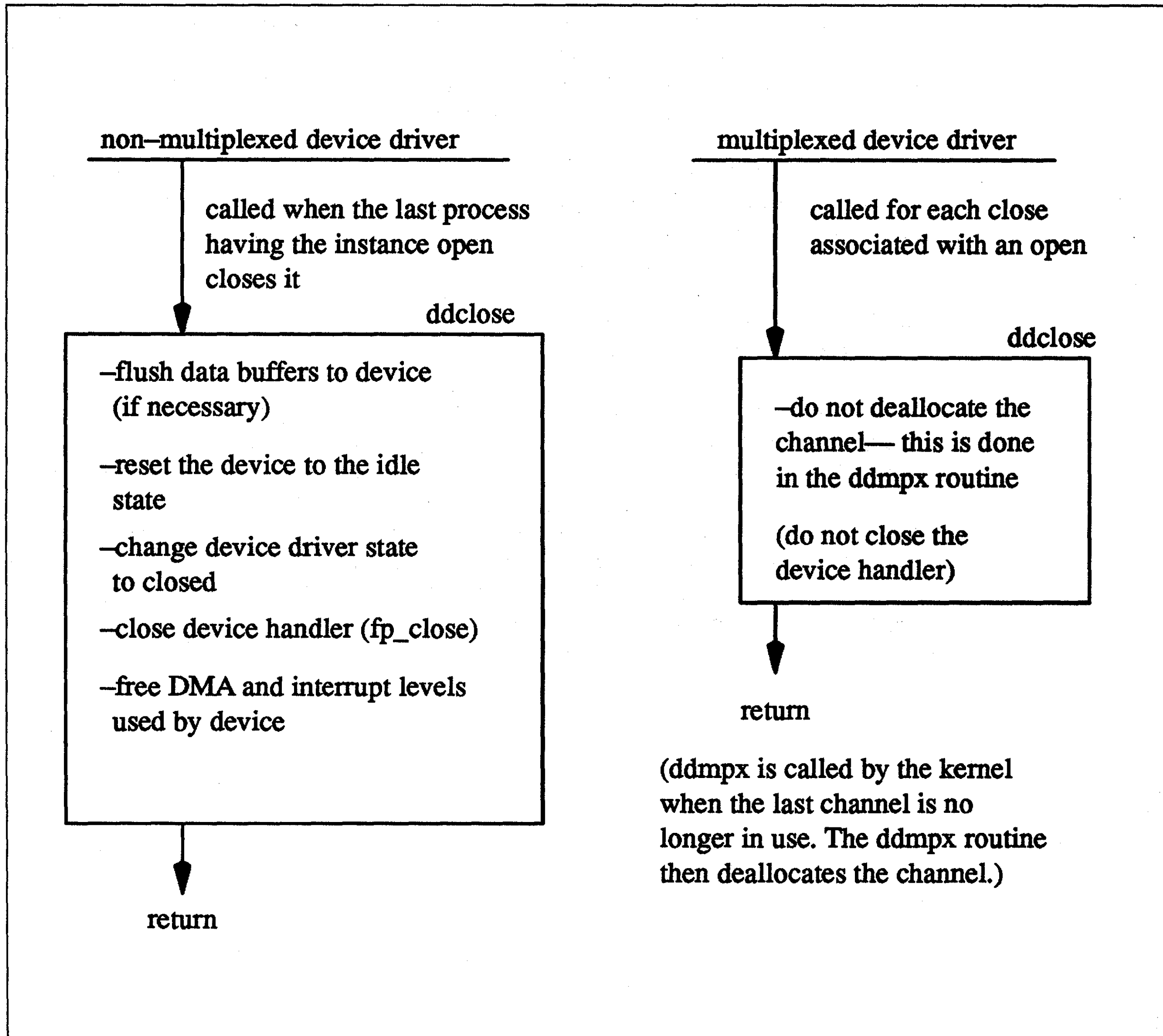


Figure 4-12. Device Driver ddclose Program Flow

The **ddclose** entry point is called when a previously opened device instance is closed by the **close** subroutine or **fp_close** kernel service. The kernel calls the routine under different circumstances for non-multiplexed and multiplexed device drivers. For non-multiplexed device drivers, the **ddclose** routine is called by the kernel when the last process having the device instance open closes it. This causes the **gnode** reference count to be decremented to **0**, and the **gnode** to be deallocated. For multiplexed device drivers, the **ddclose** routine is called for each close associated with an explicit open. In other

words, the device driver's **ddclose** routine is invoked once for each time its **ddopen** routine was invoked for the channel.

In some instances, data buffers should be written to the device before returning from the **ddclose** routine. These are buffers containing data to be written to the device that have been queued by the device driver but not yet written.

Non-multiplexed device drivers should reset the associated device to an idle state and change the device driver state to closed. This can involve calling the **fp_close** kernel service to issue a **close to an associated open device handler** for the device. Returning the device to an idle state prevents the device from generating any more interrupts or DMA requests. DMA channels and interrupt levels allocated for this device should be freed until the device is re-opened, to release limited system resources used by this device.

Multiplexed device drivers should provide the same device quiescing, but not in the **ddclose** routine. Returning the device to the idle state and freeing its resources should be delayed until the **ddmpx** routine is called to deallocate the last channel allocated on the device.

In all cases, the device instance is considered closed once the **ddclose** routine has returned to the caller, even if a nonzero return code is returned.

The **ddclose** entry point can indicate an error condition to the user mode application program by returning a nonzero return code. This causes the subroutine call to return a value of **-1**. It also makes the return code available to the user mode application in the **errno** external variable. The return code used should be one of the values defined in the **<sys/errno.h>** header file. The device is always considered closed even if a nonzero return code is returned. When applicable, the return values defined in the *POSIX 1003.1* standard for the **close** subroutine should be used.

```
 1
 2  /***************************************************************************
 3   *
 4   *       ricclose closes a single port.
 5   *
 6   ***************************************************************************/
 7  ricclose(devno, mpxchan, ext)
 8  dev_t devno;
 9  int mpxchan;
10  int ext;
11  {
12          int             adapt_num;      /* adapter number */
13          int             port_num;       /* port number  */
14          t_acb           *acb_ptr;         /* pointer to ACB */
15          t_chan_info     *tmp_chanptr;  /* temp channel info pointer   */
16          t_ric_dds       *dds_ptr;         /* pointer to DDS */
17          unsigned int    ret;             /* return values */
18          int             old_pri;        /* interrupt level      */
19          unsigned long   bus_sr;         /* bus segment reg      */
20          unsigned char   *io_ptr;          /* pointer to io reg    */
21          unsigned char   comreg;         /* COMREG on ric     */
22          unsigned int    sleep_flag;     /* que_cmd sleep flag   */
23
24       /* if minor number is invalid, return error */
25          if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
26          {
27                  return(EINVAL);
28          }
29
30       /* if the channel number out of range, return */
31          if ( mpxchan != 0 )
32          {
33                  return(ECHRNG);
34          }
35
36       /* get dds pointer from dds directory */
37          dds_ptr = dds_dir[minor(devno)];
38
39       /* if port not configured, return error */
40          if (dds_ptr == NULL)
41          {
42                  return(EINVAL);
43          }
44
45          adapt_num = dds_ptr->dds_hdw.slot_num;
46          acb_ptr = acb_dir[adapt_num];
47
48          port_num = dds_ptr->dds_dvc.port_num;
49
50       /*   remove the select queue data structure, the channel
51        *   information data structure and zero out the dds pointer
52        *   to the channel ds
53        */
54
55          tmp_chanptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
56
57       /* remove device flags */
58          tmp_chanptr->devflag = 0;
```

Figure 4-13 (Part 1 of 2). Code Sample of the ricclose Routine

```
59
60        /* last close for this adapter. notify kernel the adapter
61         *is no longer generating interrupts
62         */
63          if (--acb_ptr->n_open_ports == 0)
64          {
65               /* First disable interrupts from the adapter. */
66                  bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val,0);
67
68                  io_ptr = (unsigned char *)( acb_ptr->io_base + bus_sr);
69
70                  comreg = PIO_GETC( io_ptr + COMREG );
71
72                  PIO_PUTC(io_ptr + COMREG, comreg &  COM_IE );
73
74                  BUSIO_DET(bus_sr);
75
76                  i_clear(&acb_ptr->slih_intr);
77          }
78
79        /* set port state to closed */
80          dds_ptr->dds_dvc.port_state = CLOSED;
81
82          return(0);
83  } /* end ricclose */
84
```

Figure 4-13 (Part 2 of 2). Code Sample of the ricclose Routine

4.1.5 ddread Device Driver Entry Point

Figure 4-14 on page 4-30 shows the ddread entry point.

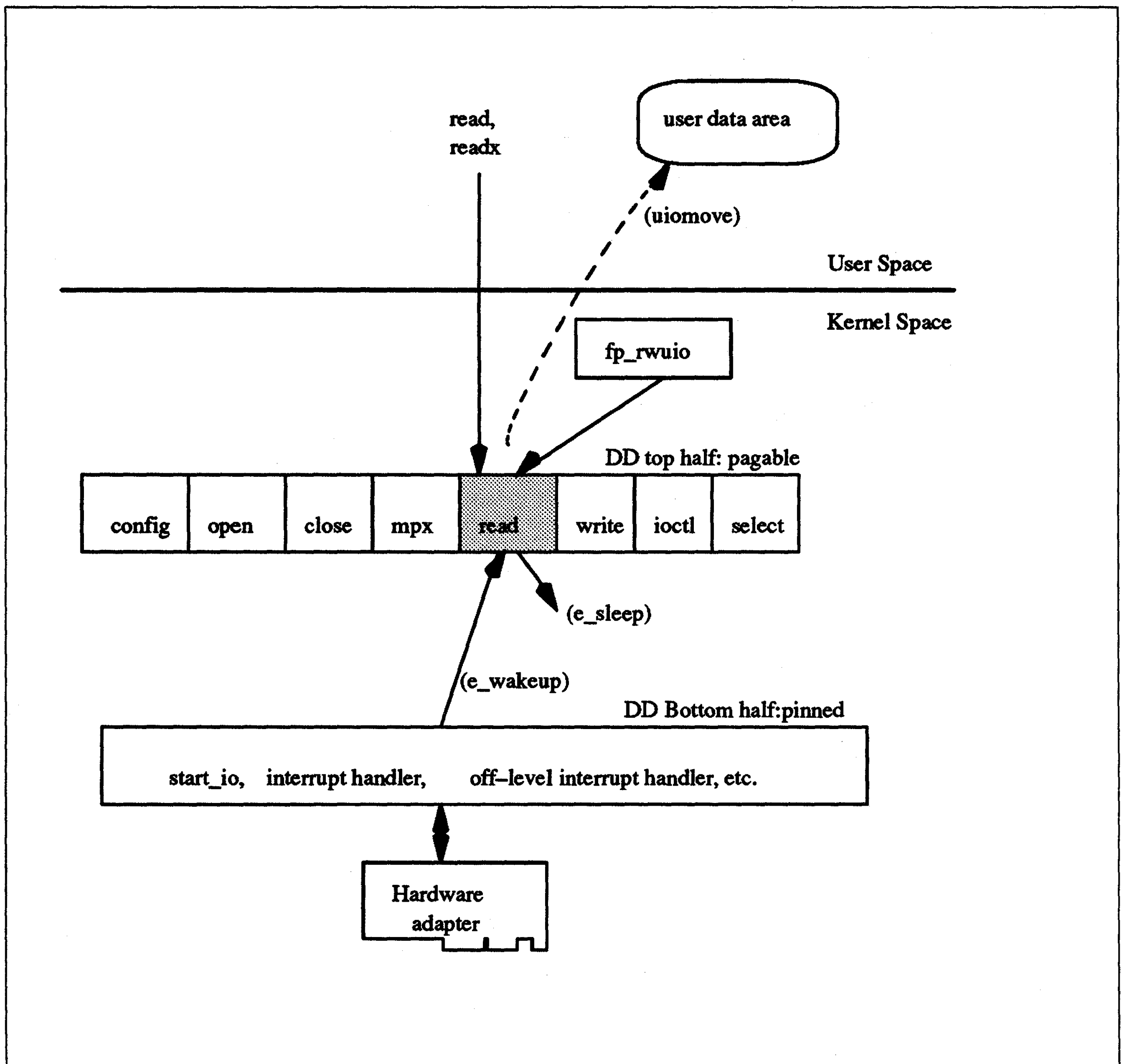


Figure 4-14. Device Driver ddread Entry Point

The **ddread** device driver entry point reads in data from a character device. The **ddread** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver. Refer to Figure 4-15 on page 4-32 for the **ricread** sample code.

The **ddread** routine expects four parameters. These are **devno**, **uiop**, **chan**, and **ext**, where:

devno Specifies the major and minor device numbers.

uiop Points to a uio structure describing the data area or areas to be written into.

chan Specifies the channel number.

ext Specifies the extension parameter.

The following example shows the syntax of a **ddread** entry point.

```
#include <sys/device.h>
#include <sys/types.h>

int ddread (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
chan_t chan;
int  ext;
```

When a program issues a **read** or **readx** subroutine call or when the **fp_rwuio** kernel service is used, the kernel calls the **ddread** entry point. This entry point receives a pointer to a **uio** structure that provides variables used to specify the data transfer operation. Character device drivers can use the **ureadc** and **uiomove** kernel services to transfer data into and out of the user buffer area during a read subroutine call. These services receive a pointer to the **uio** structure and update the fields in the structure by the number of bytes transferred. The only fields in the **uio** structure that cannot be modified by the data transfer are the **uio_fmode** and **uio_segflg** fields.

For most devices, the **ddread** routine sends the request to the *device handler* and then waits for it to finish. The waiting can be accomplished by calling the **e_sleep** kernel service. This service suspends the driver and the process that called it and permits other processes to run until a specified event occurs.

When the I/O operation completes, the device usually issues an **interrupt**, causing the device driver's *interrupt handler* to be called. The **interrupt handler** then calls the **e_wakeup** kernel service specifying the awaited event, **thus** allowing the **ddread** routine to resume.

The **uio_resid** field initially contains the total number of bytes to read from the device. If the device driver supports it, the **uio_offset** field indicates the byte offset on the device from which point the read should start. If no error occurs, the **uio_resid** field should be **0** on return from the **ddread** routine to indicate that all requested bytes were read. If an error occurs, this field should contain the number of bytes remaining to be read when the error occurred.

If a read request starts at a valid device offset but extends past the end of the device's capabilities, no error should be returned. However, the **uio_resid** field should indicate the number of bytes not transferred. If the read starts at the end of the device's capabilities, no error should be returned. However, the **uio_resid** field should not be modified, indicating that no bytes were transferred. If the read starts past the end of the device's capabilities, an **ENXIO** return code should be returned, without modifying the uio_resid field.

When the **ddread** entry point is provided for raw I/O to a block device, this routine usually translates requests into block I/O requests using the **uphysio** kernel service.

The **ddread** entry point can indicate an error condition to the caller by returning a nonzero return code. This causes the subroutine call to return a value of **-1**. It also makes the return code available to the user mode program in the **errno** external variable. The error code used should be one of the values defined in the **<sys/errno.h>** header file. When applicable, the return values defined in the *POSIX 1003.1* standard for the **read** subroutine should be used.

```
 1
 2  /***************************************************************************
 3   *
 4   *       ricread reads the adapter
 5   *
 6   ***************************************************************************/
 7  ricread(devno, uiop, mpxchan, rdext_ptr)
 8  dev_t devno;
 9  struct uio  *uiop;
10  int mpxchan;
11  struct read_extension  *rdext_ptr;
12  {
13          int  adapt_num;      /* adapter number  */
14          int  port_num;       /* port number  */
15          int  old_pri;                 /* interrupt level */
16          u_short         pkt_hdr_len;    /* packet header length        */
17          u_short         pkt_length;     /* receive data length */
18          u_short         pkt_status;     /* receive packet status */
19          t_acb           *acb_ptr;         /* pointer to ACB */
20          t_ric_dds      *dds_ptr;         /* pointer to DDS */
21          struct mbuf     *mbuf_ptr;        /* pointer to  mbuf */
22          caddr_t         p_pkt;          /* pointer to the received packet */
23          u_short         *p_shrt_pkt;    /* pointer to the received packet */
24          t_sel_que       *p_rcv_elem;    /* pointer to the receive entry */
25          volatile t_chan_info *tmp_chnptr; /* temp channel info pointer */
26          int             ret;             /* return code */
27          int             sleep_ret;       /* return code from e_sleep */
28
29      /* if minor number is invalid, return error */
30          if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
31          {
32                  return(EINVAL);
33          }
34
35      /* if the channel number out of range (only 0 is valid for now) */
36          if ( mpxchan != 0 )
37          {
38                  return (ECHRNG);
39          }
40
41      /* get dds pointer from dds directory */
42          dds_ptr = dds_dir[minor(devno)];
43
44      /* if port not configured, return error */
45          if (dds_ptr == NULL)
46          {
47                  return(ENXIO);
48          }
49
50          adapt_num = dds_ptr->dds_hdw.slot_num;
51          acb_ptr = acb_dir[adapt_num];
52
53          port_num = dds_ptr->dds_dvc.port_num;
54
55          /*
56           *  go get the channel information data struct pointer from
57           *  the DDS.
58           */
59          tmp_chnptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
```

Figure 4-15 (Part 1 of 3). Code Sample of the ricread Routine

```
 60
 61        /* disable interrupts to single thread */
 62           old_pri = i_disable(INTOFFL3);
 63
 64        /* no packets are available on the queue */
 65           while( tmp_chnptr->p_rcv_head == NULL)
 66           {
 67               /* DNDELAY set, return at once */
 68                   if( tmp_chnptr->devflag & DNDELAY )
 69                   {
 70                       /* end single thread */
 71                           i_enable(old_pri);
 72
 73                       /* set length to zero */
 74                           uiop->uio_resid = 0;
 75
 76                       /* no data, return zero */
 77                           return(0);
 78                   }
 79                   else
 80             /* NDELAY not set, wait until data is received */
 81                   {
 82                       /* do an e_sleep */
 83                           sleep_ret = e_sleep(&(tmp_chnptr->rcv_event_lst),
 84                                                  EVENT_SIGRET);
 85
 86                           if ( sleep_ret != EVENT_SUCC )
 87                           {
 88                                   i_enable( old_pri );
 89                                   return( EINTR );
 90                           }
 91                   }
 92           }
 93        /*
 94         * message waiting. deque it and copy to user's buffer
 95         */
 96        /* point to first element */
 97           p_rcv_elem = tmp_chnptr->p_rcv_head;
 98
 99        /* copy the code field to the status field of read extension */
100           if ( rdext_ptr != NULL)
101           {
102                   rdext_ptr->status = (ulong) p_rcv_elem->stat_block.code;
103           }
104
105           tmp_chnptr->p_rcv_head = p_rcv_elem->p_sel_que; /* deque it */
106
107        /* get mbuf pointer */
108           mbuf_ptr = (struct mbuf *)p_rcv_elem->stat_block.option[0];
109
110        /* receive head ptr is null, make receive tail ptr null */
111           if(tmp_chnptr->p_rcv_head == NULL)
112           {
113                   tmp_chnptr->p_rcv_tail = NULL;
114           }
```

Figure 4-15 (Part 2 of 3). Code Sample of the ricread Routine

```
115
116        /*
117         * zero out the select queue element and add it back
118         * to the select queue available chain
119         */
120           p_rcv_elem->rqe_value = 0;
121           p_rcv_elem->stat_block.code = 0;
122           p_rcv_elem->stat_block.option[0] = 0;
123           p_rcv_elem->p_sel_que = tmp_chnptr->p_sel_avail;
124           tmp_chnptr->p_sel_avail = p_rcv_elem;
125
126           i_enable(old_pri);
127
128        /* if mbuf_ptr is NULL, there is a status, not a receive buffer */
129           if (mbuf_ptr ==  NULL)
130           {
131                   return (0);
132           }
133
134        /* get buffer address */
135           p_pkt = MTOD(mbuf_ptr, caddr_t);
136
137           p_shrt_pkt = (u_short *)p_pkt;
138
139        /* get information from packet header */
140           pkt_hdr_len = PIO_GETSR(p_shrt_pkt++);
141           pkt_length = PIO_GETSR(p_shrt_pkt++);
142           pkt_status = PIO_GETSR(p_shrt_pkt);
143
144        /* point packet address to start past header */
145           p_pkt = p_pkt + pkt_hdr_len;
146
147        /* attempt to move the packet contents to the user area */
148           ret = uiomove(p_pkt, (unsigned int)pkt_length, UIO_READ, uiop);
149
150        /* free the mbuf */
151           m_free( mbuf_ptr );
152
153           return(ret);
154
155    }  /* end ricread */
156
```

Figure 4-15 (Part 3 of 3). Code Sample of the ricread Routine

4.1.6 ddwrite Device Driver Entry Point

Figure 4-16 on page 4-35 shows the ddwrite entry point.

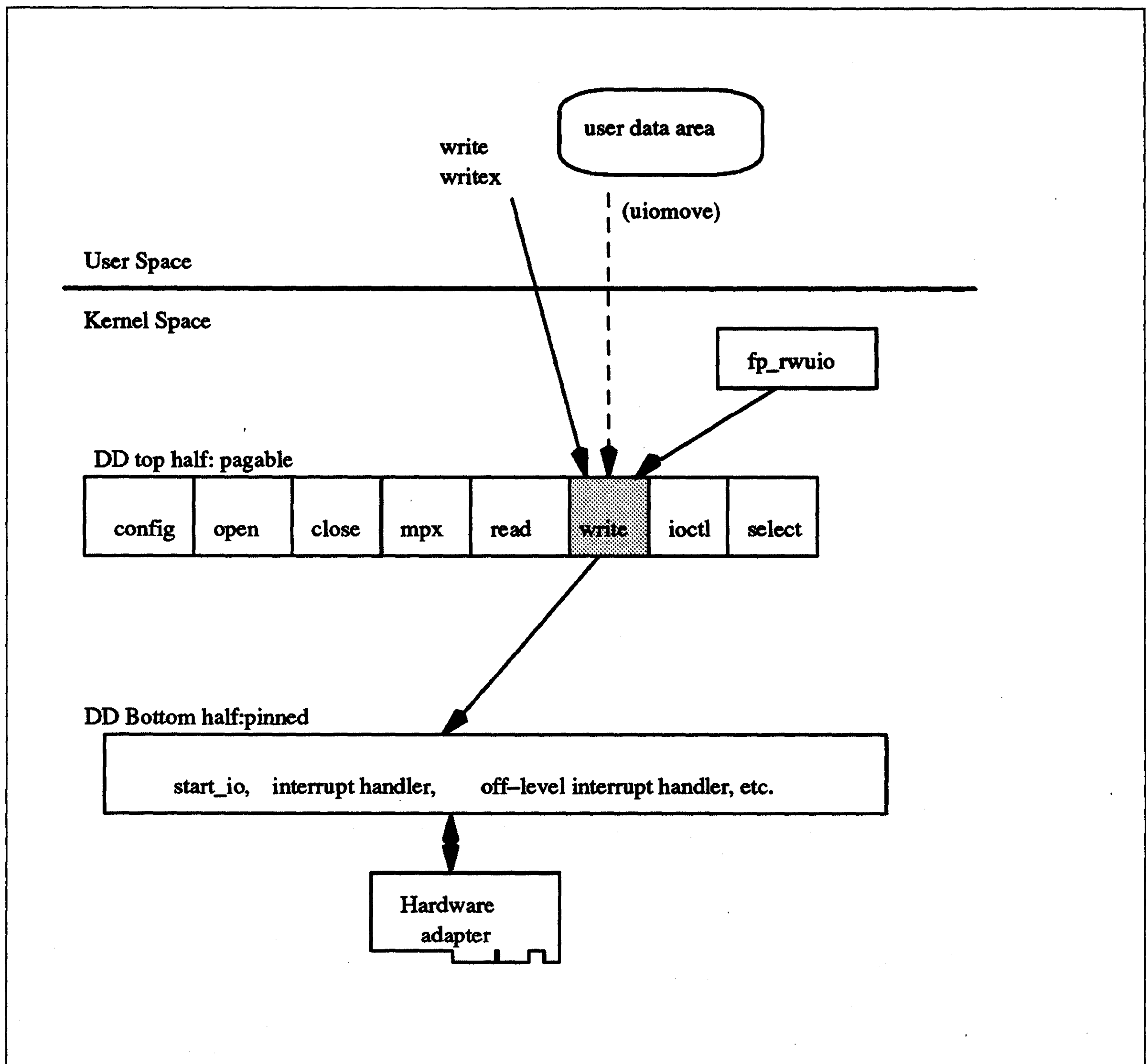


Figure 4-16. Device Driver ddwrite Entry Point

The **ddwrite** device driver entry point writes out data to a character device. The **ddwrite** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver. See Figure 4-17 on page 4-37 for the **ricwrite** sample code.

The **ddwrite** routine expects four parameters. These are **devno**, **uiop**, **chan**, and **ext**, where:

devno Specifies the major and minor device numbers.

uiop Points to a uio structure describing the data area or areas to be written from.

chan Specifies the channel number.

ext Specifies the extension parameter.

The following example shows the syntax of a **ddwrite** entry point.

```
#include <sys/device.h>
#include <sys/types.h>

int ddwrite (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
chan_t chan;
int  ext;
```

When a program issues a **write** or **writex** subroutine call or when the **fp_rwuio** kernel service is used, the kernel calls the **ddwrite** entry point. This entry point receives a pointer to a **uio** structure, which provides variables used to specify the data transfer operation. Character device drivers can use the **uwritec** and **uiomove** kernel services to transfer data into and out of the user buffer area during a **write** subroutine call. These services are passed a pointer to the **uio** structure. They update the fields in the structure by the number of bytes transferred. The only fields in the **uio** structure that are not potentially modified by the data transfer are the **uio_fmode** and **uio_segflg** fields.

For most devices, the **ddwrite** routine queues the request to the device handler and then waits for it to finish. The waiting is typically accomplished by calling the **e_sleep** kernel service to wait for an event. The **e_sleep** service suspends the (top-half) driver and the process that called it, and permits other processes to run.

When the I/O operation is completed, the device usually causes an interrupt, which causes the device driver's *interrupt handler* to be called. The *interrupt handler* then calls the **e_wakeup** kernel service specifying the awaited event, thus allowing the **ddwrite** routine to resume.

The **uio_resid** field initially contains the total number of bytes to write to the device. If the device driver supports it, the **uio_offset** field indicates the byte offset on the device from which point the write should start. If no error occurs, the **uio_resid** field should be **0** on return from the **ddwrite** routine to indicate that all requested bytes were written. If an error occurs, this field should contain the number of bytes remaining to be written when the error occurred.

If a write request starts at a valid device offset but extends past the end of the device's capabilities, no error should be returned. However, the **uio_resid** field should indicate the number of bytes not transferred. If the write starts at or past the end of the device's capabilities, no data should be transferred. An error code of **ENXIO** should be returned, and the **uio_resid** field should not be modified.

When the **ddwrite** entry point is provided for raw I/O to a block device, this routine usually translates requests into block I/O requests using the **uphysio** kernel service.

The **ddwrite** entry point can indicate an error condition to the caller by returning a nonzero return code. This causes the subroutine to return a value of **-1**. It also makes the return code available to the user mode program in the **errno** external variable. The error code used should be one of the values defined in

the <**sys/errno.h**> header file. When applicable, the return values defined in the *POSIX 1003.1* standard for the **write** subroutine should be used.

```
 1
 2  /***************************************************************************
 3   *
 4   *       ricwrite allows write or transmit for user level or kernel
 5   *       level users of the ric.
 6   *
 7   ***************************************************************************/
 8  ricwrite(devno, uiop, mpxchan, ext_ptr, sleep_flag)
 9  dev_t  devno;
10  struct uio *uiop;
11  int mpxchan;
12  t_write_ext *ext_ptr;
13  unsigned int sleep_flag;
14
15  {
16          int  adapt_num;       /* adapter number */
17          int  port_num;        /* port number    */
18          t_acb  *acb_ptr;      /* pointer to ACB */
19          t_ric_dds  *dds_ptr;  /* pointer to DDS */
20          t_write_ext     lc_ext; /* local copy of write extension */
21          int  data_len;        /* total length of chained mbuf */
22          unsigned short  lc_flags;      /* local copy of flag bits */
23          unsigned short  lc_seq_num;
24          unsigned short  lc_xmt_length;
25          char            *lc_bus_buf;
26          char            *lc_bus_base;
27          char            *lc_host_buf;
28          struct mbuf     *lc_xmt_mbuf;
29          unsigned int    old_pri;    /* interrupt priority save element */
30          t_xmt_chain     *xchn_ptr; /* pointer to the xmit chain */
31          t_xmt_map       *xmap_ptr; /* pointer to current xmit map */
32          struct mbuf     *mbuf_ptr;         /* pointer to the mbuf */
33          struct mbuf     *freembuf_ptr;     /* pointer to mbuf to free */
34          struct mbuf     *freembufc_ptr; /* ptr to mbuf chain to free */
35          struct mbuf     *allocmbuf_ptr;    /* mbuf allocated by us */
36          unsigned char   *mbufdata_ptr;    /* pointer to mbuf data to be sent */
37          struct mbuf     *tmpmbuf_ptr;     /* temp pointer to mbuf */
38          int             ret;              /* return code */
39          struct xmem     xmd;             /* cross memory descriptor for dma */
40          t_adap_cmd      xmt_adap_cmd;    /* on stack adapter command buffer */
41          unsigned char   tmp_cntrl;       /* temp var for filling in cmd blk */
42
43   /* if minor number is bad, return error */
44      if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
45      {
46          return(EINVAL);
47      }
48
49   /* if the channel number out of range, return */
50      if ( mpxchan != 0 )
51      {
52          return (ECHRNG);
53      }
54
```

Figure 4-17 (Part 1 of 6). Code Sample of the ricwrite Routine

```
 55    /* get dds pointer from dds directory */
 56       dds_ptr = dds_dir[minor(devno)];
 57
 58    /* if port not configured, return error */
 59       if (dds_ptr == NULL)
 60       {
 61           return(EINVAL);
 62       }
 63
 64       adapt_num = dds_ptr->dds_hdw.slot_num;
 65       acb_ptr = acb_dir[adapt_num];
 66
 67       port_num = dds_ptr->dds_dvc.port_num;
 68
 69    /* initialize local mbuf pointers */
 70       freembuf_ptr = NULL;
 71       freembufc_ptr = NULL;
 72       allocmbuf_ptr = NULL;
 73
 74       bzero( ( char *)&xmt_adap_cmd , sizeof(t_adap_cmd));
 75
 76       /*  if write extension provided, copyin if from user space.
 77        *  else copy directly (bcopy) if from kernel space.
 78        */
 79       bzero( &lc_ext, sizeof( t_write_ext ));
 80       if ( ext_ptr )
 81           if ( uiop->uio_segflg == UIO_USERSPACE )
 82               copyin( ext_ptr, &lc_ext, sizeof( t_write_ext ));
 83           else
 84               bcopy ( ext_ptr, &lc_ext, sizeof( t_write_ext ));
 85
 86    /* initialize local flags */
 87       if ( lc_ext.cio_write.flag & CIO_ACK_TX_DONE ) {
 88           lc_flags = XMT_STAT_REQ;
 89       }
 90       else
 91       {
 92           lc_flags = 0;
 93       }
 94
 95    /* get pointer to transmit chain */
 96       xchn_ptr = dds_ptr->dds_wrk.p_xmt_chn;
 97
 98    /* if no available transmit map elements, then return */
 99       if((xchn_ptr->elts_in_use +1 ) >= xchn_ptr->length)
100       {
101           return(EAGAIN);
102       }
103
```

Figure 4-17 (Part 2 of 6). Code Sample of the ricwrite Routine

```
104    /* a user process called the write */
105       if( uiop->uio_segflg == UIO_USERSPACE )
106       {
107           lc_xmt_length = (unsigned int)uiop->uio_resid;
108
109         /* data length is 48 bytes or less */
110           if( lc_xmt_length <= 48 )
111           {
112            /* do uiomove to get data into command block */
113              if((ret = uiomove(&(xmt_adap_cmd.u_data_area.d_ovl.data[0]),
114                             uiop->uio_resid, UIO_WRITE, uiop)) != 0)
115              {
116               /* uiomove failed, return an error */
117                  return(ret);
118              }
119           } /* end of transmit <= 48 bytes */
120           else
121           {
122             /* if request for more than one page, return */
123              if( lc_xmt_length > PAGESIZE )
124              {
125                  return(EINVAL);
126              }
127
128             /* allocate an mbuf and copy the data into it */
129              mbuf_ptr = m_get( M_DONTWAIT, MT_DATA);
130
131              /* if no mbuf available, return */
132              if( mbuf_ptr == (struct mbuf *)NULL )
133              {
134                  return(ENOMEM);
135              }
136
137            /* try to get an mbuf cluster */
138              m_clget(mbuf_ptr);
139
140            /* no mbuf clusters available */
141              if(!M_HASCL(mbuf_ptr))
142              {
143                  m_free(mbuf_ptr);
144                  return(ENOMEM);
145              }
146
147            /* save pointer to mbuf */
148              allocmbuf_ptr = mbuf_ptr;
149
150              /* set local flags */
151              lc_flags |= (XMT_FREE_MBUF |        /* mbuf to be freed */
152                           XMT_DMA_REQ);          /* will be doing dma */
153
154            /* now get a pointer to the actual data */
155              mbufdata_ptr = MTOD(mbuf_ptr, char *);
```
Figure 4-17 (Part 3 of 6). Code Sample of the ricwrite Routine

```
156
157            /* now do uiomove to get data into mbuf or mbuf extension */
158             if((ret = uiomove(mbufdata_ptr, uiop->uio_resid, UIO_WRITE,
159                               uiop)) != 0)
160              {
161               /* uiomove failed, free the mbuf and return  */
162                   m_free(mbuf_ptr);
163                   return(ret);
164               }
165          }
166      }
167
168      if (lc_ext.transparent)
169              tmp_cntrl = (ADAP_TX_ACK | ADAP_TRANSP);
170      else
171              tmp_cntrl = ADAP_TX_ACK;
172
173      lc_seq_num = ++dds_ptr->dds_wrk.cmd_seq_num;
174
175   /* need to do a DMA */
176      if(lc_flags & XMT_DMA_REQ)
177      {
178       /* will be doing a XMIT_LONG command */
179
180       /* already running max number of dma's */
181          if(xchn_ptr->num_active_dma >= XMT_TCWS_PORT)
182          {
183              if (allocmbuf_ptr)
184                  m_free(allocmbuf_ptr);
185              return( EAGAIN );
186          }
187
188          lc_xmt_mbuf = mbuf_ptr;
189          lc_host_buf = MTOD(mbuf_ptr, char *);
190          lc_bus_base = reg_alloc ( dds_ptr->dds_wrk.p_reg_list, PAGESIZE);
191          lc_bus_buf =lc_bus_base + ((unsigned int)lc_host_buf % PAGESIZE);
192
193          /* make the buffer visible to the adapter */
194          xmd.aspace_id = XMEM_GLOBAL;
195          xmd.subspace_id = NULL;
196          d_master(acb_ptr->dma_channel_id, DMA_WRITE_ONLY, lc_host_buf,
197                  lc_xmt_length, &xmd, lc_bus_buf);
198
199          /* fill in command block */
200          xmt_adap_cmd.cmd_typ = XMIT_LONG;
201          xmt_adap_cmd.port_nmbr = (unsigned char)port_num;
202          xmt_adap_cmd.seq_num = SWAPSHORT(lc_seq_num);
203          xmt_adap_cmd.u_data_area.c_ovl.tst_length =
204                                          SWAPSHORT(lc_xmt_length);
205          xmt_adap_cmd.u_data_area.c_ovl.tst_addr =
206                                          SWAPLONG((unsigned int)lc_bus_buf);
207          xmt_adap_cmd.u_data_area.c_ovl.cntl = tmp_cntrl;
208      }
```

Figure 4-17 (Part 4 of 6). Code Sample of the ricwrite Routine

```
209        else
210        {
211         /* will be doing a XMIT_SHORT command */
212            lc_xmt_mbuf = NULL;
213            lc_host_buf = NULL;
214            lc_bus_base = NULL;
215            lc_bus_buf = NULL;
216
217         /* fill in command block */
218            xmt_adap_cmd.cmd_typ = XMIT_SHORT;
219            xmt_adap_cmd.port_nmbr = (unsigned char)port_num;
220            xmt_adap_cmd.seq_num = SWAPSHORT(lc_seq_num);
221            xmt_adap_cmd.lngth = (unsigned char)lc_xmt_length;
222            xmt_adap_cmd.cntrl = tmp_cntrl;
223        }
224
225     /* get pointer to next available transmit map element */
226        xmap_ptr = &(xchn_ptr->xmt_map_chn[(int)xchn_ptr->tail]);
227
228     /* fill it in */
229        xmap_ptr->seq_num = lc_seq_num;
230        xmap_ptr->xmt_elem_flags = lc_flags;
231        xmap_ptr->xmt_length = lc_xmt_length;
232        xmap_ptr->write_id = lc_ext.cio_write.write_id;
233        xmap_ptr->p_xmt_mbuf = lc_xmt_mbuf;
234        xmap_ptr->p_host_buf = lc_host_buf;
235        xmap_ptr->p_bus_base = lc_bus_base;
236        xmap_ptr->p_bus_buf = lc_bus_buf;
237
238      /* send the command down */
239        old_pri = i_disable(INTOFFL3);
240
241      /* if unable to get available command block, return  */
242        if((ret = que_command ( acb_ptr, &xmt_adap_cmd, sleep_flag)) < 0)
243        {
244            i_enable(old_pri);
245        /* have d_mastered stuff here, d_complete it */
246            if( lc_flags & XMT_DMA_REQ )
247            {
248             /* d_complete the transmit information */
249                xmd.aspace_id = XMEM_GLOBAL;
250                xmd.subspace_id = NULL;
251                ret = d_complete(acb_ptr->dma_channel_id, 0, lc_host_buf,
252                                    lc_xmt_length, &xmd, lc_bus_buf);
253            }
254
255         /* free any mbuf allocated in this routine */
256            if (allocmbuf_ptr)
257                    m_free(allocmbuf_ptr);
258
259            return(EAGAIN);
260        } /* cmd queued to adapter */
```

Figure 4-17 (Part 5 of 6). Code Sample of the ricwrite Routine

```
261
262        /* successfully started transmit */
263
264      /* increment number of outstanding active dma's */
265        if (lc_flags & XMT_DMA_REQ)
266            xchn_ptr->num_active_dma++;
267
268      /* incrment transmit map tail pointer */
269        xchn_ptr->elts_in_use++;
270        xchn_ptr->tail = (xchn_ptr->tail + 1) % XMT_CHN_ELEM;
271
272        i_enable(old_pri);
273
274      /* free any LLC mbufs that can be freed now */
275        if (freembufc_ptr)
276            m_free(freembufc_ptr);
277        if (freembuf_ptr)
278            m_free(freembuf_ptr);
279
280        /* accumulate the transmit stats here, and have a nice day ! */
281            DDS_STAT.tx_port_cnt++;
282            if (ULONG_MAX - xmt_adap_cmd.lngth < DDS_STAT.tx_byte_lcnt)
283            {
284                    DDS_STAT.tx_byte_mcnt++;
285                    DDS_STAT.tx_byte_lcnt =
286                            ULONG_MAX - DDS_STAT.tx_byte_lcnt;
287                    DDS_STAT.tx_byte_lcnt =
288                            xmt_adap_cmd.lngth - DDS_STAT.tx_byte_lcnt;
289            }
290            else
291            {
292                    DDS_STAT.tx_byte_lcnt += xmt_adap_cmd.lngth;
293            }
294            if (xmt_adap_cmd.cmd_typ == XMIT_SHORT)
295            {
296                    DDS_STAT.tx_short++;
297                    DDS_STAT.tx_shortbytes += xmt_adap_cmd.lngth;
298            }
299            else
300                    if ((xmt_adap_cmd.cmd_typ == XMIT_LONG) ||
301                        (xmt_adap_cmd.cmd_typ == XMIT_GATHER))
302                    {
303                            DDS_STAT.tx_dma++;
304                            DDS_STAT.tx_dmabytes += xmt_adap_cmd.lngth;
305                    }
306
307        return(0);
308    }  /* end ricwrite */
309
```

Figure 4-17 (Part 6 of 6). Code Sample of the ricwrite Routine

4.1.7 ddioctl Device Driver Entry Point

Figure 4-18 on page 4-43 shows the ddioctl entry point.

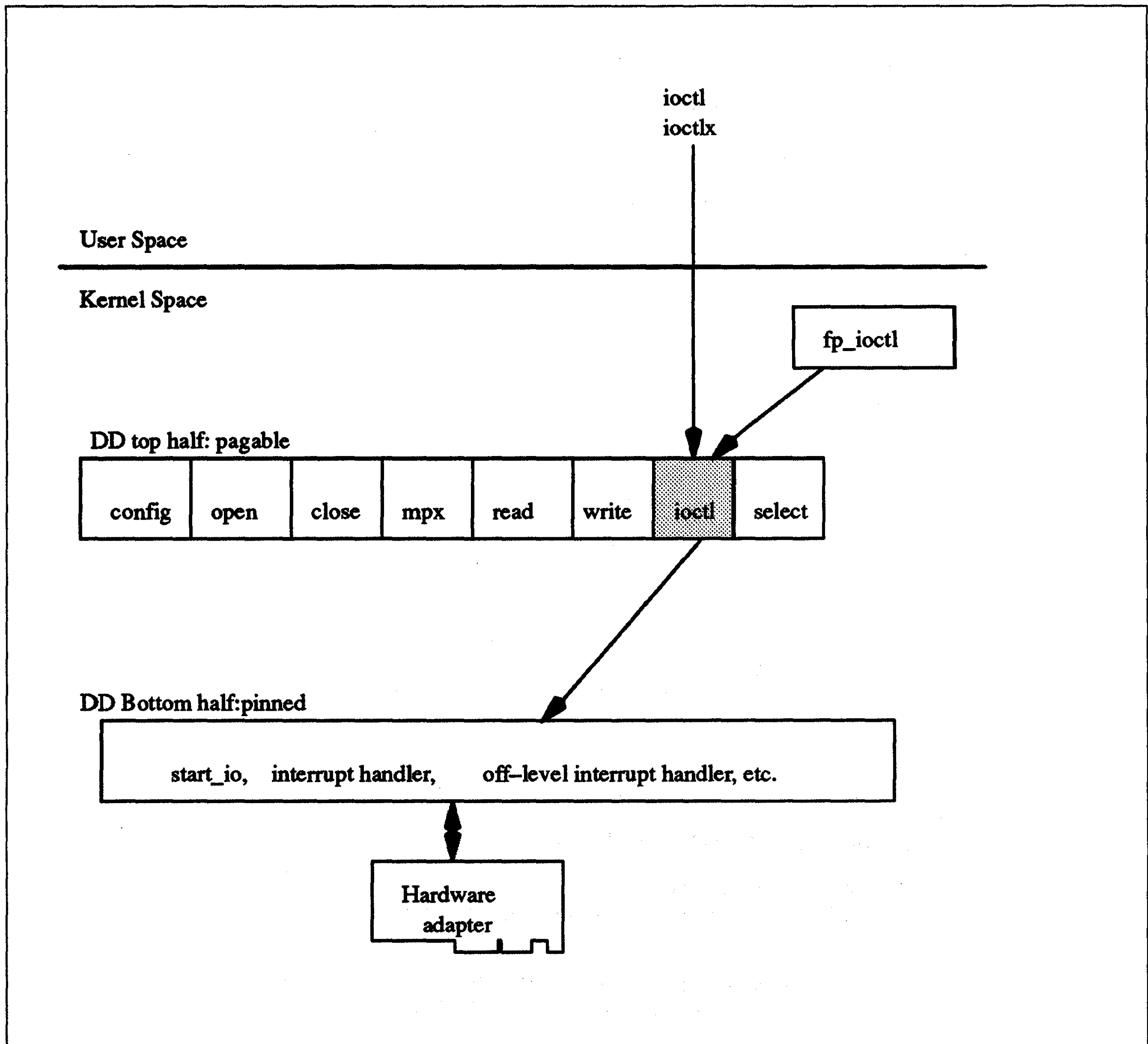


Figure 4-18. Device Driver ddioctl Entry Point

The **ddioctl** device driver entry point performs the special I/O operations requested in an **ioctl** or **ioctlx** subroutine call. The **ddioctl** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver. See Figure 4-19 on page 4-45 for the **ricioctl** sample code.

Six parameters are passed to the **ddioctl** entry point. They are **devno**, **cmd**, **arg**, **devflag**, **chan**, and **ext**, where:

devno Specifies the major and minor device numbers.

cmd The parameter from the ioctl subroutine call that specifies the operation to be performed.

arg The parameter from the ioctl subroutine call that specifies an additional argument for the cmd operation.

devflag Specifies the device open or file control flags.

chan Specifies the channel number.

ext Specifies the extension parameter.

The following example shows the syntax of **ddioctl.**

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

int ddioctl (devno, cmd, arg, devflag, chan, ext)
dev_t devno;
int cmd, arg;
ulong devflag;
chan_t chan;
int  ext;
```

When a program issues an **ioctl** subroutine call, the kernel calls the **ddioctl** routine of the specified device driver. The **ddioctl** routine is responsible for performing whatever functions are requested. In addition, it must return whatever control information has been specified by the original caller of the **ioctl** subroutine. The **cmd** parameter contains the name of the operation to be performed. Most ioctl operations depend on the specific device involved. However, all ioctl routines must respond to the following command:

IOCINFO Returns a devinfo structure, defined in the **<sys/devinfo.h>**, that describes the device. Only the first two fields of the data structure need to be returned if the remaining fields of the structure do not apply to the device.

The **devflag** parameter indicates one of several types of information. It can give conditions in which the device was opened. (These conditions can subsequently be changed by the **fcntl** subroutine call.) Alternatively, it can tell which of two ways the entry point was invoked:

- By the file system on behalf of a using application.
- Directly by a kernel routine using the fp_ioctl kernel service.

Thus, flags in the **devflag** parameter have the following definitions, as defined in the **<sys/device.h>** file:

DKERNEL Entry point called by kernel routine using the **fp_ioctl** service.

DREAD Open for reading.

DWRITE Open for writing.

DAPPEND Open for appending.

DNDELAY Device open in non-blocking mode.

The **ddioctl** entry point can indicate an error condition to the user mode application program by returning a nonzero return code. This causes the ioctl subroutine to return a value of **-1** and makes the return code available to the user mode application in the **errno** external variable. The error code used should be one of the values defined in **<sys/errno.h>**. When applicable, the return values defined in the *POSIX 1003.1* standard for the **ioctl** subroutine should be used.

```
 1
 2  /***************************************************************************
 3   *
 4   *      ricioctl
 5   *
 6   ***************************************************************************/
 7  ricioctl(devno, cmd, arg, flag, mpxchan, ext)
 8  dev_t   devno;          /* major and minor device number */
 9  int     cmd;            /* command to be performed  */
10  caddr_t arg;            /* address of parm block for ioctl system call*/
11  int     flag;           /* flag from last open system call  */
12  chan_t  mpxchan;        /* mpx channel number */
13  caddr_t ext;            /* value of "ext" passed to WRITEX           */
14  {
15      int adapt_num;      /* adapter number */
16      int port_num;       /* port number   */
17      int ret;          /* return value  */
18      t_ric_dds   *dds_ptr;        /* dds pointer */
19      t_acb        *acb_ptr;        /* pointer to ACB struct */
20      struct devinfo      *devinfo_ptr;
21      volatile unsigned long      bus_sr;         /* IO Seg Reg number mask */
22      int                         error;          /* return value  */
23      unsigned long               iob;            /* adapter io base addr */
24      unsigned long               memb;           /* adapter bus memory base */
25      unsigned int                sleep_flag;     /* sleep flag for que_command */
26
27      /* if minor number is invalid, return error */
28      if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
29      {
30              return(EINVAL);
31      }
32
33      /* if the channel number out of range (only 0 is valid for now) */
34      if ( mpxchan != 0 )
35      {
36              return(ECHRNG);
37      }
38
39      /* get dds pointer from dds directory */
40      dds_ptr = dds_dir[minor(devno)];
41
42      /* if port not configured, return error */
43      if (dds_ptr == NULL)
44      {
45              return(EINVAL);
46      }
47
48      adapt_num = dds_ptr->dds_hdw.slot_num;
49      acb_ptr = acb_dir[adapt_num];
50
51      port_num = dds_ptr->dds_dvc.port_num;
52
```

Figure 4-19 (Part 1 of 2). Code Sample of the ricioctl Routine

```
53      /* use the cmd parameter to switch for various operations */
54
55      ret = 0;
56      switch (cmd)
57      {
58          case IOCINFO:/* Standard request for devinfo */
59             devinfo_ptr = (struct devinfo*)arg;
60             devinfo_ptr->devtype = DD_RIC;
61             devinfo_ptr->flags = 0;
62             break;
63
64          case RIC_RASW:   /* Reload adapter software */
65          {
66              /* invoke reload_asw to actually do adapter software */
67              /* reload */
68              sleep_flag = 0;
69              error = reload_asw(acb_ptr, dds_ptr, mpxchan, arg, bus_sr, iob,
70                        memb, sleep_flag);
71
72              break;
73          }
74
75          default:
76             return(EINVAL);
77      }
78
79  } /* end ricioctl */
80
```

Figure 4-19 (Part 2 of 2). Code Sample of the ricioctl Routine

4.1.8 ddselect Device Driver Entry Point

Figure 4-20 on page 4-47 shows the ddselect entry point.

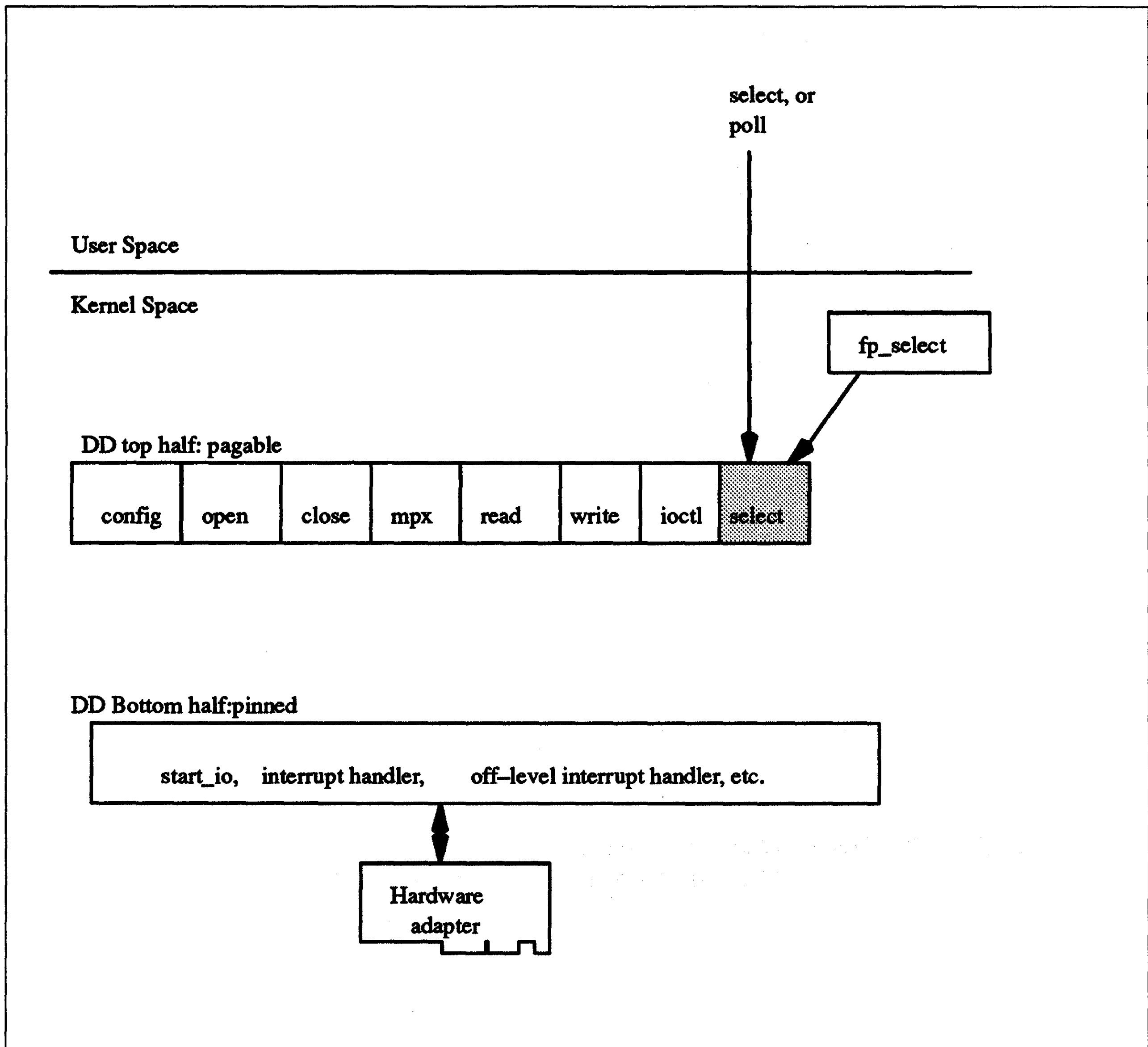


Figure 4-20. Device Driver ddselect Entry Point

The **ddselect** device driver entry point checks to see if one or more events has occurred on the device. The **ddselect** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver. See Figure 4-21 on page 4-50 for the **ricselect** sample code.

The **ddselect** routine can be called with four parameters. They are **devno**, **events**, **reventp**, and **chan**, where:

devno Specifies the major and minor device numbers.

events Specifies the events to be checked.

reventp Returned events pointer. This parameter, passed by reference, is used by the **ddselect** routine to indicate which of the selected events are true at the time of the call. The returned events location pointed to by the **reventp** parameter is set to **0** before entering this routine.

chan Specifies the channel number.

The following example shows the syntax of **ddselect**.

```
#include <sys/device.h>
#include <sys/poll.h>

int ddselect (devno, events, reventp, chan)
dev_t devno;
ushort events;
ushort *reventp;
int chan;
```

The **ddselect** entry point is called when the **select** or **poll** subroutine is used, or when the **fp_select** kernel service is invoked. It determines whether a specified event or events have occurred on the device. The **ddselect** routine can be provided only by character class device drivers. It cannot be provided by block device drivers even when providing raw read/write access.

Possible events to check for are represented as **flags**, or **bits**, in the **events** parameter. There are three basic events defined for the select and poll subroutines, when applied to devices supporting select or poll operations:

POLLIN Input is present on the device.

POLLOUT The device is capable of output.

POLLPRI An exceptional condition has occurred on the device.

A fourth event flag is used to indicate whether the **ddselect** routine should record this request for later notification of the event using the **selnotify** kernel service. This flag can be set in the events parameter if the device driver is not required to provide asychronous notification of the requested events:

POLLSYNC This request is a synchronous request only. The routine need not call the selnotify service for this request even if the events later occur.

Additional event flags in the events parameter are left for device-specific events on the poll subroutine call.

If one or more events specified in the events parameter are in fact true, the **ddselect** routine should indicate this by setting the corresponding bits in the **reventp** parameter. Note that the returned events parameter reventp is *passed by reference*. If none of the requested events are true, then the **ddselect** routine sets the returned events parameter to **0** , which is *passed by reference* through the **reventp** parameter. It also checks the **POLLSYNC** flag in the events parameter. If this flag is **true**, the **ddselect** routine should simply return, since the event request was a synchronous request only. However, if the **POLLSYNC** flag is **false**, the **ddselect** routine needs to notify the kernel when one or more of the specified events later happen. For this purpose, the routine should set separate internal flags for each event requested in the events parameter. When any of these events become true, the device driver routine should use the **selnotify** service to notify the kernel. The corresponding internal flags should then be reset to prevent renotification of the event.

Sometimes the device can be in a state in which a supported event or events can never be satisfied (such as when a communication line is not operational).

In this case, the **ddselect** routine should simply set the corresponding **reventp** flags to **1**. This prevents the select or poll subroutine from waiting indefinitely. As a result however, the caller will not in this case be able to distinguish between satisfied events and unsatisfiable ones. Only when a later request with an **NDELAY** option fails will the error be detected.

Note: Other device driver routines, such as the ddread or ddwrite routines, may require logic to support select or poll operations.

The **ddselect** routine should return with a return code of **0** if the select/poll operation requested is valid for the resource specified. Requested operations are invalid, however, if either of the following is true:

1. The device driver does not support a requested event.
2. The device is in a state in which poll and select operations are not accepted.

In these cases, the **ddselect** routine should return with a nonzero return code, typically **EINVAL**, and without setting the relevant **reventp** flags to **1**. This causes the poll subroutine to return to the caller with the **POLLERR** flag set in the returned **events** parameter associated with this resource. The select subroutine indicates to the caller that all requested events are true for this resource. When applicable, the return values defined in the *POSIX 1003.1* standard for the **select** subroutine should be used.

```
 1
 2  /**************************************************************************
 3   *
 4   *      ricselect
 5   *
 6   **************************************************************************/
 7  ricselect(devno, events, revent_ptr, mpxchan)
 8  dev_t devno;
 9  unsigned short  events;
10  unsigned short  *revent_ptr;
11  int mpxchan;
12  {
13          int             adapt_num;      /* adapter number */
14          int             port_num;       /* port number  */
15          t_acb           *acb_ptr;         /* pointer to ACB */
16          t_ric_dds       *dds_ptr;         /* pointer to DDS */
17          t_chan_info     *tmp_chnptr;  /* temporary channel info pointer */
18          unsigned char   done;
19
20       /* if minor number bad, return  */
21          if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
22          {
23                  return(EINVAL);
24          }
25
26       /* if the channel number out of range, return */
27          if ( mpxchan != 0 )
28          {
29                  return(ECHRNG);
30          }
31
32       /* get dds pointer  */
33          dds_ptr = dds_dir[minor(devno)];
34
35       /* if port not configured, return  */
36          if (dds_ptr == NULL)
37          {
38                  return(ENXIO);
39          }
40
41          dds_ptr->dds_wrk.cmd_avail_flag = FALSE;
42
43          adapt_num = dds_ptr->dds_hdw.slot_num;
44          acb_ptr = acb_dir[adapt_num];
45
46          port_num = dds_ptr->dds_dvc.port_num;
```

Figure 4-21 (Part 1 of 2). Code Sample of the ricselect Routine

```
47
48        /*
49         * get the channel information data structure
50         * pointer from the dds for this channel.
51         */
52          tmp_chnptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
53
54          done = TRUE;
55          while ( done == TRUE )
56          {
57               /* check for requested selections, one at a time */
58
59               /* select on receive data available */
60                  if( events & POLLIN )
61                  {
62                        /* at least one event on the rcv queue */
63                          if(tmp_chnptr->p_rcv_head != NULL)
64                          {
65                                  *revent_ptr |= POLLIN;
66                          }
67                          else
68                          {
69                                  if( !(events & POLLSYNC) )
70                                  {
71                                          tmp_chnptr->sync_flags |= POLLIN;
72                                  }
73                          }
74                  } /* end check for POLLIN flag */
75
76               /* select on status available */
77                  if( events & POLLPRI )
78                  {
79                       /* at least one event on the status queue */
80                          if(tmp_chnptr->p_stat_head != NULL)
81                          {
82                                  *revent_ptr |= POLLPRI;
83                          }
84                          else
85                          {
86                                  if( !(events & POLLSYNC) )
87                                  {
88                                          tmp_chnptr->sync_flags |= POLLPRI;
89                                  }
90                          }
91                  }  /* end check for POLLPRI flag */
92
93          } /* end while */
94
95        /* return of zero tells poll/select to sleep if necessary */
96          return(0);
97
98   } /* end ricselect */
99
```

Figure 4-21 (Part 2 of 2). Code Sample of the ricselect Routine

4.1.9 dddump Device Driver Entry Point

The **dddump** entry point is called by the **kernel dump routine** to set up and send dump requests to the device. The **dddump** routine is optional for a device driver. It is required only when the device driver supports a device as a target for a possible kernel dump.

The **dddump** writes system dump data to a device. The **DUMPINIT dddump** operation is called in the process environment only. The **DUMPQUERY**, **DUMPSTART**, **DUMPWRITE**, **DUMPEND**, and **DUMPTERM dddump** operations can be called in both the process environment and interrupt environment.

> **NOTE**
>
> This entry point is for making your device the target of a system dump, i.e. system data will be transferred to your device when the dump is executed. For information on including your device's own data into the system dump, please refer to "System Dump" on page 9-1.

dddump can be called with six paramters. These are **devno**, **uiop**, **cmd**, **arg**, **chan**, and **ext**, where:

devno Specifies the major and minor device numbers.

uiop Points to the uio structure describing the data area or areas to be dumped.

cmd The parameter from the kernel dump function that specifies the operation to be performed.

arg The parameter from the caller that specifies the address of a parameter block associated with the kernel dump command.

chan Specifies the channel number.

ext Specifies the extension parameter.

The following example shows the syntax for **dddump**.

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

int dddump (devno, uiop, cmd, arg, chan, ext)
dev_t devno;
struct uio *uiop;
int cmd, arg;
chan_t chan;
int ext;
```

It is important that the system state change as little as possible when performing the dump. As a result, the **dddump** routine should use the minimal number of services in writing the dump data to the device.

The **cmd** parameter can specify any of the following dump commands:

DUMPINIT Initialization in preparation for supporting a system dump.

DUMPQUERY Query minimum and maximum data transfer sizes.

DUMPSTART Device setup in preparation for doing a system dump.

DUMPWRITE Write dump data to the device.

DUMPEND Cleanup of the device state after completing dump.

DUMPTERM Release resources allocated for dump support.

The **dddump** entry point can indicate an error condition to the caller by returning a nonzero return code.

Chapter 5. Overview of a Block Device Driver

5.1 Introduction

A block device driver interacts with a special facility in the kernel called the buffer cache. Special entry points in the driver are provided because of this interaction. This driver may also support character type interaction through read and write operations refered to as **raw I/O**. The principal characteristic of block devices is to perform I/O operations using system facilities such as buffer cache management and paging.

Data read from character devices is not stored in a cache for subsequent reading from system buffers. For block device drivers, data is stored in a cache. Block devices interact with the system to keep the cache containing information that a process (or multiple processes) can read from at any time. If the information is not in the cache, the system (not the user) requests the data from the block device driver.

Like all devices, the interaction with block devices is through shared memory. In addition, there are routines to indicate when data in the shared memory (called buf structures) has completed IO processing.

The following sections cover the entry points responsible for the movement of data to and from block devices. This includes control information, the shared memory facilites, the mechanisms for programs to share the information, and the use of the Kernel cache.

Finally, it may be necessary to talk to the device directly without interacting with the system buffer cache. This topic is presented in "Character Access to Block Device Drivers" on page 5-6.

5.1.1 Block I/O Device Driver Entry Points

The device switch table contains the entry point addresses of the interface routines for each device driver in the system, just as it does for for the character device drivers. Figure 5-1 on page 5-2 shows the entry points for a block device driver. Like the character device driver, the block device driver must supply a **config** routine for configuration support as well as an **open** and a **close** routine. The open routine is called each time the device is opened and the close routine is called only on the final close of the device. Instead of having a separate read and write routines, like character device drivers, each block device driver has a **strategy** routine. This routine is called with a pointer to a buffer header, known as the buf structure, which contains the I/O request parameter.

The **strategy** routine handles requests as buffers to be written or read from the device.

© Copyright IBM Corp. 1991

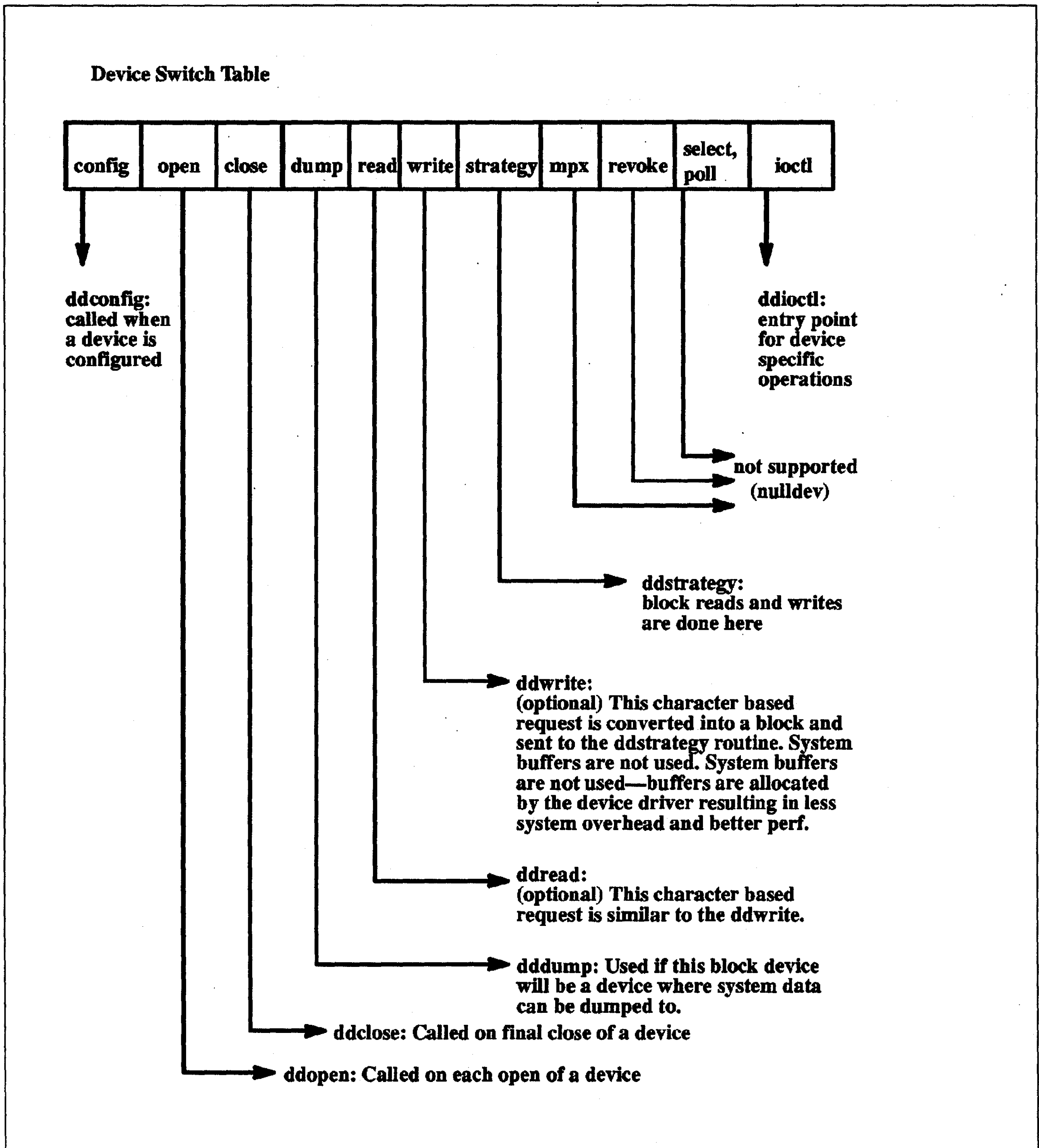


Figure 5-1. Entry Points for a Block Device Driver

5.1.1.1 ddconfig Entry Point

The configuration routine of a block device driver creates /dev entries for a block device. The device may support raw access. Raw access is character access to a block device. In which case, it must create /dev entries for a raw device. They both will use the same major number but the raw device will have names are used with a prefix of 'r' before the device name. For example, a block device named /dev/hdisk0 would also have a /dev/rhdisk0 device if it supported raw access. The system will call the read and write routines of the

raw device if /dev/rhdisk0 is opened. Notice that the raw device and block device share the same major number. Other vendor's UNIX systems may not allocate the same major number for both character and block devices.

5.1.1.2 ddopen/ddclose Entry Points

AIX supports only a few block devices in normal installation. These devices are capable of random access such as the hard disks and cdrom. When these devices are opened, they are opened by system services such as the buffer cache and paging subsystem. They should not be opened directly by user space applications during normal system operations but may be opened during maintenance by applications such as fsck.

The **ddopen** routine in AIX should verify that the device that is requested is opened by only one user. It should also verify the device is a valid device and that it is online and available.

Most of the block devices are attached to the SCSI adapter, therefore you should open the SCSI adapter device driver to communicate with the device. Please see *Kernel Extentions and Device Support Programming Concepts* for the section on the SCSI subsystem.

ddclose processing is performed by the device to release the resource. If the device is attached to the SCSI bus you should refer to *Kernel Extentions and Device Support Programming Concepts* in the SCSI Subsystem section for details.

5.1.1.3 ddstrategy Entry Point

The I/O requests to the physical device are accomplished through the strategy routine. The strategy routine provides a 'Strategy' for mapping I/O requests to the device so that it minimizes requests to the device and maximizes data transfer. When the **strategy** routine (**ddstrategy** device driver entry point) is invoked, a pointer to a buffer header or a chain of buffer headers specifies the request for device I/O. The strategy entry point is invoked in a user process context when the buffer cache does not contain the buffer requested by the user. The strategy routine however does not know or care about the user process.

The buffer header contains the following information:

- The major and the minor number of the device
- The description of the memory buffer to be used for the data transfer
- The direction of the transfer
- The transfer count
- The block number on the device for which the transfer is targeted
- The operation flag

The **strategy** routine returns to the caller as soon as the buffer headers are queued to the appropriate device queue. Note that the strategy routine provides no return code to the caller and never waits for the I/O completion before returning. This means that all requests are assumed valid in terms of parameters and that the request is asynchronous. Normal errors are caught such as out of range blocks but not returned directly as a return code.

The execution of the request completes some time later. Again, the buf struct contains fields for reporting the completion of the request.

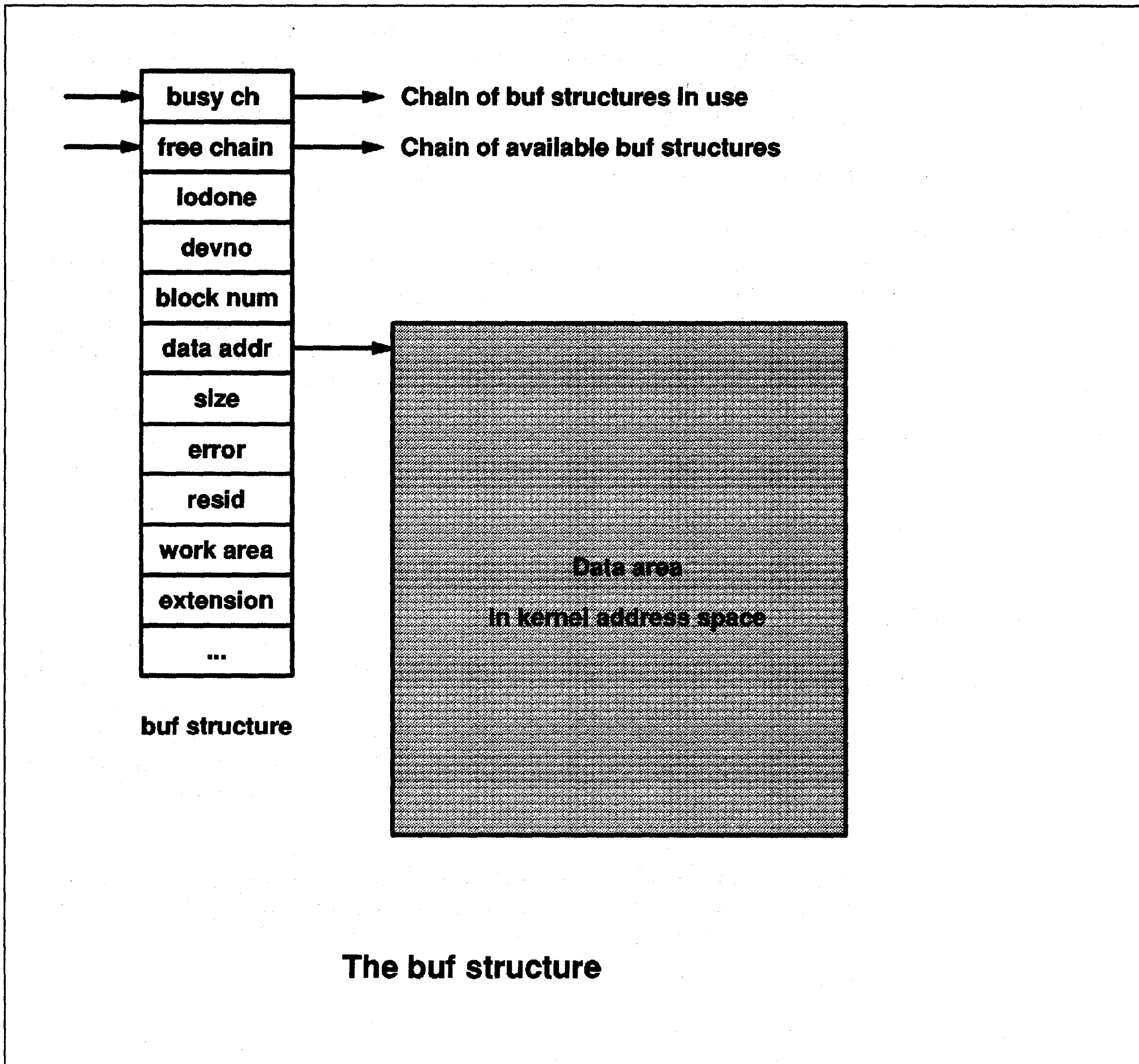


Figure 5-2. The mbuf structure

A buf header contains all the information required to perform block I/O. The buf structure is shown in Figure 5-2 It is the primary interface to the bottom half of block device drivers. In AIX version 3, the traditional **strategy** interface is extended as follows:

1. The device driver strategy routine is called with a list of buf structures, chained using the **av_forw** pointer. The last entry in this list has a NULL **av_forw** pointer.
2. When the operation is completed, and the driver calls **iodone**, the **b_iodone** function defined by the caller is scheduled to run as a software interrupt handler.

   The buf struct and its associated data page must be pinned before calling the **strategy** routine. This is by definition in the <sys/buf.h> include file.

The buf stucture contains the operation to be performed and status information to be returned to the caller and is more like a message exchanged between requestor and service provider.

The caller is notified of I/O completion (or of an error associated with the request) by the device driver's call to **iodone** kernel sevices. A residual count of the number of bytes requested but not tranfered by the operation is placed in the *b_resid* field of the *buf* structure by the device driver before the I/O is marked as complete for the buffer header. If all the requested bytes are transfered then this count will be set to 0.

5.1.1.4 Reordering Block I/O Requests

Multiple I/O request can also be presented to the **strategy** routine, where the additional buffer headers may be chained to the first by using the **av_forw** pointers. While the device **strategy** routine is free to rearrange the buffers on it's device queue with respect to the processing of single request, the ordering of the buffer headers provided in a chain to the **strategy** routine cannot be modified. The **strategy** routine also determine if the block number requested is valid for the device. In the case of a read only operation, a block number at the *end-of-media* is not considered as an error, but no data is transfered. For a write operation, if the block number is at the *end-of-media*, it is considered as an error and the B_ERROR flag in the buf structure should be set, and the *b_error* field should also contain the ENXIO value.

5.1.1.5 Categorizing Requests To The Start I/O Routine

To maintain the state of the device and it's I/O requests, the device driver will typically allocate a private data structure in the system memory associated with the device. The data structure contains device status along with the device error information and the device queue pointers. Some device drivers will maintain more than one queue of buffer headers, for example, one queue for the requests that are waiting for I/O start and another queue for the request that are currently in process.

For SCSI, the queueing process scans the pending queue for the requested device so that the number of SCSI operations is minimized. The requests will be grouped by one of the following rules:

1. Contiguous write operations
2. Operations larger than maximum transfer size
3. Operations requiring special processing

The coalesced (grouped) requests will be removed from the pending queue and placed in the *in_progress* queue so that a single command may be built to satisfy the requests. These requests are then queued and the start I/O routine is called.

5.1.1.6 Starting Processing With The Start I/O Routine

The **start I/O routine** checks to make sure that the device is not busy, and then scans the request queues in an attempt to find an operation to start. First the command stack is checked to see if a command needs to be restarted, and then the *in_progress* queue is checked to start any operations that have already been coalesced. Finally, the pending queue is checked. If it is not empty, the coalesce routine is called to group the operations into the *in_progress* queue. When a request has been found and built, the adapter device driver is called via the strategy routine in order to begin processing of the operation. While the

queues are being scanned and an operation is in progress, the device busy flag is set. It is then reset if no request is found.

Once the I/O handling routine has completed an I/O transfer it calls the iodone routine that determines if the indicated operation has completed successfully or if it has failed. If the operation was successful and complete, then the next request is processed via the start I/O routine. If the operation has failed, your general failure processing routine is called in an attempt to clear the error (retry, etc).

5.1.1.7 dddump Entry Point

The dddump entry point is supplied by a block device driver if it is to be capable of supporting system dumps. It is invoked by **devdump**. (See "System Dump" on page 9-1 for information on supporting the system dump.) The dddump routine provides a return code to devdump. See "dddump Device Driver Entry Point" on page 4-52 for information on the dddump entry point.

5.1.1.8 ddioctl Entry Point

In addition to supplying statistical information about the device, the ioctl can be used for a block device to control operations of the device. However, when the strategy routine is invoked, it should not be dependent upon any ioctl operations.

5.1.2 Character Access to Block Device Drivers

As previously mentioned, character access to block device drivers is known as **raw I/O**. While a character device driver can only be accessed by a character special file, most block device drivers provide both a block and a character special file. This dual interface supports a user being able to access the device in either block or character mode. Note that the block device driver must have a read and a write entry point as well as a strategy entry point if it is to support both character and block mode access. (If only block mode is supported, only a stategy entry point need be supported.)

Examples of the duality provided by a block device drivers are the diskette or the hard-disk device drivers. The diskette is accessed by */dev/fd0* for block mode and by */dev/rfd0* for raw mode. The hard disk is accessed by */dev/hdisk0* for block mode and */dev/rhdisk0* for raw mode.

5.1.2.1 Raw I/O Processing

The raw I/O processing is a mechanism by which a block device driver has the ability to transfer data without using the I/O buffer cache. Instead the raw I/O request is converted into a block and then sent to the the device driver **strategy** entry point to be processed while the read and the write routines are typically waiting for the I/O completion.

When your device driver is configured it will contain entries for both read/write (raw access) and strategy (block access). In addition, the configuration entry point must set up the /dev entries for both. (See "ddconfig Entry Point" on page 5-2.)

If there is no buffer cache and the user is making the request directly then a different buffering facility is involved. Namely, the user is providing a buffer passed in through the uio services. Therefore the read and write entry points are talking to a user process and translating the requests into strategy requests

but still using buf.h. Because the buf.h structure is a header that contains a pointer to the data area it can be mapped to point to a user data area.

In fact, the user buffer could come out of user data, text segments, shared memory segments or the system segment. The different areas are defined in the uio structure through the iovecs.

The read/write routines of the raw device driver use the uphysio services to map the uio areas into buf structs used by the strategy routines. This is discussed in Understanding Raw I/O Support in Kernel Extensions and Concepts.

5.1.3 Block I/O Device Device Summary

A block I/O device contains a device name for it's block device and it's optional character device. Block devices support strategy and read and write routines. mkfs and fsck use the read and write interfaces to perform maintenance on the device while the AIX file system, and virtual memory management work together to provide the traditional Unix Kernel cache.

The kernel cache speeds up access to data by allowing mulitple processes to use the same data and keeping data that is referenced often in the cache. However the cache is based on buffer sizes compatible with Unix file system block sizes and is not efficient for applications that may want to use larger block sizes. To help performance a block device driver should also provide raw access or character access to block devices.

More information

For more information on block device drivers and block I/O kernel services, please refer to the *Kernel Extensions and Device Support Programming Concepts* manual.

Chapter 6. Device Drivers Configuration

6.1 Introduction

Unlike classic Unix device drivers, in AIX Version 3, device drivers are dynamically loaded either at or after system boot time. They are not statically linked into the kernel on disk. This eliminates a great deal of administrative overhead associated with maintaining the AIX kernel, and makes the system simpler to administer and extend. However, it does add to the complexity of the device driver and associated routines.

At system boot time AIX automatically examines the configuration of the system and loads the appropriate device drivers. It also resolves conflicts between various adapters (for I/O port addresses, IRQ levels, etc.). This is called the AIX device configuration subsystem and it performs a variety of functions:

- It scans the Micro Channel bus to determine the two-byte unique POS code for the adapter in each slot.
- It examines the ODM database to determine the characteristics of these adapters.
- It assigns resources (IRQ levels, buffer addresses, DMA levels, I/O port addresses) to each adapter to avoid conflicts.
- It calls the *configuration method* for each device to be configured; this loads and initializes the device driver. Every device must have a configuration method. This will be described later in this chapter.

In the AIX device configuration subsystem, the term device has a wider range of meaning than it does in traditional Unix systems.

In both AIX and Unix systems, the term "device" refers to hardware components such as disk drives, tape drives, printers, and keyboards. Pseudo-devices, such as the console, error, and the null special file, are also included. For AIX, all of these devices are referred to as the kernel devices, that is, the devices with device drivers and known to the system by major and minor numbers.

However, in the AIX operating system, hardware components such as buses, adapters, and even enclosures (including racks, drawers, and expansion boxes) are also considered devices (Figure 6-3 on page 6-9 shows an example of connections and dependencies between these components).

Note that the system cannot use any device unless it is configured.

6.1.1 General Structure of the Device Configuration Subsystem

The Device Configuration Subsystem can be viewed from three different administration levels: the *High Level Administration Perspective*, the *Device Method Level*, and the *Low Level Perspective* (Figure 6-1 on page 6-2 illustrates the general structure of the configuration subsystem).

© Copyright IBM Corp. 1991

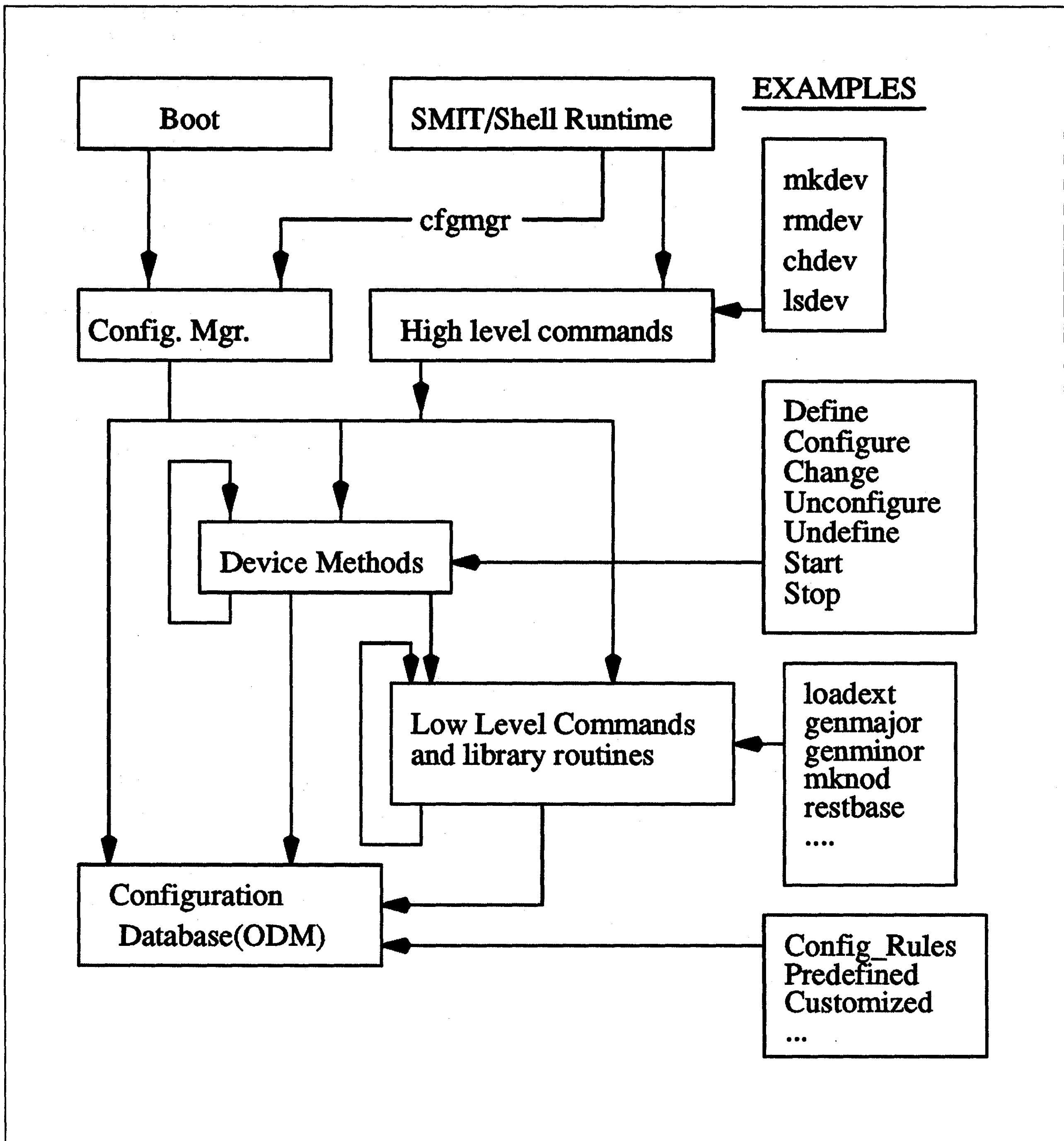


Figure 6-1. Structure of the Configuration Subsytem

6.1.1.1 High Level Perspective

From a high level, user-oriented perspective, four basic tasks comprise device configuration:

1. Adding a device to the system. (mkdev)
2. Deleting a device from the system. (rmdev)
3. Changing the attributes of a device. (chdev)
4. Showing information about a device. (lsdev)

A set of high level commands accomplish these tasks during run time: **mkdev**, **rmdev**, **chdev**, and **lsdev**. (see "The Run Time Configuration Commands" on page A-3 for a description of mkdev, chdev, and rmdev).

The Configuration database stores all information relevant to support the device configuration process. It has two components: the **Predefined Database (PdDv)** and the **Customized Database (CuDv)**. The Predefined database contains configuration data for all devices that could possibly be supported by the system. The Customized database contains configuration data for the devices actually defined and configured in that particular system.

The Configuration Manager (cfgmgr) supervises the configuration of a system's devices when the system is booted (or at run time).

6.1.1.2 Device Method Level

Beneath the high level devices commands or the Configuration Manager is a set of functions called device *methods*. These methods perform well-defined configuration steps, including these five functions:

1. Defining a device in the configuration database.
2. Configuring a device to make it available.
3. Changing a device to make a change in its characteristics.
4. Unconfiguring a device to make it unavailable.
5. Undefining a device from the configuration database.

Device methods also provide two optional functions for devices that need them:

1. Starting a device to take it from the Stopped state to the Available state.
2. Stopping a device to take it to the Stopped state.

Device methods are responsible for changing the state of a device in the system. Figure 6-2 on page 6-4 illustrates all possible device states and how the various methods affect device state changes.

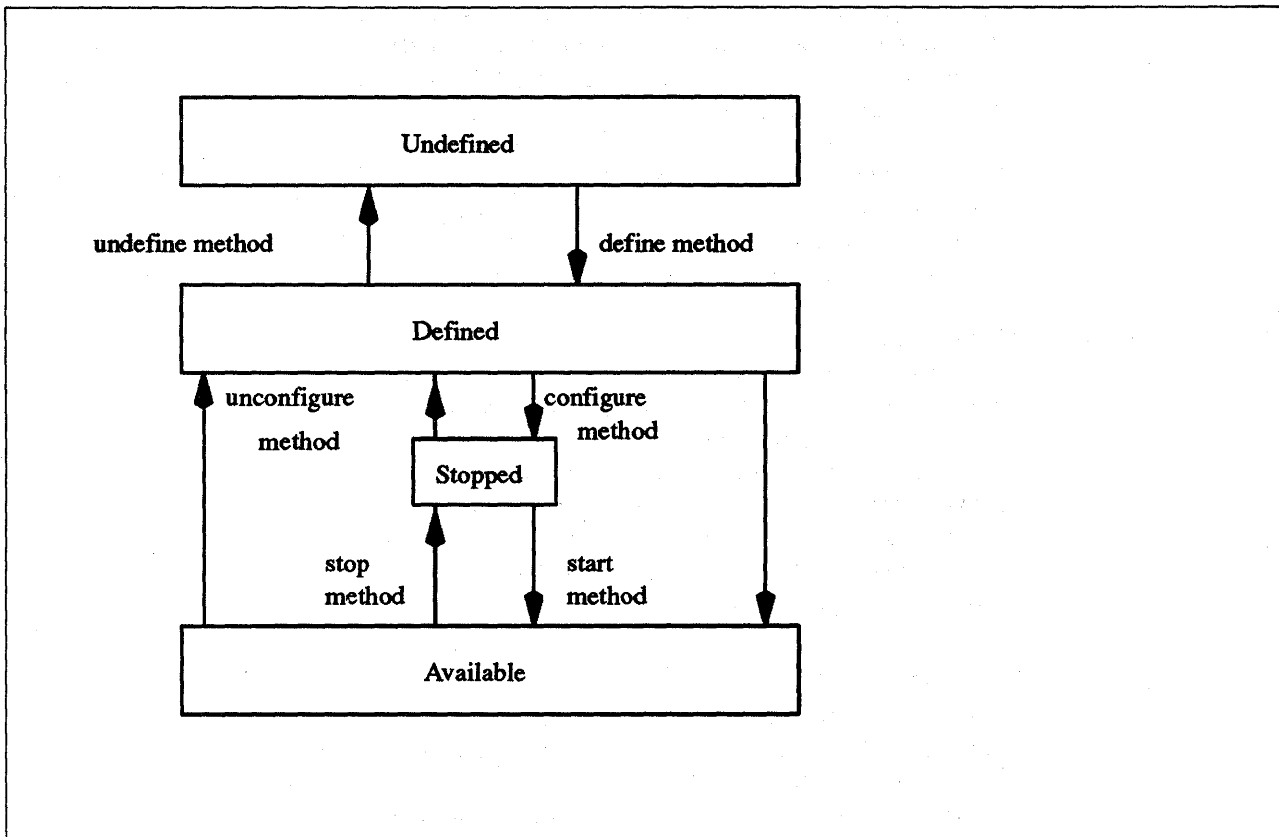


Figure 6-2. Device States

Defined The device instance is represented in the Customized database, but is not configured and not available for use in the system.

Available Configured and available for use by the user.

Undefined The device instance is not represented in the Customized database.

Stopped Configured, but not available for use by applications (optional state).

The **define** method is responsible for creating the device instance in the Customized database and setting the state to *defined*. The **configure** method performs all operations necessary to make the device usable and then sets the state to *available*.

The **change** method usually does not change the state of the device. If the device is in the *defined* state, the **change** method applies all changes to the database and leaves the device *defined*. If the device is *available*, the **change** method attempts to apply the changes to both the database and the actual device and again leave the device in the same state. However, if an error occurs when applying the changes to the actual device, the **change method** may need to unconfigure the device, thus changing the state to *defined*.

The **unconfigure method** must perform the operations necessary to make the device no longer usable. Basically, this is to undo the operations performed by the **configure** method. It will set the device state to defined. Finally, the **undefine** method actually deletes all information for a device instance from the Customized database, thus reverting the instance to the *undefined* state.

The *stopped* state is an optional state that some devices may need to use. A device that supports this state needs **start** and **stop** methods. The **stop** method changes the state from *available* to *stopped*. The **start** method changes it from *stopped* back to *available*.

Both the high level device commands and the Configuration Manager can use the device methods. These methods are implemented to insulate higher level configuration programs from kernel-specific, hardware-specific, and device-specific configuration steps.

6.1.1.3 Low Level Perspective

Beneath the device methods is a set of low level device configuration commands and library routines that can be directly called by device methods as well as by higher level configuration programs. Examples of these commands are **def**dev, **udef**dev, **cfg**dev, **ucfg**dev, **chg**dev, **stt**dev and **stpf**dev. In these commands, the **dev** is the name or your hardware device. These low level commands are usually put in /etc/methods. The device driver programmer writes these methods. They are described in more detail in "Writing Device Methods" on page 6-14.

6.1.2 Device Configuration Database Overview

The Configuration database is an object-oriented database. The Object Data Manager (ODM) provides facilities for accessing and manipulating it. The sets of objects, or **object classes**, contain different pieces of configuration information about the AIX system. The names of the different object classes are found in the /etc/objrepos directory. Although there are many different object classes, we will only be concerned with classes that deal with predefining, customizing and establishing configuration rules for the hardware device.

MORE INFORMATION

For a more extensive description of those object classes please see "ODM Object Classes" on page B-1.

Class Name Definition

PdDv *Predefined Devices*. This class contains an entry for each piece of hardware that could exist in the AIX system. Each entry contains information such as:

- The POS-ID of the adapter
- The name of the adapter
- The code to flash in the machine LEDs as this adapter is being configured
- The name of the configuration method for this adapter.

PdAt *Predefined Attributes*. This class contains specific information about each attribute of each device that could exist in the system. If an adapter can run at three different interrupt levels, for example, then the adapter will have an entry in the PdAt class specifying what those three levels are. Some attributes which might be defined are:

bus_addr_start
Address where this adapter's I/O ports will start. The number of ports is also specified.

bus_mem_start
Address where this adapter's memory will reside. The size (length) of the memory is also specified.

int_level The IRQ level(s) on which this adapter can generate interrupts.

PdCn *Predefined Connections*. This class contains connection information for intermediate devices (see "Device Classes, Subclasses, and Types Overview" on page A-1 for an overview of the AIX classification of the devices). This object class also includes predefined dependency information. For each connection location, there are one or more objects describing the subclasses of devices that can be connected. This information is useful, for example, in verifying whether a device instance to be defined and configured can be connected to a given device.

CuDv *Configured Devices*. The boot process creates this class when the system is started. It contains information about the devices that are actually present in the system at this time.

CuAt *Configured Attributes*. The boot process creates this class when the system is started. If the boot process selects a non-default value for a device's parameters (I/O address, for example) the boot process records the setting to be used here.

CuDep *Customized Dependency*. This class describes device instances that depend on other device instances. Dependency does not imply a physical connection. This object class describes the dependence links between logical devices and physical devices, as well as dependence links between logical devices. Physical dependencies of one device on another device are recorded in the **CuDv** object class.

CuDvDr *Customized Device Driver*. This class stores information about critical resources that need concurrency management through the use of the device configuration library routines. You should only access this object class through these five device Configuration Library routines: the **genmajor**, **genminor**, **relmajor**, **reldevno**, and **getminor** routines.

These routines exclusively lock this class so that accesses to it are serialized. The genmajor and genminor routines return the major and minor number to the calling method. Similarly, the reldevno or relmajor routine releases the major or minor number from this object class.

CuVPD *Customized VPD*. This class contains the Vital Product Data (VPD) for customized devices. VPD can be either machine-readable VPD or manually-entered user VPD information.

Config_Rules *Configuration Rules*. This class contains rules that define in which phase of configuration each hardware device is to be added to the system. Also included is the order in which devices are added. A program that the Configuration Manager must execute is also defined.

(This program is typically the configuration program for the device at the top of the nodes. When these programs are invoked, the names of the next lower level devices that need to be configured are returned. The Configuration Manager configures the next lower level devices by invoking the configuration methods for those devices. In turn, those configuration methods return a list of to-be-configured device names. The process is repeated until no more device names are returned. As a result, all devices in the same node are configured in transverse order.) The following section will explain this in more detail.

For the configuration process to run correctly, then, each adapter in the machine must be defined in the **PdDv** and **PdAt** classes of the ODM database. The author of a new device driver must determine which options can be set for the device and must cause this information to be added to the ODM database. The **PdDv** and **PdAt** classes come preloaded with information about all devices which AIX supports. The **odmadd** command can be used to add new information to these classes; the information to be added to the database is placed in a file whose name ends in **.add**, and this file is used as input by the **odmadd** program.

For a device to be in the *defined* state, the Configuration database must contain a complete description of it. This information includes items such as the device driver name, the device major and minor numbers, the device method names, the device attributes, connection information, and location information.

6.1.3 Device Configuration Procedure Overview

At system boot time, the Configuration Manager is automatically invoked to configure all devices detected and devices whose device information is stored in the Configuration database. At run time, you can configure a specific device by directly invoking a shell command or by using SMIT. This is illustrated in Figure 6-1 on page 6-2.

The system is dynamically configurable. Therefore, it needs a set of rules that defines how to build the system. This building is done by the configuration manager. When the configuration manager is invoked, it reads rules from the **Config_Rules** object class and performs the indicated actions.

The Config_Rules object class is described in more detail in "Configuration Rules (Config_Rules)" on page B-17.

During system boot time, the Configuration Manager is run in two phases. In phase 1 [1] it configures the base devices needed to successfully startup the system. These devices include the root volume group, which permits the Configuration database to be read in from the root file system.

In phase 2 [2] the Configuration Manager configures the remaining devices using the Configuration database from the root file system. During this phase,

---

[1] In Phase 1 the Configuration Manager is called as " **cfgmgr -f**"

[2] In Phase 2 the Configuration Manager is called as "**cfgmgr -s**"

different rules are used depending on the key switch position on the front panel. If the key position is in service position, the rules for service mode are used. Otherwise, the normal startup (phase 2) rules are used.

When invoked during run time, the Configuration Manager only uses the phase 2 rules.

Devices in the system are organized in clusters of tree structures known as **nodes** (Figure 6-3 on page 6-9 provides an example of connections and dependencies of devices in a system).

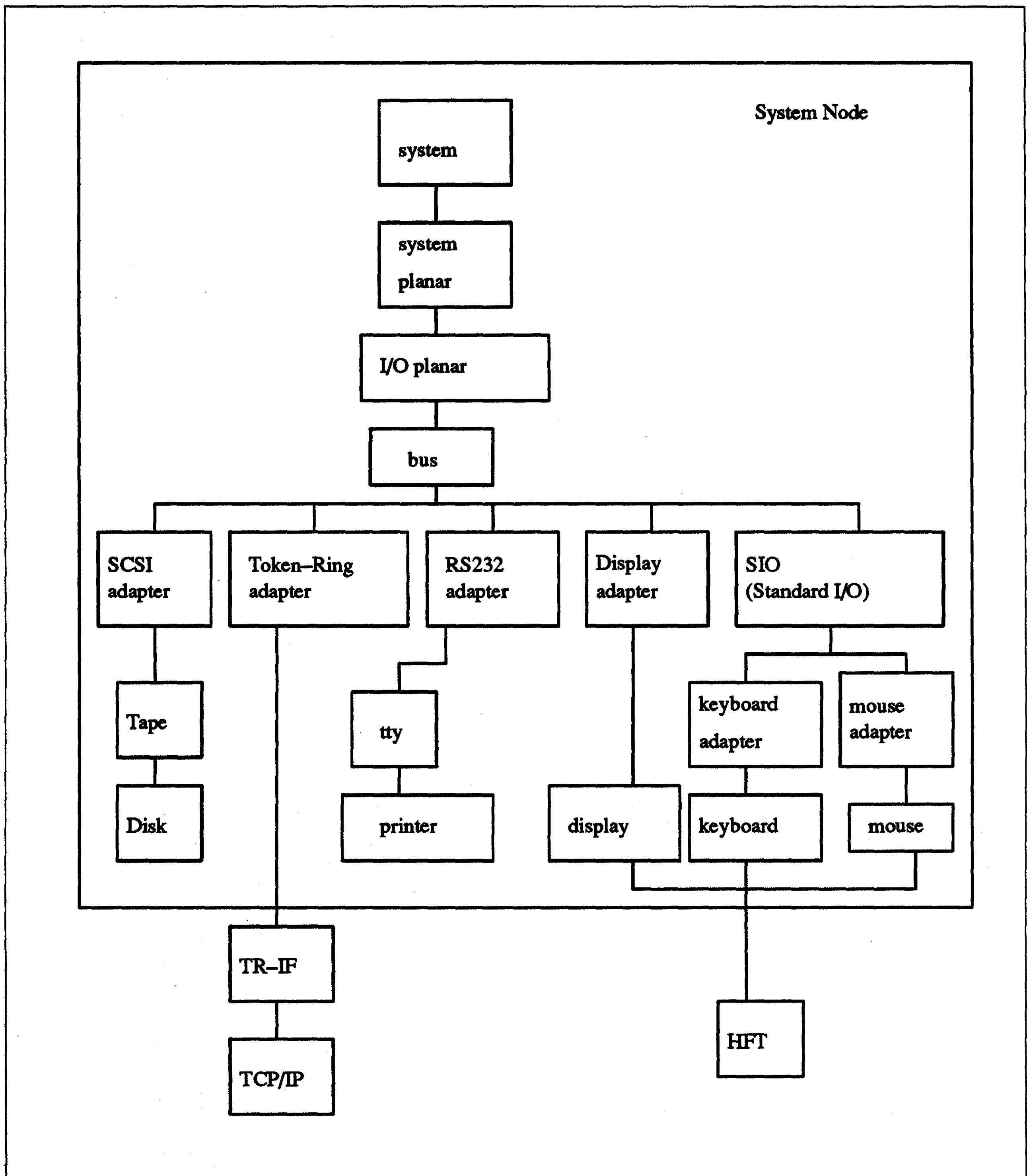


Figure 6-3. Example of Devices Graph

Each tree is a logical subsystem by itself, for example, the System node consists of all the physical devices in the system. The top of the node is the **system** device. Below the bus are the adapters, which are connected to the bus. The bottom of the hierarchy contains the devices to which no other devices are connected. Most of the pseudo-devices, including HFT and pty, are organized as separate tree structures or nodes.

Each rule in the **Config_Rules** object class specifies a program name that the Configuration Manager must execute.

HOW TO LOOK AT Config_Rules

To see the Config_Rules object class you need to use the ODM editor. This is done by issuing the "odme Config_Rules" command. The ODM editor is menu driven. Use the cursor keys so that the "Retrieve/Edit objects" selection is highlighted. Press Enter and the rules are displayed. Press PF3 twice to exit back to an AIX prompt.

The Configuration Manager invokes the programs in the order specified by the sequence value in the rule. In general, the lower the sequence number within a given phase, the higher the priority [3]. These programs are typically the configuration programs for the top of the nodes. In invoking these programs, the names of the next lower level devices that need to be configured are returned. This is shown below.

```
phase    sequence    rule
 1         1         /etc/methods/defsys
 1         2         /etc/methods/deflvm
 2         5         /bin/sysdumpdev -q
 2         10        /etc/methods/defsys
 2         15        /etc/methods/ptynode
 2         20        /etc/methods/starthft
 2         25        /etc/methods/starttty
 2         30        /etc/rc.net -2
 3         5         /bin/sysdumpdev -q
 3         10        /etc/methods/defsys
 3         15        /etc/methods/ptynode
 3         20        /etc/methods/starthft
 3         25        /etc/methods/starttty
```

The Configuration Manager begins by invoking a Node Configuration program listed in one of the rules. The Node Configuration program is responsible for starting the configuration process for a node. It does this by querying the Customized database to see if the device at the top of the node is represented in the database. If so, the program writes the logical name of the device to the stdout file and then returns to the Configuration Manager.

The Configuration Manager intercepts the Node Configuration program's stdout file to obtain the names of the devices that were written. It then invokes the **configure** method for those devices. The device's **configure** method performs the steps necessary to make the device available. If the device is not an intermediate one, the **configure** method simply returns to the Configuration Manager. However, if the device is an intermediate device that has child devices, the **configure** method must determine whether any of the children need to be configured. If so, the **configure** method writes the names of all the child devices to be configured to the stdout file and then returns to the Configuration Manager.

[3] Except for zero, which indicates a don't care condition. Any rule with a sequence number of zero is executed last.

The Configuration Manager intercepts the **configure** method's stdout file to retrieve the names of the children. It then invokes, one at a time, the **configure** methods for each child device. Each of these **configure** methods operate as described for the parent device. For example, they might simply exit when complete, or write to their stdout file a list of additional device names to be configured and then exit. The Configuration Manager will continue to intercept the device names written to the stdout file and to invoke the **configure** methods for those devices until the **configure** methods for all the devices have been run and no more names are written to the stdout file.

When a specific device is defined through its **define** method, the information from the *Predefined* database for that type of device is used to create the information describing the specific device instance. This specific device instance information is then stored in the *Customized* database.

The process of configuring a device is often highly device-specific. The **configure** method for a kernel device needs to:

- Load the device's driver into the kernel
- Pass the Device-Dependent Structure (DDS) describing the device instance to the driver
- Create a special file for the device in the /dev directory.

Of course, many devices do not have device drivers. For this type of device the configured state is not as meaningful. However, it still has a **configure** method that simply marks the device as configured.

QUICK CONFIGURATION SUMMARY

The following is a quick summary of the configuration process:

- Phase 1 configuration only runs at boot time and it configures base devices and logical volume groups.
- Phase 2 configuration runs at boot time and also on demand from the SMIT menus (via the cfgmgr command).
- All configuration phases read the rules in the Config_Rules object class of the ODM database.
- Config_Rules object class defines nodes. The system node as shown in Figure 6-3 on page 6-9 is configured in phase 1 when the Configuration Manager executes the /etc/methods/defsys rule.
- All undetectable devices, i.e. pseudo-devices (pty and HFT) must have a node configuration program so that they can be configured.
- The Configuration Manager calls the config methods of all the devices listed in the node entry of the Config_Rules.
- All devices that have children (i.e. intermediate devices) write their children's names (if they are to be configured) to stdout.
- The Configuration Manager calls the configuration method for all device names that have been written to stdout.

The following figure summarizes how your device would get configured at boot time (assuming you have an entry in the Config_Rules object class of the ODM database).

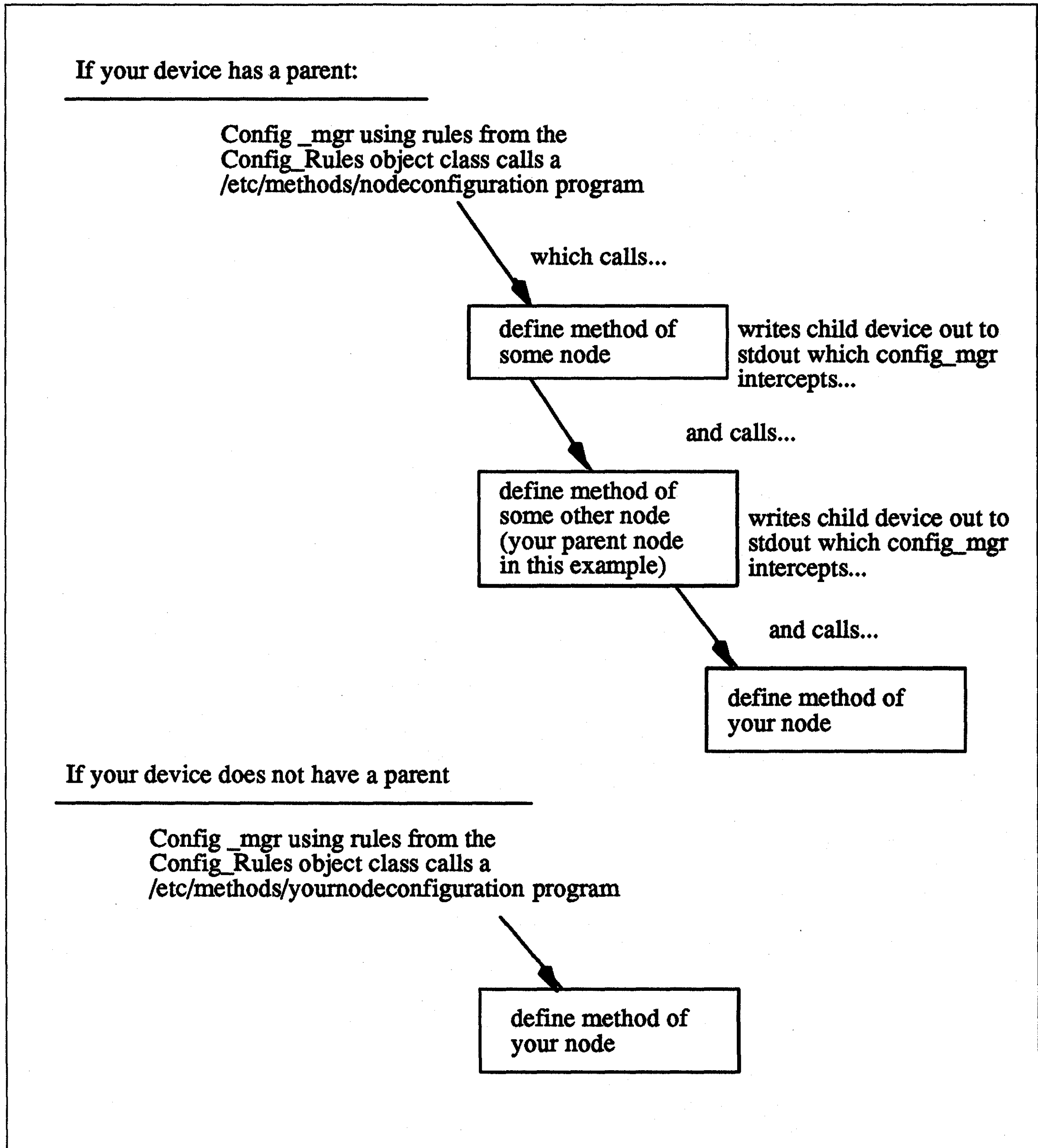


Figure 6-4. How cfgmgr Executes Config_Rules

6.2 Configuring an Unsupported Device to the System

To configure a currently unsupported device to your system, you need to:

- Modify the ODM database
- Write appropriate device methods.

6.2.1 Modifying the Predefined Database

To add a device to your system, you must modify the Predefined database. To do this, you must add information about your device to three Predefined object classes:

- Predefined Devices (PdDv) object class
- Predefined Attribute (PdAt) object class
- Predefined Connection (PdCn) object class.

To describe the device, you must add one object to the **PdDv** object class to indicate the class, subclass, and device type (see "Device Classes, Subclasses, and Types Overview" on page A-1). You must also add one object to the **PdAt** object class for each device attribute, such as interrupt level or block size. Finally, you must add objects to the **PdCn** object class if the device is an intermediate device. (An intermediate device is a device like a SCSI adapter that is used to run other "children" devices like disks and tapes.) If the device is an intermediate device, you must add an object for each different connection location on the intermediate device.

You can use the **odmadd** ODM (Object Data Manager) command from the command line or in a shell script to populate the necessary Predefined object classes from stanza files.

See "ODM Stanzas (ric.add)" on page B-22 for the stanzas necessary to populate the ODM database to support the character device driver used an example in "Overview of a Character Device Driver" on page 4-1.

6.2.1.1 Accessing Device Attributes

The predefined device attributes for each type of predefined device are stored in the **PdAt** object class. The objects in the **PdAt** object class identify the default values as well as other possible values for each attribute. The **CuAt** object class contains only attributes for customized device instances that have been changed from their default values.

When a customized device instance is created by a **define** method, its attributes assume the default values. As a result, no objects are added to the **CuAt** object class for the device. If an attribute for the device is changed from the default value by the **change** method, an object to describe the attribute's current value will be added to the **CuAt** object class for the attribute . If the attribute is subsequently changed back to the default value, the **change** method deletes the **CuAt** object for the attribute.

Any device methods that need the current attribute values for a device must access both the **PdAt** and **CuAt** object classes. If an attribute appears in the **CuAt** object class, then the associated object identifies the current value.

Otherwise, the default value from the **PdAt** attribute object identifies the current value.

6.2.1.2 Modifying an Attribute Value

When modifying an attribute value, your methods must also obtain the objects for that attribute from both the **PdAt** and **CuAt** object classes.

Here are four scenarios that your methods must be able to handle:

1. If the new value differs from the default value and no object currently exists in the **CuAt** object class, your method must add an object into the **CuAt** object class to identify the new value.
2. If the new value differs from the default value and an object already exists in the **CuAt** object class, your method must update the **CuAt** object with the new value.
3. If the new value is the same as the default value and an object exists in the **CuAt** object class, your method must delete the **CuAt** object for the attribute.
4. If the new value is the same as the default value and no object exists in the **CuAt** object class, your method does not need to do anything.

NOTE

Your methods can use the **getattr** and **putattr** subroutines to get and modify attributes.

The getattr subroutine checks both the **PdAt** and **CuAt** object classes before returning an attribute to you. It always returns the information in the form of a **CuAt** object even if returning the default value from the **PdAt** object class.

The **putattr** routine is used to modify attributes and it correctly handles the four cases that are described above. See the softcopy publications for more details.

6.2.2 Writing Device Methods

You will obviously have to write some device methods for your device. Because AIX is dynamically configurable, these methods are necessary in order to use your devices. By convention, these methods are put in /etc/methods.

Your device can have all of the following device methods:

Method	Purpose
define method	Make device ready for configuration.
undefine method	Remove device.
configure method	Put device in ready to use state.
unconfigure method	Remove device from ready to use state.
change method	Alter device parameters.
stop method	Stop device from being used by users so that diagnostics may be run on it.

start method Allow users to use the device again.

It is helpful to see how these device methods actually get called. Please see Figure 6-5 on page 6-16 for information on how the high and low level commands may be used to invoke the various device methods.

How device methods get called During Runtime

HIGH LEVEL COMMAND	METHOD(S) CALLED	ACTION PERFORMED
mkdev →	define method	define device
mkdev →	define method change method	define device with different attributes
mkdev →	define method change method config method	define device and configure with different attributes
mkdev →	config method	configure previously defined device
chdev →	change method	change attruibutes on device
rmdev →	undefine method unconfig method –or– unconfig method undefine method	remove device
cfgmgr →	define method config method	define and configure all detectable devices not configured at boot time

LOW LEVEL COMMAND	METHOD(S) CALLED
defdev →	define method
udefdev →	undefine method
ucfgdev →	unconfig method
cfgdev →	config method
chgdev →	change method
sttdev →	start device
stpdev →	stop device

Figure 6-5. How Device Methods Get Invoked

6.2.2.1 Getting Some Help

So far, the reader might think that to add support for a new device driver, he will have to write five or seven different *methods* (if you need **start/stop** methods).

Most of the time, this is not true. First, very often, you don't need a **change** method. Second, for the **define**, **undefine**, and **unconfigure** methods, as well as for the **change** method, you can try to use the generic methods provided in */etc/methods*:

- *chggen*
- *undefine*
- *ucfgdevice*
- *define.*

It is only when these fail that you will have to write such *methods*. And in that case, you should use the sample methods provided in */usr/lpp/bos/samples* as a starting point. All you have to do then is to make them work for your device!

CAUTION

The generic methods that are found in /usr/lpp/bos/samples (chggen, undefine, ucfgdevice and define) may work now, but may be changed by IBM in subsequent releases of AIX. IBM does not support this as an official programming interface. Therefore, if you are writing a device driver for real customers, do NOT use these generic methods.

Concerning the configuration method, you always need one, and you always need to write it yourself. Of course some examples are provided in */usr/lpp/bos/samples*. Another source of information is "Adapter Configuration Method (cfgrica.c)" on page B-28 and "Ric Port Configuration Method (cfgricp.c)" on page B-37, which are the configuration methods, respectively for the Real Time Interface Co-Processor Portmaster adapter, and for a port on the adapter.

NOTE

The following sections describe various device methods. The syntax specifies the options that your device method may be expected to handle. Note that the "dev" should be replaced by your device name. For example, we would use "ric" for the "Real time Interface Controller" adapter. Therefore, "defric" would be the name of our define method for this adapter.

6.2.2.2 Writing a Define Method

1. Syntax

defdev **-c class -s subclass -t type** [**-p parent -w connection**] [**-l name**]
(where "dev" is the name of your device)

-c class Specifies the class of the device being defined. Class, subclass, and type are required to identify the Predefined Device object in the **PdDv** object class for which a customized device instance is to be created.

-s subclass Specifies the subclass of the device being defined. Class, subclass, and type are required to identify the Predefined Device object in the **PdDv** object class for which a customized device instance is to be created.

-t type Specifies the type of the device being defined. Class, subclass, and type are required to identify the predefined device object in the **PdDv** object class for which a customized device instance is to be created.

-p parent Specifies the logical name of the parent device. This logical name is required for devices that connect to a parent device. This option does not apply to devices that do not have parents for example, most pseudo-devices.

-w connection Specifies where the device connects to the parent. This option applies only to devices that connect to a parent device.

-l name This option is passed in by the **mkdev** command if the user invoking the command is defining a new device and wants to select the name for the device. The **define** method assigns this name as the logical name of the device in the **CuDv** object, if the name is not already in use. If this option is not specified, the **define** method generates a name for the device. Not all devices support or need to support this option.

2. Description

The **define** method is responsible for creating a customized device instance of a device in the Customized database. It does this by adding an object for the device into the **CuDv** object class. The **define** method is invoked either by the **mkdev** configuration command, by a node configuration program, or by the **configure** method of a device that is detecting and defining child devices.

By convention, the first three characters of the name of the **define** method should be **def**. The remainder of the name can be any characters that identify the device or group of devices that use the method, subject to AIX file name restrictions.

The **define** method uses information supplied as input, as well as information in the Predefined database, for filling in the **CuDv** object. If the method is written to support a single device, it can ignore the class, subclass, and type options. In contrast, if the method supports multiple devices, it may need to use these options to obtain the **PdDv** device object for the type of device being customized.

3. Guidelines

The following list of tasks is meant to serve as a guideline for writing a **define** method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device does not have a parent, there is no need to include all of the parent and connection validation tasks. You may also find that your device has special needs that are not listed in these tasks.

Your **define** method must :

1. Validate input parameters.
2. Initialize the ODM.
3. Retrieve the predefined **PdDv** object for the type of device being defined.
4. Ensure that the parent device exists.
5. Validate that the device being defined can be connected to the specified parent device.
6. Assign a logical name to the device.
7. Determine the device's location code.
8. Create the new **CuDv** object.
9. Write the name of the device to standard output.
10. Ensure all object classes are closed and terminate the ODM.

Validate the Input Parameters

The **define** method should ensure that all of the options it requires have been supplied to it. For example, if the **define** method expects parent and connection options for the device being defined, it should ensure that the options are indeed supplied. Also, a **define** method that does not support the **-l** name specification option may want to exit with an error if the option is supplied.

Initialize the ODM

You should initialize the ODM using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine. The following code fragment illustrates this process:

```
#include <cf.h>

if (odm_initialize() < 0)
exit(E_ODMINIT);      /* initialization failed */

if (odm_lock("/etc/objrepos/config_lock",0) == -1) {
odm_terminate();
exit(E_ODMLOCK);      /* database lock failed */
}
```

Retrieve the Predefined PdDv Object for the Type of Device Being Defined

This is done by obtaining the object from the **PdDv** object class whose Class, Subclass, and Type descriptors match the class, subclass, and type options supplied to the **define** method. If no match is found, the **define** method should exit with an error. Information will be taken from the **PdDv** device object in order to create the **CuDv** device object.

Ensure That the Parent Device Exists: If the device being defined connects to a parent device and the name of the parent has been supplied, the **define** method must ensure that the specified device actually exists. It does this by retrieving the **CuDv** object whose Device Name descriptor matches the name of the parent device supplied using the **-p** flag. If no match is found, the **define** method should exit with an error.

Validate That the Device Being Defined Can Be Connected to the Specified Parent Device

If the device has a parent and that parent device exists in the **CuDv** object class, you must next validate that the device being defined can be connected to the specified parent device. To do this, retrieve the predefined connection object from the **PdCn** object class whose UniqueType, Connection Key, and Connection Location descriptors match the Link to Predefined Devices Object Class descriptor of the parent's **CuDv** object obtained in the previous step and the subclass and connection options input into the **define** method, respectively. If no match is found, an invalid connection has been specified. This may be because the specified parent is not an intermediate device, does not accept the type of device being defined (as described by subclass), or does not have the connection location identified by the connection option.

Assign a Logical Name to the Device

Each newly assigned logical name must be unique to the system. If a name has been supplied using the **-l** flag, you must make sure it is unique before assigning it to the device. This is done by checking the **CuDv** object class for any object whose Device Name descriptor matches the desired name. If a match is found, the name is already used and the **define** method must exit with an error.

If the **define** method is to generate a name, it can do so by obtaining the prefix name from the Prefix Name descriptor of the device's **PdDv** device object and invoking the **genseq** subroutine to obtain a unique sequence number for this prefix. By appending the sequence number to the prefix name, a unique name results. The genseq routine looks in the **CuDv** object class to ensure that it assigns a sequence number that has not been used with the specified prefix to form a device name.

In some cases, a **define** method may need to ensure that only one device of a particular type has been defined. For example, there can only be one PTY device customized in the **CuDv** object class. The PTY **define** method does this by querying the **CuDv** object class to see if a device by the name pty0 exists. If it does, the PTY device has already been defined. Otherwise, the **define** method proceeds to define the PTY device using the name pty0.

Determine the Device's Location Code

If the device being defined is a physical device, it has a location code (see "Devices Location Codes" on page A-3 for more information about location codes).

Create the New **CuDv** Object.

Set the **CuDv** descriptors as follows:

device name
: Use the name as determined above.

device status flag
: Set to the Defined state.

change status flag
Set to the same value as that found in the Change Status Flag descriptor in the device's **PdDv** object.

device driver instance
Typically set to the same value as the Device Driver Name descriptor in the device's **PdDv** object. It may be used later by the **configure** method.

device location code
Set to a null string if the device does not have a location code. Otherwise, set it to the value computed.

parent device logical name
Set to a null string if the device does not have a parent. Otherwise set it to the parent name as specified by the parent option.

location where connected on parent device
Set to a null string if the device does not have a parent. Otherwise, set it to the value specified by the connection option.

link to predefined devices object class
Set to the value obtained from the Unique Type descriptor of the device's **PdDv** object.

Write the Name of the Device to Standard Output: A blank should be appended to the device name to serve as a separator in case other methods write device names to standard output. Either the **mkdev** command or the **configure** method that invoked the **define** method will intercept standard output to obtain the device name assigned to the device.

Ensure That All Object Classes Are Closed and Terminate the ODM

Exit with an exit code of zero if there were no errors.

6.2.2.3 Writing an Undefine Method

1. Syntax

udefdev **-l name**
(where "dev is the name of your device)

-l name Identifies the logical name of the device to be undefined.

2. Description

The **undefine** method is responsible for deleting a Defined device from the Customized database. Once a device is deleted, it cannot be configured until it is once again defined by the **define** method.

The **undefine** method is also responsible for releasing the major and minor number assignments for the device instance and deleting the device's special files from the */dev* directory. If minor-number assignments are registered with the genminor subroutine, the **undefine** method can release the major and

minor number assignments and delete the special files by simply calling the **reldevno** subroutine.

By convention, the first four characters of the name of the **undefine** method are to be **udef**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the method.

3. Guidelines

The following list of tasks is meant to serve as a guideline for writing an **undefine** method. You may find that your device has special needs that are not listed in these tasks.

Your **undefine** method must:

1. Validate the input parameters. The **-l** flag must be supplied to identify the device that is to be undefined.
2. Initialize the ODM using the **odm_initialize** subroutine and lock the configuration database using the **odm_lock** subroutine.
3. Retrieve the **CuDv** object for the device to be unconfigured. This is done by getting the **CuDv** object whose Device Name descriptor matches the name supplied with the **-l** flag. If no object is found with the specified name, exit with an error.
4. Check the device's current state. If the Device Status descriptor indicates that the device is not in the Defined state, then it is not ready to be undefined. If this is the case, exit with an error.
5. Check for any child devices. This check is accomplished by querying the **CuDv** object class for any objects whose Parent Device Logical Name descriptor matches this device's name. If the device has any children at all, regardless of the states they are in, the **undefine** method must fail. All children must be undefined before the parent can be undefined.
6. Check to see if this device is listed as a dependency of another device. This is done by querying the **CuDep** object class for objects whose Dependency descriptor matches this device's logical name. If a match is found, exit with an error. A device may not be undefined if it has been listed as a dependency by another device (see "Device Dependencies and Child Devices" on page A-1 for more information about device dependencies).
7. If no errors have been encountered, the method can begin deleting customized information. First, delete the special files from the */dev* directory. Next, delete all minor number assignments. If the last minor number has been deleted for a particular major number, release the major number as well, using the **relmajor** subroutine. The **undefine** method should never delete objects from the **CuDvDr** object class directly, but should always use the routines provided. If the minor-number assignments are registered with the **genminor** subroutine, all of the above can be accomplished by the **reldevno** subroutine.
8. Delete all attributes for the device from the **CuAt** object class. Simply delete all **CuAt** objects whose Device Name descriptor matches this device's logical name. It is not an error if the ODM routines used to delete the attributes indicate that no objects were deleted. This simply indicates

that the device has no attributes that had been changed from the default values.

9. Delete the **CuDv** object for the device.

10. Make sure all object classes are closed and terminate the ODM via the **odm_terminate** call. Exit with an exit code of zero if there are no errors.

6.2.2.4 Writing a Configure Method

1. Syntax

cfgdev **-l name** [**-1** | **-2**]
(where "dev" is the name of your device)

-l name Identifies the logical name of the device to be configured.

-1 Specifies that the device will be configured in phase 1 of system boot. This -1 option cannot be specified along with the -2 option. If neither the -1 nor the -2 options are specified, the **configure** method is invoked at run time.

-2 Specifies that the device will be configured in phase 2 of system boot. This -2 option cannot be specified along with the -1 option. If neither the -1 nor the -2 options are specified, the **configure** method is invoked at run time.

The options specifying the phase of system boot can be used to limit certain functions to specific phases.

2. Description

The **configure** method is responsible for configuring a device, that is, making it available for use in the system. It changes a device's state from Defined to Available. If the device has a device driver, the **configure** method is responsible for loading the device driver into the kernel and describing the device characteristics to the driver. For an intermediate device (for example, a SCSI bus adapter), this method also determines which attached children are to be configured and writes their logical names to standard output.

The **configure** method is invoked by either the **mkdev** configuration command or by the Configuration Manager. Because the Configuration Manager runs a second time in phase 2 system boot, and can also be invoked repeatedly at runtime, a device's **configure** method can be invoked to configure an already available device. This is not an error condition. In the case of an intermediate device, the **configure** method should check again for the presence of child devices. If the device is not an intermediate device, the method simply returns.

By convention, the first three characters of the name of the **configure** method should be **cfg**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the method.

In general, the **configure** method obtains all the information it needs about the device from the Configuration database (made up of the CuDv, CuDvDr, CuAt, Cu...object classes).

If the device has a parent device, the parent must be configured first. The **configure** method for a device should fail if its parent is not already in the Available state.

3. Guidelines for Writing a Configure Method

This list of tasks is meant to serve as a guideline for writing a **configure** method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device is not an intermediate device or does not have a device driver, your method can be written accordingly. You may also find that your device has special needs that are not listed in these tasks.

Your **configure** method must:

1. Validate the input parameters. The **-l** logical name option must be supplied to identify the device that is to be configured. The **-1** and **-2** options cannot be supplied at the same time.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine.
3. Retrieve the **CuDv** object for the device to be configured. This is done by getting the **CuDv** object whose Device Name descriptor matches the name supplied with the **-l** logical name option. If no object is found with the specified name, exit with an error.
4. Retrieve the **PdDv** object for the device to be configured by getting the **PdDv** object whose Uniquetype descriptor matches the Link to Predefined Devices Object Class descriptor of the device's **CuDv** object.
5. If either the **-1** or **-2** option is specified, the **configure** method should obtain the LED Value descriptor of the device's **PdDv** object and display the value on the system LEDs using the **setleds** subroutine. This specifies when the **configure** method will execute at boot time. If the system hangs during configuration at boot time, the displayed LED value indicates which **configure** method the hang occurred in.
6. If the device is already configured (that is, the Device State descriptor of the device's **CuDv** object indicates that the device is in the Available state), and is an intermediate device, the **configure** method should skip to the task of detecting children devices. If the device is configured but is not an intermediate device, the **configure** method should simply exit with no error.
7. If the device is still in the Defined state, the following tasks should be performed:
   a. If the device has a parent, the **configure** method must ensure that the parent device exists and is in the available state. The method can look at the Parent Device Logical Name descriptor of the device's **CuDv** object to obtain the parent name. If the device does not have a parent, this descriptor should be a null string.

      Assuming that the device does have a parent, the **configure** method should obtain the parent device's **CuDv** object and check the Device State descriptor. If the object does not exist or is not in the Available state, exit with an error. Another check must be made if the device has a parent device. The configure method must make sure that no

other device connected to the same parent at the same connection location has been configured. This case could arise, for example, when different printers are connected to the same port using a switch box. Each of the printers would have the same parent and connection, but only one could be configured at any given time.

The **configure** method can make this check by querying the **CuDv** object class for objects whose Device State descriptor is set to available and whose Parent Device Logical Name and Location Where Connected on Parent Device descriptors match those for the device being configured. If a match is found, exit with an error.

b. If the device is an adapter card and the **configure** method has been invoked at run time (indicated by the absence of both the **-1** and **-2** options), the **configure** method should ensure that the adapter card is actually present.

   This can be done by reading POS registers from the card. This is essential, because if the card is present, the **configure** method must invoke the **busresolve** library routine to assign bus resources to the card and ensure that bus resources for the adapter do not conflict with other adapter cards in the system. If the card is not present or the **busresolve** routine fails to resolve bus resources, exit with an error.

   busresolve information

   For more information on the busresolve system call, see "The busresolve system call" on page F-1.

   The POS registers are obtained by opening and accessing the */dev/bus0* special file.

c. Determine whether or not the device has a device driver. The **configure** method obtains the name of the device driver from the Device Driver Name descriptor of the device's **PdDv** object. If this descriptor is a null string, the device does not have a device driver.

d. If the device has a device driver, the **configure** method will need to perform the following tasks:

   - First, load the device driver. The **loadext** subroutine can be used to do this. Loading a Device Driver in InfoExplorer has more information on loading the device driver.
   - Determine the device's major number using the **genmajor** subroutine.
   - Determine the device's minor number, possibly by using the **getminor** and **genminor** subroutines.
   - Create the device special files in the */dev* directory if they do not already exist. Special files are created with the **mknod** subroutine.
   - Build the device-dependent structure (DDS) for the device. This structure contains the information that describes the device's characteristics to the device driver. The information is usually obtained from the device's attributes in the Configuration database. You may need to refer to the appropriate device driver information to determine what the device driver expects the DDS to look like

(see "A Brief Discussion of the DDS" on page 4-9 for more information).

- Use the **sysconfig** subroutine to initialize and pass the DDS to the device driver.
- If there is code to be downloaded to the device, read in the required file and pass the code to the device through the interface provided by the device driver. The file to be downloaded might possibly be identified by a **PdAt** or **CuAt** object. By convention, microcode files should be in the */etc/microcode* directory.

e. After the tasks relating to the device driver are complete, or if the device did not have a device driver, the **configure** method should determine if it needs to obtain vital product data (VPD) from the device. The VPD Flag descriptor of the device's **PdDv** object specifies whether or not it has VPD. (See the next section for more details.)

f. At this point, if no errors have been encountered, the device is configured. The **configure** method should update the Device Status descriptor of the device's **CuDv** object to indicate that it is available.

8. If the device being configured is an intermediate device, the **configure** method has one final task to perform. If the child devices actually attached can be detected, the **configure** method is responsible for defining any new children not currently represented in the **CuDv** object class. This is accomplished by invoking the **define** method for each new child device. For each detected child device that is already in the **CuDv** object class, the **configure** method must look at the child device's **CuDv** Change Status Flag descriptor to see if it needs to be updated. If the descriptor's value is **DONT_CARE**, nothing needs to be done. If it has any other value, it must be set to **SAME** and the child device's **CuDv** object must be updated. The Change Status Flag descriptor is used by the system to indicate configuration changes.

   If the device is an intermediate device but cannot detect attached children, it can query the **CuDv** object class for children. The value of the Change Status Flag descriptor for these child devices should be **DONT_CARE** since the parent device cannot detect them. Sometimes a child device has an attribute specifying to the **configure** method whether the child is to be configured. The autoconfig attribute of TTY devices is an example of this type of attribute.

   Regardless of whether the child devices are detectable, the **configure** method should write the device logical names of the children to be configured to standard output, separated by space characters. If the method was invoked by the Configuration Manager, the Manager invokes the **configure** method for each of the child device names written to standard output.

9. Finally, ensure that all object classes are closed and terminate the ODM. Exit with an exit code of 0 (zero) if there are no errors.

4. Handling Device Vital Product Data (VPD)

Devices that provide vital product data (VPD) should be identified in the **PdDv** object class by setting the VPD Flag descriptor to TRUE in each of the device's **PdDv** objects. The **configure** method must obtain the VPD from the device and store it into the Customized VPD (CuVPD) object class. The appropriate

hardware documentation for the device should be consulted to determine how to retrieve VPD from the device. (In many cases, VPD can be obtained for a device from the device driver with the sysconfig subroutine.)

Once the VPD is obtained from the device, the **configure** method should query the **CuVPD** object class to see if the device already has hardware VPD stored there. If there is, the method should compare the VPD obtained from the device with that from the **CuVPD** object class. If the VPD is the same in both cases, no further processing is needed. If they are different, update the VPD in the **CuVPD** object class for the device. If there is no VPD in the **CuVPD** object class for the device, add the device's own VPD into it.

Comparing the device's VPD with that in the **CuVPD** object class first helps make modifications to the **CuVPD** object class less frequent. This results from the fact that the VPD from a device typically does not change. Reducing the number of database writes increases performance and minimizes the possibility of data loss.

5. Configure Method Errors

For many of the errors detected by the **configure** method, the method can simply exit with the appropriate exit code. In other cases, the **configure** method may need to undo some operations it has performed. For instance, after loading the device's device driver and defining the device to the device driver by passing it the device-dependent structure (DDS), the **configure** method may subsequently encounter an error while downloading microcode. If this happens, the method should terminate the device from the device driver with the sysconfig subroutine and unload the driver with the **loadext** subroutine.

The **configure** method does not need to delete the special files or unassign the major and minor numbers if the major and minor numbers were successfully allocated and the special file created before the error was encountered.

This is because the AIX configuration scheme allows both major and minor numbers and special files to be maintained for a device even though the device is unconfigured. If the device is configured again, the **configure** method should recognize that the major and minor numbers are already allocated and that the special files already exist.

By the time the **configure** method checks for child devices, it has already successfully configured the device that it was called to configure. Errors that occur while checking for child devices are indicated with the **E_FINDCHILD** exit code. The **mkdev** command detects whether the **configure** method completed successfully. It can still display a message indicating that an error occurred while looking for child devices.

6.2.2.5 Writing an Unconfigure Method

1. Syntax

ucfgdev **-l name**
(where "dev" is the name of your device)

-l name Identifies the logical name of the device to be unconfigured.

2. Description

The **unconfigure** method is responsible for unconfiguring an available device. This means taking a device that is available for use by the system and making it unusable. All the customized information about the device is to be retained in the database so that the device can be configured again exactly as it was before.

The actual operations required to make a device no longer available for use depend on what the **configure** method did to make the device available in the first place. For instance, if the device has a device driver, the **configure** method will have loaded a device driver into the kernel and described the device to the driver through a device-dependent structure (DDS). The **unconfigure** method thus needs to tell the driver to delete the device instance and then request an unload of the driver.

If the device is an intermediate device, the **unconfigure** method must check the states of the child devices. If any child is in the Available state, the **unconfigure** method will fail and leave the device configured. To ensure proper system operation, all children must be unconfigured before the parent can be unconfigured.

Although the **unconfigure** method must check child devices, it does not need to check for device dependencies recorded in the **CuDep** object class (see "Device Dependencies and Child Devices" on page A-1 for more information).

The **unconfigure** method must also fail if the device is currently open. In this case, the device driver must return a value for the errno variable of EBUSY to the **unconfigure** method when the method requests the driver to delete the device. The device driver is the only component at that instant that knows the device is open. As in the case of configured children, the **unconfigure** method will fail and leave the device configured.

When requesting the device driver to terminate the device, errno values other than EBUSY can be returned. The driver should return ENODEV if it does not know about the device. Under the best circumstances, however, this case should not occur. If ENODEV is returned, the **unconfigure** method should go ahead and unconfigure the device with respect to the database so that the database and device driver are in agreement. If the device driver chooses to return any other errno value, it must still delete any stored characteristics for the specified device instance. The **unconfigure** method should also indicate that the device is unconfigured by setting the state to Defined.

The **unconfigure** method does not generally release the major number and minor number assignments for a device, nor does it delete the device's special files in the */dev* directory.

By convention, the first four characters of the name of the **unconfigure** method should be **ucfg**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the method.

3. Guidelines

This list of tasks is meant to serve as a guideline for writing an **unconfigure** method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device is not an intermediate device or does not have a device driver, your method can be written accordingly. You may also find that your device has special needs that are not listed in these tasks.

Your **unconfigure** method must:

1. Validate the input parameters. The **-l** flag must be supplied to identify the device that is to be unconfigured.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine.
3. Retrieve the **CuDv** object for the device to be unconfigured. This is done by getting the **CuDv** object whose Device Name descriptor matches the name supplied with the **-l** flag. If no object is found with the specified name, exit with an error.
4. Check the device's current state. If the Device Status descriptor indicates that the device is in the Defined state, then it is already unconfigured. You should exit as for a successful completion.
5. Check for child devices in the Available state. This can be done by querying the **CuDv** object class for objects whose Parent Device Logical Name descriptor matches this device's name and whose Device Status descriptor is not defined. If a match is found, exit with an error.
6. Retrieve the predefined **PdDv** object for the device to be configured by getting the **PdDv** object whose UniqueType descriptor matches the Link to Predefined Devices Object Class descriptor of the device's **CuDv** object. This object will be used to get the device driver name.
7. Determine whether the device has a device driver. The **unconfigure** method obtains the name of the device driver from the Device Driver Name descriptor of the device's **PdDv** object. If this descriptor is a null string, the device does not have a device driver. In this case, skip to the task of updating the device's state.
8. If the device has a device driver, the **unconfigure** method will need to perform the following tasks:
   a. Determine the device's major and minor numbers using the **genmajor** and **getminor** subroutines. These are used to compute the device's *devno*, using the **makedev** macro defined in the *sysmacros*.h header file, in preparation for the next task.
   b. Use the **sysconfig** subroutine to tell the device driver to terminate the device. If a value of EBUSY for the errno variable is returned, exit with an error.
   c. Use the **loadext** routine to unload the device driver from the kernel. The **loadext** routine will not actually unload the driver if there is another device still configured for the driver.
9. The device is now unconfigured. The **unconfigure** method should update the Device Status descriptor of the device's **CuDv** object to defined.

10. Ensure that all object classes are closed and terminate the ODM. If there are no errors, exit with an exit code of 0 (zero).

6.2.2.6 Writing a Change Method

1. Syntax

chgdev **-l name [-p parent] [-w connection] [-P | -T] [-a attr=value]**
(where "dev" is the name opf your device)

-l name Identifies the logical name of the device to be changed.

-p parent Identifies the logical name of a new parent for the device. This option is used to move a device from one parent to another.

-w connection
Identifies a new connection location for the device. This option either identifies a new connection location on the device's existing parent, or if the **-p** option is also used, it identifies the connection location on the new parent device.

-P Indicates that the changes are to be recorded in the Customized database without those changes being applied to the actual device. This is a useful option for a device which is usually kept open by the system such that it cannot be changed. Changes made to the database with this option are later applied to the device when it is configured at system reboot.

-T Indicates that the changes are to be applied only to the actual device and not recorded in the database. This is a useful option for allowing temporary configuration changes that will not apply once the system is rebooted.

-a attr=value
Identifies an attribute to be changed and the value to which it should be changed.

2. Description

The **change** method is responsible for applying configuration changes to a device. If the device is in the Defined state, the changes are simply recorded in the Customized database. If the device is in the Available state, the **change** method must also apply the changes to the actual device by reconfiguring it.

Your **change** method does not need to support all the options described for **change** methods. For instance, if your device is a pseudo-devices with no parent, it need not support parent and connection changes. Even for devices that have parents, it may be desirable to disallow parent and connection changes. For a printer, such changes may make sense since a printer is easily moved from one port to another. An adapter card, by contrast, is not usually moved without first shutting off the system. It is then automatically configured at its new location when the system is rebooted. Consequently, there may not be a need for a **change** method to support parent and connection changes.

In deciding whether to support the **-T** and **-P** flags, remember that these options will allow a device's configuration to get out of sync with the Configuration

database. The **-P** flag can often be useful for devices that are typically kept open by the system. The **change** methods for most IBM-supported devices do not support the **-T** flag.

In applying changes to a device in the Available state, your **change** method could terminate the device from the driver, rebuild the device-dependent structure (DDS) using the new information, and define the device again to the driver using the new DDS. Your method may also need to reload adapter software or perform other device-specific operations. An alternative is to simply invoke the device's **unconfigure** method, update the Customized database, and invoke the device's **configure** method.

By convention, the first three characters of the name of the **change** method should be **chg**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices which use the method.

3. Guidelines

The following list of tasks is meant to serve as a guideline for writing a **change** method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device does not support the changing of parent or connection, there is no need to include those tasks. You may also find that your device has special needs that are not listed in these tasks.

If your **change** method is written to invoke the **unconfigure** and **configure** methods, it must:

1. Validate the input parameters. The **-l** flag must be supplied to identify the device that is to be undefined. You may want to exit with an error if options that your method does not support are specified.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine.
3. Retrieve the **CuDv** object for the device to be changed by getting the **CuDv** object whose Device Name descriptor matches the name supplied with the **-l** option. If no object is found with the specified name, exit with an error.
4. Validate all attributes being changed. Make sure that the attributes apply to the specified device, that they can be set by the user, and that they are being set to valid values. The **attrval** subroutine can be used for this purpose. If you have attributes whose values depend on each other, you need to write the code to cross check them. If invalid attributes are found, your method needs to write information to standard error describing them (See the next section for more explanations.)
5. If a new parent device has been specified, find out whether it exists by querying the **CuDv** object class for an object whose Device Name descriptor matches the new parent name. If no match is found, exit with an error.
6. If a new connection has been specified, validate that this device can be connected there. Do this by querying the **PdCn** object class for an object whose UniqueType descriptor matches the Link to the Predefined Devices Object Class descriptor of the parent's **CuDv** object, whose Connection Key descriptor matches the subclass name of the device being changed, and

whose Connection Location descriptor matches the new connection value. If no match is found, exit with an error.

If a match is found, the new connection is valid. If the device is currently available, then it should still be available after being moved to the new connection. Since only one device can be available at a particular connection, the **change** method will need to check for other available devices already at that connection. If one is found, exit with an error.

7. If the device state is Available and the **-P** flag was not specified, invoke the device's **unconfigure** method using the **odm_run_method** command. This fails if the device has available children, which is why the **change** method does not need to check explicitly for children.
8. Record new attribute values in the database. If parent or connection changed, update the Parent Device Logical Name, Location Where Connected on Parent Device, and Location Code descriptors of the device's **CuDv** object.
9. If the device state was Available before being unconfigured, invoke the device's **configure** method via the **odm_run_method** command. If this returns in error leaving the device unconfigured, you may want your **change** method to restore the Customized database for the device to its pre-change state.
10. Ensure that all object classes are closed and terminate the ODM. Exit with an exit code of 0 (zero) if there were no errors.

4. Handling Invalid Attributes

If the **change** method detects attributes that are in error, it must write information to the stderr file to identify them. This consists of writing the attribute name followed by the attribute description. Only one attribute and its description is to be written per line. If an attribute name was mistyped so that it does not match any of the device's attributes, write the attribute name supplied on a line by itself.

The **mkdev** and **chdev** configuration commands intercept the information written to standard error by the **change** method. They in turn write it out following an error message describing that there were invalid attributes. Both the attribute name and attribute description are needed to identify the attribute. If you invoked the **mkdev** or **chdev** command directly, you can recognize the attributes by attribute name. If you are using SMIT, these comands recognize attributes by description.

The attribute description is obtained from the appropriate message catalog. A message is identified by catalog name, set number, and message number. The catalog name and set number are obtained from the device's **PdDv** object. The message number is obtained from the NLS Index descriptor in either the **PdAt** or **CuAt** object corresponding to the attribute.

6.2.2.7 Writing Start/Stop Methods

1. Syntax

sttdev **-l name**

stpdev **-l name**
(where "dev" is the name of your device)

-l name Identifies the logical name of the device to be started or stopped.

2. Description

The **start** and **stop** methods are optional methods. (Most devices do not have **start** and **stop** methods.) The purpose of these methods is to allow the device to be put into a state where they are available or unavailable to users.

The **start** method takes the device from the Stopped state to the Available state. The **stop** method takes the device from the Available state to the Stopped state.

The Stopped state provides a state in which the device is configured in the system but unusable by applications. In this state, the device's driver is loaded and the device is defined to the driver. This might be implemented by having the **stop** method issue a command telling the device driver not to accept any normal I/O requests. If an application subsequently issues a normal I/O request to the device, it will fail. The **start** method can then issue a command to the driver telling it to start accepting I/O requests once again.

If you write **start** and **stop** methods for your device, your other methods must be written to account for the Stopped state. For instance, if one of your methods checks for a device state of Available, it might now need to check for both Available and Stopped states.

Additionally, write your **configure** method so that it takes the device from the Defined state to the Stopped state. However, you can have the **configure** method invoke the **start** method, thus taking the device to the Available state. The **unconfigure** method should be able to take the device to the Defined state from either the Available or Stopped states.

By convention, the first three characters of the name of the **start** method should be **stt**. The first three characters of the name of the Stop method should be **stp**. The remainder of the names can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the methods.

Start and **stop** methods, when they are used, are usually highly device-specific.

Configuration: What you need to do

The previous sections described what was needed to configure your device into the RISC/6000. The following is both a summary and an example of how to configure a new device into the system.

1. Create an ASCII file that will be used to update the ODM for the Predefined Devices (PdDv), Predefined Attributes (PdAt) and Predefined Connections (PdCn) object classes. For the following three steps, refer to "ODM Stanzas (ric.add)" on page B-22 for an example of how our stanzas were written.
2. Add the PdDv object class stanza. See Table B-1 on page B-2 for a table that lists the possible PdDv descriptors.
3. Add the PdAt object class stanza. See Table B-2 on page B-7 for a table that lists the possible PdAt descriptors.
4. Add the PdCn object class stanza. See Table B-3 on page B-10 for a table that lists the possible PdCn descriptors.
5. Add the information from the ASCII file into the ODM by using the **odmadd** command or by using the **odm_add_obj** subroutine.
6. Write the device configuration method. See "Writing a Configure Method" on page 6-23 for a description. Also see "Adapter Configuration Method (cfgrica.c)" on page B-28 for an example of the device configuration method that we wrote for the RIC device driver.
7. Write the device unconfiguration method. See "Writing an Unconfigure Method" on page 6-27 for a description. Also see /usr/lpp/bos/samples/ucfgxxx.c for an example of a device unconfiguration method. (Please note that this is not the one that we used for our device driver.)
8. Write the device define method. See "Writing a Define Method" on page 6-17 for a description. Also see /usr/lpp/bos/samples/defxxx.c for an example of a device define method. (Please note that this is not the one that we used for our device driver.)
9. Write the device undefine method. See "Writing an Undefine Method" on page 6-21 for a description. Also see /usr/lpp/bos/samples/udefxxx.c for an example of an device undefine method. (Please note that this is not the one that we used for our device driver.)
10. Write the device change method. This can be optional. For this device driver, we have not included it. Refer to "Writing a Change Method" on page 6-30 for a description. Also see /usr/lpp/bos/samples/chgyyy.c for an example.
11. Write the device start method. This is optional. For our example device driver, we have not included it.
12. Write the device stop method. This is optional. For our example device driver, we have not included it.

Configuration: How you actually do it

The following is a procedure for actually configuring your device:

1. Do all applicable adding of stanzas and writing of device methods as defined in the previous box.
2. Issue the "mkdev ...." command.

   Your device should now be defined and configured.

Chapter 7. SMIT Interface

7.1 Introduction

SMIT (System Management Interface Tool) is an interactive and extensible screen-oriented command interface. It prompts users for the information needed to construct command strings and presents appropriate predefined selections or run time defaults where available. This shields users from many sources of extra work or error, including the details of complex command syntax, valid parameter values, system command spelling, or custom shell path names.

New tasks consisting of one or more commands or inline ksh shell scripts can be added to SMIT at any time by adding new instances of predefined screen objects to SMIT's database. These screen objects (described by stanza files) are used by the ODM (Object Data Manager) to update SMIT's database. This database controls SMIT's run time behavior. Items that can be specified include:

- The sequence of screens presented to the user
- The data displayed for the user
- The method for generating default entry field values
- The input requested from the user
- The method in which user input is used to build and run auxiliary and task command strings.

You can also build and use alternate databases instead of modifying SMIT's default system database.

There are three main screen types that a user can traverse in order to perform a task, any of which can be optional in certain cases. These occur in a hierarchy consisting of **menu** screens, **selector** screens, and **dialog** screens. In performing a task, a user typically traverses one or more **menus**, then zero or more **selectors**, and finally one **dialog**.

Table 7-1. SMIT Screen Types

Screen type	What the user sees on the screen	What SMIT does internally
menu	a list of choices	uses the choice to select the next screen display
selector	either a list of choices or an entry field	obtains a datavalue for subsequent screen, optionally selects alternative dialogs or selectors
dialog	a sequence of entry fields	uses data from the entry fields to construct and run the target task command string

© Copyright IBM Corp. 1991

The table above shows SMIT screen types, what the user sees on each screen, and what SMIT does internally with each screen. **Menus** present a list of alternative subtasks; a selection can then lead to another menu screen, or to a selector or dialog screen. A **selector** is generally used to obtain one item of information that is needed by a subsequent screen and which can also be used to select which of several selector or dialog screens to use next. A **dialog** screen is where any remaining input is requested from the user and where the chosen task is actually run.

The Figure 7-1 shows some possible relationships among SMIT menus, selectors, and dialogs. A **menu** is the basic entry point into SMIT and can be followed by another menu, a selector, or a dialog. A **selector** can be followed by a dialog. A **dialog** is the final entry panel in a SMIT sequence.

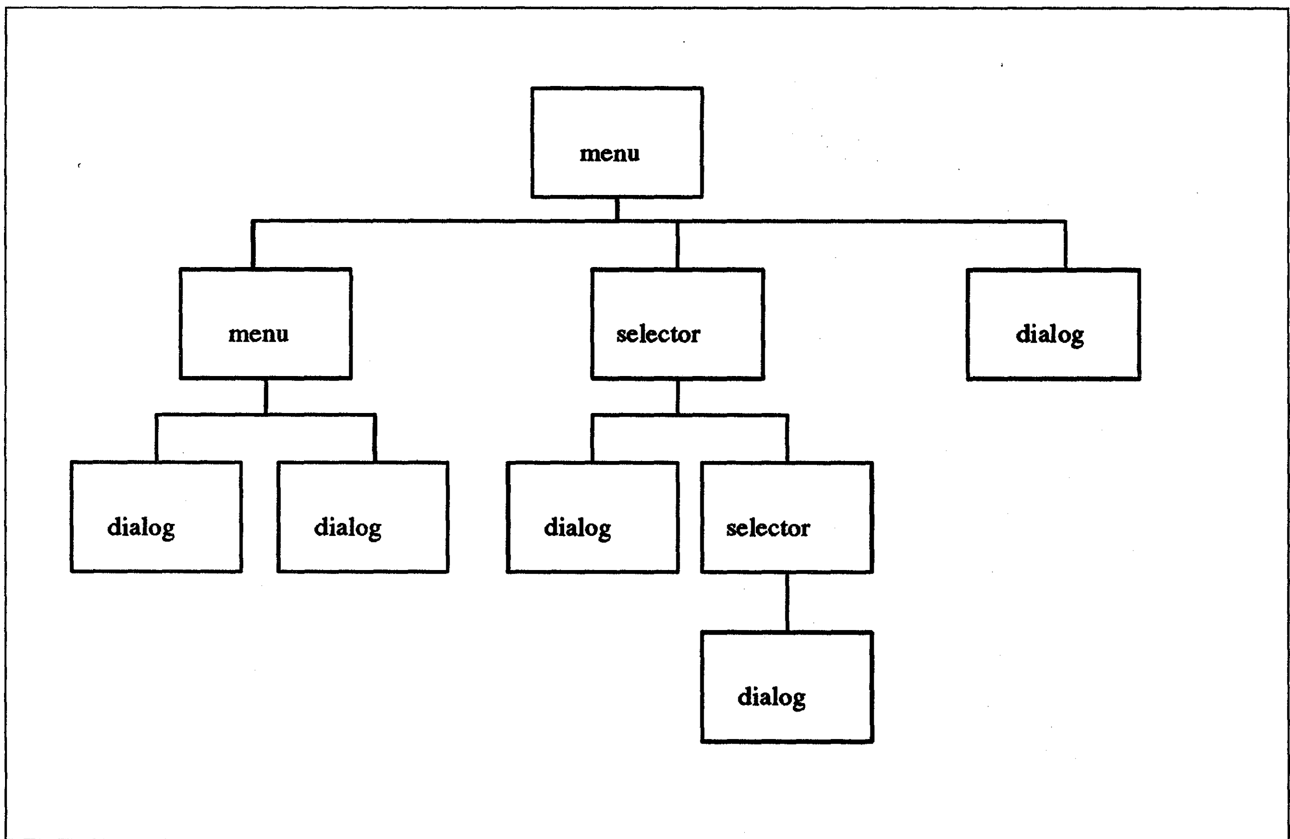


Figure 7-1. Some Relationships among SMIT Menus, Selectors and Dialogs

7.2 SMIT Screens

7.2.1 Menu Screens

A SMIT menu is a list of user-selectable items. Menu items are typically tasks or classes of tasks that can be performed from SMIT. A user starting with the main SMIT menu selects an item defining a broad range of system tasks. A selection from the next and subsequent menus progressively focuses the user's choice, until finally a dialog is typically displayed to collect information for performance of a particular task.

You design menus to help a user of SMIT narrow the scope of choice to a particular task. Your design can be as simple as a new menu and dialog attached to an existing branch of SMIT, or as complex as an entire new hierarchy of menus, selectors, and dialogs starting at the SMIT applications menu.

At run time, SMIT retrieves all menu objects with a given ID (ID descriptor value) from the specified object repository. Therefore, to add an item to a particular menu of SMIT, you add a menu object having an ID value equal to the value of the ID descriptor of other non-title objects in the same menu. See "Menu Object Class (sm_menu_opt)" on page C-1 for a detailed explanation of the class of menu objects.

7.2.2 Selector Screens

A SMIT selector prompts a user to specify a particular item, typically a system object (such as a printer) or attribute of an object (such as a printer mode of serial or parallel). This information is then generally used by SMIT in the following dialog.

You design a selector to request a single piece of information from the user. A selector, when used, falls between menus and dialogs. Selectors can be strung together in a series to gather several pieces of information before a dialog is displayed.

Selectors should usually contain a prompt displayed in user-oriented language and either a response area for user input or a pop-up list from which to select a value, i.e., one question field and one answer. Typically the question field is displayed and the SMIT user enters a value in the response area by typing the value or by selecting a value from a list or an option ring.

To give the user a run time list of choices, the selector object can have an associated command that lists the valid choices. The list is not hand-coded; it is developed by the command in conjunction with stdout. The user gets this list by invoking the **F4=List** function of the SMIT interface. See "Selector Header Object Class (sm_name_hdr)" on page C-3 for a detailed explanation of the selector object classes.

7.2.3 Dialog Screens

A dialog in SMIT is the interface to a command or task a user performs. Each dialog executes one or more commands, shell functions, and so on. A command can be run from any number of dialogs.

To design a dialog, you need to know the command string you want to build and the command options and operands for which you want user-specified values. In the dialog display, each of these command options and operands is represented by a prompt displayed in user-oriented language and a response area for user input. Each option and operand is represented by a dialog command option object in the ODM database. The entire dialog is held together by the dialog header object.

The SMIT user enters a value in the response area by typing the value, or by selecting a value from a list or an option ring. To give the user a run time list of choices, each dialog object can have an associated command that lists the valid choices. The user gets this list by invoking the **F4=List** function of the SMIT interface. See "Dialog Header Object Class (sm_cmd_hdr)" on page C-5 for a detailed explanation of the selector object classes.

7.3 SMIT Database

An object class created with ODM defines a common format or record data type for all individual objects that are instances of that object class. Therefore a SMIT object class is basically a record data type and a SMIT object is a particular record of that type.

SMIT **menu**, **selector**, and **dialog** screens are described by objects that are instances of one of four object classes:

- **sm_menu_opt**
- **sm_name_hdr**
- **sm_cmd_hdr**
- **sm_cmd_opt**

The following table shows the objects used to create each screen type:

Table 7-2. Object Used to Create Screens

Screen Type	Object Class	Object's Use (typical case)
menu	sm_menu_opt	1 for title of screen
	sm_menu_opt	1 for first item
	sm_menu_opt	1 for second item
	...	...
	sm_menu_opt	1 for last item
selector	sm_name_hdr	1 for title of screen and other attribute
	sm_cmd_opt	1 for entry field or pop-up list
dialog	sm_cmd_hdr	1 for title of screen and command string
	sm_cmd_opt	1 for first entry field
	sm_cmd_opt	1 for second entry field
	...	...
	sm_cmd_opt	1 for last entry field

Each object consists of a sequence of named fields and associated values. These are represented in stanza format in ASCII files that can be used by the odmadd command to initialize or extend SMIT databases. Stanzas in a file should be separated with one or more blank lines.

[1] **Note:** comments in an ODM input file (ASCII stanza file) used by the odmadd command must be alone on a line with a # (pound sign) in column one. A comment cannot be on the same line as a line of the stanza. Comments in the following examples that are at the end of a stanza line are there only to clarify this documentation; such comments should not be included in your object stanzas.

The following is an example [1] of a stanza for an **sm_menu_opt** object:

```
sm_menu_opt:                             #name of object class
   id              = "top_menu"  #object's (menu screen) name
   id_seq_num      = "050"
   next_id         = "commo"     #id of objects for next menu screen
   text            = "Communications Applications & Services"
   text_msg_file   = ""
   text_msg_set    = 0
   text_msg_id     = 0
   next_type       = "m"         #next_id specified another menu
   alias           = ""
   help_msg_id     = ""
   help_msg_loc    = ""
```

The notation **ObjectClass**.*Descriptor* is commonly used to describe the value of the fields of an object. For instance, in the preceeding **sm_menu_opt** object, the value of **sm_menu_opt**.*id* is "top_menu".

See "Menu Object Class (sm_menu_opt)" on page C-1 for a detailed explanation of each item in the **sm_menu_opt** object stanza.

The following is an example of a stanza for an **sm_name_hdr** object:

```
sm_name_hdr:                              #---- used for selector screens
   id                        = "   #the name of this selector screen
   next_id                   = ""  #next sm_name_hdr or sm_cmd_hdr screen object
   option_id                 = ""  #specifies one associated sm_cmd_opt object
   has_name_select           = ""
   name                      = ""  #title for this screen
   name_msg_file             = ""
   name_msg_id               = 0
   type                      = ""
   ghost                     = ""
   cmd_to_classify           = ""
   cmd_to_classify_postfix   = ""
   raw_field_name            = ""
   cooked_field_name         = ""
   next_type                 = ""
   help_msg_id               = ""
   help_msg_loc              = ""
```

See "Selector Header Object Class (sm_name_hdr)" on page C-3 for a detailed explanation of each item in the **sm_name_hdr** object stanza.

The following is an example of a stanza for an **sm_cmd_hdr** object:

```
sm_cmd_hdr:                           #---- used for dialog screens
   id                          = ""   #the name of this dialog screen
   option_id                   = ""   #defines associated set of sm_cmd_opt objects
   has_name_select             = ""
   name                        = ""   #title for this screen
   name_msg_file               = ""
   name_msg_set                = 0
   name_msg_id                 = 0
   cmd_to_exec                 = ""
   ask                         = ""
   exec_mode                   = ""
   ghost                       = ""
   cmd_to_discover             = ""
   cmd_to_discover_postfix     = ""
   name_size                   = 0
   value_size                  = 0
   help_msg_id                 = ""
   help_msg_loc                = ""
```

See "Dialog Header Object Class (sm_cmd_hdr)" on page C-5 for a detailed explanation of each item in the **sm_cmd_hdr** object stanza.

The following is an example of a stanza for an **sm_cmd_opt** object:

```
sm_cmd_opt:                       #---- used for both selector and dialog screens
   id                      = ""   #name of this object
   id_seq_num              = ""   #"0" if associated with selector screen
   disc_field_name         = ""
   name                    = ""   #text describing this entry
   name_msg_file           = ""
   name_msg_set            = 0
   name_msg_id             = 0
   op_type                 = ""
   entry_type              = ""
   entry_size              = 0
   required                = ""
   prefix                  = ""
   cmd_to_list_mode        = ""
   cmd_to_list             = ""
   cmd_to_list_postfix     = ""
   multi_select            = ""
   value_index             = 0
   disp_values             = ""
   values_msg_file         = ""
   values_msg_set          = 0
   values_msg_id           = 0
   aix_values              = ""
   help_msg_id             = ""
   help_msg_loc            = ""
```

See "Dialog/Selector Command Option Object Class (sm_cmd_opt)" on page C-7 for a detailed explanation of each item in the **sm_cmd_opt** object stanza.

All SMIT objects have an *"id"* field that provides a name used for looking up that object. The **sm_menu_opt** objects used for menu titles are also looked up using their *next_id* field. The **sm_menu_opt** and **sm_name_hdr** objects also have *next_id* fields that point to the *"id"* fields of other objects. These are how

the links between screens are represented in the SMIT database. Likewise, there is an *option_id* field in **sm_name_hdr** and **sm_cmd_hdr** objects that points to the *"id"* fields of their associated **sm_cmd_opt** object(s).

The Figure 7-2 on page 7-9 shows a hierarchy of **sm_menu_opt** objects and the menu screens displayed for these objects. Note that the value of each **sm_menu_opt**.*id* field that is part of the same menu screen is equal to the value of the immediately-preceding **sm_menu_opt**.*next_id* field (and this object serves as the title). This provides a link between a menu item and the items in a menu that immediately follow selection of the item.

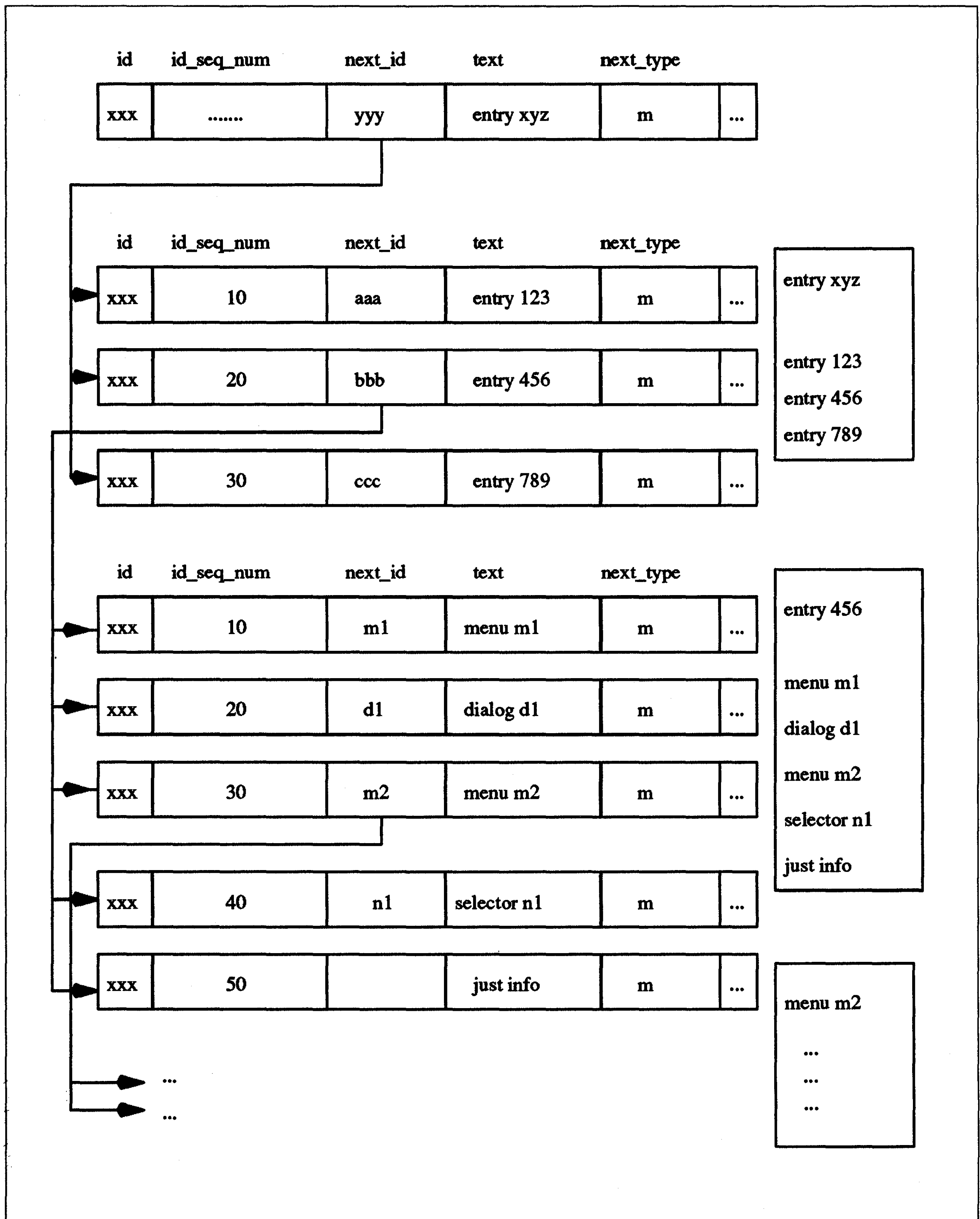


Figure 7-2. Hierarchy of sm_menu_opt Objects

The Figure 7-3 on page 7-10 shows a dialog using a **sm_cmd_hdr** object and three **sm_cmd_opt** objects, and the resulting dialog screen. Note that the **sm_cmd_hdr**.*option_id* object field is equal to each **sm_cmd_opt**.*id* object field;

this defines the link between the **sm_cmd_hdr** object and its associated **sm_cmd_opt** objects.

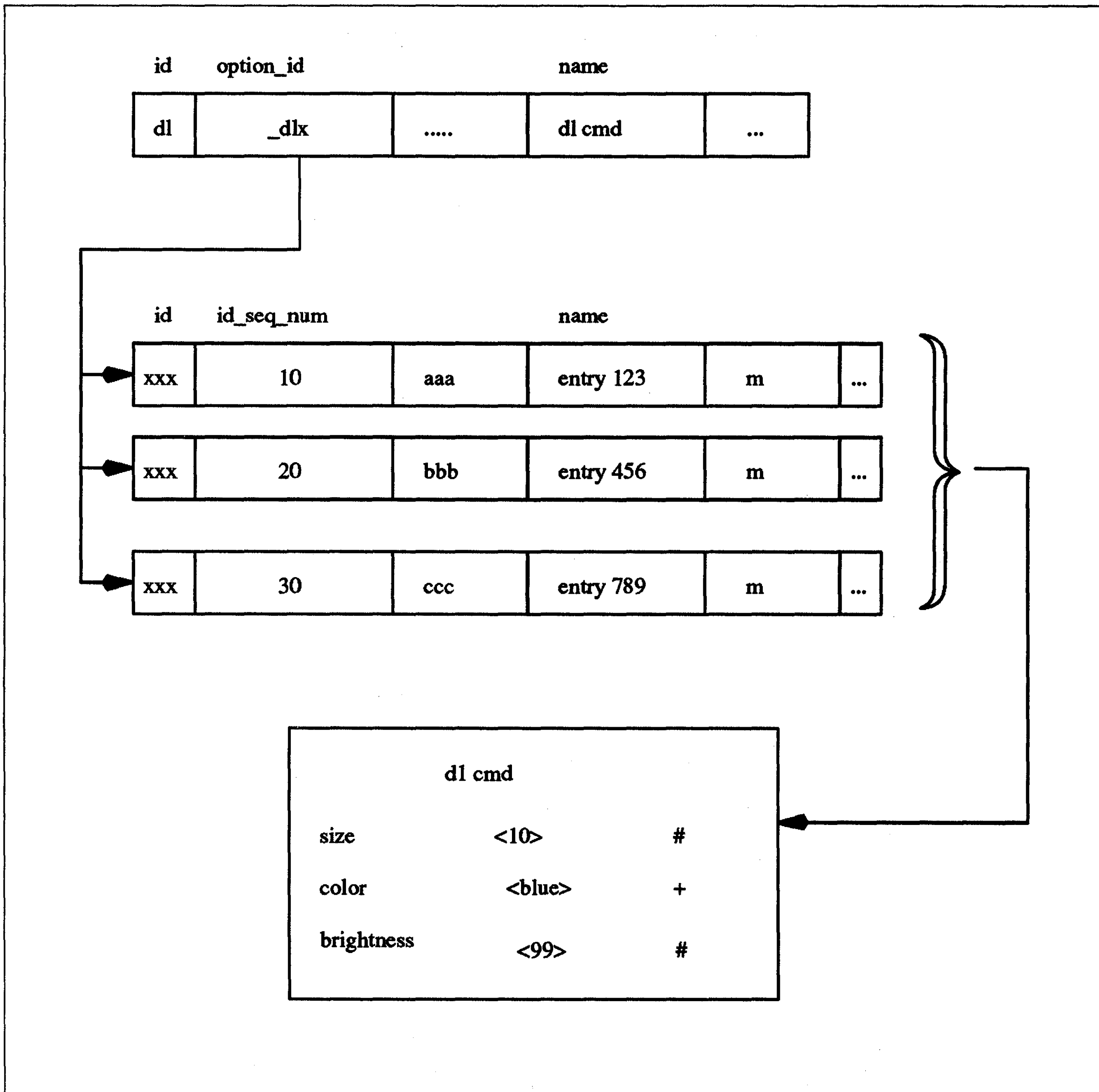


Figure 7-3. SMIT Dialogs

Two or more dialogs can share common **sm_cmd_opt** objects since SMIT uses the ODM LIKE operator to look up objects with the same **sm_cmd_opt**.*id* field values. SMIT allows up to five IDs (separated by commas) to be specified in a **sm_cmd_hdr**.*option_id* field, so that **sm_cmd_opt** objects with any of five different **sm_cmd_opt**.*id* field values can be associated with the **sm_cmd_hdr** object.

The following table shows how the value of an **sm_cmd_hdr**.*option_id* field relates to the values of **sm_cmd_opt**.*id* and **sm_cmd_opt**.*id_seq_num* fields. Note that the values in the **sm_cmd_opt**.*id_seq_num* fields are used to sort the retrieved objects for screen display.

Table 7-3. sm_cmd_hdr Relationships		
IDs of the Objects to Retrieve (sm_cmd_hdr.option_id)	**Objects Retrieved (sm_cmd_opt.id)**	**Display Sequence of Retrieved Objects (sm_cmd_opt.id_seq_num)**
"demo.[AB]"	"demo.A"	"10"
	"demo.B"	"20"
	"demo.A"	"30"
	"demo.A"	"40"
"demo.[ACD]"	"demo.A"	"10"
	"demo.C"	"20"
	"demo.A"	"30"
	"demo.A"	"40"
	"demo.D"	"50"
"demo.X,demo.Y,demo.Z"	"demo.Y"	"20"
	"demo.Z"	"40"
	"demo.X"	"60"
	"demo.X"	"80"

SMIT objects are generated with ODM creation facilities and stored in files in a designated database. The default SMIT database consists of eight files:

1. The sm_menu_opt file
2. The sm_menu_opt.vc file
3. The sm_name_hdr file
4. The sm_name_hdr.vc file
5. The sm_cmd_hdr file
6. The sm_cmd_hdr.vc file
7. The sm_cmd_opt file
8. The sm_cmd_opt.vc file.

The files are stored by default in the */etc/objrepos* directory. They should always be saved and restored together.

7.4 Command Building and Running

Each dialog in SMIT builds and executes a version of a standard command. The command to be executed by the dialog is defined by the *cmd_to_exec* descriptor in the **sm_cmd_hdr** object that defines the dialog header.

7.4.1 Task Building

In building the command defined in an **sm_cmd_hdr**.*cmd_to_exec* descriptor, SMIT uses a two-pass scan over the dialog's set of **sm_cmd_opt** objects to collect prefix and parameter values. The parameter values collected include those that the user changed from their initially displayed values and those with the **sm_cmd_opt**.*required* descriptor set to "y".

The first pass gathers all of the values of the **sm_cmd_opt** objects (in order) for which the prefix descriptor is either an empty string ("") or starts with a **-** (a dash, such as with a flag: **-f**). These parameters are not position-sensitive and are added immediately following the command name, together with the contents of the prefix descriptor for the parameter.

The second pass gathers all of the values of the remaining **sm_cmd_opt** objects (in order) for which the prefix descriptor is **--** (two dashes). These parameters are position-sensitive and are added after the flagged options collected in the first pass.

Command parameter values in a dialog are filled in automatically when the *disc_field_name* descriptors of its **sm_cmd_opt** objects match names of values generated by preceding selectors or a preceding discovery command. These parameter values are effectively default values and are normally not added to the command line. Initializing an **sm_cmd_opt**.*required* descriptor to "y" or "+" causes these values to be added to the command line even when they are not changed in the dialog. These parameter values are built into the command line as part of the regular two-pass process.

Leading and trailing white space (spaces and tabs) are removed from parameter values except when the **sm_cmd_opt**.*entry_type* descriptor is set to "r". If the resulting parameter value is an empty string, no further action is taken unless the **sm_cmd_opt**.*prefix* descriptor starts with an option flag. Surrounding single quotation marks are added to the parameter value if the prefix descriptor is not set to "--" (two dashes). Each parameter is placed immediately after the associated prefix, if any, with no intervening spaces. Also, if the *multi_select* descriptor is set to "m", tokens separated by white space in the entry field are treated as separate parameters.

7.4.2 Command Execution

SMIT runs the command string specified in a **sm_cmd_hdr**.*cmd_to_exec* descriptor by first creating a child process. The stderr (standard error) and stdout (standard output) of the child process are handled as specified by the contents of the **sm_cmd_hdr**.*exec_mode* descriptor. SMIT next runs a setenv("ENV=") subroutine in the child process to prevent commands specified in the $HOME/.env file of the user from being run automatically when a new shell is invoked. Finally, SMIT calls the execl subroutine to start a ksh shell, using the command string as the ksh -c parameter value.

You can override SMIT default output redirection of the (child) task process by setting the **sm_cmd_hdr**.*exec_mode* field to "i". This setting gives output management control to the task, since the task process simply inherits the standard error and standard output file descriptors.

You can cause SMIT to shut down and replace itself with the target task by setting the **sm_cmd_hdr**.*exec_mode* field to "e".

7.5 Dialogs Example

7.5.1 List All Defined Ric Ports

First the **Real Time Interface Co-Processor Adapter** *menu* is called (from "System Management"/"Devices"/"Communication Devices"). The following screen will be another menu (next_type = m), and the ID will be **ric** (next_id field).

```
sm_menu_opt:
    id                  = "commodev"
    id_seq_num          = "070"
    next_id             = "ric"
    text                = "Realtime Interface Co-Processor Portmaster Adapter"
    text_msg_file       = ""
    text_msg_set        = 0
    text_msg_id         = 0
    next_type           = "m"
    alias               = ""
    help_msg_id         = ""
    help_msg_loc        = ""
```

On the screen, you will see the different choices (see Figure 7-4). The corresponding objects all have the same ID (**ric**), but with different **id_seq_num**.

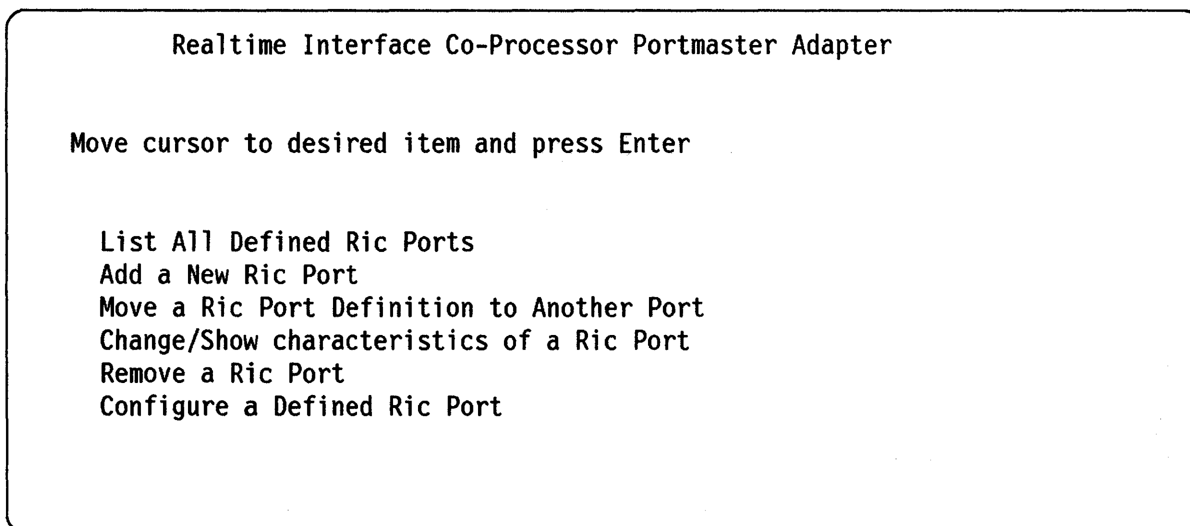


Figure 7-4. SMIT Screen Example

They are shown on the screen in ascending order according to their **id_seq_num**.

The first one is:

```
sm_menu_opt:
    id                       = "ric"
    id_seq_num               = "010"
    next_id                  = "lsdric"
    text                     = "List All Defined Ric Ports"
    text_msg_file            = ""
    text_msg_set             = 0
    text_msg_id              = 0
    next_type                = "d"
    alias                    = ""
    help_msg_id              = ""
    help_msg_loc             = ""
```

The following stanza represents the final dialog screen of this chain. It is a *ghost* dialog, which means that there is actually no new screen displayed; it just executes the **lsdev -C -c ricport -H** command.

```
sm_cmd_hdr:
    id                       = "lsdric"
    option_id                = ""
    has_name_select          = "n"
    name                     = "List All Defined Ric Ports"
    name_msg_file            = "ric.cat"
    name_msg_set             = 3
    name_msg_id              = 0
    cmd_to_exec              = "lsdev -C -c ricport -H"
    ask                      = ""
    exec_mode                = ""
    ghost                    = "y"
    cmd_to_discover          = ""
    cmd_to_discover_postfix  = ""
    name_size                = 0
    value_size               = 0
    help_msg_id              = ""
    help_msg_loc             = ""
```

7.5.2 Add a Ric Port

The beginning of this chain is the same as in the previous example, up to the **commodev** menu (with **id** = 070). We start here with the next available choice on the screen: to add a port. This object has **id_seq_num** = 020. Note that the next object is a dialog (**next_type** = d) with **id** = makric.

```
sm_menu_opt:
    id                       = "ric"
    id_seq_num               = "020"
    next_id                  = "makric"
    text                     = "Add a Ric Ports"
    text_msg_file            = ""
    text_msg_set             = 0
    text_msg_id              = 0
    next_type                = "n"
    alias                    = ""
    help_msg_id              = ""
    help_msg_loc             = ""
```

The following object is a ghost selector with one option (second stanza). It is used to put up a list of defined ric adapters for the user to select from:

```
sm_name_hdr:
    id                        = "makric"
    next_id                   = "makric_hdr"
    option_id                 = "ric_mk_parent"
    has_name_select           = "n"
    name                      = "Add a Ric Port"
    name_msg_file             = "ric.cat"
    name_msg_set              = 3
    name_msg_id               = 1
    type                      = ""
    ghost                     = "y"
    cmd_to_classify           = ""
    cmd_to_classify_postfix   = ""
    raw_field_name            = "parent"
    cooked_field_name         = ""
    next_type                 = "d"
    help_msg_id               = ""
    help_msg_loc              = ""

# Name selector command option for parent adapter
sm_cmd_opt:
    id                        = "ric_mk_parent"
    id_seq_num                = "0"
    disc_field_name           = ""
    name                      = "Parent Adapter"
    name_msg_file             = "ric.cat"
    name_msg_set              = 3
    name_msg_id               = 8
    op_type                   = "l"
    entry_type                = "t"
    entry_size                = 0
    required                  = "y"
    prefix                    = ""
    cmd_to_list_mode          = "1"
    cmd_to_list               = "lsparent -C -k ricp"
    cmd_to_list_postfix       = ""
    multi_select              = ""
    value_index               = 0
    disp_values               = ""
    values_msg_file           = ""
    values_msg_set            = 0
    values_msg_id             = 0
    aix_values                = ""
    help_msg_id               = ""
    help_msg_loc              = ""
```

The next five stanzas represent a dialog which puts up a list of four user configurable attributes. It then executes the **mkdev** command to create the port.

```
sm_cmd_hdr:
    id                          = "makric_hdr"
    option_id                   = "ric_add,ric_common"
    has_name_select             = "y"
    name                        = "Add a Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 1
    cmd_to_exec                 = "mkdev -c ricport -s ricp -t port "
    ask                         = ""
    exec_mode                   = ""
    ghost                       = "n"
    cmd_to_discover             = "lsattr -c ricport -s ricp -t port -D -O"
    cmd_to_discover_postfix     = ""
    name_size                   = 0
    value_size                  = 0
    help_msg_id                 = ""
    help_msg_loc                = ""

# Displays rdto attribute.
sm_cmd_opt:
    id                          = "ric_common"
    id_seq_num                  = "010"
    disc_field_name             = "rdto"
    name                        = "RECEIVE DATA TRANSFER OFFSET"
    name_msg_file               = "ric.cat"
    name_msg_set                = 2
    name_msg_id                 = 2
    op_type                     = "l"
    entry_type                  = "#"
    entry_size                  = 0
    required                    = "n"
    prefix                      = "-a rdto="
    cmd_to_list_mode            = "r"
    cmd_to_list                 = "lsattr -c ricport -s ricp -t port -a rdto -R"
    cmd_to_list_postfix         = ""
    multi_select                = "n"
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
# Displays autoconfig attribute.
sm_cmd_opt:
    id                      = "ric_common"
    id_seq_num              = "020"
    disc_field_name         = "autoconfig"
    name                    = "STATE to be configured at boot time"
    name_msg_file           = "ric.cat"
    name_msg_set            = 2
    name_msg_id             = 3
    op_type                 = "l"
    entry_type              = "t"
    entry_size              = 0
    required                = "n"
    prefix                  = "-a autoconfig="
    cmd_to_list_mode        = "1"
    cmd_to_list             = "lsattr -c ricport -s ricp -t port -a autoconfig -R"
    cmd_to_list_postfix     = ""
    multi_select            = "n"
    value_index             = 0
    disp_values             = ""
    values_msg_file         = ""
    values_msg_set          = 0
    values_msg_id           = 0
    aix_values              = ""
    help_msg_id             = ""
    help_msg_loc            = ""

# Displays ric port's parent adapter.
sm_cmd_opt:
    id                          = "ric_add"
    id_seq_num                  = "001"
    disc_field_name             = "parent"
    name                        = "Parent Adapter"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 8
    op_type                     = ""
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "y"
    prefix                      = "-p "
    cmd_to_list_mode            = ""
    cmd_to_list                 = ""
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
# Displays physical port number being defined.
sm_cmd_opt:
    id                          = "ric_add"
    id_seq_num                  = "002"
    disc_field_name             = ""
    name                        = "PORT number"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 9
    op_type                     = "l"
    entry_type                  = "t"
    entry_size                  = 0
    required                    = "+"
    prefix                      = "-w "
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "lsconn -k ricp "
    cmd_to_list_postfix         = "-p parent"
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

7.6 Additions to the SMIT Database

7.6.1 Database Creation

Whenever you are developing new objects for the SMIT database, it is recommended that you set up a separate test database for development.

TESTING HINT

To create a test database, do the following:

1. Create a directory for testing use. For example, the following command creates a /u/smit/test directory:

   ```
   mkdir /u/smit /u/smit/test
   ```

2. Make the test directory the current directory:

   ```
   cd /u/smit/test
   ```

3. Define the current directory as the default object repository by setting the ODMDIR environment variable to "."

   ```
   export ODMDIR=.
   ```

4. Take a copy of all the SMIT object classes from */etc/objrepos*.

   ```
   cp /etc/objrepos/sm_* .
   ```

You have now a copy of the original database on which you can try your additions without destroying anything. You could also create an empty database to test your dialogs with the following command (to be used instead of Step 4):

```
odmcreate -c /usr/lpp/smit/samples/smit_class.cre
```

7.6.2 SMIT Extensions Debugging

You can test and debug newly-created SMIT menus, selectors, and dialogs by starting and running SMIT with one or more of the following command options, as appropriate:

v Produces a more verbose log file showing command strings and output from auxiliary commands.

t Provides more information about what objects were read from the SMIT database.

o Specifies a directory for use as an alternate SMIT object database. For example, you can start SMIT with the command smit -o. to specify the current directory.

l Specifies an alternate log file path. For example, you can enter the command **smit -l log** to start SMIT and specify a short file name in the current directory for use as the log file for the session.

x Prevents the command string in the *command_to_exec* field from being run. This can be useful when the target command has a major impact on the system, because SMIT logs, but does not run, the command string.

7.6.3 Task Additions

To add tasks to the SMIT database :

1. Design the **dialog** for the command you want SMIT to build.
2. Design the hierarchy of **menus** and, optionally, **selectors**, needed to get a SMIT user to the **dialog**, and determine where and how this hierarchy should be linked into the existing SMIT database. The following strategy may save you time if you are developing SMIT database extensions for the first time:
   a. Start SMIT (run the **smit** command); look for existing **menu**, **selector**, and **dialog** screens that perform tasks similar to the one you want to add, and find the menu screen(s) to which you will add the new task.
   b. Exit from SMIT, then remove the existing SMIT log file. Instead of removing the log file, you can use the **-l** flag of the smit command to specify a different log file when starting SMIT in the following step. This enables you to isolate the trace output of your next SMIT session.
   c. Start SMIT again with the **-vt** command flags and again look at the screen(s) to which you will add the new task. This logs the object IDs accessed for each screen for the next step.
   d. Look at the SMIT log file to determine the ID for each object class used as part of the menu(s) (see "SMIT Log File" on page C-35 for a trace of the example exposed in "List All Defined Ric Ports" on page 7-14).
   e. Use the object class IDs with the **odmget** command to retrieve the stanzas for these objects. The stanzas can be used as rough examples to guide your implementation and to learn from the experience of others.
   f. Look in the SMIT log file for the command strings used when running through the screens to see if special tools are being utilized behind the scenes (such as sed or awk scripts, ksh shell functions, environment variable assignment, and so on). When entering command strings, keep in mind that they are processed twice: the first time by the **odmadd** command and the second time by the ksh shell. Therefore, be careful when using special escape meta-characters (such as \) or quotation characters (′ and ″). Note also that the output of the odmget command does not always match the input to the **odmadd** command, especially when these characters or multi-line string values are used.
3. Code the **dialog**, **menu**, and **selector** objects by defining them in the ASCII object stanza file format required by the **odmadd** command. For examples of stanzas used to code SMIT objects, see "List All Defined Ric Ports" on page 7-14, "Add a Ric Port" on page 7-15, and "ODM Stanzas for Ric Dialogs (sm_ric.add file)" on page C-15.
4. Add the **dialog**, **menu**, and **selector** objects to the SMIT test database with the **odmadd** command, using the name of your ASCII object stanza file in place of *test_stanzas*:

   ```
   odmadd test_stanzas
   ```

5. Test and debug your additions by running SMIT using the local test database:

   ```
   smit -o.
   ```

Chapter 8. Device Drivers Packaging

8.1 Introduction

The objective of the installp procedure is to make installation of application programs as painless as possible, while maintaining a high degree of functionality. This is certainly something highly desirable for a device driver installation where the following tasks have to be done:

- Installation of the device driver code under */etc/drivers*.
- Installation of the different methods under */etc/methods*.
- Populating of different Predefined object classes.
- Installation of new SMIT dialogs.
- Optionally, installation of error and trace report templates.

In this chapter, we will present a summary of the installp procedure, illustrated with the files necessary to install the **ric** device driver used as an example in this book.

8.2 Design Guidelines

The installation procedures for any two application programs are never exactly the same. However, there are general guidelines that should be used by every installation procedure:

- Ensure that the **prereq** file is set correctly for the order in which installation procedures should occur.
- Require minimal user interaction.
- Select least disruptive state that permits installation.
- Ensure that the system is in the required state. Reject the installation if the state is not correct.
- Make sure error recovery is comprehensive.

8.3 The installp Command

Usage:

installp [**-d Device**] [**-F**] [**-X**] {[**-f File**] | **Options**}
installp -l [**-d Device**]
installp -C

The command has a standard interface through SMIT. The following tools are available to installp programmers:

inuumsg for displaying messages.

inurest for restoring files from medium.

© Copyright IBM Corp. 1991

ckprereq for checking prerequisites.

sysck for entering inventory information.

8.4 Ensuring installp Command Compatibility

For programs to be installed, they must be compatible with the installp command. This command implements the following processing for an application program:

- Restores list of program options from distribution medium (*./lpp_name* file).
- Verifies that specified program options exist.
- Checks the level of the program to be installed.
- Restores files that installp needs from medium.
- Checks that disk space suffices for specified options.
- Echoes copyright file.
- Executes the instal script file (**instal Device Filename**), where **Device** is the device name to pass to the installp command and **Filename** is the name of the file that contains the list of installable options to pass to the installp command.
- Checks the status of each option's installation: If instal is successful, updates the Vital Product Data (VPD) database using information from the **lpp_name** file and the **prereq** file; otherwise, execute the lpp.cleanup file (**lpp.cleanup Device Filename**), where **Device** is the device name passed to the installp command and **Filename** is the name of the file that contains the list of installable options that failed.
- Delete installp files from *./usr/lpp/Program* (all files without **lpp.** prefix, subdirectories excepted).

8.5 Files for installp Operation

NOTE

We give here a list of the most commonly used files for installp operations, but a complete list of required and optional files for **installp** and **updatep** commands can be found in "Installp/Updatep Files" on page **D-1**.

./lpp_name (required)
list of program options

./usr/lpp/liblpp.a (required)
archive containing installp files:

instal (required)
script to perform installation

config or Option.config (optional)
script for configuring program

prereq or Option.prereq (optional)
list of prerequisites for program

lpp.doc (optional)
document pages for the program

Filename.err (optional)
error report templates

Filename.trc (optional)
trace report templates

Filename.evt (optional)
trace event types

al or Option.al (required)
apply list (of files in program option)

lpp.acf (optional)
names of files and libraries updated by installation

copyright (required)
can be empty

productid (optional)
line of text to enter into VPD database

size or Option.size (required)
size file (of each file in option)

inventory (optional)
information about each application file

lpp.cleanup (required)
script to clean up after failure of an option

lpp.deinst (optional)
script for de-installing program.

Other files
Any other file that needs to be updated, named relative to the top of the file system (for example *./usr/lib/sendmail*).

8.5.1 LPP Option List File: lpp_name

```
Syntax of entries

<format> <platform> <medium_type> {
<Program> <level> <vol> <quiesce> <type> <lang> <descr> #<comment>
<Program> <level> <vol> <quiesce> <type> <lang> <descr> #<comment>
.
.
}

<format>      = 1
<platform>    = R
<medium_type> = I (installation) or M (multiple updates)

<Program> = Program or Option name (string)
<level>   = version, release, modification and fix levels (string)
<vol>     = volume number where files for this option are located
<quiesce> = Y (subsystem should be stopped prior to installation)
          = N (not necessary to stop subsystem)
<type>    = type of information on medium (code/documentation/...)
<lang>    = NLS language token of language used
```

```
<descr>   = description of Program Option (string)
<comment> = everything following a # sign after <descr>
```

8.5.2 Instal Script

In order to be compatible with the installp command, the user-provided instal script file for an application program must perform the following procedures:

- Verify compatibility of the program (*Program* file) or program option (*Program.Option* file) with other installed programs via the **ckprereq** command.
- Perform pre-installation processing.
- Restore all required files from the medium via the **inurest** command.
- Execute the configuration procedure (*config* file) if it exists.
- Create a status file indicating success or failure for each program option.
- Return an exit code indicating the status of the installation or update.

8.5.3 al (Option.al)

This file contains a list of files for program (Program.Option), one name per line, with path given relative to root.

8.5.4 size (Option.size)

This file contains a list of major directories and how many 512-byte blocks are required by program (Program.Option) in each directory.

8.5.5 copyright

This file contains any ASCII text.

8.5.6 lpp.cleanup

This file is a script or executable to handle failed installation attempt. It must at least remove files that were restored, but other actions may be necessary (e.g., undoing changes to files).

8.5.7 Prereq (Option.prereq)

This file contains a list of prerequisites for installing program (Program.Option). Two types of prerequisites are possible:

1. Required program level prerequisite

   ```
   <prog-name> <lev-expr> <lev-expr> ...
   ```

   where <lev-expr> is **v** (version), **r** (release), **m** (modification), or **f** (fix), followed by an equality or inequality sign, and a number. For example :

   ```
   database v=1 r>3
   ```

   means that Version 1, Release 4 or later of the program database is required.

2. Relational prerequisites

```
> <int> {
<prereq type 1>
<prereq type 1>
.
.
}
```

For example:

```
>1 {
Prog1 v=2
Prog2 v>3
}
```

means that Prog1, Version 2, and Prog2, Version 4 or later are required.

8.5.8 config (Option.config)

This file is an executable for configuring system after installation (this is done after error templates are updated).

8.5.9 lpp.deinst

This file is a script for manually de-installing program. It must restore prior state which may require saving copies of files that have been modified for installation (in this case, the size of these files must be included in the size file).

8.5.10 inventory (Option.inventory)

This is a highly recommended file which contains specific information about each file in the program (Program.Option). It is an ASCII text in stanza format, to be entered into the VPD database. Installp automatically calls sysck to do this.

8.5.11 productid

This file contains the part number of the program. It contains one line of text to be entered into the VPD database at the time of installation.

8.5.12 lpp.acf

The archive control file is needed whenever the application adds to or modifies a library owned by another application. It is recommended that files to be archived in the library */LibPath/Lib* be placed in */LibPath/inst_updt/Lib*. The files are entered into lpp.acf as follows:

```
Filename ArchiveName
```

where **Filename** is the complete path name of the file (relative to root), and **ArchiveName** is the complete path name of the archive where the file is to be archived. These names are to be separated by one or more white spaces, and each entry must occur on a new line.

8.6 Installp Example

8.6.1 Introduction

As an example, we have packaged the device driver for the Real Time Interface Co-processor Adapter. The package contains two options. First, it is possible to install the device driver (ricdd), and the configuration methods (cfgrica and cfgricp), and update the ODM database from stanza files (ric.add for PdDv, PdAt and PdCn, and sm_ric.add for the SMIT dialogs). The second option consists of the sources of the device driver and the methods. If installed, they are put respectively in the directories */usr/lpp/ricdd/src/driver* and */usr/lpp/ricdd/src/methods*.

Most of the files necessary for the package are listed in "Real Time Interface Co-Processor Device Driver Package" on page D-16. In the same section, you will also find a listing of the **Makefile** used to create the package from the individual files. To use it, you need to put all the files necessary to create the liblpp.a archive file used by the installp command in the same directory as the Makefile. The files to be installed (and lpp_name) should be in another directory (referenced as $(ROOT) in the Makefile). For convenience, this directory is a sub-directory named root under the first one.

8.6.2 How to Use the Makefile

In order to generate a package, you need first to modify a certain number of variables inside the Makefile:

1. DEV: points to the destination of image of distribution medium.
2. PROG: name of application program (ricdd).
3. ROOT: application files directory. It is assumed that every file below $ROOT other than $ROOT/lpp_name and $ROOT/usr/lpp/$PROG/liblpp.a is part of the application program package.

The Makefile needs the following files:

1. lpp_name (the list of options) (must be in directory $(ROOT))
2. instal (installation script)
3. lpp.cleanup and Option.cleanup (cleanup scripts if installation fails)
4. al or Option.al (apply list for each option)
5. size or Option.size (size of file for each option).

If the package to build has different options, those options must be listed, one per line in a file called *Options*. Thus, for this package, we have an *Options* file:

```
ricdd.src
ricdd.driver
```

an *Option.al* (i.e. *ricdd.driver.al* and *ricdd.src.al*) and an *Option.size* (i.e. *ricdd.driver.size* and *ricdd.src.size*) files. The Makefile is also ready for a package with no options (no *Option* file, but an *al* and a *size* file).

The Makefile will try to build files that could be missing. It knows how to make *al* (everything under $(ROOT)), and can calculate *size* from *al* or *Option.size* or from *Option.al*. In this case the files *ricdd.driver.size* and *ricdd.src.size* are therefore optional.

The following two files are also required for an installp package, but Makefile knows how to supply them:

1. liblpp.a (archive containing all installp files except lpp_name): Makefile will create or remake it if it is out of date.
2. copyright: if missing, Makefile will supply an empty file.

It is possible to add a certain number of optional files. In this example, we have a *config* file (configuration of the installed package, named *ricdd.driver.config*), *prereq* (list of prerequisites), and *lpp.deinst* (a deinstallation script). Both *ricdd.driver.config* and *lpp.deinst* are listed in "Real Time Interface Co-Processor Device Driver Package" on page D-16, while *prereq* contains the following lines:

```
bos.obj v>0
bosext2.games.obj v>0
```

In other words, we need the Base Operating System, and the Games to be installed before we can install the ricdd package.

All the optional files to be included in distribution must be listed in the file *optionalfiles*, one per line. The file used in this case thus contains three lines with:

```
prereq
lpp.deinst
ricdd.driver.config
```

Finally, you need the following files to be under the *root* sub-directory:

```
./root:
etc
lpp_name
usr

./root/etc:
drivers
methods

./root/etc/drivers:
ricdd
```

```
./root/etc/methods:
cfgrica
cfgricp

./root/usr:
lpp

./root/usr/lpp:
ricdd

./root/usr/lpp/ricdd:
liblpp.a
ric.add
ric.msg
sm_ric.add
src

./root/usr/lpp/ricdd/src:
driver
methods

./root/usr/lpp/ricdd/src/driver:
ric.h
ricmisc.h
ricstruct.h
riccfg.c
ricdd.c
ricutil.c

./root/usr/lpp/ricdd/src/methods:
Makefile
cfgrica.c
cfgricp.c
debug.h
ric.add
ric.h
ric.msg
ricmisc.h
ricstruct.h
sm_ric.add
```

Then, just type "make". The package will be created on the specified medium.

8.6.3 Root/lpp_name File

```
1 R I {
ricdd.src 01.00.0000.0000 1 N O US_ENG ric device driver sources
ricdd.driver 01.00.0000.0000 1 N O US_ENG ric device driver
}
```

8.6.4 Apply List Files

8.6.4.1 Ricdd.driver.al

```
./etc/drivers/ricdd
./etc/methods/cfgrica
./etc/methods/cfgricp
./usr/lpp/ricdd/ric.add
./usr/lpp/ricdd/sm_ric.add
./usr/lpp/ricdd/ric.msg
```

8.6.4.2 Ricdd.src.al

```
./usr/lpp/ricdd/src/methods/Makefile
./usr/lpp/ricdd/src/methods/cfgrica.c
./usr/lpp/ricdd/src/methods/cfgricp.c
./usr/lpp/ricdd/src/methods/debug.h
./usr/lpp/ricdd/src/methods/ric.add
./usr/lpp/ricdd/src/methods/sm_ric.add
./usr/lpp/ricdd/src/methods/ric.msg
./usr/lpp/ricdd/src/driver/ric.h
./usr/lpp/ricdd/src/driver/ricmisc.h
./usr/lpp/ricdd/src/driver/ricstruct.h
./usr/lpp/ricdd/src/driver/riccfg.c
./usr/lpp/ricdd/src/driver/ricdd.c
./usr/lpp/ricdd/src/driver/ricutil.c
```

INSTALL PACKAGE EXAMPLE

Please see "Real Time Interface Co-Processor Device Driver Package" on page D-16 for an example of the install package that we have done for the RIC device driver.

Chapter 9. Tools for Debugging Device Drivers

9.1 Debugging Overview

This chapter provides information regarding the available procedures for debugging a device driver which is under development. The procedures discussed include:

- How to save device driver information in a system dump
- How to use the "crash" command to interpret and format system structures
- How to use the kernel debugger to set breakpoints and display variables and registers
- How to use Trace to monitor entry/exit of device drivers and selectable system events
- Error logging to record device-specific hardware or software abnormalities.

The system kernel dump routine contains all the vital structures of the running system, such as the process table. the kernel's global memory segment, each process' data segment and stack segment. Be sure to refer to the source of the system header files in /usr/include/sys. The various system structures are defined in files with a suffix of ".h". The name of the file tells which structure and associated information it contains. For example, the user block is defined in /usr/include/sys/user.h. The process block is defined in /usr/include/sys/proc.h. When you examine system data which maps into these structures, you will be able to gain valuable kernel information that may explain why the dump was called.

9.2 System Dump

The system dump copies selected kernel structures to the dump device when an unexpected system halt occurs, when the reset button is pressed, or when the special system dump key sequences are entered. You can also initiate a system dump through SMIT. The dump device can be dynamically configured, which means that either the sysdumpdev, tape or logical volumes on hard disk can be used to receive the system dump. Dynamic configuration can be achieved using the **sysdumpdev** command. Primary and secondary dump devices can be defined. A primary dump device is a dedicated dump device, while a secondary dump device is a shared one.

9.2.1 Initiating a System Dump

A system dump initiated by a kernel panic is written to the primary dump device. If you initiate a system dump by pressing the reset button (be sure the key is in the service position!), the system dump is written to the primary dump device. You can determine whether the write of a system dump goes to the primary dump device or to the secondary dump device by use of the special key sequences. To use the special key sequences the key must be in the service position. To write to the primary dump device, use the sequence **<Ctrl> <Alt> <NumPad 1>**. To write to the secondary dump device, use the sequence **<Ctrl> <Alt> <NumPad 2>**.

© Copyright IBM Corp. 1991

To use SMIT, select **Problem Determination** from the main menu, then select **System Dump**. This presents a menu which allows you to not only elect to initiate a system dump to either the primary or secondary device, but also options for manipulating the dump devices and the system dump files. If you prefer to initiate the system dump from the command line, use the **sysdumpstart** command. This command used with the **-p** flag will write to the primary device, while the **-s** flag will write to the secondary device.

Note: The system halts after system dump completes.

9.2.2 Including Device Driver Information in a System Dump

The system dump is table driven. It consists of a **master dump table** and a **component dump table**. A master dump table entry is a pointer to a function which is provided by the device driver. The function is called by the kernel dump routine when a system dump occurs. The function must return a pointer to a component dump table The component dump table specifies memory areas to be included in a system dump. Both the master dump table and the component dump table must reside in pinned global memory.

When a dump occurs, the kernel dump routine calls the function pointed to in the master dump table twice. On the first call, an argument of **1** indicates that the kernel dump routine is starting to dump the data specified by the component dump table. On the second call, an argument of **2** indicates that the kernel dump routine has finished dumping the data specified by the component dump table. The component dump table should be allocated and pinned during initialization. The entries in the component dump table can be filled in later. The function pointed to in the master dump table must not attempt to allocate memory when it is called. Figure 9-1 on page 9-3 shows the flow of a system dump.

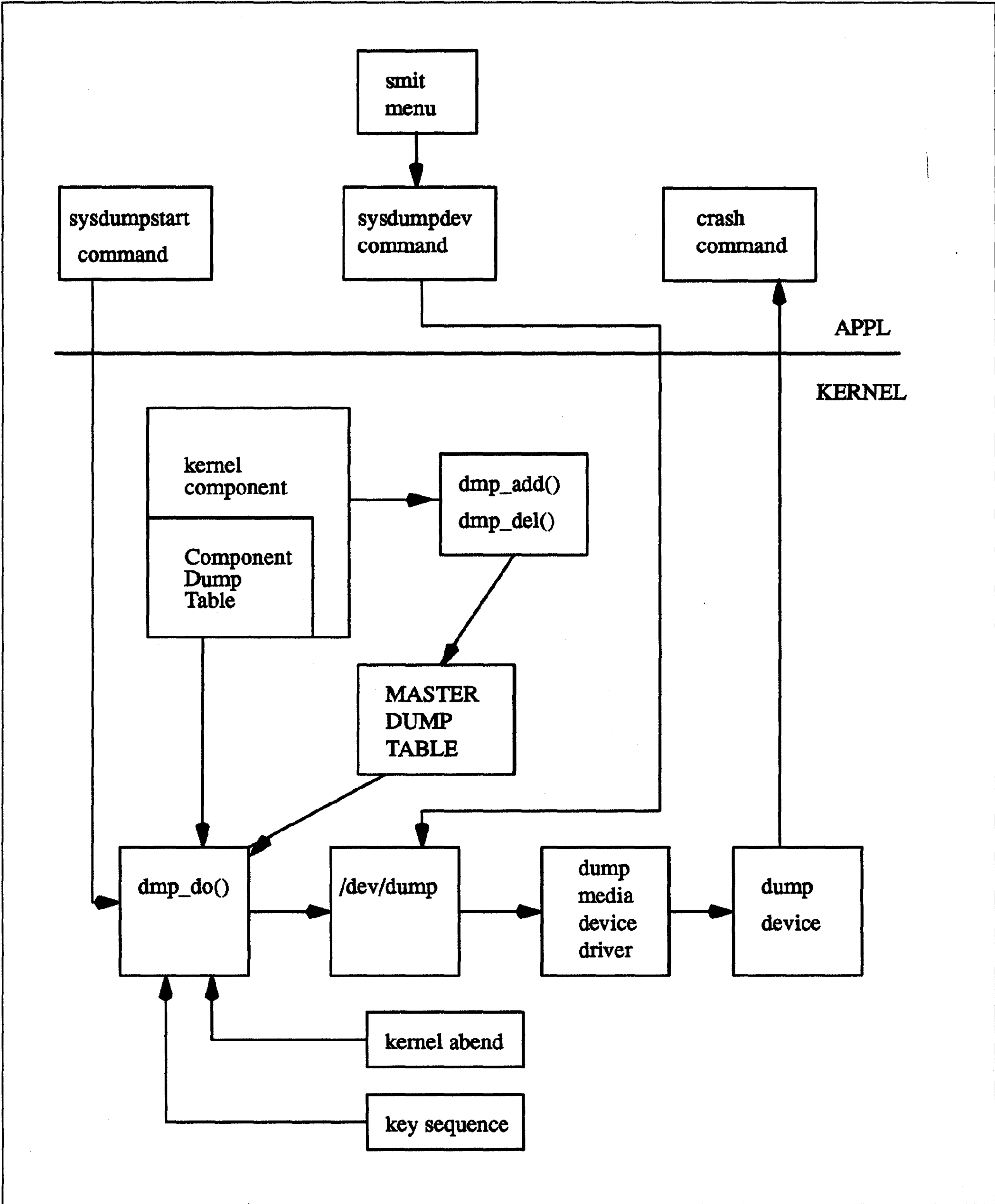


Figure 9-1. System Dump Flow

In order to have your device driver data areas included in a system dump, you must register the data areas in the master dump table. Use the **dmp_add** service to add an entry to the master dump table. Conversely, use the **dmp_del** service to delete an entry from the master dump table. The syntax is as follows:

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dump.h>

int dmp_add(cdt_func)  or  int dmp_del(cdt_func)
int cdt * ((*cdt_func) ());
```

The structure type (**struct cdt**) is defined in /usr/include/sys/dump.h. A **cdt** structure consists of a fixed-length header (**cdt_head** structure) and an array of one or more **cdt_entry** structures. The **cdt_head** structure contains a component name field, which should be filled in with the name of the device driver, and the length of the component dump table. Each **cdt_entry** structure describes a contiguous data area, giving a pointer to the data area, its length, a segment register, and a name for the data area. The name supplied for the data area can be used to refer to it when the **crash** command formats the dump. Refer to Figure 9-2 on page 9-5 for an illustration of the dump image.

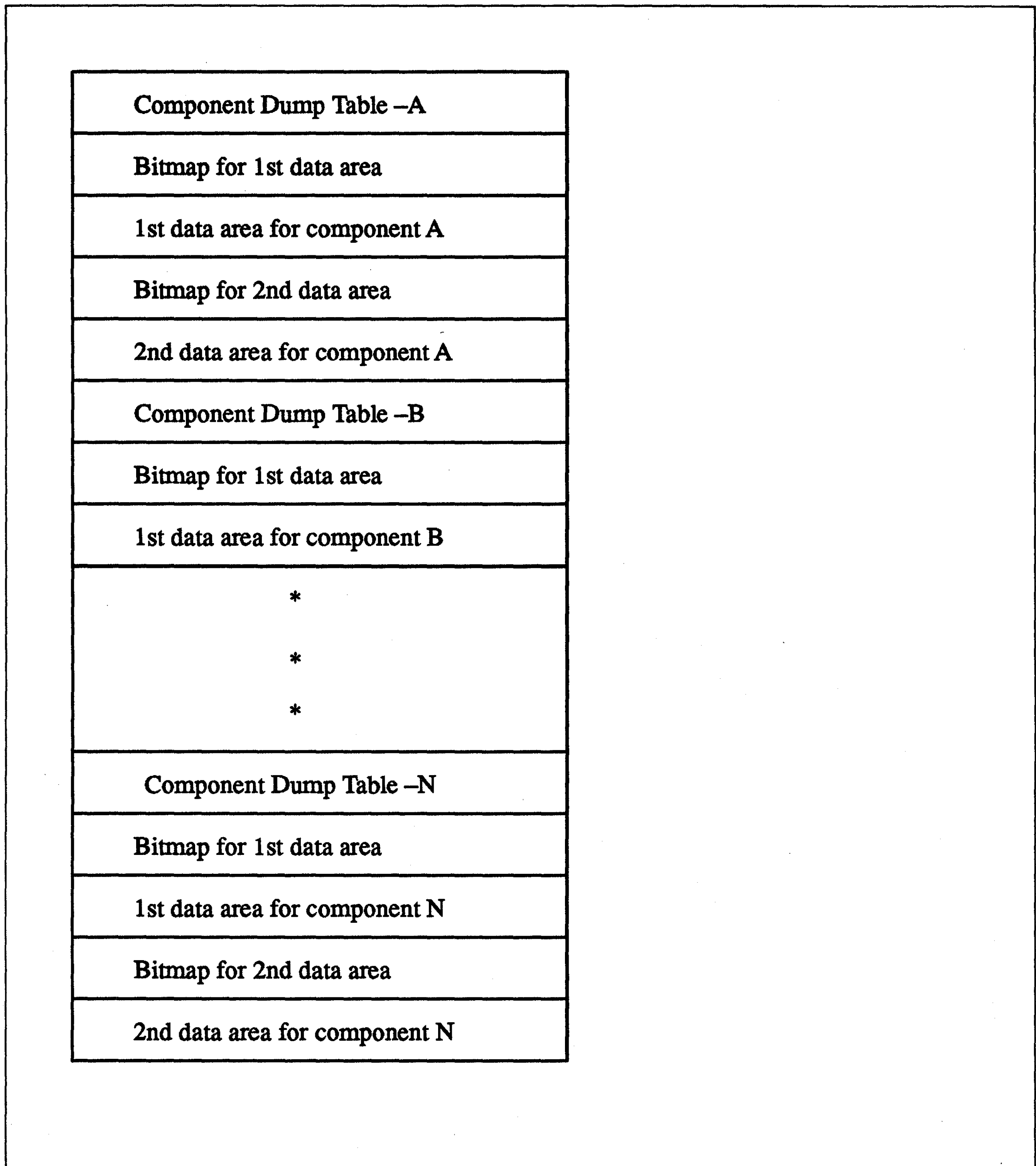


Figure 9-2. Kernel Dump Image

9.2.3 Formatting a System Dump

Each device driver that includes data in a system dump can install a unique formatting routine in the **/usr/adm/ras/dmprtns** directory. A formatting routine is a command that is called by the **crash** command. The name of the formatting routine must match the component name field of the corresponding component dump table. The **crash** command forks a child process that executes the formatting routines. If a formatting routine is not provided for a

component name, the **crash** command executes the **_default_dmp_fmt** default formatting routine, which prints out the data areas in hex.

The **crash** command calls the formatting routine as a command, passing the file descriptor of the open dump image file as a command line argument. The syntax for this argument is **-f***file_descriptor*.

The dump image file includes a copy of each component dump table used to dump memory. Before calling a formatting routine, the **crash** command positions the file pointer for the dump image file to the beginning of the relevant component dump table copy.

The memory dumped is laid out in the dump image file with the component dump table followed by a bit map for the first data area, then the first data area itself. Then will follow a bit map for the next data area, the next data area itself, and so on.

The bit map for a given data area indicates which pages of the data area are actually present in the dump image and which are not. Pages that were not in memory when the dump occurred were not dumped. The least significant bit of the first byte of the bit map is set to 1 if the first page is present. The next least significant bit indicates the presence or absence of the second page and so on. A macro for determining the size of a bit map is provided in /usr/include/sys/dump.h.

9.3 The crash Command

The crash command is a particularly useful tool for device driver development and debug which interprets and formats the system structures. The crash command is interactive and allows you to examine an operating system image or an active system. An operating system image is held in a system dump file existing either as a file or on the dump device. When you examine an active system the crash command opens /dev/mem; therefore, you must be running with root user permissions.

To invoke the crash command on the active system, simply type **crash** on the command line. To invoke the crash command on a system image file, type **crash** *system image*. The *system image* parameter can be either a file name or the name of the dump device. The default is /dev/mem. You may also specify a *kernel symbol definition* as a parameter following the *system image*. The default is /unix.

Note that by convention the symbol names for function entry points always begin with a period (.), while symbol names for data areas always begin with an underscore (_). There is usually a data address corresponding to an external entry point address, and the **od** subcommand will display the data address for a name with no prefix. To be safe, be sure to use the proper prefix when looking for addresses.

There is a flag which can be used with the crash command to generate a list of data structures without using subcommands. This is the **-a** flag. Use of this flag will generate a huge listing to standard out, so it is wise to redirect the output to either a file or to a printer.

In order to view the system structures a variety of subcommands may be used. These subcommands may have flags which can modify the format of the data. If you do not use a flag to specify what you want to see, all valid entries will be displayed.

9.3.1 crash Subcommands

Once you initiate the crash command, you will receive a line prompt which uses the "**>**" character. For a list of the subcommands available in the crash command, type "**?**" after the prompt. This will generate the subcommand list with a brief description of each. Some of the subcommands have aliases, or short forms, which will be noted with the subcommand descriptions. To exit the crash command, type **q** to quit.

Any shell command can be executed from within the crash command by preceding it with "!". Note that many structures displayed are longer than one screen length. Make sure that you can halt scrolling if it is important to view something in detail. To do this, first set the terminal with the stty command. To do this from within the crash command, type "**!stty ixon ixany** " after the prompt. Then you may type "**<Ctrl> <s>**" to stop scrolling and "**<Ctrl> <q>**" to resume scrolling.

- **buffer**[*Format*][*BufferHeaderNumber*]

  The **buffer** subcommand displays the data in a system buffer according to the *Format* parameter. When specifying a buffer header number, the buffer associated with that buffer header is displayed. If you do not provide a *Format* parameter, the previous *Format* is used. Valid options are **decimal**, **octal**, **hex**, **character**, **byte**, **directory** and **write**. The **write** creates a file in the current directory containing the buffer data.

  Aliases = b

  ```
  > buffer hex 3

  BUFFER FOR BUF_HDR 3
  00000: 41495820  4c564342  00006a66  73000000
  00020: 00000000  00000000  00000000  00000000
  00040: 00000000  00000000  00003030  30303033
                             .
                             .
                             .
                         and so on
  ```

- **buf**[*BufferHeaderNumber*]

  The **buf** subcommand displays the system buffer headers. A buffer header contains the information required to perform block I/O. If you type the **buf** subcommand with no *BufferHeaderNumber*, a summary of the system buffer headers will be displayed:

  Aliases = bufhdr, hdr

  ```
  > buf
  BUF MAJ  MIN    BLOCK   FLAGS
    0 000a 000b       8   done stale
    1 000a 000b     243   done stale
    2 000a 000b      24   done stale
  ```

```
                .
                .
                .
          and so on
```

If you type the **buf** subcommand with a *BufferHeaderNumber* a single complete header is displayed:

```
> buf 3

BUFFER HEADER 3:
    b_forw:     0x014d0528,  b_back:     0x014d0160,  b_vp:      0x00000000
    av_forw:    0x014d0160,  av_back:    0x014d0528,  b_iodone:  0x000185f8
    b_dev:      0x000a000b,  b_blkno:             0,  b_addr:    0x014e9000
    b_bcount:         4096,  b_error:             0,  b_resid:            0
    b_work:     0x80000000,  b_options: 0x00000000,  b_event:   0xffffffff
    b_start.tv_sec:             0,  b_start.tv_nsec:             0
    b_xmemd.aspace_id: 0x00000000,  b_xmemd.subspace_id: 0x00000000
    b_flags: read done stale
```

Please refer to /usr/include/sys/buf.h for the structure definition.

- **callout**

The **callout** subcommand displays all active entries on the active **trblist**. When the kernel extension **time-out** is used in a device driver, this timer request will be entered on a system-wide list of active timer requests. This list of timer requests is the trblist. Any timer which is active will be on this list until it expires.

Aliases = **c**, **call**, **calls**, **time**, **timeout**, **tout**

```
> callout
TRB #1 on Active List

   Timer address:  0x018d7080
   Timer flags:    0x00000002
   Timeout function:    0x000273dc
   Timeout function data:    0x018d7080
   Scheduled timeout (secs):    0270286ad
   Scheduled timeout (nanosecs):   0x1935a300
   trb.timerid:    0x00000000
   trb.timeout.it_interval.tv_sec:   0x00000000
   trb.timeout.it_interval.tv_nsec:  0x00000000
   trb.knext:    0x018d7180
   trb.kprev:    0x00000000

TRB #2 on Active List
                .
                .
                .
          and so on
```

Please refer to /usr/include/sys/timer.h for the structure definitions, and to InfoExplorer for a description of the time-out mechanism.

- **cm**[*SlotNumber SegmentNumber*]

The **cm** is used by the **od** subcommand to change the current segment map. The **cm** subcommand changes the map of the crash command internal pointers for any segment of a process not

swapped out, if you specify the process *SlotNumber* and *SegmentNumber*. This allows the **od** subcommand to display data relative to the beginning of the segment desired. The following example sets the map to process *SlotNumber* 3 to *SegmentNumber* 2, then displays ten words starting from the offset 0:

Aliases = none

```
> cm 3 2
p3,2 >> od 0 10
00000000: 00000000 00000000 00000000 00000000
00000010: 00000000 00000000 00000000 00000000
00000020: 00000000 00000000
p3,2 >>
          .
          .
          .
          and so on
```

Typing the **cm** subcommand without any parameters resets the map of internal pointers. Use this subcommand only when analyzing the currently running system.

- **ds**[*Address*]

  The **ds** subcommand returns the symbols closest to the given address. The **ds** subcommand can take either a text address or a data address as input.

  Aliases = none

```
> ds 012345
       .ioctl_systrace  + 0x000001b5
```

  When a number such as *0x000001b5* is displayed, it shows the number of assembly language instructions by which the given address is offset from the routine's entry point.

- **du**[*SlotNumber*]

  The **du** subcommand uses the specified process *SlotNumber* to display a combined hex and ASCII dump of the user block for any process that is not swapped out. The default is the current process.

  Aliases = none

```
> du 3
    00000000  00000000 00000000 2ff7fec0 00000000  *......../.......*
    00000010  00000303 00000000 00030644 000010b0  *...........D....*
    00000020  22222828 00030644 00006244 00000009  *""((...D..bD....*
              .
              .
              .
           and so on
```

- **dump**

  The **dump** subcommand displays the name of each component for which there is data present. After you select a component name from the list, the **crash** program loads and runs the associated formatting routine contained in the **/usr/adm/ras/dmprtns** directory. If there is more than one data area for the selected component, the

formatting routine displays a list of the data areas and allows you to select one. The **crash** command then displays the selected data area. You may enter the **quit** subcommand to return to the previously displayed list and make another selection or enter **quit** a second time to leave the **dump** subcommand loop.

Aliases = none

- **file**[*FileTableEntry*]

The **file** subcommand displays the file table. Unless specific file entries are requested, only those with a non-zero reference are displayed.

Aliases = **files**, **f**

```
> f 3
SLOT  REF     INODE    FLAGS
   3    1  0x018e53f0   read
```

Please refer to /usr/include/sys/file.h for the structure definition.

- **fs**[*SlotNumber*]

The **fs** subcommand traces a kernel stack for the process specified by the process *SlotNumber* for any process that has not been swapped out. The **fs** subcommand displays the called subroutines with a hex dump of the stack frame for the subroutine that contains the parameters passed to the subroutine. If no *SlotNumber* is enterred, the currently running process is displayed. If the process specified is swapped out, the message number **0425-074** will be displayed, and it will tell you that the frame pointer is not valid.

Aliases = none

```
> fs
STACK TRACE:


        **** .e_wait ****
2ff97e78  2FF97ED8 0080D568 00000000 018F4C60   /.¬....h......L
2ff97e88  2FF97EE8 0080D568 00082BC0 000BA020   /.¬....h..+....
2ff97e98  2FF97ED8 28008044 00082418 2FF98000   /.¬.(..D..B./...
2ff97ea8  00000000 000B8468 00000000 00000000   ......h........
2ff97eb8  2FF97F38 0000000B 00000004 00000004   /..8...........
2ff97ec8  00000005 01DFE258 00000000 E3000600   .......X........
```

- **inode**[-][<*MAJ*> <*MIN*> <*INUMB*>]

The **inode** subcommand displays the i-node table and the i-node data block addresses. A specific inode can be displayed by specifying the major and minor device numbers of the device where the inode resides and the inode number. The inode will only be displayed if it is currently on the system hash list.

Aliases = **ino**, **i**

```
>inode
 ADDRESS    MAJ  MIN INUMB REF LINK  UID  GID   SIZE   MODE  SMAJ SMIN FLAG
0x018e4e50 010 0007 11264   0    1    2    2     30 ----777    -    -
0x018f9fd0 010 0009 16384   1    6  201    0    512 d---755    -    -
     addr: 16448   0    0   0    0    0    0    0
0x018ea940 010 0011     0   1    0    0    0      0 ----  0    -    -
                .
                .
                .
           and so on
```

- **kfp**[*FramePointer*]

  If the **kfp** subcommand is entered without parameters, it displays the last kernel frame pointer address that was set using **kfp**. If a frame pointer address is provided, then it sets the kernel frame pointer to the new address. This subcommand is used in conjunction with the **-r** flag on the **trace** subcommand.

  Aliases = fp, rl

```
> kfp
```

- **knlist**[*Symbol*]

  The **knlist** subcommand displays the addresses of all the symbol names given. If the symbol is not found, a **no match** message is returned. This subcommand runs on an active system only. The **knlist** subcommand runs a subroutine to the active kernel to obtain the address from the system's knlist. The **nm** subcommand provides the same function but searches the symbol table in the Kernel Image File for the address and therefore can be used on a dump.

  Aliases = none

```
> knlist open
        open:0x000bbc98
```

- **mbuf**[-][*Clusters*|<*Address*>*...*]

  The **mbuf** subcommand displays the system mbuf structures. mbuf structures are memory buffers which are chained and which can be manipulated by the Memory Buffer Kernel Services. If the "**-**" flag is used, the data associated with the mbuf is also displayed. The **mbuf** subcommand with no additional arguments will display the chain of mbufs pointed to by the **mbuf** pointer. If the *Clusters* argument is supplied, then the chain of mbufs pointed to by the kernel mbclusters pointer is displayed. If the *Address* argument is given, then the mbuf at the given address is displayed. Note that valid mbuf pointers must be on a 128-byte boundary.

  Aliases = mbuff

```
> mbuf
  ADDRESS    SIZE    TYPE      LINK       DATAPTR
0x01a67000     0     free  0x00000000  0x01a67000
    DATA:  0x00000000 0x00000000 0x00000000 0x00000000
```

Refer to /usr/include/sys/mbuf.h for the structure definition.

• **nm**[*Symbol*]

The **nm** subcommand displays symbol value and type as found in *KernelImage*.

Aliases = none

```
> nm open
        00095484 000C70  PR SD    <.open>
        00095484         PR LD    .open
        000BBC98 00000C  SV SD    open
```

•**od**[*SymbolNameOrAddress*][*Count*] [*Format*]

The **od** subcommand dumps *Count* number of data values starting at *Symbol* value or *Address* according to *Format*. Possible formats are **octal**, **longoct**, **decimal**, **longdec**, **character**, **hex**, and **byte**. The default is **hex**. Note that if you use the *Format* parameter, you must also use *Count*.

The **od** subcommand is especially useful during program development in order to see structure values at a given point in time.

Aliases = none

```
> od open 10
00095484: 7c0802a6 bf21ffe4 90010008 9421ff30
00095494: 609c0000 832202e0 607b0000 60bd0000
000954a4: 63230000 38800000

> od open 10 byte
00095484: 0174 0010 0002 0246 0277 0041 0377 0344
0009548c: 0220 0001

> od 12345
warning: word alignment performed
00012344: 480001d8
```

• **pcb**[*ProcessTableEntry*]

The **pcb** subcommand displays the mstsave portion of the user structure of the named *Process*, or process control block. If you do not specify an entry, information about the last running process is displayed. If you attempt to display a paged process, you will receive the message **Cannot read uarea for this process.** Note that the **Segment Registers**, the **General Purpose Registers**, and the **Floating Point Registers** are displayed by this subcommand, but this data is too lengthy for this example.

Aliases = none

```
> pcb
        USER AREA FOR crash (ProcTable Address 0xe3003700)

SAVED MACHINE STATE
    curid:0x00003727  m/q:0x70000000  iar:0x00005fd8  cr:0x28442888
    msr:0x000090b0  lr:0x0004d170  xer:0x00000001
```

```
    ctr:0x0001c7a4  *bus:0x00000000
    *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0000000b
    backtrace:0x00  tid:0x00000000  fpeu:0x00  ecr:0x00000000
    Exception Struct
      0x30035d00  0x42000000  0x40001acc  0x30035d00
                .
                .
                .
            and so on
```

```
> pcb 30
        USER AREA FOR cron (ProcTable Address 0xe3001e00)

SAVED MACHINE STATE
    curid:0x00001e70  m/q:0x00000067  iar:0x00030644  cr:0x28424028
    msr:0x000010b0  lr:0x00030644  xer:0x00000008
    ctr:0x0009a6e4  *bus:0x00000000
    *prevmst:0x00000000  *stackfix:0x2ff97bd8  intpri:0x00000000
    backtrace:0x00  tid:0x00000000  fpeu:0x00  ecr:0x00000000
    Exception Struct
      0x00000000  0x22000000  0x2000343a  0xc008d780
                .
                .
                .
            and so on
```

Please refer to /usr/include/sys/user.h and /usr/include/sys/mstsave.h for the relevent structure definitions.

- **proc**[-][*-r*][*ProcessTableEntry*] .
  The **proc** subcommand displays the process table. The **-r** displays only runnable processes. The "**-**" flag displays a longer listing of the process table.

  Aliases = **ps**, **p**

```
> p
SLT ST    PID  PPID   PGRP   UID  EUID  PRI   CPU   EVENT  NAME
  0 s       0     0     0     0     0   16   120          swapper
        FLAGS: swapped_in no_swap fixed_pri kproc wake/sig
  1 s       1     0     0     0     0   60     0          init
        FLAGS: swapped_in no_swap wake/sig
  2 r     202     0     0     0     0  127   106          wait
        FLAGS: swapped_in no_swap fixed_pri kproc
                .
                .
                .
            and so on

> p 30
SLT ST    PID  PPID   PGRP   UID  EUID  PRI   CPU   EVENT  NAME
 30 s    1e70     1  1e70     0     0   26     0 001e83bc cron
        FLAGS: swapped_in wake/sig

> p -
                .
                .
```

```
SLT ST    PID   PPID   PGRP   UID  EUID  PRI   CPU   EVENT  NAME
  0 s       0      0      0     0     0   16   120          swapper
        FLAGS: swapped_in no_swap fixed_pri kproc wake/sig

Links:  *child:0xe3000100  *siblings:0x00000000  *uidl:0xe3003300
    *wchanl(real):0x00000000  *lcklst:0x00000000
    selchn:0x00000000
Dispatch Fields:  *prior:0xe3000000  *next:0xe3000000
    pevent:0x00000020  wevent:0x00000003
    polevel:0x00000000  *lockwait:0x00000000
    *eventlst:0x00000000  *wchan(hashed):0x00000000  suspend:0x7fff
    process waiting for:  event(s)
                .
                .
                .
            and so on
```

Please refer to /usr/include/sys/proc.h for the structure definition.

- **socket**[-]

  The **socket** subcommand displays the system socket structures. If the "-" flag is used, the socket buffers are also displayed.

  Aliases = **sock**

```
> sock
SLOT:   26  type:0x0002  opts:0x0000  linger:0x0000
     state:0x0080  pcb:0x01d32d8c  proto:0x01c65cf0
     q0:0x00000000  q0len:    0 q:0x00000000
     qlen:    0  qlimit:    0  head:0x00000000
     timeo:    0  error:    0  oobmark:    0  pgrp:    0
                .
                .
                .
            and so on
```

Please refer to /usr/include/sys/socket.h for structure definitions.

- **stack**[*ProcessTableEntry*]

  The **stack** subcommand displays a dump of the kernel stack of a process. The addresses shown are virtual data addresses rather than true physical addresses. If you do not specify an entry, information about the last running process is displayed. You cannot trace the stack of the current running process on a running system.

  Aliases = **s**, **stk**, **k**, **kernel**

```
> s 30
KERNEL STACK:

2ff97a50:          8eaa4        16  2ff97ac8         2
2ff97a60:           90b0     8e8b4  2ff97ad8         0
2ff97a70:              1        26  2ff97ac8  2ff98938
```

```
        .
        .
   and so on
```

- **stat**

  The **stat** subcommand displays statistics found in the dump. These statistics include the panic message (if there is one), time of crash, and system name.

  Aliases = none

```
> stat
        sysname: AIX
        nodename: funk
        release: 1
        version: 3
        machine: 000003961000
        time of crash: Fri Sep 28 17:50:38 1990
        age of system: 15 day, 6 hr., 25 min.
```

- **trace**[*-r*][*ProcessTableEntry*]

  The **trace** subcommand displays a kernel stack trace of the current process. The trace starts at the bottom of the stack and attempts to find valid stack frames deeper in the stack. If you do not specify a *ProcessTableEntry*, information about the current process is displayed. The **-r** flag will cause trace to use the kernel frame pointer set up by the **kfp** subcommand as its starting address instead of the frame pointer found in the *SystemImageFile.* The **trace** subcommand will stop and an error will be reported if an invalid frame pointer is encountered.

  Aliases = t

```
> t 30
STACK TRACE:
        .e_wait ()
        .e_sleep ()
        .e_sleepl ()
        .sleepx ()
        .fifo_read ()
        .fifo_rdwr ()
        .vno_rw ()
        .rwuio ()
        .rdwr ()
        .kreadv ()
```

- **ts**[*TextAddress*]

  The **ts** subcommand finds the text symbols closest to the given address.

  Aliases = none

```
> ts 012345
        .ioctl_systrace
```

•**tty**[*a* | *o* | *ResourceName* | *major*[*minor*[*channel*]]]

The **tty** subcommand displays the tty structures. Entering **tty** without parameters defaults to the *o* option and displays abbreviated data for all the open tty structures. The *a* option displays abbreviated data for all tty structures. A *ResourceName* is the name by which a device is known to the system. For tty devices this is typically *tty0*, *tty1*, *etc*. Therefore entering the **tty** subcommand with a *ResourceName* option will produce a detailed display of the tty structure for the device known to the system by the name. Combinations of *major*, *minor*, and *channel* numbers also give detailed data for the corresponding devices. For example, to get a detailed display of the tty structures for all devices associated with *major* device **4**, type:

```
>tty 4
```

To get a detailed display of the tty structures for all devices associated with *minor* device **8** on *major* device **4**, type:

```
>tty 4 8
```

To get a detailed display of the tty structure for the device on *channel* **0** of *minor* device **8** on *major* device **4**, type:

```
>tty 4 8 0
```

Aliases = **term**, **dz**, **dh**

```
> tty
(maj,min)[chan]:    sid      grp  raw can out flags
            tty0:00002833 00002833   0   0   0 isopen iaslp ccnt=0
           hft/0:0000151a 0000151a   0   0   0 isopen iaslp ccnt=0
           pts/0:00002949 0000325f   0   0   0 isopen iaslp ccnt=0
           pts/1:00002c4e 00002f76   0   0   0 isopen ccnt=0

> tty tty0
            tty0: tp=0x01b348f8, dev:(27,0) chan:(0x0)
sid:0x00002833 group:0x00002833 tsm:0x00000000 id:0
port status: isopen iaslp ccnt=0
ctl=0x01377240 lctl=0x01a85300 hptr=0x01b34800 evt=0x00002833 lck=0xfffff
rbuf: cc=0   ''
tbuf: cc=58   %   Blks Cp Rnk¬M¬J------- ----- -------''
raw queue: cc=0, actual=0:   ''
can queue: cc=0, actual=0:   ''
out queue: cc=0, actual=0:   ''
                 .
                 .
                 .
            and so on
```

Please refer to /usr/include/sys/tty.h for the structure definition.

• **user**[*ProcessTableEntry*]

The **user** subcommand displays the user structure of the named process as determined by the information contained in the process entry table. If you do not specify the entry, the information about the last running process is displayed. If you attempt to display a paged process, an error message is displayed.

Aliases = u, uarea, u_area

```
> u 30
        USER AREA FOR biod (ProcTable Address 0xe3001e00)

SAVED MACHINE STATE
    curid:0x00001e08  m/q:0x00000039  iar:0x00030644  cr:0x28424028
    msr:0x000010b0  lr:0x00030644  xer:0x0000000c
    ctr:0x000303b8  *bus:0x00000000
    *prevmst:0x00000000  *stackfix:0x2ff97d48  intpri:0x00000000
    backtrace:0x00  tid:0x00000000  fpeu:0x00  ecr:0x00000000
    Exception Struct
      0x00000000  0x40000000  0x4000172b  0x20045134
    Segment Regs
      0:0x00000000   1:0x40000522   2:0x0000172b   3:0x007fffff
      4:0x007fffff   5:0x007fffff   6:0x007fffff   7:0x007fffff
      8:0x007fffff   9:0x007fffff  10:0x007fffff  11:0x007fffff
     12:0x007fffff  13:0x4000140a  14:0x00000c06  15:0x007fffff
                .
                .
                .
            and so on
```

Please refer to /usr/include/sys/user.h for the structure definition.

- **var**

  The **var** subcommand displays the tunable system parameters.

  Aliases = tune, tunable, tunables

```
> var
buffers    20
files     328
e_files   328
procs     131071
e_procs    64
clists    16384
maxproc    40
iostats     1
locks     200
e_locks   8456344
```

- **vfs**[-][*Vfs SlotNumber*]

  The **vfs** uses the specified *Vfs SlotNumber* to display an entry in the vfs table. The "-" displays the vnodes associated with the vfs. The default is to display the entire vfs table.

  Aliases = m, mnt, mount

```
> vfs 3
VFS ADDRESS   TYPE OBJECT    STUB NUMBER FLAGS  PATHS
  3 1a62494    jfs 1a6d47c 1a6d650      5 D     /dev/hd1 mounted over /u

        flags:  C=disconnected D=device   I=remote P=removable
                R=readonly     S=shutdown U=unmounted Y=dummy

> vfs - 3
VFS ADDRESS   TYPE OBJECT    STUB NUMBER FLAGS  PATHS
```

```
  3 1a62494    jfs 1a6d47c 1a6d650      5 D     /dev/hd1 mounted over /u
VSLOT ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT    INODE FLAGS
   83 1a6e0ac  3    -   vreg    jfs     1     -  18f82c0
   84 1a6e218  3    -   vreg    jfs     1     -  18f8770
   85 1a6e24c  3    -   vreg    jfs     1     -  18f8590
   86 1a6e17c  3    -   vdir    jfs     3     -  18f7f00
   87 1a6dea4  3    -   vreg    jfs     2     -  18f65b0
   88 1a6dfa8  3    -   vdir    jfs     5     -  18f6100
   89 1a6d47c  3    -   vdir    jfs     1     -  18ea580 vfs_root
```

Please refer to /usr/include/sys/vfs.h for structure definitions.

- **vnode**[*VNodeSlotNumber*]

  The **vnode** subcommand uses the specified *VNodeSlotNumber* to display an entry in the vnode table. The default is to display the entire vnode structure.

  Aliases = none

```
> vnode 3
VSLOT ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT  DATAPTR FLAGS
    3 1a6e078  0    -   vreg    jfs     4     -  18f6790
 Total VNODES printed 1
```

  Please refer to /usr/include/sys/vnode.h for the structure definition.

- **xmalloc**

  The **xmalloc** subcommand displays information concerning the allocation and usage of kernel memory (the pinned_heap and the kernel_heap).

  Aliases = xm, malloc

```
> xmalloc
Kernel heap usage
heap size = 247803904 amount used =        0
Pinned heap usage
heap size = 247803904 amount used =        0
Kernel and pinned heap usage
```

9.4 The Kernel Debugger

The kernel debug program helps determine errors in code running in the kernel. The kernel debug program is used throughout the kernel development area. One important use of the kernel debug program is debugging device drivers.

The kernel debug program can run in any configuration that includes an asynchronous terminal connected to a serial adapter. The kernel debug program does <u>not</u> support any displays connected to any of the IBM graphics adapters.

You might find it useful to have listings of the various kernel structures, such as the process block or the user block. You can find these in /usr/include/sys. You should also have a symbol map of your device driver. The way to get a symbol map is to use the *binder option* **map** at the link step of compiling your driver. Refer to the **ld** command on how to do this.

You may enter the kernel debug program through a key sequence from the keyboard, or by using breakpoints. To enter the kernel debug program from the keyboard ***the key must be in service position !*** To invoke the kernel debug program from a native keyboard press the key sequence **<Ctrl><Alt><NumPad 4>**. While you can invoke the kernel debug program from the native keyboard, the kernel debug program will come up on an asynchronous terminal rather than on the display associated with the native keyboard. To invoke the kernel debug program from an IBM 3151 Asynchronous Terminal, use the key sequence **<Ctrl><\>**. To invoke the kernel debug program from an IBM 3161 Asynchronous Terminal, use the key sequence **<Ctrl><4>**.

You can use the **crash** command to determine whether the kernel debug program is available. Type:

```
#crash
>od dbg_avail
```

The string that is returned will indicate whether the kernel debug program is available. A **0** or **1** indicates that the kernel debug program is *available*. A **2** indicates that the kernel debug program is not available.

If the kernel debug program is not available, i.e., nothing happens when you type in the appropriate key sequences, you must load it. To do this, type **bosboot -a -D** or **bosboot -a -I**. The **-D** flag causes the kernel debug program to be loaded. The **-I** flag also causes the kernel debug program to be loaded, but it also is invoked at system initialization time. This means that at boot-up the system will trap the kernel debug program. Type **<go><Enter>** to get to the login screen. Note that you must shut down and reboot the system after the **bosboot** command completes in order to make the changes available.

There are two ways to enter the kernel debug program using breakpoints. One way is to set a static debug program trap, or **SDT** in the compiled code. This is done by placing the assembly language instruction **t 0x4, r1, r1** at the desired address.

NOTE

A way to do this is to make an assembly language routine that does this, then call it from your driver code. There is no way to generate an assembly language instruction from your C code.

The second way is to set a breakpoint by issuing a **break** command from within the kernel debug program. You can set breakpoints only in memory that is currently paged into RAM.

The kernel debug program has other major functions besides that of setting breakpoints. You may also use it to display and change processor memory and

registers or memory manager segment registers. Note that the crash command allows you to display these structures, but it does not allow you to change them. You may use the kernel debug program to format kernel data structures or display formatted trace buffers. Conventional debugger features such as stepping, looping, searches, and variables are also available.

9.4.1 The Kernel Debug Program Commands

When you enter the kernel debug program, it is the only process running on the machine. Every other process is suspended until you leave the debugger. Interrupts are disabled. The the kernel debug program maintains its own **mstsave** (machine state save) area. Once in the kernel debug program, use the commands to investigate and make alterations. Each command has an alias or a shortened form. This is the minimum number of letters required by the kernel debug program to recognize the alias as unique. See Table 9-1 for a complete list of the kernel debugger commands and for a summary of the command function.

9.4.1.1 Commands List and Summary

Table 9-1 (Page 1 of 2). Kernel Debugger Commands List

Command	Alias	Description
alter	**a**	Alter memory
back	**b**	Decrements the **IAR**
break	**br**	Sets a breakpoint
breaks	**b**	List currently set breakpoints
clear	**c**	Clears (removes) breakpoints
display	**d**	Displays a specified amount of memory
drivers	**dr**	Displays the contents of the device driver (devsw) table
find	**f**	Finds a pattern in memory
float	**fl**	Displays the floating point registers
go	**g**	Starts execution of the program
? or **help**	**h**	Displays the list of valid commands
loop	**l**	Execute until control returns to this point
map	**m**	Displays the system loadlist
next	**n**	Increments the **IAR**
origin	**o**	Sets the origin
proc	**p**	Displays the formatted process table
quit	**q**	Ends a debugging session
reset	**r**	Releases a user-defined variable
screen	**s**	Displays a screen containing registers and memory
set	**se**	Define or initialize a variable

Table 9-1 (Page 2 of 2). Kernel Debugger Commands List		
Command	**Alias**	**Description**
sregs	**sr**	Displays segment registers
st	**st**	Stores a fullword in memory
stack	**sta**	Displays a formatted kernel stack trace
stc	**stc**	Stores one byte in memory
step	**ste**	Performs an instruction single-step
sth	**sth**	Stores a halfword in memory
swap	**sw**	Switches from the current display & keyboard to another RS232 port
trace	**tr**	Displays formatted trace information
trb	**trb**	Displays the timer request blocks
tty	**tt**	Displays the tty structure
user	**u**	Displays a formatted user area
vars	**v**	Displays a listing of the user-defined variables
vmm	**vm**	Displays the virtual memory data structure
xlate	**x**	Translates a virtual address to a real address

Note that **help** is available by typing "?" at the debugger prompt. The following represents a quick guide to the kernel debugger.

- **Alter address data**

```
alter 1000 ffff     Store the 16 bit value ffff at address 1000
a 1000 2C           Store the 8 bit value 2C in the high order
                    byte at address 1000
```

Modifies the contents of memory, replacing the old value with the new specified value. If data is only one byte, then it is stored in the high order byte of the word at address. If data is a halfword, it is stored in the high order halfword at address. Alter is normally used to change data while a program is running.

- **Back [byte-count]**

```
back               Decrement the IAR by 4 bytes
b 16               Decrement the IAR by 16 bytes
```

Decrements the IAR by the specified amount.

- **Break [address]**

```
break              Set a breakpoint at the IAR
break 521a         Set a breakpoint at address 521A
br 8300+A0         Set a breakpoint at A0 + 8300
break +A0          Set a breakpoint at the origin + A0
break lr           Set a breakpoint at address in the link
                   register
```

Sets a breakpoint which will cause the debugger to be entered when the instruction at the target address is next executed. There is a maximum of 32 breakpoints.

• **Breaks**

```
breaks          List breakpoints
```

Displays the breakpoints as a combination of segment register values, and offsets into the segment. Since the segment registers may change during program execution, an address such as 1000052C may refer to different regions of memory. The segment register value is needed, therefore, to specify a unique address.

• **Clear [address]**

```
clear           Clear breakpoint at IAR
cl 10000200     Clear breakpoint at address
                10000200
```

Removes a previously set breakpoint. If more than one breakpoint has the specified offset into the segment (low order 28 bits of address), then the user will be prompted with the segment values for each of the breakpoints, and asked which to clear.

• **Display address [byte-count]**

```
display iar     Display 16 bytes at the IAR
d 152f 12       Display 12 bytes at address 152F
display +B7     Display 16 bytes at the origin + B7
disp r3         Display 16 bytes at the address in r3
d r3>           Display from the address contained in the
                address in r3 (1 level of indirection)
```

Displays memory in hexadecimal (base 16) and ASCII (characters). The byte-count is the number of bytes to display, and is used to determine how memory is to be accessed. A byte-count of 1 will cause a single byte to be loaded, a byte-count of 2 will cause a halfword to be loaded, a byte-count of 4 will cause a word to be loaded; any other byte-count (including none) will cause memory to be loaded one byte at a time.

• **Drivers [slot | address]**

```
drivers         Display device driver (devsw) table
drivers 10      Display slot 10 of the device driver table
dr 130000f      Display last entry point before address
                130000F
```

Displays the contents of a devsw table entry. Each devsw entry consists of a number of entry points (Read, Write, Ioctl, etc) into the specified driver. Each entry consists of a function descriptor, and the address of the function. If no arguments are given to the drivers command, the entire devsw table is displayed. If an argument is given, and that argument is a valid slot number, the corresponding entry is displayed. If the argument is not a valid slot, it is taken to be an address and the slot with the last entry point before the address is displayed (along with the name of the entry point).

• **Find pattern|* [start|*] [end|* [align|*]]**

```
find 7c81          Find first occurence of 7C81 in virtual
                   memory starting at 0
find "AIX"         Find first occurence of string AIX
f 7c81 10000       Find first 7C81 after address 10000
f 7c81 0 top       Find first 7C81 between 0 and
                   user-de fined variable "top"
find 7c81 *        Find first 7C81 starting at last address
                   used
f 7c81 * * 2       Same as above, but aligned on halfword
f 7c fx+1 * 2      Find next 7c starting at 1 + last address
                   find stopped at
```

Locates instances of the specified pattern in the current virtual address space. If, for any argument to find, an asterisk (*) is used instead of specifying a value then the previous value is used. For example "find *" searches for the last pattern used. Find sets a variable, fx, to the address of the last item found. This feature coupled with the asterisk can be used to continue the search for the pattern, e.g., "f * fx+1" searches for the last pattern starting at the next location. Find remembers the alignment which was used in the previous search.

- **Float**

```
float              Display floating point registers
```

Float will display the floating point registers.

- **Go [address]**

```
go                 Continue execution at the IAR
g 1000             Set the IAR to 1000 and begin execution
                   there
```

Go will resume execution after a breakpoint. It can be used to set the IAR to a new address and begin execution there.

- **Help**

```
help               Display the list of valid commands
```

Help displays one line for each debugger command.

- **Loop [iterations]**

```
loop 2             Execute until the second time the IAR has
                   the current value
```

Loop is similar to setting a breakpoint at the current IAR, but allows you to stop, for example, on the sixth time the IAR returns to the current point. If the IAR is at the beginning of a function, entering the command "loop 6" will cause the program to continue execution and allow the IAR to return to the current point five times without entering the debugger. Upon returning for a sixth time, execution stops and the debugger is entered.

- **Map [address|symbol]**

```
map                Display the system loadlist
m e3000000         Display symbol with value closest to
                   E3000000
map execexit       Display the value of the symbol "execexit"
```

Displays information from the system loadlist (the list of symbols exported from the kernel). If the map command is given with no arguments, then the entire loadlist is displayed one page at a time. If an address is given as an argument then the symbol value which is closest to, but less than, the address is displayed. Since map only knows of symbols which were exported from the kernel, this information may not be exact. If a symbol name is given as an argument, then the loadlist is searched for the symbol, and any entries (there may be more than one) which match are displayed.

NOTE: The symbol value for a data structure will be the address of the data structure. The symbol value for a function will NOT be the address of the function, it will instead be the address of the function descriptor for that function. The first word of the function descriptor will be the address of the function. For example: if "map execexit" displays 0x1000, then "display 1000" will display the address of the function execexit.

- **Next** [**byte-count**]

```
next            Increment the IAR by 4
n 20            Increment the IAR by 20
```

Increments the IAR by the specified amount.

- **Origin expression**

```
origin 178D     Set the origin to 178D
o 592cc         Set the origin to 592cc
```

Sets the origin variable to the value of the specified expression. Origins are used to match addresses with assembly language listings (which express addresses as offsets from the start of the file). Set the origin to the address in the executable of the first function in the assembly listing. Offsets from the origin are equivalent to offsets into the assembly listing. This is useful if you wish to know where the instruction you are about to execute is in the listing.

- **Proc** [**pid**]

```
p               Display the process table
proc 1          Display the process table entry for
                process with process id 1
```

The proc command with no arguments displays one line for each process in the process table (similar to the ps command). If a process ID is passed as an argument then a long listing of the process table entry is displayed. This listing shows every field in use in the process table for that process table entry

- **Quit** [**dump**]

```
quit dump       Dump the kernel data structures
```

Quiting the debugger frees the memory that the debugger occupied. When the debugger has been entered due to a fatal system error the argument "dump" instructs the system to dump kernel data structures to the primary dump device

- **Reset variable**

```
reset foo       Deletes the user-defined variable "foo"
```

Resetting a variable effectively deletes it, and allows the variable slot to be re-used. Currently 16 user-defined variables are allowed, and when they are all in use no more may be set. Variables which are not user-defined (such as registers) may not be reset. The value of a non user-deifined variable is passed to the reset command and an error message of the form "invalid parameter 14C5" is displayed.

- **Screen [track] [+ | - | address | on half | off | on]**

```
screen +          Display the next 112 bytes of memory
screen -          Display the previous 112 bytes of memory
s 20000ff7        Display memory starting at 20000ff7
s 200>            Display memory at address contained in
                  location 200 (one level of indirection)
screen on         Turns on the display
screen off        Turns off the display
screen on half    Display format uses about 1/2 the screen
sc track r3       Tracks memory starting at the value in
                  general purpose register 3
```

The screen command controls the display: which information is displayed, and how much at a time. The screen command by itself shows the current screen format (which may have been scrolled by some other command). Arguments may select another area of memory to display, may modify the format of the screen so that only half of the physical screen is used, or even turn off the screen display entirely. The format modification arguments are useful if important information may be scrolled off the screen when the debugger is entered. The default screen may be restored by entering the "screen on" command, which restores the full screen. The track option changes the address that the screen displays as the expression which is being tracked changes. This is useful in a case where at each stop, at a breakpoint, the memory to be displayed is addressed by a register.

- **Set variable|register value**

```
set start 100     Assign value 100 to variable "start"
set r12 0         Set general purpose register 12 to 0
se s3  10000      Set segment register 3 to 10000
set iar 45F0      Assign 45F0 to the IAR
se name "AIX"     Assign string "AIX" to variable "name"
```

Debugger variables are set via the set command. Set will create new variables or modify the value of old variables. Certain debugger variables are symbolic names for machine registers, which may also be modified using the set command. They are iar, r0 through r31 (for general purpose registers), lr (link register), s0 through s15 (segment registers), frp0 through fpr31 (floating piont registers), fpscr, dsisr, dar, eim0, eim1, eis0,eis1, mq, msr, cr, ctr, tid, xer, sdr0, sdr1, rtcu, rtcl and dec.

- **Sregs**

```
sregs         Display the segment registers
```

Sregs creates a display similar to the screen command, but shows the segment registers along with certain other registers.

- **St address word**

```
st 1000 5        Store the 32-bit value 5 at address
                 1000
```

Stores a 32-bit value to memory. This is similar to the alter command, but the word size is implicit in the command.

- **Stc address byte**

```
stc 1000 ff     Store the 8 bit value FF at address 1000
```

Similar to the st command, stc stores a single byte at the specified address.

- **Sth address halfword**

```
sth 1000 0014     Store the 16 bit value 14 at address 1000
```

Similar to the st command, sth stores a halfword at the specified address.

- **Stack** [**pid**]

```
stack          Format any existing stack frames
sta 251        Format stack frames for process with process id 251
```

Formats the stack frames for the specified process. If no process ID is given, the currently running process is used. Stack frames show return addresses and may be used to trace the program's calling sequence. Be aware that the first few parameters to the called functions are passed in registers, and will not usually be available on the stack. Generally, only the chain (stack back-chain pointer) and return address (caller of the owner of this stack frame) are valid. To thoroughly interpret the stack, it is necessary to use the assembly language listing for a procedure to determine what has been stored on the stack. Stack frames for the specified process may not always be accessible.

- **Step** [**s | instructions**]

```
step            Single step the processor
step s          Single step (skip over a subroutine call)
ste 20          Step for 20 instructions
```

Step allows the processor to single step (execute a single instruction). The argument "s" will cause step to skip over subroutine calls, so that the main flow of control may be followed. An integer argument will be used as the number of instructions to execute before returning control to the debugger.

- **Swap port**

```
swap 1          Switch display to RS-232 port 1
```

The swap command allows control of the debugger to be transferred to another terminal. s0 is port 1, s1 is port 2.

- **Trace**

```
trace               Display the kernel trace buffers
```

The trace command displays the last 128 entries of the kernel trace buffers. Currently there are eight trace channels which may trace any combination of events. Event entries are placed in a trace buffer before they are written out to disk (if they are logged to disk), or the

buffers are used in a circular fashion and reused when full. The trace command allows examination of the trace headers which contain pointers into the trace buffers. Also the trace entries are shown (the hook IDs are given in text form, all data is displayed in hex).

- **Trb**

```
trb                Display timer request block information
```

A menu of commands to display timer request block information is given.

- **Tty [v] [major [minor [channel]]]**

```
tty                Display all open ttys
tty v              Verbose listing of all open ttys
tty 7              Display all open ttys with major number 7
```

The tty structure is displayed. Major and minor specify which device to look at. If minor is omitted, all open ttys for the specified major number are listed. If both major and minor are omitted, all open ttys are listed. The argument "v" specifies a more verbose output.

- **User [pid]**

```
user         Display current user area
u 315        Display user area for process with process ID 315
```

The user command with no arguments displays the user area for the currently running process. If an optional process ID is given, then the user area for that process is displayed instead.

- **Vars**

```
vars                Display current user-defined variables
```

A list of the user-defined debugger variables is displayed. Each variable's name and value is displayed, and an indication of what the base of the value might be is also given. Since the value 10 may be either decimal or hexadecimal it will be displayed as HEX/DEC. String variables are displayed, but there are no quotes around the string value.

- **Vmm**

```
vmm                 Display virtual memory data structures
```

Displays a menu of commands for examining the virtual memory data structures. Useful information such as the number of free pages is available.

- **Xlate virtual-address**

```
xlate 10054000      Translate virtual 10054000 to a real
                    address
```

Translates a virtual address to a real address.

Please refer to the *Commands Reference* volumes (all three) for more detailed information on these commands.

9.4.1.2 Numeric Values and Strings

Numeric arguments are required to be hex for all commands except the **loop** and **step** commands, which take decimal. Decimal numbers must either be decimal constants, variables or expressions involving constants and variables.

A string is either a hex constant or character constant of the form *"String"*. Double quotes seperate the string from other data.

9.4.1.3 Variables

Variable names must start with a letter and may be up to eight characters long. Variable names may not contain special symbols. Variables usually represent locations or values which are used again and again. A variable must not represent a valid number. Use the **set** command to define and initialize variables. Variables may contain from 1 to 4 bytes of numeric data or up to 32 characters of string data. You can release a variable with the **reset** command. Note that the **reset** command cannot be used with reserved variables.

```
ex: set name 1234        (sets your variable called name=1234)
    set s8 820c00e0      (sets seg reg 8 to point to the IOCC)
```

Note that s8 is a reserved variable (see below).

9.4.1.4 Reserved Variables

There is a set of variables which have a reserved meaning for the kernel debug program. These variables may be referenced and changed, but they do represent the actual hardware registers. There are also two variables reserved for use by the kernel debug program which may be changed or set. If you change the segment registers or the general purpose registers while in the kernel debug program, the change will remain in effect when you leave the kernel debug program.

r0-r31	General purpose registers 0 - 31
s0-s15	Segment registers 0 - 15
fp0-fp31	Floating point registers 0 - 31
iar	Instruction address register (program counter)
mq	Multiply quotient
msr	Machine state register
cr	Condition register
lr	Link register
ctr	Count register
tid	Transaction ID register
xer	Exception register (fixed point)
fpscr	Floating point status and control register
srr0	Machine status save/restore 0
srr1	Machine status save/restore 1
disr	Data storage interrupt status register
dar	Data address register
eim0	External interrupt mask (low)
eim1	External interrupt mask (high)
eis0	External interrupt summary (low)
eis1	External interrupt summary (high)
sdr0	Storage description register 0
sdr1	Storage description register 1
rtcu	Real time clock (seconds)
rtcl	Real time clock (nanoseconds)
dec	Decrementer

fx	Address of the item found by the find command
org	Current value of the origin

9.4.1.5 Expressions

The kernel debug program does not allow full expression processing. Expressions may only contain certain terms:

- Decimal or hex constants
- Variables
  Variable operators include:
  - + (addition)
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - > (dereference)

The > operator indicates that the value of the preceding expression is to be taken as the address of the target value. The contents of the address specified by the evaluated expression are used in place of the expression.

Expressions can be entered in the form *Expression(Expression)*. This form causes the two expressions to be evaluated separately and then added together. This is similar to the base address syntax used in the assembler.

Expressions are processed from left to right only. The type of data specified must be the same for all terms in the expression.

9.4.1.6 Pointer Dereferences

A pointer dereference may be used to refer indirectly to the contents of a memory location. For example, assume location **0xc50** contains a counter. Then an expression of the form **c50>** may be used to refer to the counter. Any expression may be placed before the >, including an expression involving another >. In this case, multiple levels of indirection are used. To extend the previous example, if location **ff7** contains the value c50, the expression **ff7>>** will refer to the previously discussed counter.

9.5 Using the Kernel Debugger to Debug Device Drivers

9.5.1 Setting Breakpoints in Device Driver Routines

You need a map file or an assembler listing file for the device driver to get relative addresses within the device driver routines (e.g. read, write, open, close, config). The map file can be generated by the map option of the loader (ld) command. For example, in our device driver this is done by:

```
ld -o ourdd ricdd.o ricutil.o riccfg.o -T512 -ericconfig \
-bimport:/lib/my.exp -bimport:/lib/syscalls.exp -lsys -lcsys \
-bmap:ourmap
```

After you load your driver with sysconfig, you must find out the address of its entry points. You can use the kernel debugger to find out where the device driver routines are loaded in the kernel. First you must know the major number for it. This can be found by the **li -l /dev** command. The device drivers are indexed by the major number. Use the **drivers** command in the kernel debugger to reveal the entry points. (Make sure that you look for your major number.) The func addr field will have the starting address of the routine.

One way to compute a breakpoint address is based on the assumption that config is the first routine in the device driver map. From the device switch table get the address of where config is loaded in the kernel. Since config is the first routine in the device driver, all other routine addresses in the map file will be relative to the load point of config. So, in order to set a break point, add the address of config to the displacement in the map or assembler listing file.

Another way to compute a breakpoint address is to find the absolute address of the device driver function in the device switch table. Add the displacement value corresponding to the instruction you wish to stop at (from the assembler listing) to the address in the device switch table.

9.5.2 Setting Breakpoints in System Routines

Sometimes it is desirable to set breakpoints in a system routine. Here is an example of how to set a breakpoint in the getpid system routine. (You can easily extend this example to set a breakpoint wherever you desire.)

- Use the **nm -xv /unix >unix.m** command to create a map file of the kernel.
- Search for the getpid entry point.
- An alternate way of getting this entry point is to get into the kernel debugger and use the kernel **map** command. Then by reading the function descriptor for the getpid routine, use the **d function descriptor** command to get the entry point address.
- Use the **break xxxxxxxx** command to set the breakpoint.
- Type **g** (for go) to exit the debugger. The kernel debugger will trap whenever the getpid entry point is called.
- Remember **clear xxxxxxxx** will clear the breakpoint at the xxxxxxxx address.

9.5.3 Displaying Registers on a Micro Channel Adapter

When writing a device driver for a new Micro Channel adapter, it is often desirable to be able to read and write to registers that reside on the adapter. This gives the programmer the feeling that the hardware is functioning correctly. For example, let us look at a register on the token-ring adapter. First, we need to see where this adapter resides in the bus I/O space. You can do this in the following manner:

$lsdev -C

```
sys0        Available 00-00       System Object
sysunit0    Available 00-00       RISC System/6000 System Unit
sysplanar0  Available 00-00       CPU Planar
        *
        *
        *
scsi0       Available 00-01       SCSI I/O Controller
tok0        Available 00-02       Token-Ring High-Performance Adapter
ent0        Available 00-03       Ethernet High-Performance LAN Adapter
```

$lsattr -l tok0 -E

```
bus_intr_lvl  3        Bus interrupt level                  False
intr_priority 3        Interrupt priority                   False
            *
            *
            *
rdto          92       RECEIVE DATA TRANSFER OFFSET         True
bus_io_addr   0x86a0   Bus I/O address                      False
dma_lvl       0x5      DMA arbitration level                False
dma_bus_mem   0x202000 Address of bus memory used for DMA   False
```

We now know that the token-ring adapter is located at 0x86a0.

- Get into the kernel debugger.
- Use **sregs** to display the segment registers. Find an unused segment register (=007FFFFF). For example, say s9 is unused.
- **set s9 820c0020** This enables the Micro Channel bus addressing.
- **sregs** will now display the segment register values so you know that you typed it in correctly.
- From the *Hardware Technical Reference*, we know that the address of the Adapter Communication and Status register is P6a6. From above, we know that P=8, therefore the address is 86a6.
- **d 900086a6 2** will now display the two-byte register.

The key is to load a segment register with **820c0020** and then use that segment register to reference registers/memory on your adapter. You may use the same method to access registers resident on the IOCC. In that case, you would load the segment register with a value of **820c00e0**.

9.5.4 How to read/write Data Variables in your Device Driver

The following method is general enough to be applicable to both device drivers (dd) and application code. Of course when it comes to debug application code, you want to use some symbolic debugger instead of the Kernel Debugger (Kdbg). Nevertheless for the sake of simplicity and because the steps in both cases are the same, we will use a short and easy-to-compile application program as an example. For the RISC/6000 the best way of debugging your dd is probably the tracing tool but those of you still inquisitive about the obscure world of Kdbg hopefully will find these lines illuminating.

In our example, we just use simple integers. When you deal with complex data structures you have just one additional concern: to find out how the compiler resolves the structures in terms of byte alignment. (See note in STEP 3.)

The objective of this example is to find out the Effective Address of our variables. Once this is done, Kdbg will take care of translating it, using segment regs, to the virtual address first and eventually to the physical address. To build the Effective Address, some sort of Base Address and Offsets are needed. We will obtain the offsets from the symbol table and the Base Addresses from the stack via Kdbg. Once we have the Effective Address we can use the Kdbg commands to read/write at that address.

STEP 1) write your code, e.g. pippo.c

```
================== pippo.c file ==================
int pippo_init=0x5a5a5a5a;
int pippo_not_init;
struct pippo_struct_type{
                           int     field1;
                           char    field2;
                           int     field3;
                        }pippo_struct_var;
main()
{
int pippo_stack=0x3e3e3e3e;

printf("ADDRESS OF TEXT: %x0,main);
printf("ADDRESS OF pippo_init:  %x0,&pippo_init);
printf("ADDRESS OF pippo_not_init:  %x0,&pippo_not_init);
printf("ADDRESS OF pippo_stack:  %x0,&pippo_stack);

pippo_data_not_init = 0x88888888;
while(1);
}
```

In Italian, Pippo means "Goofy" ** the Walt Disney character.

STEP 2) compile and link it

```
cc -g pippo.c -o pippo
```

Compiling and Binding directives for a device driver will be different (see "Compiling Device Drivers" on page 10-3) but but do not forget to add the "-g" option, otherwise you will not be able to have a symbol table information for pippo_stack.

STEP 3) read the symbol table of our executable into a file:

```
nm -xv pippo > pippo.nm
```

We are interested in the following entries:

```
TOC               |0x000002e4|unamex|              |      |     |.data
pippo_stack:-1    |0x00000038|lsym  |              |      |     |
pippo_init        |0x00000088|extern|              |      |     |.data
pippo_init        |0x000002fc|unamex|              |      |     |.data
pippo_not_init    |0x00000320|extern|              |      |     |.bss
pippo_not_init    |0x00000300|unamex|              |      |     |.data
```

Notes:

```
 -) there are 2 values per each global variable: one identifies the
variable the other is a pointer to that variable (see step 7)

 -) for data structures you may want to consider also the type
declaration line:
```

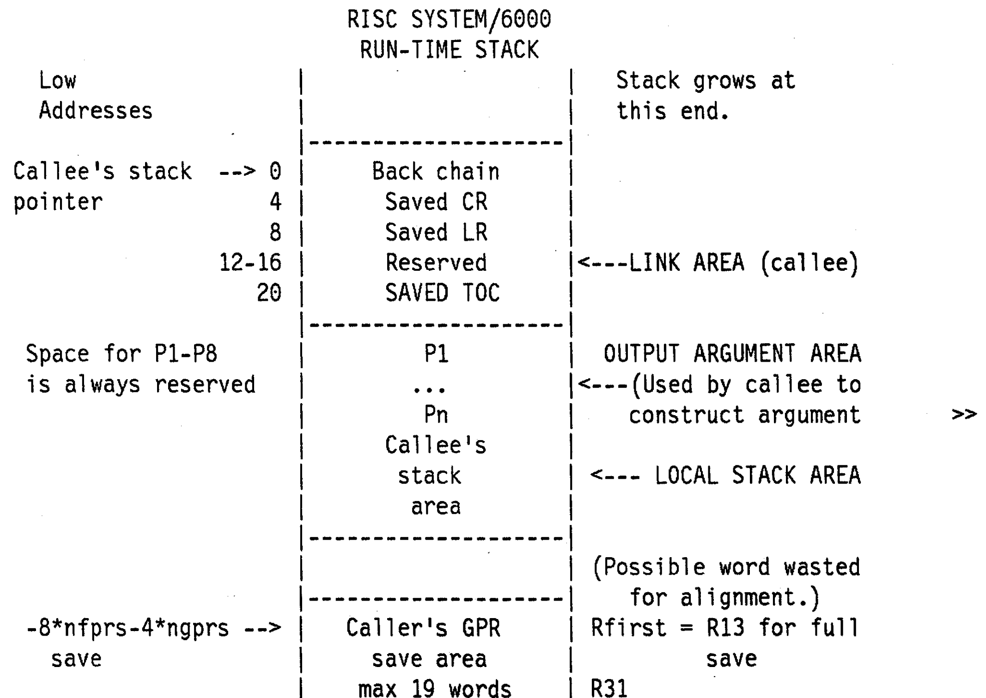
pippo_struct_type:T9=s12field1:-1,0,32;field2:-5,32,8;field3:-1,64,32;;| |decl

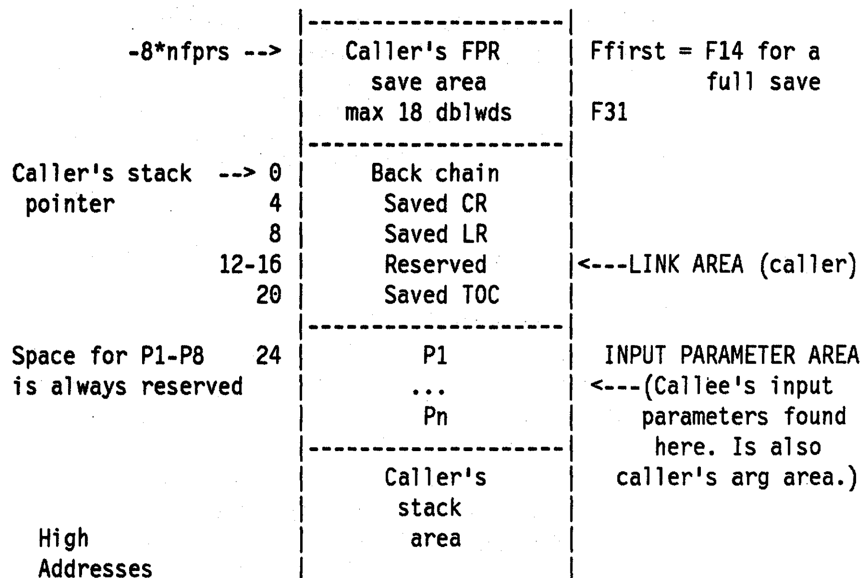
This will help to find out for each field the offset relative to the beginning of the structure and the size of the field, both in bits. In our example, field2 is at offset 32 and its size is 8.

STEP 4) Run your executable and break into the Kdbg. This is done via keyboard with the key in SERVICE position, and hopefully the running process is pippo (you can check it using "proc" Kdbg). In a device driver environment you add the brkpoint(MyParm) function call exactly where you want. Note that MyParm is an integer that will be displayed in GPR03 to help keeping track of the calling brkpoint() in case you have more than one in your code.

STEP 5) Use the Kdbg command "stack". You will see information about the last Stack Frame pushed into the stack before invoking the Kdbg (see figure below). From there we need two Effective Addresses: the saved TOC and the Beginning Stack.

```
In our example typing "stack" we obtained:
Beginning IAR: 0x10000558       Beginning Stack: 0x2FF7FBF8
Chain: 0x2FF7FC48  CR:0x22222088 Ret Addr: 0x10000538 TOC: 0x2003EAE4
P1:0x2003E864 P2:0x2FF7FC30 P3:0xDEADBEEF P4:0xDEADBEEF
P5:0xDEADBEEF P6:0xDEADBEEF P7:0xDEADBEEF P8:0x00000000
2FF7FC30        3E3E3E3E 00000000 00000000 00000000
2FF7FC40        DEADBEEF DEADBEEF 00000000 00000000
```





STEP 6) Just for clarity, use the "set" Kdbg command to set mnemonics for base addresses and offsets from steps 3 and 5.

```
Base Addresses (see step 5):
set toc_r                       2003EAE4
set sp                          2FF7FBF8

Offsets (see step 3):
set TOC                                000002e4
set pippo_init                         00000088
set pippo_not_init                     00000320
set pippo_init_P                       000002fc
set pippo_not_init_P                   00000300
set pippo_stack                        00000038

Notes:
 -) toc_r (alias the saved TOC) is the relocated value of TOC.
 -) the suffix "_P" stands for pointer.
```

STEP 7) Read pippo_init and pippo_not_init using "display" and write them using "alter" Kdbg commands. (Hopefully our variables are in real memory.)

The base address is toc_r, for the offset we have two alternatives: (pippo_init_P-TOC) and (pippo_init-TOC); the first one locates a pointer to pippo_init while the second one locates pippo_init itself. They are both allocated in the DATA section. Same for pippo_not_init_p and pippo_not_init but in this case (pippo_not_init-TOC) will be allocated in the BSS section.

```
For pippo_init:
you type        d toc_r+pippo_init_P-TOC>
or
you type        d toc_r+pippo_init-TOC
Kdbg output     2003E888        5A5A5A5A 00000000 00000000 00000000
you type        a 2003E888 77777777

Similar sequence for pippo_not_init.

Note:  the ">" which stands for "indirect addressing" in the Kdbg syntax.
```

STEP 8) Read pippo_stack using "display" and write it using "alter" Kdbg commands. The base address is sp, the offset is pippo_stack.

```
you type        d sp+pippo_stack
Kdbg output     2FF7FC30        3E3E3E3E 00000000 00000000 00000000
you type        a 2FF7FC30 77777777
```

9.6 Error Logging

The error logging facility allows a device driver to have entries recorded in the system error log. These error log entries record any software or hardware failures which need to be available either for informational purposes or for fault detection and corrective action. The device driver, using the **errsave** kernel service, adds error records to the special file **/dev/error**. The **errdemon** will then pick up the error record and create an error log entry. When the error log is accessed either through **SMIT** or with the **errpt** command, the error record is formatted according to the **error template** in the **error template repository** and presented in either a summary or detailed report. Refer to Figure 9-3 on page 9-36 for the flow of the error logging facility.

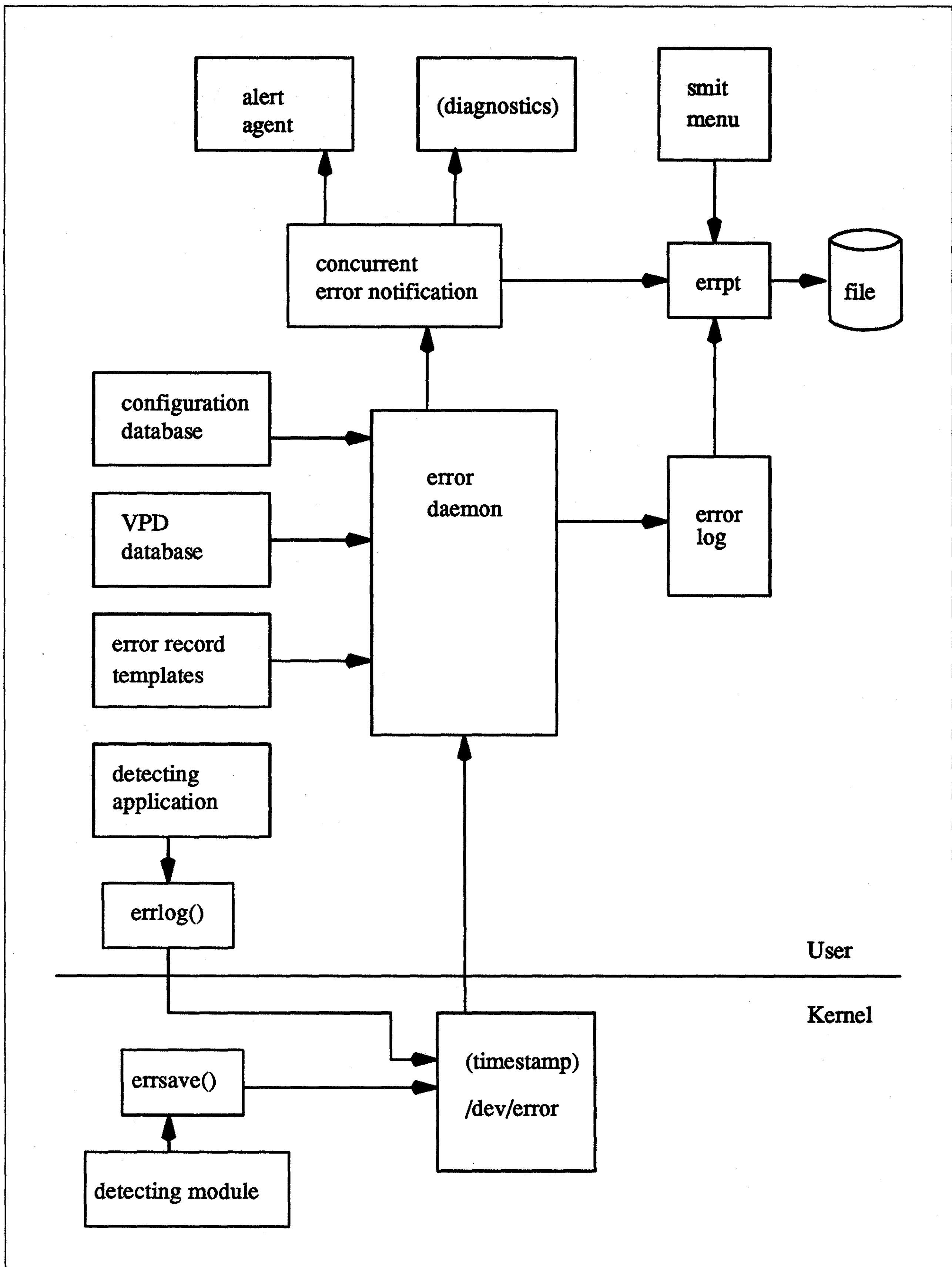


Figure 9-3. Flow of the Error Logging Facility

9.6.1 Pre-Coding Steps to Consider

9.6.1.1 Review the Error Logging Documentation

The first thing to do is to review the error logging documentation. It is beneficial to understand what services are available to developers, and what the customer, service person and defect organization will see.

9.6.1.2 Determine the Importance of the Error

Secondly, determine the importance of the error. The developer should determine whether a particular error should be logged. This may seem a trivial point but it is very important. There is no point in using system resources, i.e. service time and cost, or machine storage, for logging information which is unimportant or confusing to the intended audience. It is, however, a worse mistake **not** to log an error that merits logging. The device driver writer should work in concert with the hardware developer, if possible, to identify detectable errors and the information which should be relayed concerning those errors.

9.6.1.3 Determine the Text of the Message

Next, determine the text of the message. The developer should use the **errmsg** command with the **-w** flag to browse the system error messages file for a list of available messages. If messages are needed that are not already defined, additional steps are required. If your product is an IBM logo product, contact the the AWD Error Logging Component owner who will allocate new message identifiers. Otherwise you have the option of contacting the Error Logging Component owner or allocating your own within the user-defined range of each message set. The drawback of this is that customer-defined error messages could be overwritten.

9.6.1.4 Determine the Correct Level of Thresholding

Finally, determine the correct level of thresholding. Each error to be logged, regardless of whether it is a software or hardware error, would be limited by thresholding to avoid filling the error log with duplicate information. Side effects of *runaway error logging* include overwriting existing error log entries and unduly alarming the end user. The error log is not unlimited in size. When its size limit is reached, the log wraps. If a particular error is repeated many times needlessly, existing information will be overwritten, possibly causing inaccurate diagnostic analyses. The end user or service person may believe that a situation is more serious or pervasive than it really is if hundreds of identical or nearly identical error entries are seen. The developer is responsible for implementing the proper level of thresholding in the device driver code. The levels should be determined by the device driver developer and the hardware developer, if possible.

The error log is now 1MB long. As shipped, it will clean up any entries older than 30 days. Therefore, in order to ensure that your error log entries are actually informative, that they are noticed, and that they remain intact, ***please test your driver thoroughly !!***

9.6.2 Coding Steps

9.6.2.1 Solidifying the Error Text

The first task is to solidify the error text. After browsing the contents of the system message file (using the *errmsg -w* command), three possible paths exist for solidifying the error text. Either all of the desired messages for the new errors exist in the message file, none of the messages exist, or some combination exists.

1. If all of the messages to be used exist in the system message file, make a notation of what the four-digit hex identification number associated with it is, as well as the message set identification letter. For instance, a desired error description may be:

   ```
   SET E
   E859"The wagon wheel is broken."
   ```

2. If none of the system error messages meet requirements, determine the text of the new messages and contact the AWD Error Log owner who will allocate the new message identifiers. After the new message numbers have been allocated, build an input file suitable for use by the **errinstall** command. InfoExplorer explains the command syntax and construction of the necessary input file.

3. It is also possible to use a combination of existing messages and new messages within the same error record template definition. If new messages are needed, follow the instructions given in the previous section.

9.6.2.2 Construct Error Record Templates

Next, construct your error record templates. An **error record template** defines the text that appears in the error report. Each error record template has the following general form:

```
Error Record Template
        +LABEL:
                Comment =
                Class =
                Log =
                Report =
                Alert =
                Err_Type =
                Err_Desc =
                Probable_Causes =
                User_Causes =
                User_Actions =
                Inst_Causes =
                Inst_Actions =
                Fail_Causes =
                Fail_Actions =
                Detail_Data = <data_len>, <data_id>, <data_encoding>
```

Each field in this stanza has well defined criteria for input values. Excerpted information may be found in InfoExplorer in the discussion of the **errupdate** command.

Label A unique lable must be provided for each entry to be added. It must follow C-language rules for identifiers and must not exceed 16 characters in length.

Comment This is a comment field. The comment must be enclosed in double quotes and must not exceed 40 characters.

Class Class values are either **H** (hardware) or **S** (software).

Log The values for this field are either **TRUE** or **FALSE**. If the failure occurs, the error will only be logged if this field value is set to **TRUE**. When this value is **FALSE** the *Report* and *Alert* fields are ignored.

Report The values for this field are **TRUE** or **FALSE**. If the logged error is to be displayed using error report, the value of this field must be **TRUE**.

Alert Errors that need to be forwarded to the Network Management Alert Manager program for Alert generation should have this field set to **TRUE**. For non-alertable errors, this field should be set to **FALSE**.

Err_Type This field describes the severity of the failure that occurred. The allowable values in this field include **PERM**, **TEMP**, **PERF**, **PEND**, and **UNKN**, where:

PERM A permanent failure is defined as a condition that could not be recovered from, e.g. an operation was retried a prescribed number of times without success.

TEMP A temporary failure is defined to be a condition that was recovered from, yet the number of unsuccessful attempts exceeded a predetermined threshold.

PERF A condition in which the performance of a device or component was degraded below an acceptable level.

PEND A condition in which it has been determined that the loss of availability of a device or component is imminent.

UNKN A condition in which it is not possible to assess the severity of a failure.

Err_desc This field describes the failure that occurred. Proper input for this field would be the four-digit hex identifier of the error description message to be displayed from **SET E** in the message file.

Prob_Causes This field describes one or more probable causes for the failure that occurred. A list of up to four probable cause identifiers separated by commas may be specified. A probable cause identifier identifies a probable cause text message from **SET P** in the message file. Probable causes should be listed in the order of decreasing probability. At least one probable cause identifier is required.

User_Causes A user cause is defined to be a condition that an operator can resolve without contacting any service organization. A list of up to four user causes identifiers separated by commas may be specified. A user cause identifier identifies a user cause text message from **SET U** in the message file. User causes should be listed in the order of decreasing probability. This field may be left blank if it does not apply to the failure that occurred. If this field is left blank, either the Inst_Causes or the Fail_Causes field must be non-blank.

User_Actions This field describes recommended actions for correcting a failure that resulted from a user cause. A list of up to four recommended action identifiers separated by commas may be specified. A recommended action identifier identifies a recommended action text message, **SET R** in the message file. This field must be left blank if

User_Causes was left blank. The order in which the recommended actions are listed is determined by the expense of the action and the probability that the action will correct the failure. Actions that have little or no cost and little or no impact on system operation should always be listed first. When actions for which the probability of correcting the failure is equal or nearly equal, the least expensive action should be listed first. Remaining actions should be listed in order of decreasing probability.

Inst_Causes An install cause is defined to be a condition that resulted from the intitial installation or setup of a resource. A list of up to four install cause identifiers separated by commas may be specified. An install cause identifier identifies an install cause text message, **SET I** in the message file. The install causes should be listed in the order of decreasing probability. This field may be left blank if it is not applicable to the failure that occurred. If this field is left blank, either the User_Causes or the Failure_Causes field must be non-blank.

Inst_Actions This field is used to describe recommended actions for correcting a failure that resulted from an install_cause. A list of up to four recommended action identifiers separated by commas may be specified. A recommended action identifier identifies a recommended action text message, **SET R** in the message file. This field must be left blank if the Inst_Causes field was left blank. The order in which the recommended actions are listed is determined by the expense of the action and the probability that the action will correct the failure. See User_Actions for the list criteria.

Fail_Causes A failure cause is defined to be a condition that resulted from the failure of a resource. A list of up to four failure cause identifiers separated by commas may be specified. A failure cause identifier identifies a failure cause text message, **SET F** in the message file. The failure causes should be listed in the order of decreasing probability. This field may be left blank if it is not applicable to the failure that occurred. If this field is left blank, either the User_Causes or the Inst_Causes fields must be non-blank.

Fail_Actions This field is used to describe recommended actions for correcting a failure that resulted from a failure cause. A list of up to four recommended action identifiers separated by commas may be specified. The identifiers must correspond to recommended action messages found in **SET R** of the message file. This field must be left blank if the Fail_Causes field was blank. Refer to the description of User_Actions for criteria in listing these recommended actions.

Detail_Data This field is used to describe the detailed data, such as detecting module name, sense data, or return codes, that will be logged with the error when the failure occurs. This field must be left blank if no detailed data will be logged with the error. Three values are required for each detail data entry:

data_len The number of bytes of data to be associated with the **data_id**. **data_len** is interpreted as a decimal value.

data_id A detailed data identifier identifies a text message to be printed in the error report in front of the detailed data. These identifiers refer to messages in **SET D** of the message file.

data_encoding Describes how the detailed data is to be printed in the error report. Valid values for this field are:

ALPHA The detailed data is a printable ASCII character string.

DEC The detailed data is the binary representation of an integer value, the decimal equivalent is to be printed.

HEX The detailed data is to be printed in hex.

The Detail_Data field may be repeated. The amount of data logged with an error must not exceed the maximum error record length defined in **/usr/include/sys/erec.h**. Failure data that cannot be contained in an error log entry should be saved elsewhere, e.g. in a file, and the detailed data in the error log entry should contain information that can be used to correlate the failure data to the error log entry.

An example of an error record template is:

```
+ MISC_ERR:
        Comment = "Interrupt:I/O bus timeout or channel check"
        Class = H
        Log = TRUE
        Report = TRUE
        Alert = FALSE
        Err_Type = UNKN
        Err_Desc = E856
        Prob_Causes = 3300, 6300
        User_Causes =
        User_Actions =
        Inst_Causes =
        Inst_Actions =
        Fail_Causes = 3300, 6300
        Fail_Actions = 0000
        Detail_Data = 4, 8119, HEX              *IOCC bus number
        Detail_Data = 4, 811A, HEX              *Bus Status Register
        Detail_Data = 4, 811B, HEX              *Misc. Interrupt Register
```

Construct the error templates for all new errors to be added in a file suitable for entry in the **errupdate** command. Execute the **errupdate** command with the **-h** flag and the input file. The new errors are now part of the error record template repository. A new header file is also created (*file.h*) in the same directory in which the **errupdate** was executed. This header file must be included in the device driver code at compile time. Note that the **errupdate** command has a built-in syntax checker for the new stanza which may be invoked with the **-c** flag.

9.6.2.3 Put Error Logging Calls into the Device Driver Code

Now put error logging calls into the device driver code. The **errsave** kernel service allows the kernel and kernel extensions to write to the error log. The syntax for this command can be found in InfoExplorer and in Calls and Subroutines Vol. 5. Typically you will define a routine in the device driver which can be called by other device driver routines when a loggable error is

encountered. This function will take the data passed to it, put it into the proper structure and call **errsave**. The syntax for **errsave** is as follows:

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

void errsave(buf, cnt)
char *buf;
unsigned int cnt;
```

where,

buf pointer to a buffer that contains an error record as described in /usr/include/sys/errids.h

cnt specifies the number of bytes in the error record contained in the buffer pointed to by **buf**.

In the example of a device driver error logging routine (Figure 9-5 on page 9-43), the data to be passed to **errsave** has been defined in a structure, **dderr** and defined in a local header file, **dderr.h** (refer to Figure 9-4). You do not have to do it this way; you could simply list the fields to be filled within the function.

```
typedef struct dderr {
      struct  err_rec0 err;
      int     data1;      /* use data1 and data2 to show detail  */
      int     data2;      /* data in the errlog report. Define   */
                        /* these fields in the errlog template */
                        /* These fields may not be used in all */
                        /* cases.                              */
} dderr;
```

Figure 9-4. dderr.h. An example of an error logging structure defined in a header file.

In Figure 9-4, the first field of dderr.h is comprised of the **err_rec0** structure, which is defined in **/usr/include/sys/err_rec.h**. This structure contains the ID, or **label**, and a field for the resource name. The two **data** fields will hold the detail data for the error log report.

```
void
errsv_ex(int err_id, unsigned int port_num,
      int line, char *file, uint data1, uint data2)
{
      dderr          log;
      char           errbuf[255];
      ddex_dds          *p_dds;

      p_dds = dds_dir[port_num];
      log.err.error_id = err_id;


      if (port_num = BAD_STATE) {
            sprintf(log.err.resource_name, "%s :%d",
                  p_dds->dds_vpd.adpt_name, data1);
            data1 = 0;
      }
      else
            sprintf(log.err.resource_name, "%s",
                  p_dds->dds_vpd.devname);

      sprintf(errbuf, "line: %d file: %s", line, file);

      strncpy(log.file, errbuf, (size_t)sizeof(log.file));

      log.data1 = data1;
      log.data2 = data2;

      errsave(&log, (uint)sizeof(dderr)); /* execute actual logging  */

} /* end errlog_ex                                         */
```

Figure 9-5. errlog_ex. An example of using errsave from a device driver routine.

In Figure 9-5, the error logging routine takes data passed to it from some part of the main body of the device driver. All this code does is fill in the structure with the pertinent information, then pass it on with **errsave**.

By the way, you can log a message into the error log from the command line. To do this you use the **errlogger** command. Refer to InfoExplorer or Commands Reference Vol. 1 for a description and syntax.

After the templates have been added using **errupdate**, the device driver code should be compiled along with the new header file. The error should be simulated enabling a check that it was written to the error log correctly. Some details to check for include:

1. Is the error demon running? This can be verified by executing **ps -ef** and checking for **/usr/lib/errdemon** as part of the output.
2. Is the error part of the error template repository? This can be verified by running **errpt -at**.

3. Was the new header file, which was created by **errupdate** and which contains the error label and unique error identification number, included in the device driver code when it was compiled?

9.6.3 What Really Happens in /dev/error

Once the **errsave** information arrives at **/dev/error**, the first thing that happens is that the **errdemon** time stamps it. Then the **errdemon** looks for a match in the error template repository. A match is determined by the label and/or the eight digit hex ID found in **/usr/include/sys/errids.h**. It is possible to have errors which are not loggable, such as an *alertable* error. If the error is loggable, it is written to the error log.

The **errdemon** also checks to see if the system is set to be **concurrently notifiable**. This means that the error will be written to the screen as well as to the error log. The system is shipped with notify **on**, but you can turn it off with **SMIT**.

9.7 Performance Tracing for AIX

9.7.1 Introduction

The AIX 3.1 **trace** facility is very useful for observing device driver and system execution. The **trace** facility captures a sequential flow of time-stamped system events, providing a fine level of detail on system activity. Events are shown in time sequence and in the context of other events. The **trace** facility is useful in expanding the trace event information to understand who, when, how, and even why the event happened.

The operating system is shipped with permanent trace event points. These events provide general visibility to system execution. Users can extend the visibility into their applications by inserting additional events and providing formatting rules.

Care was taken in the design and implementation of this facility to make the collection of **trace** data efficient, so that system performance and flow would be minimally altered by activating **trace**. Because of this, the facility is extremely useful as a performance analysis tool and as a problem determination tool.

The **trace** facility is more flexible than traditional system monitor services that access and present statistics maintained by the system. With traditional monitor services, data reduction (conversion of system events to statistics) is largely coupled to the system instrumentation. For example, the system can maintain the minimum, maximum and average elapsed time observed for executions of task A and permit this information to be extracted. The **trace** facility does not strongly couple data reduction to instrumentation, but provides a stream of system events. It is not required to presuppose what statistics will be needed; the statistics or data reduction is to a large degree separated from the instrumentation. The user may choose to develop the minimum, maximum and average time for task A from the flow of events. But it is also possible to extract the average time for task A when called by process B; or the average time for task A when conditions XYZ are met; or develop a standard deviation for task A; or even decide that some other task, recognized by a stream of

events, is more meaningful to summarize. This flexibility is invaluable for diagnosing performance or functional problems.

The **trace** facility generates large volumes of data. This data cannot be captured for extended periods of time without overflowing the storage device. This allows two practical ways that the **trace** facility can be used natively. First, the **trace** facility can be triggered in multiple ways to capture small increments of system activity. It is practical to capture seconds to minutes of system activity in this way for post processing. This is sufficient time to characterize major application transactions or interesting sections of a long task. Secondly, the **trace** facility can be configured to direct the event stream to standard output. This allows a real-time process to connect to the event stream and provide data reduction in real-time, thereby creating long-term monitoring capability. A logical extension for specialized instrumentation is to direct the data stream to an auxiliary device that can either store massive amounts of data or provide dynamic data reduction.

There are three methods of starting the system trace. You can start the trace from the command line, from SMIT or from software. As shown in Figure 9-6, trace causes predefined events to be written to a trace log. The tracing action is then stopped. Tracing from a command line is discussed in "Controlling trace" on page 9-50. Tracing from a software application is discussed and an example is presented in "Examples of Coding Events and Formatting Events" on page 9-72.

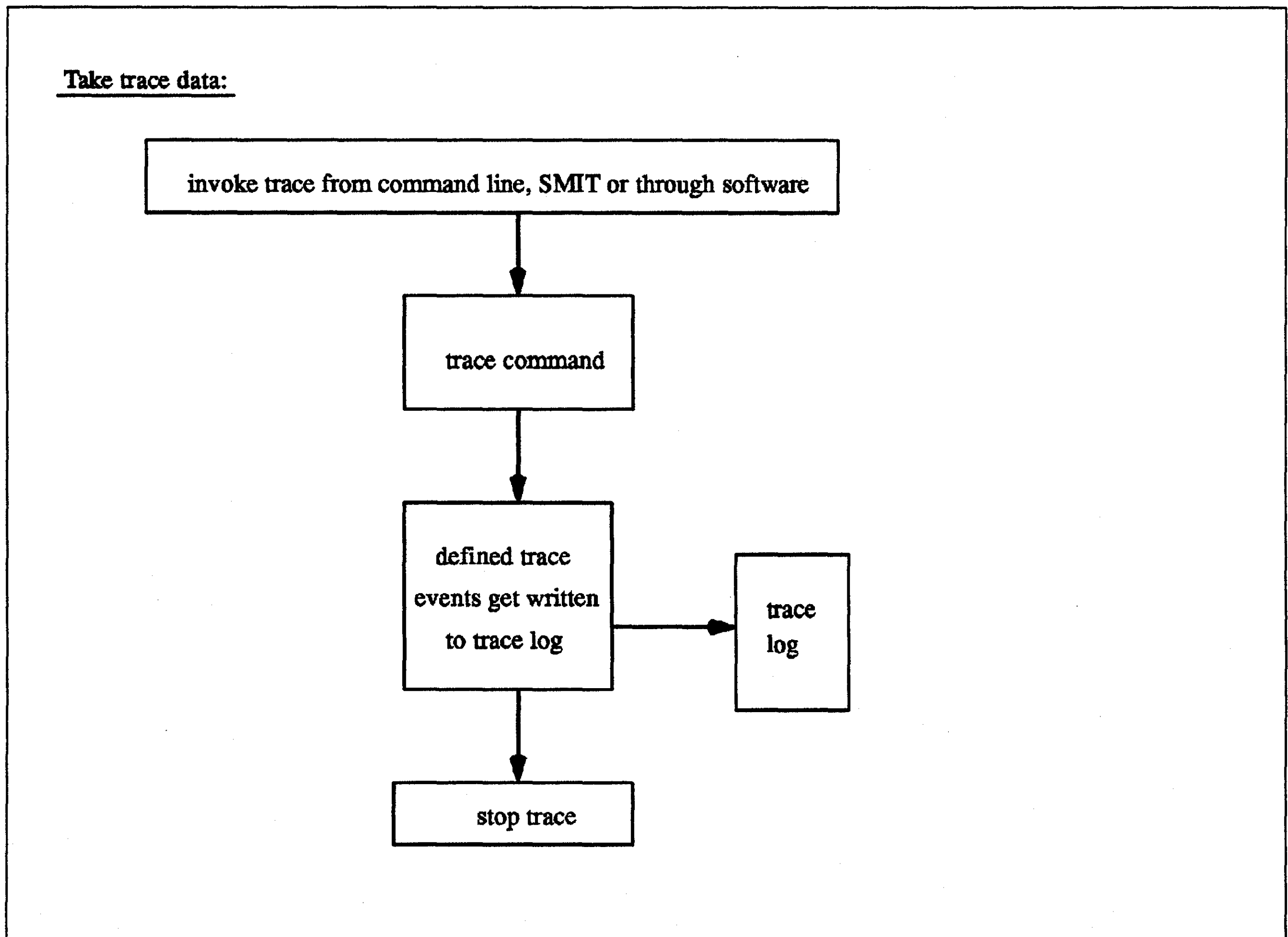


Figure 9-6. Flow Involved in Starting/Stopping Trace

After a trace has been taken (i.e. started and stopped), it must be formatted before it can be viewed. This is shown in Figure 9-7 on page 9-46. To format the trace events that you have defined, you must provide a stanza that describes how the trace formatter is to interpet the data that has been collected. This is described in "Syntax for Stanzas in the trace Format File" on page 9-60.

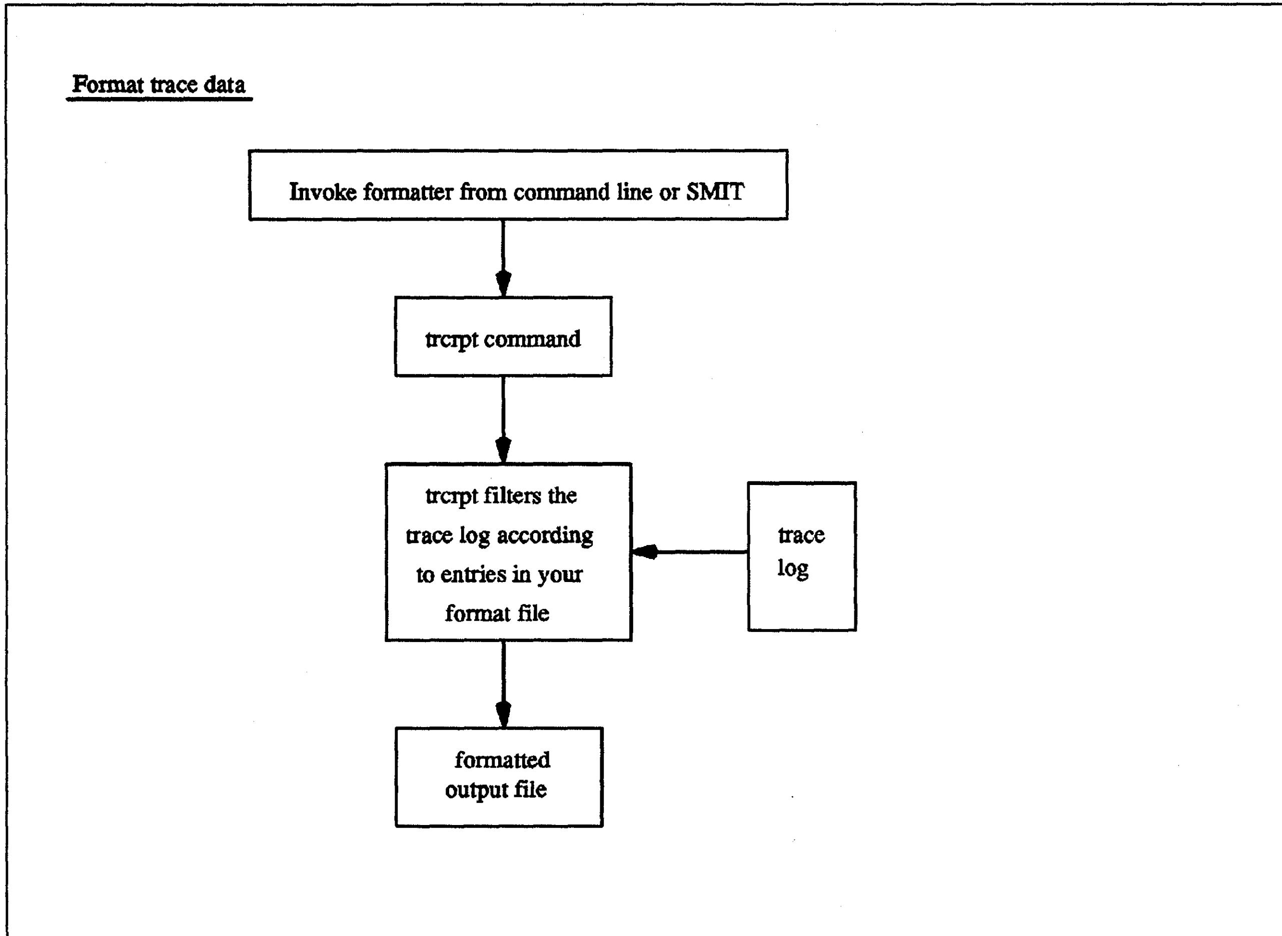


Figure 9-7. Trace Formatting

For an event to be traced, an event hook (sometimes called a trace hook) must be written into the code that you want to trace. Tracing can be done on either the system channel (channel 0) or on a generic channel (channels 1-7). All preshipped trace points are output to the system channel. Usually, when you want to show interaction with other system routines, the system channel is used. The generic channels are provided so that you can control how much data is written to the trace log. Only your data is written to one of the generic channels. The trace hooks for both the system and the generic channels are summarized in Figure 9-8 on page 9-47.

Defining trace events

Use the following trace macros to define trace events in your software:

For the system trace channel (channel 0):

```
TRCHKL0T(hw)
TRCHKL1T(hw,D1)
TRCHKL2T(hw,D1,D2)
TRCHKL3T(hw,D1,D2,D3)
TRCHKL4T(hw,D1,D2,D3,D4)
TRCHKL5T(hw,D1,D2,D3,D4,D5)
TRCHKL0(hw)
TRCHKL1(hw,D1)
TRCHKL2(hw,D1,D2)
TRCHKL3(hw,D1,D2,D3)
TRCHKL4(hw,D1,D2,D3,D4)
TRCHKL5(hw,D1,D2,D3,D4,D5)
```

For the generic trace channels (channels 1–7):

```
TRCGEN(ch,hw,dl,len,buf)
TRCGENT(ch,hw,dl,len,buf)
```

Figure 9-8. Trace Hook Summary

A general-purpose report facility is provided by the **trcrpt** command. The report facility provides little data reduction, but converts the raw binary event stream to a readable ASCII listing of the event stream. Data can be visually extracted by a reader, or tools can be developed to further reduce the data.

9.7.2 Use of the trace Facility

The following sections describe the use of the AIX 3.1 trace facility.

9.7.2.1 Configuring and Starting trace Data Collection

The trace facility is configured and data collection optionally started by the **trace** command. The syntax of this command is as follows:

```
trace [-fl] [-ad] [-s] [-h] [-jk events] [-m message] [-o outfile ]
        [-1234567] [-T buf_sz] [-L log_sz]
```

The various options of the **trace** command are described as follows:

-f or -l
Controls the capture of trace data in system memory. If neither the **-f** nor **-l** option is specified, the trace facility creates two buffer areas in system memory to capture the trace data. These buffers are alternately written to the log file (or standard output if specified) as they become full. The **-f** or **-l** flag provides the user with the ability to prevent data from being written to the file during data collection. The options are to collect data only until the memory buffer becomes full (**-f** for first), or to use the memory buffer as a circular buffer that will capture only the last set of events that occurred before **trace** was terminated (**-l**). The **-f** and **-l** options are mutually exclusive. With either the **-f** or **-l** option, data is not transferred from the memory collection buffers to file until **trace** is terminated.

-a
Specifies that the **trace** collection is to run asynchronously (as a background task), returning a normal command line prompt to the user. Without this option, the **trace** command runs in a subcommand mode (similar to crash) and returns a prompt of >. Subcommands and regular shell commands can be issued from the **trace** subcommand mode by preceding the shell commands with an exclamation point "!".

-d
Specifies that data collection is to be delayed. That is, the trace facility is only configured. Data collection is delayed until one of the collection trigger events occurs. Various methods for triggering data collection on and off are provided. These include the following:

- **trace** subcommands
- **trace** commands
- **ioctls** to **/dev/systrctl**.

-j events or -k events
Allows the user to specify a specific set of events to include (**-j**) or exclude (**-k**) from the collection process. A list of events to include or exclude is specified by a series of three-digit hexadecimal event IDs separated by a space.

-s
Specifies that **trace** data collection should terminate if the **trace** log file reaches its maximum specified size. The default without this option is to wrap and overwrite the data in the log file on a FIFO basis.

-h
Suppresses writing a **date/sysname/message** header to the **trace** log file.

-m message
Allows the user to specify a text string (message) to be included in the **trace** log header record. The message is included in reports generated by the **trcrpt** command.

-o outfile Allows the user to specify a file to use as the log file. When the **-o** option is not specified, the default log file is **/usr/adm/ras/trcfile**. The trace data can be directed to standard output by coding the **-o** option as **-o** -. This technique should be used only to pipe the data stream to another process since the trace data contains raw binary events that are not displayable.

-1234567 The **trace** design is duplicated for multiple channels. Channel 0 is the default channel and is always used for recording system events. The other channels are generic channels, and their use is not predefined. There are various uses of generic channels in the system. The generic channels are also available to user applications. Each created channel is a separate events data stream. Events recorded to channel 0 are mixed with the predefined system events data stream. The other channels have no predefined use and are assigned generically. A program can request that a generic channel be opened by using the trcstart subroutine. A channel number is returned, similar to the way a file descriptor is returned when it opens a file. The program can record events to this channel and, thus, have a private data stream. The **trace** command allows a generic channel to be specifically configured by defining the channel number with this option. However, this is not generally the way a generic channel is started. It is more likely to be started from a program using the **trcstart** subroutine, which uses the returned channel ID to record events.

-T size and -L size
Permit the user to specify the size of the collection memory buffers and the maximum size of the log file in bytes. The trace facility pins the data collection buffers, making this amount of memory unavailable to the rest of the system. It is important to be aware of this, because it means that the trace facility can impact performance in a memory-constrained environment. If the application being monitored is not memory-constrained, or if the percentage of memory consumed by the trace routine is small compared to what is available in the system, the impact of **trace** "stolen" memory should be small. If no value is specified, a default size is used. The trace facility pins a little more than the specified buffer size. This additional memory is required for the trace facility itself. A little more than the amount specified is pinned for first buffer mode (**-f** option). A little more than twice the amount specified is pinned for **trace** configured in alternate buffer or last (circular) buffer mode.

The **trace** command can be initiated from a command line. **trace** can also be initiated from a program with a subroutine call. The subroutine is **trcstart** and is in the **librts.a** library. The syntax of the **trcstart** subroutine is as follows:

```
int trcstart(char *args)
```

where args is simply the options list desired that you would enter using the trace command if starting a system trace (channel 0). If starting a generic trace, a -g option should be included in the arg string. On successful

completion trcstart returns the channel ID. For generic tracing this channel ID can be used to record to the private generic channel.

See the example of using this subroutine in Figure 9-10 on page 9-53.

When compiling a program using this subroutine, the link to the **librts.a** library must be specifically requested (use **-l rts** as a compile option).

9.7.3 Controlling trace

Once **trace** is configured by the **trace** command or the **trcstart** subroutine, controls to **trace** exist to trigger the collection of data on, trigger the collection of data off, and to stop the trace facility (stop deconfigures **trace** and unpin buffers). These basic controls are surfaced as subcommands, commands, subroutines, and ioctl controls to the **trace** control device, **/dev/systrctl**. These controls are described in the following sections.

9.7.3.1 Controlling trace in Subcommand Mode

If the **trace** routine is configured without the **-a** option, it runs in subcommand mode. Instead of the normal shell prompt, "->" is given as a prompt. In this mode the following subcommands are recognized:

trcon Triggers collection of **trace** data on.

trcoff Triggers collection of **trace** data off.

q or quit Stops collection of **trace** data (like **trcoff**) and terminates **trace** (deconfigures).

!command Runs the specified shell command.

Figure 9-9 on page 9-51 shows an example of a trace session in which the trace subcommands are used. First, the system trace points have been displayed. Second, a trace on the system calls have been selected. Of course, you can trace on more than one trace point. Be aware that trace takes a lot of data. After the trace has been taken, a trace format has been displayed. The actual formatted file was 759 lines long and only the first few lines have been displayed.

```
# trcrpt -j |pg
004 TRACEID IS ZERO
100 FLIH
200 RESUME
102 SLIH
103 RETURN FROM SLIH
101 SYSTEM CALL
104 RETURN FROM SYSTEM CALL
106 DISPATCH
10C DISPATCH IDLE PROCESS
11F SET ON READY QUEUE
134 EXEC SYSTEM CALL
139 FORK SYSTEM CALL
107 FILENAME TO VNODE (lookuppn)
15B OPEN SYSTEM CALL
130 CREAT SYSTEM CALL
19C WRITE SYSTEM CALL
163 READ SYSTEM CALL
10A KERN_PFS
10B LVM BUF STRUCT FLOW
116 XMALLOC size,align,heap
117 XMFREE address,heap
118 FORKCOPY
11E ISSIG
169 SBREAK SYSTEM CALL

# trace -d -j 101 -m "system calls trace example" -> trcon
-> !cp /tmp/xbugs . -> trcoff
-> quit
# trcrpt -O "exec=on,pid=on" >cp.trace # pg cp.trace pr  3 11:02:02 1991
System: AIX smetco Node: 3
Machine: 000247903100
Internet Address: 00000000 0.0.0.0

system calls trace example

trace -d -j 101 -m -m system calls trace example


ID  PROCESS NAME   PID      I    ELAPSED_SEC      DELTA_MSEC   APPL    SYSCALL KERNEL  INTERRUPT

001 trace          13939         0.000000000       0.000000                    TRACE ON channel 0
101 trace          13939         0.000251392       0.251392            kwritev
101 trace          13939         0.000940800       0.689408            sigprocmask
101 trace          13939         0.001061888       0.121088            kreadv
101 trace          13939         0.001501952       0.440064            kreadv
101 trace          13939         0.001919488       0.417536            kioctl
101 trace          13939         0.002395648       0.476160            kreadv
101 trace          13939         0.002705664       0.310016            kioctl
```

Figure 9-9. Trace Example Using Subcommands

9.7.3.2 Controlling trace by Commands

If the **trace** routine is configured to run asynchronously (the **-a** option), **trace** can be controlled by the following commands:

trcon Triggers collection of **trace** data on.

trcoff Triggers collection of **trace** data off.

trcstop Stops collection of **trace** data (like **trcoff**) and terminates the **trace** routine.

9.7.3.3 Controlling trace by Subroutines

The controls for the **trace** routine are available as subroutines from the **librts.a** library. The subroutines return int=0 on successful completion. The subroutines are:

trcon Triggers collection of **trace** data on.

trcoff Triggers collection of **trace** data off.

trcstop Stops collection of **trace** data (like **trcoff**) and terminates the **trace** routine.

9.7.3.4 Controlling trace with ioctls Calls

The subroutines for controlling **trace** open the trace control device (**/dev/systrctl**), issue the appropriate **ioctl**, close the control device and return. To control tracing around sections of code, it may be more efficient for a program to issue the **ioctl** controls directly. This avoids the unnecessary, repetitive opening and closing of the trace control device, at the expense of exposing some of the implementation details of **trace** control. To use the **ioctls** in a program, include <**sys/trcctl.h**> to define the **ioctl** commands. The syntax of the **ioctl** is as follows:

```
ioctl (fd, CMD, Channel)
```

where:

fd is the file descriptor returned from opening /dev/systrctl

CMD is the requested CMD TRCON TRCOFF or TRCSTOP

Channel is the trace channel (0 for system trace).

Figure 9-10 on page 9-53 shows how to start a **trace** from a program and only trace around a specified section of code.

```
#include <sys/trcctl.h>
extern int trcstart(char *arg);
char *ctl_dev ="/dev/systrctl";
int ctl_fd
main()
{
     printf("configuring trace collection \n");
     if (trcstart("-ad")){
         perror("trcstart");
         exit(1);
     }
     if((ctl_fd =open (ctl_dev))<0){
         perror("open ctl_dev");
         exit(1);
     }
     printf("turning trace collection on \n");
     if(ioctl(ctl_fd,TRCON,0)){
         perror("TRCON");
         exit(1);
     }
     /* code between here and trcoff ioctl will be traced */
     printf("turning trace off\n");
     if (ioctl(ctl_fd,TRCOFF,0)){
         perror("TRCOFF");
         exit(1);
     }
     exit(0);
}
```

Figure 9-10. Sample Code - trace Triggers

9.7.4 Producing a trace Report

9.7.4.1 Introduction

A trace report facility formats and displays the collected event stream in readable form. This report facility displays text and data for each event according to rules provided in the trace format file. The default trace format file is **/etc/trcfmt** and contains a stanza for each event ID. The stanza for the event provides the report facility with formatting rules for that event. This technique allows users to add their own events to programs and insert corresponding event stanzas in the format file to have their new events formatted. This report facility does not attempt to extract summary statistics (such as CPU utilization and disk utilization) from the event stream. This can be done in several other ways. To create simple summaries, consider using **awk** scripts to process the output obtained from the **trcrpt** command.

9.7.4.2 The trcrpt Command

The syntax of the **trcrpt** command is as follows:

```
trcrpt [-hcrjq] [-d id_list] [-k idlist] [-p process_list] [-n symbolfile]
 [-t format_file] [-O option], [option], ... [logfile]
```

Normally the **trcrpt** output goes to standard output. However, it is generally more useful to redirect the report output to a file. The **trcrpt** options are described in the following sections:

-h Causes the **trcrpt** command to omit the column headings of the report.

-c Causes the **trcrpt** command to check the syntax of the **trace** format file. The **trace** format file checked is either the default (**/etc/trcfmt**) or the file specified by the **-t** flag with this command. The user can check the syntax of the new or modified format files with this option before attempting to use them.

-r Produces a raw binary format of the trace data. Each event is output as a record in the order of occurrence. This is not necessarily the order in which the events are in the trace log file since the logfile can wrap. If this option is used, the output should be directed to a file (or process), since the binary form of the data is not displayable.

-j Causes the **trcrpt** command to produce a list of all the defined events from the specified trace format file. This option is useful in creating an initial file that the user can edit to use as an include or exclude list for the **trcrpt** or **trace** command.

-q Suppresses detailed output of syntax error messages. This is not an option the user typically uses.

-d id_list Permits the user to specify a list of events to be included in the **trcrpt** output. This is useful for eliminating information that is superfluous to a given analysis and making the volume of data in the report more manageable. A user may have commonly used event profiles, which are lists of events that are useful for a certain type of analysis.

-k id_list Similar to the **-d** flag, but specifies a list of events to exclude from the **trcrpt** output.

-p process_list
Permits the user to limit the **trcrpt** output to events that occurred during the execution of specific processes. The processes may be listed by process name or process ID.

-n symbolfile
Allows the user to specify a **symbolfile** to be used by **trcrpt** to convert kernel addresses to routine names. If not specified, the report facility uses the symbol table in **/unix**. A **symbolfile** that matches the system the data was collected on is necessary to produce an accurate **trace** report. A **symbolfile** can be created for a given level of system with the **trcnm** command as follows:

```
trcnm /unix > symbolfile
```

-t format_file
Allows the user to specify a trace format file other than the default (**/etc/trcfmt**).

-O option, option, ...
Allows the user to specify formatting options to the **trcrpt** command in a comma-separated list. (Do not put spaces after the commas.) These options take the form of option=selection. If unspecified, the default selection for the option is used. The possible options are discussed in the following sections. Each option is introduced by showing its default selection.

timestamp=0
: The report can contain two time columns. One column is elapsed time since the **trace** command was initiated. The other possible time column is the delta time between adjacent events. This option controls if and how these times are displayed. The selections are as follows:

  =0 Provides both an elapsed time from the start of **trace** and a delta time between events. The elapsed time is shown in seconds and the delta time is shown in milliseconds. Both fields show resolution to a nanosecond.

  =1 Provides only an elapsed time column displayed as seconds with resolution shown to microseconds.

  =2 Provides both an elapsed time and delta time column; elapsed time shown in seconds with nanosecond resolution, and delta time shown in microseconds with microsecond resolution.

  =3 Omits all time stamps from the report.

pagesize=0
: Permits the user to specify how often the column headings should be reprinted. The default selection of 0 displays the column headings initially only. A selection of 10 displays the column heading every 10 lines.

ids=on Permits the user to specify whether to display a column that contains the event IDs. If the selection is **on**, a three digit hex ID is shown for each event. The alternate selection is **off**.

exec=off Lets the user specify whether a column showing the path name of the current process should be displayed. This is useful in showing what process (by name) was active at the time of the event. The user typically wants to specify this option. It is recommended that the exec=on and PID=on be specified.

pid=off Lets the user specify whether a column showing the process ID of the current process is displayed. It is useful to have the process ID displayed to distinguish between several processes with the same executable name. It is recommended that exec=on and pid=on be specified.

svc=off Lets the user specify whether the report should contain a column that indicates the active system call for those events that occur while a system call is active.

starttime=nnn.nnnnnnnnn
: The **starttime** and **endtime** option permit the user to specify an elapsed time interval in which the **trcrpt** produces output. The elapsed time interval is specified in seconds with nanosecond resolution.

endtime=nnn.nnnnnnnnn
The **starttime** and **endtime** option permit the user to specify an elapsed time interval in which the **trcrpt** produces output. The elapsed time interval is specified in seconds with nanosecond resolution.

2line=off The **2line** option lets the user specify whether the lines in the event report are split and displayed across two lines. This is useful when more columns of information have been requested than can be displayed on the width of the output device.

logfile The **logfile** specifies the name of the file that contains the event data to be processed by **trcrpt**. The default is the **/usr/adm/ras/trcfile** file.

9.7.5 Defining trace Events

The operating system is shipped with predefined trace hooks (events). The user need only activate **trace** to capture the flow of events from the operating system. Device driver developers may want to define trace events in their program code during development for tuning purposes. This provides them with insight into how their program is interacting with the system. The following sections provide the information that is required to do this.

9.7.5.1 Possible Forms of a trace Event Record

A **trace** event can take several forms. An event consists of a hook word, optional data words, and an optional time stamp. This is pictured in Figure 9-11 on page 9-57. A four-bit type is defined for each form the event record can take. The type field is imposed by the recording routine so that the report facility can always skip from event to event when processing the data, even if the formatting rules in the **trace** format file are incorrect or missing for that event.

12 bit hook id	4 bit type	16 bit data field
D1 Optional data word 1		
D2 Optional data word 2		
D3 Optional data word 3		
D4 Optional data word 4		
D5 Optional data word 5		
Optional Timestamp		

Figure 9-11. Format of a trace Event Record

An event record should be as short as possible. Many system events use only the hookword and timestamp. There is another event type that is mentioned but should seldom be used because it is less efficient. It is a long format that allows the user to record a variable length of data. In this long form, the 16-bit data field of the hookword is converted to a *length* field that describes the length of the event record.

9.7.5.2 Macros for Recording trace Events

There is a macro to record each possible type of event record. The macros are defined in the **<sys/trcmacros.h>** header file. The event IDs are defined in the **<sys/trchkid.h>** header file. These two include files should be in any program that is recording **trace** events. The macros to record system (channel 0) events with a time stamp are listed as follows:

- **TRCHKL0T**(hw)
- **TRCHKL1T**(hw,D1)
- **TRCHKL2T**(hw,D1,D2)
- **TRCHKL3T**(hw,D1,D2,D3)
- **TRCHKL4T**(hw,D1,D2,D3)
- **TRCHKL5T**(hw,D1,D2,D3,D4,D5).

Similarly, to record non-time stamped system events (channel 0), the following macros should be used:

- **TRCHKL0**(hw)
- **TRCHKL1**(hw,D1)

- **TRCHKL2**(hw,D1,D2)
- **TRCHKL3**(hw,D1,D2,D3)
- **TRCHKL4**(hw,D1,D2,D3,D4)
- **TRCHKL5**(hw,D1,D2,D3,D4,D5).

There are only two macros to record events to one of the generic channels (channels 1-7). These are as follows:

- **TRCGEN**(ch,hw,d1,len,buf)
- **TRCGENT**(ch,hw,d1,len,buf).

These macros record a hookword (hw), a data word (d1) and a length of data (len) specified in bytes from the user's data segment at the location specified (buf) to the event stream specified by the channel (ch).

9.7.5.3 Use of Event IDs (hookids)

Event IDs are 12 bits (or 3-digit hexadecimal), for a possibility of 4096 IDs. Event IDs that are permanently left in and shipped with code need to be permanently assigned by IBM. Permanently assigned event IDs are defined in the **<sys/trchkid.h>** header file. To allow users to define events in their environments or during development, a range of event IDs has been set aside for temporary use. The range of event IDs for temporary use is hex 010 through hex 0ff. No permanent (shipped) events are assigned in this range. Users can freely use this range of IDs *in their own environment.* It is important that users who make use of this event range do not let the code leave their environment.

Permanent events must have event IDs assigned by the current owner of the trace component. Event IDs should be conserved because they are limited. Event IDs can be extended by the data field. The only reason to have a unique ID is that an ID is the level at which collection and report filtering is available in the trace facility. An ID can be collected or not collected by the trace collection process and reported or not reported by the trace report facility. Whole applications can be instrumented using only one event ID; the only restriction is that the granularity on choosing visibility is to choose whether events for that application are visible.

A new event can be formatted by the trace report facility (**trcrpt** command) if a stanza is created for the event in the trace format file. The trace format file is an editable ASCII file. The syntax for a format stanza "Syntax for Stanzas in the trace Format File" on page 9-60. All permanently assigned event IDs should have an appropriate stanza in the default trace format file shipped with the base operating system.

9.7.5.4 Suggested Locations and Data for Permanent Events

The intent of permanent events is to give an adequate level of visibility to determine execution, and data flow and have an adequate accounting for how CPU time is being consumed. During code development, it may be desirable to make very detailed use of trace for a component. For example, one may choose to trace the entry and exit of every subroutine in order to understand and tune pathlength. However, this would generally be an excessive level of instrumentation to ship for a component. It is suggested that a performance analyst be consulted for decisions regarding what events and data to capture

as permanent events for a new component. The following paragraphs provide some guidelines for these decisions.

Events should capture execution flow and data flow between major components or major sections of a component. For example, there are existing events that capture the interface between the virtual memory manager and the logical volume manager. If work is being queued, data that identifies the queued item (a handle) should be recorded with the event. When a queue element is being processed, the "dequeue" event should provide this identifier as data also, so that the queue element being serviced is identified.

Data or requests that are identified by different handles at different levels of the system should have events and data that allow them to be uniquely identified at any level. For example, a read request to the physical file system is identified by a file descriptor and a current offset in the file. To VMM the same request is identified by a segment ID and a virtual page address. At the disk device driver level this request is identified as pointer to a structure (which contains pertinent data for the request). The file descriptor or segment information is not available at the device driver level. Events must provide the necessary data to link these identifiers so that, for example, when a disk interrupt occurs for incoming data the identifier at that level (which may simply be the buffer address for where the data will be copied) can be linked to the original user request for data at some offset into a file.

Events should provide visibility to major protocol events such as requests, responses, acknowledgements, errors, retries, etc. If a request at some level is fragmented into multiple requests, a trace event should indicate this and supply linkage data to allow the multiple requests to be tracked from that point. If multiple requests at some level are coalesced into a single request a trace event also should indicate this and provide appropriate data to track the new request.

Events should be used to give visibility to resource consumption. Whenever resources are claimed, returned, created or deleted an event should record the fact. For example, claiming or returning buffers to a buffer pool or growing or shrinking the number of buffers in the pool.

TRACE HOOK GUIDELINES

The following represent some guidelines in determining where and when you should have trace hooks in your code:

- Tracing entry and exit points of every function is not necessary. Provide only key actions and data.
- Show linkage between major code blocks or processes.
- If work is queued, associate a name (handle) with it and output it as data.
- If a queue is being serviced, the trace event should indicate the unique element being serviced.
- If a work request/response is being referenced by different handles as it passes through different software components, trace the transactions so the action/receipt can be identified.
- Place trace hooks so that requests, responses, errors and retries can be observed.
- Identify when resources are claimed, returned, created or destroyed.

Please note:

- A trace ID can be used for a group of events by "switching" on one of the data fields. This means that a particular data field can be used to identify where the trace point was called from. The trace format routine can be made to format the trace data for that unique trace point.
- The trace hook is the level at which a group of events can be enabled or disabled.

9.7.5.5 Syntax for Stanzas in the trace Format File

The intent of the trace format file is to provide rules for presentation and display of the expected data for each event ID. This allows new events to be formatted without changing the report facility. Rules for new events are simply added to the format file. The syntax of the rules provide flexibility in the presentation of the data.

It may be helpful to refer to /etc/tcrfmt for examples of the syntax described below.

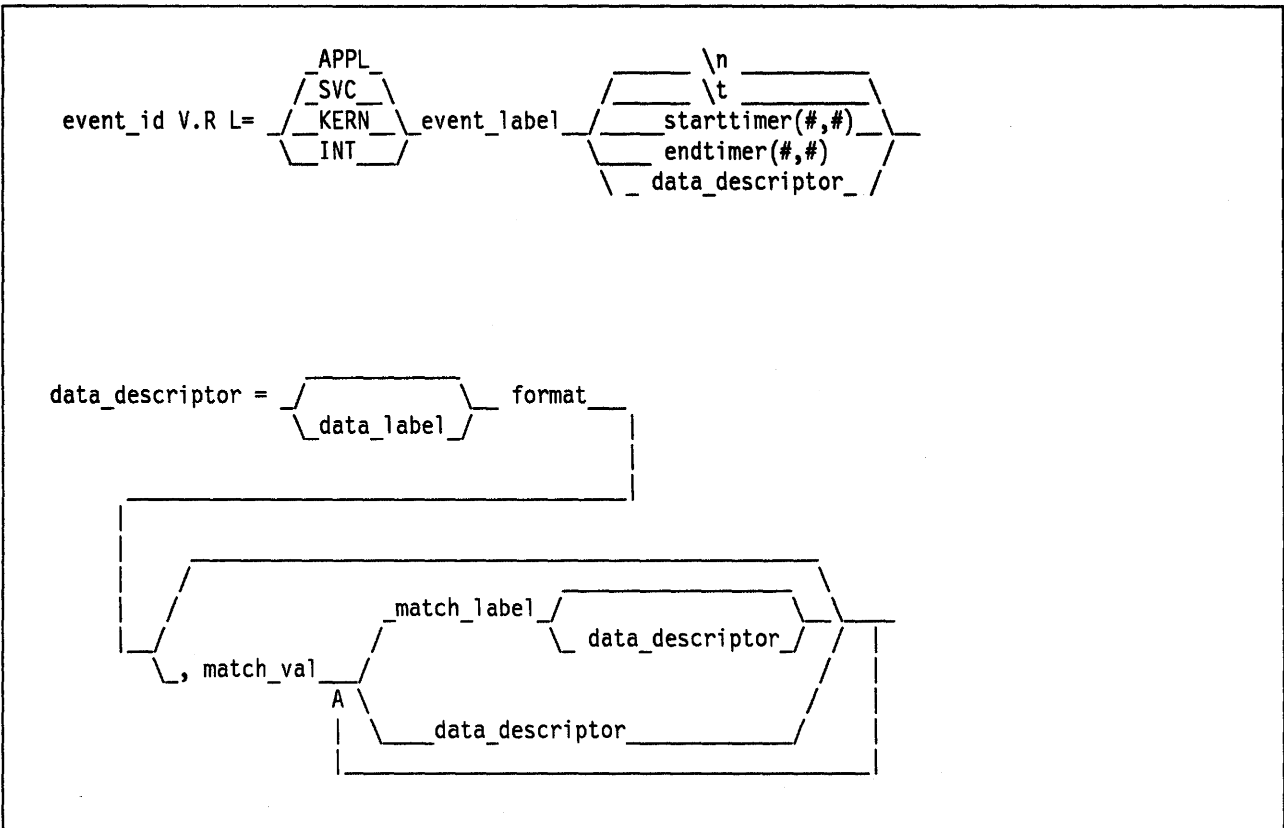


Figure 9-12. Syntax of Stanza in Format File

Figure 9-12 is a syntax diagram for a stanza in the trace format file. A trace format stanza can be as long as required to describe the rules for any particular event. The stanza can be continued to the next line by terminating the present line with a "\" character. The syntax looks complex, but is readily explainable. The descriptions of the fields follow:

event_id Each stanza begins with the three-digit hexadecimal event ID that the stanza describes, followed by a space.

V.R This field describes the version (V) and release (R) that the event was first assigned. Any integers will work for V and R, and users may want to keep their own tracking mechanism.

L= The text description of an event can begin at various indentation levels. This improves the readability of the report output. The indentation levels correspond to the level at which the system is executing. The recognized levels are application level (APPL), a transitioning system call (SVC), kernel level (KERN), and interrupt (INT).

event_label
The event_label should be an ASCII text string that describes the overall use of the event ID. This is used by the -j option of the **trcrpt** to provide a listing of events and their first level description. The event label also appears in the formatted output for the event unless the event_label starts with an @ character.

\n The event stanza describes how to parse, label and present the data contained in an event record. The \n (newline) function can be embedded in the event stanza to continue data presentation of the

data on a new line. This allows the presentation of the data for an event to be several lines long.

\t
The \t (tab) function inserts a tab at the point it is encountered in parsing the description. This is similar to the way the "\n" function inserts new lines. Spacing can also be inserted by spaces in the data_label or match_label fields.

starttimer(#,#)
The starttimer and endtimer fields work together. The (#,#) field can be thought of as a unique identifier that associates a particular starttimer with an endtimer with the same identifier. By convention, if possible, the identifiers should be (ID of starting event, ID of ending event). When the report facility encounters a starttimer directive while parsing an event, it associates the starting events time with the unique identifier. When an endtimer with the same identifier is encountered, the report facility outputs the delta time (this appears in brackets) that elapsed between the starting event and ending event. The begin and end system call events make use of this capability. On the return from system call event, a delta time is shown that indicates how long the system call took.

endtimer(#,#)
See the starttimmer field in the preceding paragraph.

data_descriptor
The data_descriptor field is the fundamental field that describes how the data should be consumed, labeled, and presented by the report facility. The syntax of the data_descriptor field is expanded in the second part of Figure 9-12 on page 9-61. The various fields of the data_descriptor are described as follows:

format
Review the format of an event record depicted in Figure 9-11 on page 9-57. The user can think of the report facility as having a pointer into the data portion of an event. This data pointer is initialized to point to the beginning of the event data (the 16-bit data field in the hookword). The format field describes how much data the report facility should consume from this point and how the data should be considered. For example, a format field of Bm.n tells the report facility to consume m bytes and n bits of data and to consider it as binary data. (The possible format fields are described in following sections.) If the format field is not followed by a comma, the report facility outputs the consumed data in the format specified. If, however, the format field is followed by a comma, it signifies that the data is not to be displayed but instead compared against the following match_values. The data descriptor associated with the matching match_value is then applied to the remainder of the data.

data_label
The data label is an ASCII string that can optionally precede the output of data consumed by the following format field.

match_value
The match value is data of the same format described by the preceding format fields. Several match values

typically follow a format field that is being matched. The successive match fields are separated by commas. The last match value is not followed by a comma. A * is used as a pattern-matching character to match anything. A pattern-matching character is frequently used as the last match_value field to specify default rules if the preceding match_values field did not occur.

match_label
The match label is an ASCII string that will be output for the corresponding match.

Each of the possible format fields are described in the comments of the **/etc/trcfmt** file. A brief introduction to the possibilities is provided here:

Format field descriptions

Am.n This specifies that m bytes of data should be consumed as ASCII text, and that it should be displayed in an output field that is n characters wide. The data pointer is moved m bytes.

S1, S2, S4 Left justified string. The length of the field is defined as 1 byte (S1), 2 bytes (S2), or 4 bytes (S4). The data pointer is moved accordingly.

Bm.n Binary data of m bytes and n bits. The data pointer is moved accordingly.

Xm Hexadecimal data of m bytes. The data pointer is moved accordingly.

D2, D4 Signed decimal format. Data length of 2 (D2) bytes or 4 (D4) bytes is consumed.

U2, U4 Unsigned decimal format. Data length of 2 or 4 bytes is consumed.

F4, F8 Floating point of 4 or 8 bytes.

Gm.n This format field merely positions the data pointer. It specifies that the data pointer should be positioned m bytes and n bits into the data.

Om.n This format field skips or omits data. It omits m bytes and n bits.

Rm This reverses the data pointer m bytes.

Some macros are provided that can be used as format fields to quickly access data. For example:

$D1, $D2, $D3, $D4, $D5
These macros access data words 1 through 5 of the event record without moving the data pointer. The data accessed by a macro is hexadecimal by default. A macro can be cast to a different data type (X, D, U, B) by using a % character followed by the new format code. For example:

```
$D1%B2.3
```

This macro causes data word one to be accessed, but to be considered as 2 bytes and 3 bits of binary data.

$HD This macro accesses the first 16 bits of data contained in the hookword, in a similar manner as the $D1 through $D5 macros access the various data words. It is also considered as hexadecimal data, and also can be cast.

You can define other macros and use other formatting techniques in the trace format file. This is shown in Figure 9-13 on page 9-65.

```
# Licensed Materials - Property of IBM
#
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# I. General Information
#
# A. Binary format for the tracehook calls. (1 column = 4 bits)
#    trchk       MMmTDDDD
#    trchkt      MMmTDDDDtttttttt
#    trchkl      MMmTDDDD11111111
#    trchklt     MMmTDDDD11111111tttttttt
#    trchkg      MMmTDDDD1111111122222222333333334444444455555555
#    trchkg      MMmTDDDD1111111122222222333333334444444455555555tttttttt
#    trcgen      MMmTLLLL11111111vvvvvvvvvvvvvvvvvvvvvvvvvvvvxxxxxx
#    trcgent     MMmTLLLL11111111vvvvvvvvvvvvvvvvvvvvvvvvvvvvxxxxxxtttttttt
#
#        legend:
#      MM = major id
#      m  = minor id
#      T  = hooktype
#      D  = hookdata
#      t  = nanosecond timestamp
#      1  = d1   (see trchkid.h for calling syntax for the tracehook routines)
#      2  = d2, etc.
#      v  = trcgen variable length buffer
#      L  = length of variable length data in bytes.
#
#   The DATA_POINTER starts at the third byte in the event, ie.,
#     at the 16 bit hookdata DDDD.
#   The trcgen() type (6,7) is an exception. The DATA_POINTER starts at
#     the fifth byte, ie., at the 'd1' parameter 11111111.
#
# B. Indentation levels
#    The left margin is set per template using the 'L=XXXX' command.
#    The default is L=KERN, the second column.
#    L=APPL moves the left margin to the first  column.
#    L=SVC  moves the left margin to the second column.
#    L=KERN moves the left margin to the third  column.
#    L=INT  moves the left margin to the fourth column.
#    The command if used must go just after the version code.
#
#    Example usage:
#113 1.7 L=INT "stray interrupt" ... \
#
# C. Continuation code and delimiters.
#    A '\' at the end of the line must be used to continue the template
#      on the next line.
#    Individual strings (labels) can be separated by one or more blanks
#      or tabs. However, all whitespace is squeezed down to 1 blank on
#      the report. Use '\t' for skipping to the next tabstop, or use
#      A0.X format (see below) for variable space.
#
```

Figure 9-13 (Part 1 of 7). Trace Format File Syntax

```
#
# II. FORMAT codes
#
# A. Codes that manipulate the DATA_PONTER
# Gm.n
#     "Goto"    Set DATA_POINTER to byte.bit location m.n
#
# Om.n
#     "Omit"    Advance DATA_POINTER by m.n byte.bits
#
# Rm
#     "Reverse" Decrement DATA_POINTER by m bytes. R0 byte aligns.
#
# B. Codes that cause data to be output.
# Am.n
#     Left justified ascii.
#     m=length in bytes of the binary data.
#     n=width of the displayed field.
#     The data pointer is rounded up to the next byte boundary.
#     Example
#      DATA_POINTER|
#                  V
#             xxxxxhello world\0xxxxxx
#
#  i.   A8.16 results in:                       |hello wo        |
#       DATA_POINTER--------|
#                           V
#              xxxxxhello world\0xxxxxx
#
#  ii.  A16.16 results in:                      |hello world     |
#       DATA_POINTER----------------|
#                                   V
#              xxxxxhello world\0xxxxxx
#
#  iii. A16 results in:                         |hello world|
#       DATA_POINTER----------------|
#                                   V
#              xxxxxhello world\0xxxxxx
#
#  iv.  A0.16 results in:                       |                |
#       DATA_POINTER|
#                   V
#              xxxxxhello world\0xxxxxx
```
Figure 9-13 (Part 2 of 7). Trace Format File Syntax

```
#
# S1, S2, S4
#     Left justified ascii string.
#     The length of the string is in the first byte (half-word, word) of
#      the data. This length of the string does not include this byte.
#     The data pointer is advanced by the length value.
#     Example
#      DATA_POINTER|
#                  V
#             xxxxxBhello worldxxxxxx     (B = hex 0x0b)
#
#  i.   S1 results in:                          |hello world|
#       DATA_POINTER-----------|
#                              V
#              xxxxBhello worldxxxxxx
#
# $reg%S1
#     A register with the format code of 'Sx' works "backwards" from
#     a register with a differnet type. The format is Sx, but the length
#     of the string comes from $reg instead of the next n bytes.
#
# Bm.n
#     Binary format.
#     m = length in bytes
#     n = length in bits
#     The length in bits of the data is m * 8 + n. B2.3 and B0.19 are the same.
#     Unlike the other printing FORMAT codes, the DATA_POINTER
#     can be bit aligned and is not rounded up to the next byte boundary.
#
# Xm
#     Hex format.
#     m = length in bytes. m=0 thru 16
#     The DATA_POINTER is advanced by m.
#
# D2, D4
#     Signed decimal format.
#     The length of the data is 2 (4) bytes.
#     The DATA_POINTER is advanced by 2 (4).
#
# U2, U4
#     Unsigned decimal format.
#     The length of the data is 2 (4) bytes.
#     The DATA_POINTER is advanced by 2 (4).
#
# F4
#     Floating point format. (like %0.4E)
#     The length of the data is 4 bytes.
#     The format of the data is that of C type 'float'.
#     The DATA_POINTER is advanced by 4.
#
# F8
#     Floating point format. (like %0.4E)
#     The length of the data is 8 bytes.
#     The format of the data is that of C type 'double'.
#     The DATA_POINTER is advanced by 8.
#
```

Figure 9-13 (Part 3 of 7). Trace Format File Syntax

```
# HB
#     Number of bytes in trcgen() variable length buffer.
#     This is also equal to the 16 bit hookdata.
#     The DATA_POINTER is not changed.
#
# HT
#     The hooktype. (0 - E)
#     trcgen  = 0, trchk  = 1, trchl  = 2, trchkg  = 6
#     trcgent = 8, trchkt = 9, trchlt = A, trchkgt = E
#     HT & 0x07 masks off the timestamp bit
#     This is used for allowing multiple, different trchkx() calls with
#       the same template.
#     The DATA_POINTER is not changed.
#
# C. Codes that interpret the data in some way before output.
# T4
#     Output the next 4 bytes as a data and time string,
#     in GMT timezone format.  (as in ctime(&seconds))
#     The DATA_POINTER is advanced by 4.
#
# E1,E2,E4
#     Output the next byte (half_word, word) as an 'errno' value, replacing
#     the numeric code with the corresponding #define name in
#     /usr/include/sys/errno.h
#     The DATA_POINTER is advanced by 1, 2, or 4.
#
# P4
#     Use the next word as a process id (pid), and
#     output the pathname of the executable with that process id.
#     Process ids and their pathnames are acquired by the trace command
#     at the start of a trace and by trcrpt via a special EXEC tracehook.
#     The DATA_POINTER is advanced by 4.
#
# \t
#     Output a tab. \t\t\t outputs 3 tabs. Tabs are expanded to spaces,
#     using a fixed tabstop separation of 8. If L=0 indentation is used,
#     the first tabstop is at 3.
#     The DATA_POINTER advances over the \t.
#
# \n
#     Output a newline. \n\n\n outputs 3 newlines.
#     The newline is left-justified according to the INDENTATION LEVEL.
#     The DATA_POINTER advances over the \n.
#
# $macro
#     The value of 'macro' is output as a %04X value. Undefined macros
#     have the value of 0000.
#     The DATA_POINTER is not changed.
#     An optional format can be used with macros:
#        $v1%X4    will output the value $v1 in X4 format.
#        $zz%B0.8  will output the value $v1 in 8 bits of binary.
#     Understood formats are: X, D, U, B. Others default to X2.
#
```

Figure 9-13 (Part 4 of 7). Trace Format File Syntax

```
# "string"     'string' data type
#     Output the characters inside the double quotes exactly. A string
#     is treated as a descriptor. Use "" as a NULL string.
#
#  string format $macro  If a string is backquoted, it is expanded
#     as a quoted string, except that FORMAT codes and $registers are
#     expanded as registers.
#
# III. SWITCH statement
#     A format code followed by a comma is a SWITCH statement.
#     Each CASE entry of the SWITCH statement consists of
#       1. a 'matchvalue' with a type (usually numeric) corresponding to
#          the format code.
#       2. a simple 'string' or a new 'descriptor' bounded by braces.
#          A descriptor is a sequence of format codes, strings, switches,
#          and loops.
#       3. and a comma delimiter.
#       The switch is terminated by a CASE entry without a comma delimiter.
#     The CASE entry is selected to as the first entry whose matchvalue
#     is equal to the expansion of the format code.
#     The special matchvalue '\*' is a wildcard and matches anything.
#     The DATA_POINTER is advanced by the format code.
#
#
# IV. LOOP statement
#     The syntax of a 'loop' is
#     LOOP format_code { descriptor }
#     The descriptor is executed N times, where N is the numeric value
#       of the format code.
#     The DATA_POINTER is advanced by the format code plus whatever the
#       descriptor does.
#     Loops are used to output binary buffers of data, so descriptor is
#       usually simply X1 or X0. Note that X0 is like X1 but does not
#       supply a space separator ' ' between each byte.
#
#
# V. macro assignment and expressions
#   'macros' are temporary (for the duration of that event) variables
#   that work like shell variables.
#   They are assigned a value with the syntax:
#   {{ $xxx = EXPR }}
#   where EXPR is a combination of format codes, macros, and constants.
#   Allowed operators are + - / *
#   For example:
#{{ $dog = 7 + 6 }} {{ $cat = $dog * 2 }} $dog $cat
#
#   will output:
#000D 001A
#
#   Macros are useful in loops where the loop count is not always
#   just before the data:
#G1.5 {{ $count = B0.5 }} G11 LOOP $count {X0}
#
```

Figure 9-13 (Part 5 of 7). Trace Format File Syntax

```
#   Up to 25 macros can be defined per template.
#
#
# VI. Special macros:
# $RELLINENO   line number for this event. The first line starts at 1.
# $D1 - $D5    dataword 1 through dataword 5. No change to datapointer.
# $HD          hookdata (lower 16 bits)
# $SVC         Output the name of the current SVC
# $EXECPATH    Output the pathname of the executable for current process.
# $PID         Output the current process id.
# $ERROR       Output an error message to the report and exit from the
#              template after the current descriptor is processed.
#              The error message supplies the logfile, logfile offset of the
#              start of that event, and the traceid.
# $LOGIDX      Current logfile offset into this event.
# $LOGIDX0     Like $LOGIDX, but is the start of the event.
# $LOGFILE     Name of the logfile being processed.
# $TRACEID     Traceid of this event.
# $DEFAULT     Use the DEFAULT template 008
# $STOP        End the trace report right away
# $BREAK       End the current trace event
# $SKIP        Like break, but don't print anything out.
# $DATAPOINTER The DATA_POINTER. It can be set and manipulated
#              like other user-macros.
#              {{ $DATAPOINTER = 5 }} is equivalent to G5
# $BASEPOINTER Usually 0. It is the starting offset into an event. The actual
#              offset is the DATA_POINTER + BASE_POINTER. It is used with
#              template subroutines, where the parts on an event have the
#              same structure, and can be printed by the same template, but
#              may have different starting points into an event.
#
# VII. Template subroutines
#    If a macro name consists of 3 hex digits, it is a "template subroutine".
#    The template whose traceid equals the macro name is inserted in place
#    of the macro.
#
#    The data pointer is where is was when the template
#    substitution was encountered. Any change made to the data pointer
#    by the template subroutine remains in affect when the template ends.
#
#    Macros used within the template subroutine correspond to those in the
#    calling template. The first definition of a macro in the called template
#    is the same variable as the first in the called. The names are not
#    related.
#
```

Figure 9-13 (Part 6 of 7). Trace Format File Syntax

```
#    Example:
#    Output the trace label ESDI STRATEGY.
#    The macro '$stat' is set to bytes 2 and 3 of the trace event.
#    Then call template 90F to interpret a buf header. The macro '$return'
#    corresponds to the macro '$rv', since they were declared in the same
#    order. A macro definition with no '=' assignment just declares the name
#    like a place holder. When the template returns, the saved special
#    status word is output and the returned minor device number.
#
#900 1.0 "ESDI STRATEGY" {{ $rv = 0 }} {{ $stat = X2 }} \
#       $90F \n\
#special_esdi_status=$stat for minor device $rv
#
#90F 1.0 "" G4 {{ $return }} \
#       block number X4 \n\
#       byte count   X4 \n\
#       B0.1, 1 B_FLAG0 \
#       B0.1, 1 B_FLAG1 \
#       B0.1, 1 B_FLAG2 \
#       G16 {{ $return = X2 }}
#
#
#    Note: The $DEFAULT reserved macro is the same as $008
#
# VII. BITFLAGS statement
#     The syntax of a 'bitflags' is
#     BITFLAGS [format_code|register],
#         flag_value string {optional string if false},   or
#         '&' mask field_value string,
#                 ...
#
#     This statement simplifies expanding state flags, since it look
#       a lot like a series of #defines.
#     The '&' mask is used for interpreting bit fields.
#     The mask is anded to the register and the result is compared to
#       the field_value. If a match, the string is printed.
#     The base is 16 for flag_values and masks.
#     The DATA_POINTER is advanced if a format code is used.
#     Note: the default base for BITFLAGS is 16. If the mask or field value
#     has a leading 0, the number is octal. 0x or 0X makes the number hex.


# A 000 traceid will use this template
# This id is also used to define most of the template functions.
#  filemode(omode)    expand omode the way  ls -l does. The
#                     call to setdelim() inhibits spaces between the chars.
#
```

Figure 9-13 (Part 7 of 7). Trace Format File Syntax

9.7.5.6 Examples of Coding Events and Formatting Events

There are five basic steps involved in generating a trace from your software program.

Step 1: Enable and disabling of trace

Enable the trace from your software that has the trace hooks defined. Figure 9-14 shows the use of **trace** events to time the execution of a program loop:

```
#include <sys/trcctl.h>
#include <sys/trcmacros.h>
#include <sys/trchkid.h>
char *ctl_file = "/dev/systrctl";
int ctlfd;
int i;
main()
{
     printf("configuring trace collection \n");
     if (trcstart("-ad")){
         perror("trcstart");
         exit(1);
     }
     if((ctlfd = open(ctl_file,0))<0){
          perror(ctl_file);
          exit(1);
     }
     printf("turning  trace on \n");
     if(ioctl(ctlfd,TRCON,0)){
          perror("TRCON");
          exit(1);
     }
     /* here is the code that is being traced */
     for(i=1;i<11;i++){
          TRCHKL1T(HKWD_USER1,i);
     /*   sleep(1) */
     /* you can uncomment sleep to make the loop take longer */
          /* If you do you will want to filter the output */
          /* Or you will be overwhelmed with 11 seconds of data */
     }
     /* stop tracing code */
     printf("turning trace off\n");
     if(ioctl(ctlfd,TRCSTOP,0)){
          perror("TRCOFF");
          exit(1);
     }
     exit(0);
}
```

Figure 9-14. Sample C Code---Trace Program Loop

Step 2: Compile your program

When you compile the sample program, you need to link to the **librts.a** library as follows:

```
cc -o sample sample.c -l rts
```

Step 3: Run the program

Run the program. In this case, it can be done with the following command:

```
./sample
```

(Note that you need to be "su" to do this if the default file is used to collect the trace information (/usr/adm/ras/trcfile)).

Step 4: Add stanza to format file

This provides the report generator with the information to correctly format your file. The report facility does not know how to format the **HKWD_USER1** event, unless rules are provided in the **trace** format file. The following example of a stanza for HKWD_USER1 could be used. HKWD_USER1 is event ID 010 hexadecimal (you can verify this by looking at the **<sys/trchkid.h>** file.

```
# User event HKWD_USER1 Formatting Rules Stanza
# An example that will format the event usage of the sample program
010 1.0 L=APPL "USER EVENT - HKWD_USER1" 02.0 \n\
               "The # of loop iterations =" U4\n\
               "The elapsed time of the last loop = "\
                endtimer(0x010,0x010) starttimer(0x010,0x010)
```

Figure 9-15. Sample Trace Event Stanza

PROGRAMMING HINT

When entering the example stanza (Figure 9-15), do not modify the master format file **/etc/trcfmt**. Instead, make a copy and keep it in your own directory. This will enable you to always have the original trace format file available.

Step 5: Run the format/filter program

You probably want to filter the output report to get only your events. To do this, run the **trcrpt** command as follows:

```
trcrpt -d 010 -t mytrcfmt -O exec=on -o sample.rpt
```

See the results in Figure 9-16 on page 9-74.

```
ID  PROCESS NAME   I     ELAPSED_SEC      DELTA_MSEC   APPL    SYSCALL KERNEL  INTERRUPT

010 sample               0.000105984        0.105984   USER HOOK 1
                                                       The data field for the user hook = 1
010 sample               0.000113920        0.007936   USER HOOK 1
                                                       The data field for the user hook = 2 [7 usec]
010 sample               0.000119296        0.005376   USER HOOK 1
                                                       The data field for the user hook = 3 [5 usec]
010 sample               0.000124672        0.005376   USER HOOK 1
                                                       The data field for the user hook = 4 [5 usec]
010 sample               0.000129792        0.005120   USER HOOK 1
                                                       The data field for the user hook = 5 [5 usec]
010 sample               0.000135168        0.005376   USER HOOK 1
                                                       The data field for the user hook = 6 [5 usec]
010 sample               0.000140288        0.005120   USER HOOK 1
                                                       The data field for the user hook = 7 [5 usec]
010 sample               0.000145408        0.005120   USER HOOK 1
                                                       The data field for the user hook = 8 [5 usec]
010 sample               0.000151040        0.005632   USER HOOK 1
                                                       The data field for the user hook = 9 [5 usec]
010 sample               0.000156160        0.005120   USER HOOK 1
                                                       The data field for the user hook = 10 [5 usec]
```

Figure 9-16. Formatted Trace Results

9.7.6 Usage Hints

The following sections provide some examples and suggestions for use of the trace facility.

9.7.6.1 Viewing trace Data

Include several optional columns of data in the **trace** output. This causes the output to exceed 80 columns. It is best to view the reports on an output device that supports 132 columns.

9.7.6.2 Bracketing Data Collection

trace data accumulates rapidly. Bracket the data collection as closely around the area of interest as possible. One technique for doing this is to issue several commands on the same command line. For example:

```
trace -a; cp /etc/trcfmt /tmp/junk; trcstop
```

captures the total execution of the copy command.

Note: This example is more educational if the source file is not already cached in system memory. The **trcfmt** file may be in memory if you have been modifying it or producing **trace** reports. In that case, choose as the source file some other file that is 50 to 100 KB and has not been touched.

9.7.6.3 Reading a trace Report

The **trace** facility displays system activity. It is a useful learning tool to observe how the system actually performs. The output from the above copy is a very interesting example to browse. To produce a report of the copy use the following:

```
trcrpt -O "exec=on,pid=on" > cp.rpt
```

In cp.rpt you can see the following activities:

- The fork, exec, and page fault activities of the **cp** process.

- The opening of the **/etc/trcfmt** file for reading and the creation of the **/tmp/junk** file.
- The successive read/write system calls to accomplish the copy.
- The process **cp** becoming blocked while waiting for I/O completion, and the wait process being dispatched.
- How logical volume requests are translated to physical volume requests.
- The files are mapped rather than buffered in traditional kernel buffers and that the read accesses cause page faults that must be resolved by the virtual memory manager.
- The virtual memory manager senses sequential access and begins to prefetch the file pages.
- That the size of the prefetch becomes larger as sequential access continues.
- That the writes are delayed until the file is closed (unless you captured execution of the **sync** daemon that periodically forces out modified pages).
- That the disk device driver coalesces multiple file requests into one I/O request to the drive when possible.

The trace output looks a little overwhelming at first. This is a good example to use as a learning aid. If you can discern the activities described, you are well on your way to being able to use the trace facility to diagnose system performance problems.

9.7.6.4 Effective Filtering of the trace Report

The full detail of the trace data may not be required. You can choose specific events of interest to be shown. For example, it is sometimes useful to find the number of times a certain event occurred. Answer the question "how many opens occurred in the copy example?" First, find the event ID for the open system call. This can be done as follows:

```
trcrpt -j |pg
```

You should be able to see that event ID 15b is the open event. Now, process the data from the copy example (data is probably still in the log file) as follows:

```
trcrpt -d 15b -O "exec=on"
```

The report is written to standard output and you can determine the number of opens that occurred. If you want to see only the opens that were performed by the **cp** process, run the report command again using the following:

```
trcrpt -d 15b -p cp  -O "exec=on"
```

Only the opens performed by the **cp** process are shown.

Chapter 10. Hints and Tips

10.1 Crash and Kernel Debugging Addresses

The virtual addresses for different memory objects such as the process table and the file table, are not likely to be the same for two debugging tools such as the kernel debugger and crash. Both utilities read the target process' u-area into a buffer allocated from entirely different segments in memory. Therefore the virtual addresses referenced by the two utilities are different.

10.2 Pinning Device Driver Code

The **pincode()** call allows portions of a device driver to be explicitly locked in real memory. This function takes as its input the address of a C function which is to be pinned. All code and static data associated with this function will be pinned in memory.

One appropriate strategy is for the *ddopen* routine to pin the interrupt handler via **pincode()**, as well as any buffers that the interrupt handler will touch. The *ddclose* routine, of course, would unpin all pinned storage. This pinning should occur before the device is initialized enough to generate any interrupts.

Unfortunately, **pincode()** actually pins significantly more memory than just the referenced function; in fact, it pins the entire *a.out* in which that function appears. This has implications on the proper packaging of device drivers.

A simple device driver is packaged on disk as a single *a.out* file, created via the **cc** command. This file is loaded into kernel memory by the configuration procedure at boot time. If the entire device driver is bound into a single a.out file, then when the open routine uses **pincode()** to lock the interrupt handler into memory the entire device driver will actually be pinned. This is normally unnecessary and wasteful.

The solution to this problem is to package the device driver as two separate *a.out* files: one containing code to be pinned and one containing code that should not be pinned. These two files can be constructed so that they are cross-linked; loading one file will automatically load the other one, and references in one file to functions in the other file will be automatically resolved. This powerful feature of the AIX 3 loader is well described elsewhere; a brief how-to discussion follows.

To split a device driver:

1. Determine which functions should be pinned. Move those functions into a ".c" file together, or into a single set of ".c" files away from those functions that should not be pinned.
2. Determine what static external data should be pinned; move such data items into the C files containing the pinned functions.
3. Compile both the pinned and the unpinned routines (but do not link them yet).

© Copyright IBM Corp. 1991

```
/* compile the .c files */
cc -c ddpin.c
cc -c dd.c
```

4. Construct an *export file* listing the functions and external static variables from the pinned portion that the unpinned portion needs to have access to. An example of an export file follows.

```
#!/path/name/of/the/pinned/portion/of/the/driver
*
* This is file "dd.exp"
*
function_to_be_seen_by_the_pageable_part
another_function
another_function
*
* Another comment
*
static_external_variable_name
another_static_external
yet_another
```

5. Link the pinned portion into its *a.out* file, telling the linker to explicitly *export* the functions listed in the export file:

```
cc -o ddpin ddpin.o -e ddintr -bexport:./dd.exp ...
```

6. Link the unpinned portion into its *a.out* file, telling the linker to explicitly *import* the functions listed in the export file:

```
cc -o dd dd.o -e ddconfig -bimport:./dd.exp ...
```

As a result of this procedure, whenever the **dd** file is loaded, the **ddpin** file will automatically be loaded and linked with it. This happens "magically" without any involvement by the user, programmer or configuration routine. Yet when the **pincode()** function is used, it will only pin data in the **ddpin** file, not the code in the **dd** file. As a result, portions of the driver that do not need to be pinned will not be, at a potentially large savings in memory consumption.

BE CAREFUL

It is technically possible for the unpinned routine to export functions and data areas for the pinned portion to reference. However, this is bad form; the pinned portion of the device driver should have no dependencies on unpinned code or data. If you decode to export symbols in both directions, do so with extreme caution.

While this procedure isn't strictly necessary for small and simple drivers, it should be strongly considered for large drivers and drivers that will be active for long periods of time. If a "quick-and-dirty" job is all that is required, the driver can be left in one executable file and can be written without concern for paging considerations, with the following caveats:

- Allocate all data from the **pinned_heap**.
- Explicitly issue **pincode()** to lock the interrupt handler (and thus the entire device driver) in real memory.

10.3 Compiling Device Drivers

While sample device drivers and configuration routines are supplied with AIX, no sample Makefiles are provided. Building these programs involves some compexities not seen in normal programming.

In general, compiling a device driver involves no special considerations. Linking one, however, requires some special flags. If the device driver has been split into a pageable portion and a pinned portion, see the discussion above about some flags to use.

A simple command for linking a single-piece device driver is given below.

```
cc -o mydd mydd.o -emyconfig -bimport:/lib/kernex.exp \
   -bimport:/lib/syscalls.exp -lsys -lcsys
```

The **-emyconfig** clause specifies that the myconfig() function is the entry point for this program; this will be the function called as the configuration routine at boot time.

The two **-bimport** clauses define lists of functions that are exported from the AIX kernel. These lists insure that the various AIX functions used by your device driver are properly bound at run time and are not reported as unresolved references at link time.

The two **-l** clauses list libraries of subroutines which may be used in kernel mode. The standard libc.a library **cannot** be used.

Configuration methods have no special compilation or link time dependencies.

10.4 Working with Kernel Processes

The following section is included for those programmers who will be writing kernel processes. Kernel processes are similar to device drivers. The similarities and differences are discussed in the following sections.

10.4.1 Writing a Kernel Process

A kernel process is written more like a device driver than a user program. Like a device driver its entry point is not (normally) named "main". A device driver has several entry points (config, read, open, etc...) and a kernel process may have only one which the programmer may name as he pleases. The name of that entry point (which is the name of a function) must be made available to the program (device driver, user process, etc..) that will call it.

10.4.2 Compiling a Kernel Process

Device drivers and kernel processes should be compiled with the -c flag and linked in a separate step. If compiled and linked in the same step then the standard c library will be linked to the device driver or kernel process. Many functions in the standard c library (such as printf) will not work in the kernel environment. A command to compile a kernel process may look like this:

```
cc -c my_kproc.c
```

10.4.3 Linking a Kernel Process

The name of the entry point to a kernel process must be available to the calling entity. This is done by exporting the entry point name at kernel process link time and importing this name at the "calling entity" link time. An export file needs to be generated that contains the name of the entry point. We could call such an export file "my_kproc.exp" and it may look like this:

```
#!
*
* kproc entry point  (this is a comment)
*
my_kproc_main
```

The **link command for the kernel process** could then look like this:

```
ld -o my_kproc my_kproc.o -emy_kproc_main                  \
-bimport:/lib/kernex.exp -bimport:/lib/syscalls.exp  \
-bexport:./my_kproc.exp                                    \
-lsys -lcsys
```

Notice that the -e flag (specifies an entry point) indicates the same symbol that is in the export file. The -bexport flag indicates the my_kproc.exp file which should be in the current directory.

10.4.4 Loading a Kernel Process

Before the kernel process can be executed it must be loaded into the kernel. There are at least three ways to do this:

1. sysconfig subroutine (loading from user space)
2. loadext subroutine (loading from user space)
3. kmod_load kernel service (loading from kernel space).

The same or similar functions provide for unloading kernel processes.

When a kernel process should be loaded and unloaded depends on the requirements of the application and the resources that may be dedicated to that application. Kernel processes that are loaded and then not used consume memory, while kernel processes that are not loaded until they are absolutely needed may affect throughput. The most popular places to load a kernel process are in a configure method, the driver config entry point, the driver mpx entry point, the driver open entry point and maybe the driver ioctl entry point.

10.4.5 Starting a Kernel Process

A loaded kernel process is started by using both the creatp() and then the initp() functions. Both of these kernel services are called from the process environment. creatp() creates a slot in the kernel process table and puts the process in an "idle" state. initp() initializes the process and puts it in a "ready" state.

The initp() function requires the name of the kernel process entry point. This name must have been imported by the calling program at link time. The link command for a process that would call the above example of a kernel process would be as follows:

```
ld -o mydd mydd.o -emy_config                                   \
-bimport:./my_kproc.exp                                         \
-bimport:/lib/kernex.exp -bimport:/lib/syscalls.exp    \
-lsys -lcsys
```

Note that the mydd program is importing the file that contains the name of the entry point of the kernel extention. (In fact, the same file that the kernel extention used for exporting can be used to import this entry point name.)

Appendix A. AIX Devices

A.1 Device Classes, Subclasses, and Types Overview

To manage its wide variety of devices more easily, the AIX Operating System classifies them hierarchically. One advantage of this arrangement is that device methods and high level commands can operate against a whole set of similar devices.

Devices are categorized into these three main groups:

- Functional classes
- Functional subclasses
- Device types.

Devices are organized into a set of functional classes at the highest level. From a user's point of view, all devices belonging to the same class perform the same functions. For example, all printer devices basically perform the same function of generating printed output.

However, devices within a class can have different interfaces. A class can therefore be partitioned into a set of functional subclasses where devices belonging to the same subclass have similar interfaces. For example, serial printers and parallel printers form two subclasses of the class of printer devices.

Finally, a device subclass is a collection of device types. All devices belonging to the same device type share the same manufacturer's model name and/or number. For example, IBM 3812-2 (Model 2 Pageprinter) and IBM 4201 (Proprinter) printers comprise two types of printers.

Devices of the same device type can be managed by different drivers if the type belongs to more than one subclass. For example, the IBM 4201 printer belongs to both the serial interface and parallel interface subclasses of the printer class, and there are different drivers for the two interfaces. But a device of a particular class, subclass, and type can be managed by only one device driver.

A.2 Device Dependencies and Child Devices

The dependencies that one device has on another can be represented in the Configuration database in two ways. One way usually represents physical connections such as a keyboard device connected to a particular keyboard adapter. The keyboard device has a dependency on the keyboard adapter in that it cannot be configured until after the adapter is configured. This relationship is usually referred to as a parent-child relationship with the adapter as parent and the keyboard device as child. These relationships are represented with the Parent Device Logical Name and Location Where Device Is Connected descriptors in the **CuDv** objects.

© Copyright IBM Corp. 1991

A device method can also add to the **CuDep** object class an object identifying both a dependent device and the device upon which it depends. The dependent device is considered to have a dependency, and the depended-upon is considered to be a dependency. Customized Dependency objects are usually added to the database to represent a situation in which one device requires access to another device. For example, the **hft0** device depends upon a particular keyboard or display device.

These two types of dependencies differ significantly. The configuration process uses parent-child dependencies at boot time to configure all devices that make up a node. The **CuDep** dependency is usually only used by a device's **configure** method to retrieve the names of the devices on which it depends. The **configure** method can then check to see if those devices exist.

For device methods, the parent-child relationship is the more important. Parent-child relationships affect device-method activities in these ways:

- A parent device cannot be unconfigured if it has a configured child.
- A parent device cannot be undefined if it has a defined or configured child.
- A child device cannot be defined if the parent is not defined or configured.
- A child device cannot be configured if the parent is not configured.
- A parent device's configuration cannot be changed if it has a configured child. This guarantees that the information about the parent which the child's device driver may be using remains valid.

However, when a device is listed as a dependency for another device in the **CuDep** object class, the only effect is to prevent the depended-upon device from being undefined. The name of the dependency is important to the dependent device. If the depended-upon device were allowed to be undefined, a third device could be defined and be assigned the same name.

Writers of **unconfigure** and **change** methods for a depended-upon device need not worry about whether the device is listed as a dependency. If the depended-upon device is actually opened by the other device, the unconfigure and change operations will fail because their device is busy. But if the depended-upon device is not currently open, the unconfigure or change operations can be performed without affecting the dependent device.

The possible parent-child connections are defined in the **PdCn** object class. Each predefined device type that can be a parent device is represented in this object class. There is an object for each connection location (such as slots or ports) describing the subclass of devices that can be connected at that location. Subclass is used to identify the devices since it indicates the devices' connection type (e.g., SCSI or rs232).

There is no corresponding predefined object class describing the possible **CuDep** dependencies. A device method can be written so that it already knows what the dependencies are. If predefined data is required, it can be added as predefined attributes for the dependent device in the **PdAt** object class.

A.3 The Run Time Configuration Commands

A.3.1 The mkdev Command

The **mkdev** command is invoked to define or configure, or define and configure, devices at run time. If just defining a device, the mkdev command invokes the **define** method for the device. The **define** method creates the customized device instance in the **CuDv** object class and writes the name assigned to the device to the stdout file. The mkdev command intercepts the device name written to the stdout file by the Define method to learn the name of the device. If user-specified attributes are supplied with the **-a** flag, the mkdev command then invokes the **change** method for the device.

If defining and configuring a device, the mkdev command invokes the **define** method, gets the name written to the stdout file by the define method, invokes the **change** method for the device if user-specified attributes were supplied, and finally invokes the device's **configure** method.

If only configuring a device, the device must already exist in the **CuDv** object class and its name must be specified to the mkdev command. In this case, the mkdev command simply invokes the **configure** method for the device.

A.3.2 The chdev Command

The **chdev** command is used to change the characteristics, or attributes, of a device. The device must already exist in the **CuDv** object class, and the name of the device must be supplied to the chdev command. The chdev command simply invokes the **change** method for the device.

A.3.3 The rmdev Command

The **rmdev** command can be used to undefine or unconfigure, or unconfigure and undefine, a device. In all cases, the device must already exist in the **CuDv** object class and the name of the device must be supplied to the rmdev command. The rmdev command then invokes the **undefine** method, the **unconfigure** method, or the **unconfigure** method followed by the **undefine** method, depending on the function requested by the user.

A.3.4 The cfgmgr Command

The **cfgmgr** command can be used to configure all detectable devices that did not get configured at boot time. This might occur if the devices had been powered off at boot time. The cfgmgr command is the Configuration Manager and operates in the same way at run time as it does at boot time.

A.4 Devices Location Codes

The location code for a device is a path from the adapter in the CPU drawer or system unit, through the signal cables and the asynchronous distribution box, if there is one, to the device or workstation. This code is another way of identifying physical devices.

The location code consists of four fields of information: Drawer, Slot, Connector, and Port. The format for a location code is **AA-BB-CC-DD**, where **AA** corresponds to drawer, **BB** to slot, **CC** to connector, and **DD** to port.

A.4.1 Printer and Plotter Devices

For the printer and plotter class, the location code format is **AA-BB-CC-DD**. The **AA-BB** field is the location code of the adapter card controlling the printer or plotter.

A value of 00 for the **AA** field identifies that the card is located in the CPU drawer or system unit, depending on the type of system. Any other value for the **AA** field indicates that the card is located in an I/O expansion drawer; in which case, the value identifies the slot number in the CPU drawer that contains the asynchronous expansion adapter.

The **BB** field identifies the slot number of the slot containing the card. A value of 00 for this field indicates the Standard I/O planar.

For a serial printer and plotter attached to a Standard I/O serial port, the **CC-DD** field is either a value of S1-00 or S2-00, depending on whether the device is attached to port s1 or port s2. Otherwise, the **CC** field identifies the connector on the adapter card to which the asynchronous distribution box is connected. Possible values are 01, 02, 03, and 04. The **DD** field identifies the port number on the asynchronous distribution box to which the printer or plotter is attached.

For a parallel printer attached to the Standard I/O parallel port, the value for the **CC-DD** field is always 0P-00.

In order to find the physical connection to cable the printer or plotter device to, use the first three fields of the location code. Look for these fields on a label found on the async distribution box. If you are configuring the device to S1, S2, or P, you will find the connector on the back of the RISC System/6000 system unit.

A.4.2 TTY Devices

For the tty device class, the location code format is **AA-BB-CC-DD**. The **AA-BB** field is the location code of the adapter card controlling the tty device.

A value of 00 for the **AA** field identifies that the card is located in the CPU drawer or system unit, depending on the type of system. Any other value for the **AA** field indicates that the card is located in an I/O expansion drawer; in which case, the value identifies the slot number in the CPU drawer that contains the asynchronous expansion adapter.

The **BB** field identifies the slot number of the slot containing the card. A value of 00 for this field indicates the Standard I/O planar.

For a tty device attached to a Standard I/O serial port, the **CC-DD** field is either a value of S1-00 or S2-00, depending on whether the device is attached to port s1 or port s2. Otherwise, the **CC** field identifies the connector on the adapter card to which the asynchronous distribution box is connected. Possible values are 01, 02, 03, and 04. The **DD** field identifies the port number on the asynchronous distribution box to which the tty device is attached.

In order to find the physical connection to cable the tty device to, use the first three fields of the location code. Look for these fields on a label found on the asynchronous distribution box. If you are configuring the device to S1 or S2, you will find the connector on the back of the RISC System/6000 system unit.

A.4.3 Direct-Attached Disks and SCSI Devices

For a direct-attached disk device, the location code format is **AA-BB** where the **AA** field is a value of 00 indicating that the disk is located in the system unit. The **BB** field indicates the slot number.

For all SCSI devices, including disks, CD-ROMs, and tapes, the location code format is **AA-BB-CC-DD**. The **AA-BB** field identifies the location code of the SCSI adapter controlling the SCSI device.

A value of 00 for the **AA** field identifies that the card is located in the CPU drawer or system unit, depending on the type of system.

The **BB** field identifies the slot number of the slot containing the card.

The **CC** field is always a value of 00.

The **DD** field identifies the SCSI ID and logical unit number (LUN) of the SCSI device. The first number is the SCSI ID, and the second number is the LUN.

For an external SCSI device, the device is attached to the slot specified by the value in the field, and the device physical address is set to the SCSI ID shown in the **DD** field.

A.4.4 Diskette Drive Devices

For diskette drives attached to the Standard I/O planar, the possible values of the location codes are 00-00-0D-01 and 00-00-0D-02 for diskette ports 0 and 1, respectively.

A.4.5 Adapter Devices

The location code for all adapter cards consists of just the first two fields: AA-BB. A value of 00 for the **AA** field identifies that the card is located in the CPU drawer or system unit, depending on the type of system. Any other value for the **AA** field indicates that the card is located in an I/O expansion drawer; in which case, the value identifies the slot number in the CPU drawer that contains the asynchronous expansion adapter.

The **BB** field for an adapter card identifies the slot number of the slot containing the card. A value of 00 indicates the Standard I/O planar.

A.4.6 Multiprotocol Port Devices

For a multiprotocol port, the location code format is **AA-BB-CC-DD**. The **AA-BB** field identifies the location code of the multiprotocol adapter to which the port corresponds. A value of 00 for the **AA** field identifies that the card is located in the CPU drawer or system unit, depending on the type of system. The **BB** field identifies the slot number of the slot containing the card. The **CC** field is always a value of 01. It identifies the connector on the adapter where the multiprotocol distribution box is connected. The **DD** field indicates the physical port number on the multiprotocol distribution box. Possible values are 00, 01, 02, and 03.

In order to find the physical connection to cable the multiprotocol device to, use the first three fields of the location code. Look for these fields on a label found on the asynchronous distribution box.

Appendix B. ODM

B.1 ODM Object Classes

B.1.1 Predefined Devices (PdDv)

The Predefined Devices (PdDv) object class contains entries for all known device types supported by the system. The term devices is used in the general sense in this context. Devices include intermediate devices (for example, adapters) and terminal devices (for example, disks, printers, display terminals, and keyboards). Pseudo-devices, including pseudo terminals, logical volumes, and TCP/IP, are also included under devices. Pseudo-devices can either be intermediate or terminal devices.

Each device type, as determined by class-subclass-type information, is represented by an object in the **PdDv** object class. These objects contain basic information about the devices, such as device method names and how to access information contained in other object classes. The **PdDv** object class is referenced by the **CuDv** object class by a link that keys into the **unique type** descriptor. This descriptor is uniquely identified by the class-subclass-type information.

Typically, the Predefined database is consulted, but never modified during system boot or run time. One exception occurs when a new device is to be added to the Predefined database. In this case, the predefined information for the new device must be added into the Predefined database.

You build a Predefined Device object by defining the objects in a file in stanza format and then processing the file with the **odmadd** command or the **odm_add_obj** subroutine.

> **NOTE**
>
> When coding an object in this object class, set unused empty strings to "" (two double quotation marks with no separating space) and unused integer fields to 0 (zero).

Each Predefined Device object corresponds to an instance of the **PdDv** object class. The descriptors for this class are:

© Copyright IBM Corp. 1991

Table B-1. PdDv Object Class Descriptors		
Descriptor name	**Description**	**Descriptor status**
type	Device Type	Required
class	Device class	Required
subclass	Device subclass	Required
prefix	Prefix name	Required
devid	Device ID	Optional
base	Base device flag	Required
has_vpd	VPD flag	Required
detectable	Detectable/nondetectable flag	Required
chgstatus	Change status flag	Required
bus_ext	Bus extender flag	Required
inventory_only	Inventory only flag	Required
fru	FRU flag	Required
led	LED value	Required
setno	Set number	Required
msgno	Message number	Required
catalog	Catalog file name	Required
DvDr	Device driver name	Optional
Define	Define method	Required
Configure	Configure method	Required
Change	Change method	Required
Unconfigure	Unconfigure method	Optional
Undefine	Undefine method	Optional
Start	Start method	Optional
Stop	Stop method	Optional
uniquetype	Unique type	Required

These fields have the following descriptions:

Device Type
The Device Type descriptor is derived from the product name or model number. For example, IBM 3812-2 Model 2 Pageprinter and IBM 4201 Proprinter are two types of printer device types. Each device type supported by the system should have an entry in the PdDv object class.

Device Class
Associated functional class name. A Functional class is a group of device instances sharing the same high-level function. For example,

a printer is a functional class name representing all devices that generate hardcopy output.

Device Subclass
Identifies the device subclass associated with the device type. A device class can be partitioned into a set of device subclasses whose members share the same interface and typically are managed by the same device driver. For example, parallel and serial printers form two subclasses within the class of printer devices.

The configuration process uses the subclass to determine valid parent-child connections. For example, an rs232 8-port adapter has information that indicates that each of its eight ports only supports devices whose subclass is rs232. Also, the subclass for one device class can be a subclass for a different device class. In other words, several device classes can have the same device subclass.

Prefix Name
The Assigned Prefix in the Customized database, used to derive the device instance name and */dev* name. For example, **tty** is a Prefix Name assigned to the tty port device type. Names of tty port instances would then look like tty0, tty1, or tty2. The rules for generating device instance names are given in the **CuDv** object class under the Device Name descriptor.

Device ID Device ID describes card IDs for Micro Channel adapter cards. These card IDs are read from POS registers and uniquely identify the card type. The bus *configure method* obtains the card IDs from the Micro Channel adapter cards and uses this descriptor to find the predefined information corresponding to the cards. The format is **0xAA** where AA identifies the *POS(0)* value and the *POS(1)* value.

Base Device Flag
A base device is any device that forms part of a minimal base system. During the first phase of system boot, a minimal base system is configured to permit access to the root volume group and hence to the root file system. This minimal base system can include, for example, the standard I/O diskette adapter and a SCSI hard drive.

This flag is not used to determine which devices are to be configured in the first phase of system boot. It serves only to identify at run time which devices need to be updated in the boot image when configuration changes are made. A value of TRUE means that the device is a base device, and a value of FALSE that it is not.

VPD Flag Certain devices contain Vital Product Data (VPD) that can be retrieved from the device itself. This attribute specifies whether device instances belonging to the device type contain extractable VPD or not. A value of TRUE means that the device has extractable VPD, and a value of FALSE that it does not.

Detectable/Nondetectable Flag
Specifies whether the device instance is detectable or nondetectable. A device whose presence and type can be electronically determined, once it is actually powered on and attached to the system, is said to be detectable. A value of TRUE

means that the device is detectable, and a value of FALSE that it is not.

Change Status Flag
Indicates the initial value of the Change Status flag to be used in the **CuDv** object class. Refer to the corresponding descriptor in the **CdDv** object class for a complete description of this flag. A value of NEW means that the device is to be flagged as new, and a value of FALSE that it is to be flagged as don't care.

Bus Extender Flag
Indicates that the device is a bus extender. The Bus Configurator uses the Bus Extender Flag descriptor to determine whether it should directly invoke the device's *configure method*. A value of TRUE means that the device is a bus extender, and a value of FALSE that it is not a bus extender.

Inventory Only Flag
Distinguishes devices that are represented solely for their replacement algorithm from those that actually manage the system. There are several devices that are represented solely for inventory or diagnostic purposes. Racks, drawers, and planars represent such devices. A value of TRUE means that the device is used solely for inventory or diagnostic purposes, and a value of FALSE that it is not used solely for diagnostic or inventory purposes.

FRU Flag Identifies the type of FRU (Field Replaceable Unit) for the device. The three possible values for this field are:

NO_FRU Indicates that there is no FRU (for pseudo-devices).

SELF_FRU Indicates that the device is its own FRU.

PARENT_FRU Indicates that the FRU is the parent.

LED Value
Indicates the hex value to be displayed on the LEDs when the *configure method* executes [1].

Catalog File Name
Identifies the file name of the NLS message catalog that contains all messages pertaining to this device. This includes the device description and its attribute descriptions. All NLS messages are identified by a catalog file name, set number, and message number.

Set Number
Identifies the set number that contains all the messages for this device in the specified NLS message catalog. This includes the device description and its attribute descriptions.

Message Number
Identifies the message number in the specified set of the NLS message catalog. The message corresponding to the message number contains the textual description of the device.

[1] Refer to *RISC System/6000 System Problem-Solving Guide* for a list of valid LED values.

Device Driver Name

Identifies the base name of the device driver associated with all device instances belonging to the device type. For example, a device driver name for a keyboard could be ktsdd. For the tape device driver, the name could be tapedd. The Device Driver Name can be passed as a parameter to the **loadext** routine to load the device driver, if the device driver is located in the */etc/drivers* directory. If the driver is located in a different directory, the full path must be appended in front of the Device Driver Name before passing it as a parameter to the **loadext** subroutine.

Define Method

Name of the define method associated with the device type. All define method names start with the prefix def.

Configure Method

Name of the configure method associated with the device type. All configure method names start with the prefix cfg.

Change Method

Name of the change method associated with the device type. All change method names start with the prefix chg.

Unconfigure Method

Name of the unconfigure method associated with the device type. All unconfigure method names start with the prefix ucfg. This field is optional for those devices (for example, the bus) that are never unconfigured or undefined. For all other devices, this descriptor is required.

Undefine Method

Name of the undefine method associated with the device type. All undefine method names start with the prefix udef. This field is optional for those devices (for example, the bus) that are never unconfigured or undefined. For all other devices, this descriptor is required.

Start Method

Name of the start method associated with the device type. All start method names start with the prefix stt. The start method is optional and only applies to devices that support the Stopped device state.

Stop Method

Name of the stop method associated with the device type. All stop method names start with the prefix stp. The stop method is optional and only applies to devices that support the Stopped device state.

Unique Type

A key that is referenced by the **PdDvLn** link in **CdDv** object class. The key is a concatenation of the Device Class, Device Subclass, and Device Type values with a / (backslash) used as a separator. For example, for a class of disk, a subclass of SCSI, and a type of 670MB, the Unique Type is disk/SCSI/670MB.

This descriptor is needed so that a device instance's object in the **CdDv** object class can have a link to its corresponding **PdDv** object. Other object classes in both the Predefined and Customized databases also use the information contained in this descriptor.

B.1.2 Predefined Attribute (PdAt)

The Predefined Attribute (PdAt) object class contains an entry for each existing attribute for each device represented in the **PdDv** object class. An attribute, in this sense, is any device-dependent information not represented in the **PdDv** object class. This includes information such as interrupt levels, bus I/O address ranges, baud rates, parity settings, block sizes, and microcode file names.

Each object in this object class represents a particular attribute belonging to a particular device class-subclass-type. Each object contains the attribute name, default value, list or range of all possible values, width, flags, and an NLS description. The flags provide further information to describe an attribute.

Note: for a device being defined or configured, only the attributes that take a nondefault value are copied into the **CuAt** object class. In other words, for a device being customized, if its attribute value is the default value in the **PdDv** object class, then there will not be an entry for the attribute in the **CuAt** object class.

Types of Attributes: there are three types of attributes. Most are regular attributes, which typically describe a specific attribute of a device. The group attribute type provides a grouping of regular attributes. The shared attribute type identifies devices that must all share the given attribute.

A shared attribute identifies another regular attribute as one that must be shared. This attribute is always a bus resource. Other regular attributes (for example, bus interrupt levels) can be shared by devices but are not themselves shared attributes. Shared attributes require that the relevant devices have the same values for this attribute. The Attribute Value descriptor for the shared attribute gives the name of the regular attribute that must be shared.

A group attribute specifies a set of other attributes whose values are chosen as a group, as well as a group attribute number used to choose the default values. Each attribute listed within a group has an associated list of possible values it can take. These values must be represented as a list, not as a range. For each attribute within the group, the list of possible values must also have the same number of choices. For example, if the possible number of values is n, the group attribute number itself can range from 0 to n-1. The particular value chosen for the group indicates the value to pick for each of the attributes in the group. For example, if the group attribute number is 0, then the value for each of the attributes in the group is the first value from their respective lists.

The **PdAt** object class contains the following fields:

Table B-2. PdAt Object Class Descriptors		
Descriptor name	**Description**	**Descriptor status**
uniquetype	Unique type	Required
attribute	Attribute name	Required
deflt	Default value	Required
values	Attribute values	Required
width	Width	Optional
type	Attribute type flags	Required
generic	Generic attribute flags	Optional
rep	Attribute representation flags	Required
nls_index	NLS index	Optional

These fields are described as follows:

Unique Type
Identifies the class-subclass-type name of the device to which this attribute is associated. This descriptor is the same as the Unique Type descriptor in the **PdDv** object class.

Attribute Name
Identifies the name of the device attribute. This is the name that can be passed to the **mkdev** and **chdev** configuration commands and device methods in the attribute-name and attribute-value pairs.

Default Value
If there are several choices or even if there is only one choice for the attribute value, the default is the value that the attribute is normally set to. For groups, the default value is the group attribute number. For example, if the possible number of choices in a group is n, the group attribute number is a number between 0 and n-1. For shared attributes, the default value is set to a null string.

When a device is defined in the system, attributes that take nondefault values are found in the **CuAt** object class. Attributes that take the default value are found in this object class. Attributes that take on the default value are not copied over to the CuAt object class. Therefore, both attribute object classes must be queried to get a complete set of customized attributes for a particular device.

Possible Values
Identifies the possible values that can be associated with the attribute name. The format of the value is determined by the Attribute Representation flags. For regular attributes, the possible values can be represented as *a string*, *hexadecimal*, *octal*, or *decimal*. In addition, they can be represented as a range or an enumerated list. If there is only one possible value, then the value can be represented either as a single value or as an enumerated list with one entry. The latter is recommended, since the use of enumerated lists allows the attrval subroutine to check that a given value is in fact a possible values.

If the value is hexadecimal, then it is prefixed with the 0x notation. If the value is octal, the value is prefixed with a leading zero. If the value is decimal, the value is its significant digits. If the value is a string, the string itself should not have embedded commas since commas are used as separators of items in an enumerated list.

A range is represented as a triplet of values: lowerlimit-upperlimit, increment value. The lowerlimit variable indicates the value of the first possible choice. The upperlimit variable indicates the value of the last possible choice. The lowerlimit and upperlimit values are separated by a - (hyphen). Values between the lowerlimit and upperlimit values are obtained by adding multiples of the increment value variable to the lowerlimit variable. The upperlimit and increment value variables are separated by a comma.

Only numeric values are used for ranges. Also, discontinuous ranges (for example, 1-3, 6-8) are disallowed. A combination of list and ranges is not allowed. An enumerated list contains values that are comma-separated.

If the attribute is a group, the Possible Values descriptor contains a list of attributes composing the group, separated by commas.

If the attribute is shared, the Possible Values descriptor contains the name of the bus resource regular attribute that must be shared with another device.

Width If the attribute is a regular attribute, the Width descriptor identifies the amount of resource used by the attribute. For example, if the attribute indicates the starting bus memory address for an adapter card, this field indicates the range of bus memory that must be allocated to the adapter. Width only applies to attributes with the M (bus memory address) and the O (bus I/O address) Attribute Types. For all other attributes, a null string is used to fill in this field.

Attribute Type
Identifies the attribute type. Only one Attribute Type must be specified. The characters A, M, I, O, and P represent bus resources that are regular attributes.

For regular attributes, the following Attribute Types are defined:

R Indicates a regular attribute that is not a bus resource.

The following are the bus resources types for regular attributes:

A Indicates DMA arbitration level.

M Indicates bus memory address.

I Indicates bus interrupt level.

O Indicates bus I/O address.

P Indicates priority class.

For non-regular attributes, the following Attribute Types are defined:

G Indicates a group.

S Indicates a shared attribute.

Generic Attribute Flags

Identifies the flags that can apply to any regular attribute. Any combination, one, both, or none, of these flags is valid. This descriptor should be a null string for group and shared attributes.

These are the defined Generic Attribute flags:

D Indicates a displayable attribute. The **lsattr** command displays only attributes with this flag.

U Indicates an attribute whose value can be set by the user.

Attribute Representation Flags

Indicates the representation of the regular attribute values. For group and shared attributes, which have no associated attribute representation, this descriptor is set to a null string. Either the **n** or **s** flag, both of which indicate value representation, must be specified.

The **r** and **l** flags indicate, respectively, a range and an enumerated list, and are optional. If neither r nor l is specified, then the **attrval** subroutine will not verify that the value falls within the range or the list.

These are the defined Attribute Representation flags:

n Indicates that the attribute value is numeric, either decimal, hex, or octal.

s Indicates that the attribute value is a character string.

r Indicates that the attribute value is a range of the form: lowerlimit-upperlimit,increment value.

l Indicates that the attribute value is an enumerated list of values.

NLS Index

Identifies the message number in the NLS message catalog of the message containing a textual description of the attribute. Only displayable attributes, as identified by the Generic Attribute flags descriptor, need an NLS message. If the attribute is not displayable, the NLS Index can be set to a value of 0. The catalog file name and the set number associated with the message number are stored in the **PdDv** object class.

B.1.3 Predefined Connection (PdCn)

The Predefined Connection (PdCn) object class contains connection information for intermediate devices. This object class also includes predefined dependency information. For each connection location, there are one or more objects describing the subclasses of devices that can be connected. This information is useful, for example, in verifying whether a device instance to be defined and configured can be connected to a given device.

The **PdCn** object class contains the following descriptors:

Table B-3. PdCn Object Class Descriptors		
Descriptor name	**Description**	**Descriptor status**
uniquetype	unique type	Required
connkey	connection key	Required
connwhere	connection location	Required

These fields are described as follows:

Unique Type
Identifies the intermediate device's class-subclass-type name. For a device with dependency information, this descriptor identifies the unique type of the device on which there is a dependency. This descriptor contains the same information as in the Unique Type descriptor in the **PdDv** object class.

Connection Key
Identifies a subclass of devices that can connect to the intermediate device at the specified location. For a device with dependency information, this descriptor serves to identify the device indicated by the Unique Type field to the devices that depend on it.

Connection Location
Identifies a specific location on the intermediate device where a child device can be connected. For a device with dependency information, this descriptor is not always required and consequently may be filled in with a null string.

The term location is used in a generic sense. For example, for a bus device, the location can refer to a specific slot on the bus, with values 1, 2, 3 ... For a multiport serial adapter device, the location can refer to a specific port on the adapter with values 0, 1, ...

B.1.4 Customized Devices (CuDv)

The Customized Devices (CuDv) object class contains entries for all device instances defined in the system. As the name implies, a defined device object is an object that a define method has created in the **CuDv** object class. A defined device instance may or may not have a corresponding actual device attached to the system.

A **CuDv** object contains attributes and connections specific to the device instance. Each device instance, distinguished by a unique logical name, is represented by an object in the **CuDv** object class. The Customized database is updated twice, during system boot and at run time, to define new devices, remove undefined devices, or update the information for a device whose attributes have been changed.

The **CuDv** object class contains the following fields:

Table B-4. CuDv Object Class Descriptors		
Descriptor name	**Description**	**Descriptor status**
name	Device name	Required
status	Device status flag	Required
chgstatus	Change status flag	Required
ddins	Device driver instance	Optional
location	Location code	Optional
parent	Parent device logical name	Optional
connwhere	Location where device is connected	Optional
PdDvLn	LINK to Predefined Devices object class	Required

These fields have the following descriptions:

Device Name

A Customized Device object for a device instance is assigned a unique logical name to distinguish the instance from other device instances. The device logical name of a device instance is derived during *define method processing*. The rules for deriving a device logical name are:

1. The name should start with a prefix name pre-assigned to the device instance's associated device type. The prefix name can be retrieved from the Prefix Name descriptor in the **PdDv** object associated with the device type.
2. To complete the logical device name, a sequence number is usually appended to the prefix name. This sequence number is unique among all defined device instances using the same prefix name. Use the following subrules when generating sequence numbers:
   a. A sequence number is a non-negative integer represented in character format. Therefore, the smallest available sequence number is 0 (zero).
   b. The next available sequence number relative to a given prefix name should be allocated when deriving a device instance logical name.
   c. The next available sequence number relative to a given prefix name is defined to be the smallest sequence number not yet allocated to defined device instances using the same prefix name.

      For example, if tty0, tty1, tty3, tty5 and tty6 are currently assigned to defined device instances, then the next available sequence number for a device instance with the tty prefix name is 2. This results in a logical device name of tty2.

The **genseq** subroutine can be used by a **define method** to obtain the next available sequence number.

Device Status Flag
Identifies the current status of the device instance. The device methods are responsible for setting the Device Status flags of device instances. When the **define method** defines a device instance, the device's device status is set to defined. When the **configure method** configures a device instance, the device's device status is typically set to available. The configure method takes a device to the Stopped state only if the device supports the Stopped state.

When the **start method** starts a device instance, its device status is changed from the *stopped state* to the *available state*. Applying a **stop method** on a started device instance changes the device status from the *available state* to the *stopped state*. Applying an **unconfigure method** on a configured device instance changes the device status from the *available state* to the *defined state*. If the device supports the *stopped state*, the **unconfigure method** takes the device from the *stopped state* to the *defined state*.

The possible status values are:

DEFINED Identifies a device instance in the defined state.

AVAILABLE Identifies a device instance in the available state.

STOPPED Identifies a device instance in the stopped state.

Change Status Flag
This flag tells whether the device instance has been altered since the last system boot. The diagnostics facility uses this flag to validate system configuration. The flag can take on these values:

NEW Specifies whether the device instance is new to the current system boot.

DONT_CARE Identifies the device as one whose presence or uniqueness cannot be determined. For these devices, the new, same, and missing states have no meaning.

SAME Specifies whether the device instance was known to the system prior to the current system boot.

MISSING Specifies whether the device instance is missing. This is true if the device is in the **CuDv** object class, but is not physically present.

Device Driver Instance
This field typically contains the same value as the Device Driver Name descriptor in the **PdDv** object class if the device driver supports only one major number. For a driver that uses multiple major numbers (for example, the logical volume device driver), unique instance names must be generated for each major number. Since the logical volume uses a different major number for each volume group, the volume group logical names would serve this purpose. This field is filled in with a null string if the device instance does not have a corresponding device driver.

Location Code

Identifies the location code of the device. This field provides a means of identifying physical devices. The location code format is defined as **AB-CD-EF-GH** where:

AB Is the drawer ID used to identify the CPU and asynchronous drawers.

CD Is the slot ID used to identify the location of an adapter, memory card, or SLA (Serial Link Adapter).

EF Is the connector ID used to identify the adapter connector that something is attached to.

GH Is the port or device or FRU ID used to identify a port, device, or FRU, respectively.

Parent Device Logical Name

Identifies the logical name of the parent device instance. In the case of a real device, this indicates the logical name of the parent device to which this device is connected. More generally, the specified parent device is the device whose **configure method** is responsible for returning the logical name of this device to the Configuration Manager for configuring this device. This field is filled in with a null string for a node device.

Location Where Device Is Connected

Identifies the specific location on the parent device instance where this device is connected. The term location is used in a generic sense. For some device instances such as the AIX bus, location indicates a slot on the bus. For device instances such as the SCSI adapter, the term indicates a logical port (that is, a SCSI ID and Logical Unit Number combination).

For example, for a bus device, the location can refer to a specific slot on the bus, with values 1, 2, 3 ... . For a multiport serial adapter device, the location can refer to a specific port on the adapter, with values 0, 1,

LINK to Predefined Devices Object Class (PdDvLn)

Provides a link to the device instance's predefined information through the Unique Type descriptor in the **PdDv** object class.

B.1.5 Customized Attribute (CuAt)

The Customized Attribute (CuAt) object class contains customized device-specific attribute information.

Device instances represented in the **CuDv** object class have attributes found in either the **PdAt** object class or the **CuAt** object class. There is an entry in the **CuAt** object class for attributes that take non-default values. Attributes taking the default value are found in the **PdAt** object class. Each entry describes the current value of the attribute.

When changing the value of an attribute, the Predefined Attribute object class must be referenced to determine other possible attribute values.

Both attribute object classes must be queried to get a complete set of current values for a particular device's attributes. Use the **getattr** and **putattr** routines to retrieve or modify customized attributes.

Table B-5. CuAt Object Class Descriptors

Descriptor name	Description	Descriptor status
name	Device name	Required
attribute	Attribute name	Required
value	Attribute value	Required
type	Attribute type	Required
generic	Generic attribute flags	Optional
rep	Attribute representation flags	Required
nls_index	NLS index	Optional

These fields are described as follows:

Device name
Identifies the logical name of the device instance to which this attribute is associated.

Attribute name
Identifies the name of a customized device attribute.

Attribute value
Identifies a customized value associated with the corresponding Attribute Name. This value is a non-default value.

Attribute type
Identifies the attribute type associated with the Attribute Name. This field is copied from the Attribute Type descriptor in the corresponding Predefined Attribute object when the Customized Attribute object is created.

Generic attribute flags
Identifies the Generic Attribute flag or flags associated with the Attribute Name. This field is copied from the Generic Attribute Flags descriptor in the corresponding Predefined Attribute object when the Customized Attribute object is created.

Attribute representation flags
Identifies the Attribute Value's representation. This field is copied from the Attribute Representation flags descriptor in the corresponding Predefined Attribute object when the Customized Attribute object is created.

NLS index
Identifies the message number in the NLS message catalog that contains a textual description of the attribute. This field is copied from the NLS Index descriptor in the corresponding Predefined Attribute object when the Customized Attribute object is created.

B.1.6 Customized Dependency (CuDep)

The Customized Dependency (CuDep) object class describes device instances that depend on other device instances. Dependency does not imply a physical connection. This object class describes the dependence links between logical devices and physical devices as well as dependence links between logical devices. Physical dependencies of one device on another device are recorded in the **CuDv** object class.

Figure 6-3 on page 6-9 demonstrates instances of dependency and connection between devices.

The **CuDep** object class contains the following descriptors:

Table B-6. CuDep Object Class Descriptors

Descriptor name	Description	Descriptor status
name	Device name	Required
dependency	Dependency (device logical name)	Required

These descriptors have the following descriptions:

Device Name Identifies the logical name of the device having a dependency.

Dependency Identifies the logical name of the device instance on which there is a dependency. For example, a mouse, keyboard, and display might all be dependencies of a device instance of hft0.

B.1.7 Customized Device Driver (CuDvDr)

The Customized Device Driver (CuDvDr) object class stores information about critical resources that need concurrency management through the use of the Device Configuration Library routines. You should only access this object class through these five Device Configuration Library routines: the **genmajor**, **genminor**, **relmajor**, **reldevno**, and **getminor** routines.

These routines exclusively lock this class so that accesses to it are serialized. The genmajor and genminor routines return the major and minor number to the calling method. Similarly, the reldevno or relmajor routine releases the major or minor number from this object class.

The **CuDvDr** object class contains the following fields:

Table B-7. CuDvDr Object Class Descriptors

Descriptor name	Description	Descriptor status
resource	Resource name	Required
value1	Value1	Required
value2	Value2	Required
value3	Value3	Required

The Resource descriptor determines the nature of the values in the Value1, Value2, and Value3 descriptors. Possible values for the Resource descriptor are the strings devno and ddins.

The following table specifies the contents of the Value1, Value2, and Value3 descriptors, depending on the contents of the Resource descriptor.

Table B-8. Contents of Value1, Value2, and Value3 Descriptors

Resource	**Value1**	**Value2**	**Value3**
devno	Major number	Minor number	Device instance name
ddins	dd instance name	Major number	Null string

When the resource field contains the **devno** string, the Value1 field contains the device major number, Value2 the device minor number, and Value3 the device instance name. These value fields are filled in by the **genminor** subroutine, which takes a major number and device instance name as input, and generates the minor number and resulting **devno CuDvDr** object.

When the resource field contains the **ddins** string, the Value1 field contains the device driver instance name. This is typically the device driver name obtained from the Device Driver Name descriptor of the **PdDv** object. However, this name can be any unique string and is used by device methods to obtain the device driver major number. The Value2 field contains the device major number and the Value3 field is not used. These value fields are set by the **genmajor** subroutine, which takes a device instance name as input, and generates the corresponding major number, and resulting **ddins CuDvDr** object.

B.1.8 Customized VPD (CuVPD)

The Customized VPD (CuVPD) object class contains the Vital Product Data (VPD) for customized devices. VPD can be either machine-readable VPD or manually-entered user VPD information.

The **CuVPD** object class contains the following descriptors:

Table B-9. CuVPD Object Class Descriptors

Descriptor name	**Description**	**Descriptor status**
name	Device name	Required
vpd_type	VPD type	Required
vpd	VPD	Required

These fields are described as follows:

Device Name
Identifies the device logical name to which this VPD information belongs.

VPD Type Identifies the VPD as either machine-readable or manually-entered. The possible values:

HW_VPD Identifies machine-readable VPD.

USER_VPD Identifies manually-entered VPD.

VPD Identifies the Vital Product Data for the device. For machine-readable VPD, an entry in this field might include such information as serial numbers, engineering change levels, and part numbers.

Manually-entered VPD is intended for accounting purposes. For example, the user may want the name of the individual responsible for the device as well as his or her office number.

B.1.9 Configuration Rules (Config_Rules)

The Config_Rules object class contains the following descriptors:

Table B-10. Config_Rules Object Class Descriptors

ODM type	Descriptor name	Description	Descriptor status
ODM_SHORT	phase	Configuration manager phase	Required
ODM_SHORT	seq	Sequence value	Required
ODM_VCHAR	rulevalue[RSIZE]	Rule value	Required

These descriptors are described as follows:

Cfgmgr Phase This descriptor indicates which phase a rule should be executed under: phase 1, phase 2, or phase 2 service.

1 Indicates that the rule should be executed in phase 1.

2 Indicates that the rule should be executed in phase 2.

3 Indicates that the rule should be executed in phase 2 service mode.

Sequence Value In relation to the other rules of this phase, seq indicates the order in which to execute this program. In general, the lower the seq number, the higher the priority. For example, a rule with a seq number of 2 is executed before a rule with a seq number of 5. There is one exception to this: a value of 0 indicates a DON'T_CARE condition, and any rule with a seq number of 0 will be executed last.

Rule Value This is the full path name of the program to be invoked. The Rule Value descriptor may also contain any options that should be passed to that program. However, options must follow the program name, as the whole string will be executed as if it has been typed in on the command line.

Note: there is one rule for each program to execute. If multiple programs are needed, then multiple rules must be added.

Table B-11. Rule Values		
phase	**sequence**	**rule value**
1	1	/etc/methods/defsys
1	5	/etc/methods/deflvm
2	1	/etc/methods/defsys
2	5	/etc/methods/ptynode
2	10	/etc/methods/starthft
2	15	/etc/methods/starttty
2	20	/etc/methods/netstart.sh
3	1	/etc/methods/defsys
3	5	/etc/methods/ptynode
3	10	/etc/methods/starthft
3	15	/etc/methods/starttty

B.2 ODM Commands

ODM commands are entered on the command line. You can create, add, change, retrieve, display, delete, and remove objects and object classes with ODM.

B.2.1 ODM Commands That Handle Objects

odmadd
: Adds objects to an object class. The odmadd command takes an ASCII stanza file as input and populates object classes with objects found in the stanza file.

odmchange
: Changes specific objects in a specified object class.

odmdelete
: Removes objects from an object class.

odmget
: Retrieves objects from object classes and puts the object information into odmadd command format.

B.2.2 ODM Commands That Handle Object Classes

odmcreate
: Creates empty object classes. The odmcreate command takes an ASCII file describing object classes as input and produces C language .h and .c files to be used by the application accessing objects in those object classes.

odmdrop
: Removes an entire object class.

odmshow
: Displays the description of an object class. The odmshow command takes an object class name as input and puts the object class information into odmcreate command format.

B.3 ODM Routines

ODM subroutines can be put in a C language program to handle objects and object classes. An ODM subroutine returns a value of -1 if the subroutine is unsuccessful. The specific error diagnostic is returned as the odmerrno external variable (defined in the *odmi.h* include file). ODM error diagnostic constants are also included in the odmi.h include file.

B.3.1 ODM Subroutines That Handle Objects

odm_add_obj Adds a new object to the object class.

odm_change_obj Changes the contents of an object.

odm_get_by_id Retrieves an object by specifying its ID.

odm_get_first Retrieves the first object that matches the specified criteria in an object class.

odm_get_list Retrieves a list of objects that match the specified criteria in an object class.

odm_get_next Retrieves the next object that matches the specified criteria in an object class.

odm_get_obj Retrieves an object that matches the specified criteria from an object class.

odm_rm_by_id Removes an object by specifying its ID.

odm_rm_obj Removes all objects that match the specified criteria from the object class.

odm_run_method Invokes a method for the specified object.

B.3.2 ODM Subroutines That Handle Object Classes

odm_close_class Closes an object class.

odm_create_class Creates an empty object class.

odm_lock Locks an object class or group of classes.

odm_mount_class Retrieves an object class description.

odm_open_class Opens an object class.

odm_rm_class Removes an object class.

odm_set_path Sets the default path for object classes.

odm_set_perms Sets default permissions for object class creation.

odm_unlock Unlocks an object class or group of classes.

B.3.3 ODM Subroutines That Handle Other ODM Functions

odm_err_msg Retrieves a message string.

odm_free_list Frees memory allocated for the odm_get_list subroutine.

odm_initialize Initializes an ODM session.

odm_terminate Ends an ODM session.

B.4 Device Configuration Library Routines

Following are the pre-existing conditions for using the device configuration library routines:

- The caller has initialized the ODM before invoking any of these library routines. This is done using the **initialize_odm** routine. Similarly, the caller must terminate the ODM (using the **terminate_odm** routine) after these library routines have completed. The only one of these routines that does not require initialization and termination is the **attrval** routine.
- Since all of these library routines (except **attrval**, **getattr**, and **putattr**) access the Customized Device Driver object class, this class must be exclusively locked and unlocked at the proper times. The application does this by using the **odm_lock** and **odm_unlock** routines. In addition, those library routines that access the Customized Device Driver object class exclusively lock this class with their own internal locks.
- The caller has set the path to */etc/objrepos* (where all the Device Configuration object classes reside) by using the **odm_set_path** ODM routine.

Following are the 11 device configuration library routines:

attrval Verifies that attributes are within range.

genmajor Generates the next available major number for a device.

genminor Generates the smallest unused minor number or a requested minor number for a device.

genseq Generates a sequence number.

getattr Returns attribute objects from the Predefined Attribute object class or the Customized Attribute object class, or from both.

getminor Gets from the Customized Device Driver object class the minor numbers for a given major number.

loadext Loads or unloads and binds or unbinds device drivers to or from the kernel.

putattr Updates attribute information in the Customized Attribute object class or creates a new object for the attribute information.

reldevno Releases the minor number or major number, or both, for a device instance.

relmajor Releases the major number associated with a specific device driver instance.

relseq Releases the specified sequence number.

B.5 Real Time Interface Co-Processor Adapter Configuration Files

B.5.1 ODM Stanzas (ric.add)

```
### Stanzas for populating PdDv

PdDv:
* device is of class adapter
        class = "adapter"
* device is of subclass mca, indicating its connection type
        subclass = "mca"
* device is an ric type of adapter card
        type = "ric"
* prefix to be used when naming customized devices of this type
        prefix = "rica"
* the card id obtained from pos(0) and pos(1)
        devid = "0x708f"
* this devices is not a base device
        base = 0
* this devices has no VPD
        has_vpd = 0
* this device is detectable
        detectable = 1
* change status is to be set to NEW when defining a device of this type
        chgstatus = 0
* this device is not a bus extension
        bus_ext = 0
* this device is a FRU (field replacable unit)
        fru = 1
* the LED value to be displayed when being configured at boot time
        led = 0x777
* the NLS message catalog containing text descriptions of adapter
        catalog = "ric.cat"
* the NLS message set number containing text descriptions of adapter
        setno = 1
* the NLS message number of the text description of the adapter
        msgno = 1
* there is no device driver to be loaded when the adapter is configured
        DvDr = ""
* the name of the define method
        Define = "/etc/methods/define"
* the name of the configure method
        Configure = "/etc/methods/cfgrica"
* this device does not have a change method
        Change = ""
* the name of the unconfigure method
        Unconfigure = "/etc/methods/ucfgdevice"
* the name of the undefine method
        Undefine = "/etc/methods/undefine"
* this device does not have a start method
        Start = ""
* this device does not have a stop method
        Stop = ""
* this device does not provide inventory information
        inventory_only = 0
* the adapters unique type consisting of class, subclass, and type
        uniquetype = "adapter/mca/ric"
```

```
PdDv:
* device is of class ricport
        class = "ricport"
* device is of subclass ricp, indicating its connection type
        subclass = "ricp"
* device is a port type
        type = "port"
* prefix to be used when naming customized devices of this type
        prefix = "ric"
* this devices does not have a card id
        devid = 0
* this devices is not a base device
        base = 0
* this devices has no VPD
        has_vpd = 0
* this devices is not detectable
        detectable = 0
* change status is to be set to NEW when defining a device of this type
        chgstatus = 0
* this device is not a bus extension
        bus_ext = 0
* this device is a FRU (field replacable unit)
        fru = 1
* the LED value to be displayed when being configured at boot time
        led = 0x778
* the NLS message catalog containing text descriptions of device
        catalog = "ric.cat"
* the NLS message set number containing text descriptions of device
        setno = 2
* the NLS message number of the text description of the device
        msgno = 1
* the name of the device driver in /etc/drivers directory
        DvDr = "ricdd"
* the name of the define method
        Define = "/etc/methods/define"
* the name of the configure method
        Configure = "/etc/methods/cfgricp"
* the name of the change method
        Change = "/etc/methods/chggen"
* the name of the unconfigure method
        Unconfigure = "/etc/methods/ucfgdevice"
* the name of the undefine method
        Undefine = "/etc/methods/undefine"
* this device does not have a start method
        Start = ""
* this device does not have a stop method
        Stop = ""
* this device does not provide inventory information
        inventory_only = 0
* the devices unique type consisting of class, subclass, and type
        uniquetype = "ricport/ricp/port"

### Stanzas for populating PdAt

PdAt:

* the adapters unique type consisting of class, subclass, and type
        uniquetype = "adapter/mca/ric"
* attribute name
```

```
        attribute = "bus_intr_lvl"
* default value for attribute
        deflt = "3"
* possible values the attribute can be set to
        values = "3,4,7,9,10,11,12"
* width not used for this type of attribute
        width = ""
* this is a bus interrupt level attribute
        type = "I"
* this attribute is displayable but not user changeable
        generic = "D"
* this attribute is numeric and possible values are represented as a list
        rep = "nl"
* the NLS message number of the text description for this attribute
        nls_index = 5

PdAt:
        uniquetype = "adapter/mca/ric"
        attribute = "bus_io_addr"
        deflt = "0x2a0"
        values = "0x2a0-0x1ea0,0x400"
* range of addresses to be assigned (8 bytes)
        width = "0x08"
* this is a bus I/O address attribute
        type = "O"
        generic = "D"
* this attribute is numeric and possible values are represented as a range
        rep = "nr"
        nls_index = 4

PdAt:
        uniquetype = "adapter/mca/ric"
        attribute = "dma_lvl"
        deflt = "0"
        values = "0-14,1"
        width = ""
* this is a dma level attribute
        type = "A"
        generic = "D"
        rep = "nr"
        nls_index = 7

PdAt:
        uniquetype = "adapter/mca/ric"
        attribute = "bus_mem_addr"
        deflt = "0x10000"
        values = "0x10000-0xff0000,0x10000"
        width = "0x10000"
* this is a bus memory address attribute
        type = "M"
        generic = "D"
        rep = "nr"
        nls_index = 2

PdAt:
        uniquetype = "adapter/mca/ric"
        attribute = "dma_bus_mem"
        deflt = "0x100000"
        values = "0x100000-0xfffc0000,0x1000"
```

```
        width = "0x40000"
        type = "M"
        generic = "D"
        rep = "nr"
        nls_index = 3

PdAt:
        uniquetype = "adapter/mca/ric"
        attribute = "intr_priority"
        deflt = "3"
        values = "3"
        width = ""
* this is an interrupt priority class attribute
        type = "P"
        generic = "D"
        rep = "n"
        nls_index = 6

PdAt:
        uniquetype= "ricport/ricp/port"
        attribute= "rdto"
        deflt= "92"
        values= "6-128,1"
        width= ""
* this is a regular attribute that is not a bus resource
        type= "R"
* this attribute is displayable and user changeable
        generic= "DU"
        rep= "nr"
        nls_index= 2

PdAt:
        uniquetype= "adapter/mca/ric"
        attribute= "ucode"
        deflt="/etc/asw/ricasw"
        values= "mpqpasw"
        width= ""
        type= "R"
        generic= "D"
        rep = "s"
        nls_index= 8

### Stanzas for populating PdCn

#
# These identify eight connection locations on the ric adapter
# and that devices of subclass ricp can be attached.
#

PdCn:
* the adapters unique type consisting of class, subclass, and type
        uniquetype = "adapter/mca/ric"
* the subclass (connection type) of devices that can be attached
        connkey = "ricp"
* the connection location where a device can be attached
        connwhere = "0"

PdCn:
        uniquetype = "adapter/mca/ric"
```

```
        connkey = "ricp"
        connwhere = "1"

PdCn:
        uniquetype = "adapter/mca/ric"
        connkey = "ricp"
        connwhere = "2"

PdCn:
        uniquetype = "adapter/mca/ric"
        connkey = "ricp"
        connwhere = "3"

PdCn:
        uniquetype = "adapter/mca/ric"
        connkey = "ricp"
        connwhere = "4"

PdCn:
        uniquetype = "adapter/mca/ric"
        connkey = "ricp"
        connwhere = "5"

PdCn:
        uniquetype = "adapter/mca/ric"
        connkey = "ricp"
        connwhere = "6"

PdCn:
        uniquetype = "adapter/mca/ric"
        connkey = "ricp"
        connwhere = "7"
```

B.5.2 Message Catalog for Ric Adapter and Ports

```
$
$  ric.msg
$
$  Realtime Interface Co-Processor configuration message catalog
$
$quote "
$
$  RIC adapter
$set 1
1 "Realtime Interface Co-Processor Portmaster Adapter"
2 "Bus memory address"
3 "Address of bus memory used for DMA"
4 "Bus I/O address"
5 "Bus interrupt level"
6 "Interrupt priority"
7 "DMA arbitration level"
8 "Adapter micro-code file name"
$
$ RIC Ports
$set 2
1 "Ric Adapter Port"
2 "Receive Data Transfer Offset"
3 "STATE to be configured at boot time"
$
$set 3
0 "List All Defined Ric Ports"
1 "Add a Ric Port"
2 "Move a Ric Port Definition to Another Port"
3 "Change / Show Characteristics of a Ric Port"
4 "KEEP definition in database"
5 "Remove a Ric Port"
6 "Configure a Defined Ric Port"
7 "Ric Port"
8 "Parent Adapter"
9 "PORT number"
10 "Status"
11 "Location"
12 "yes,no"
13 "Apply change to DATABASE only"
```

B.5.3 Adapter Configuration Method (cfgrica.c)

```
1    /*
2     *
3     * FUNCTION:    Configure method for Realtime Interface
4     *              Co-Processor Portmaster Adapter/A
5     *
6     * INTERFACE:   cfgrica -l <logical_name> [-<1|2>]
7     *
8     */
9
10   /* header files needed for compilation */
11   #include <stdio.h>
12   #include <sys/types.h>
13   #include <sys/cfgdb.h>
14   #include <sys/cfgodm.h>
15   #include <sys/sysconfig.h>
16   #include <sys/device.h>
17   #include <cf.h>
18   #include <fcntl.h>
19   #include <sys/mdio.h>
20
21   /* local header files */
22   #include "debug.h"
23
24   /* main function code */
25   main(argc, argv)
26   int     argc;
27   char    *argv[];
28   {
29       char    *logical_name;     /* logical name to configure */
30       char    *phase1, *phase2;  /* ipl phase flags */
31       char    sstring[256];      /* search criteria pointer */
32       char    conflist[1024];    /* busresolve() configured devices */
33       char    not_resolved[1024];/* busresolve() not resolved devices */
34
35       struct    cfg_dd cfg;      /* sysconfig command structure */
36       struct Class *cusdev;      /* customized devices class ptr */
37       struct Class *predev;      /* predefined devices class ptr */
38       struct CuDv cusobj;        /* customized device object storage */
39       struct PdDv preobj;        /* predefined device object storage */
40       struct CuDv parobj;        /* customized device object storage */
41       struct CuDv dmyobj;        /* customized device object storage */
42
43       ushort    devid;           /* Device id - used at run-time */
44       int    ipl_phase;          /* ipl phase: 0=run,1=phase1,2=phase2 */
45       int    slot;               /* slot of adapters */
46       int    rc;                 /* return codes go here */
47       int     errflg,c;          /* used in parsing parameters   */
48
49       extern  int     optind;    /* for getopt function */
50       extern  char    *optarg;   /* for getopt function */
```

```
51          /*****                                                          */
52          /***** Parse Parameters                                         */
53          /*****                                                          */
54          ipl_phase = RUNTIME_CFG;
55          errflg = 0;
56          logical_name = NULL;
57
58          while ((c = getopt(argc,argv,"l:12")) != EOF) {
59                  switch (c) {
60                  case 'l':
61                          if (logical_name != NULL)
62                                  errflg++;
63                          logical_name = optarg;
64                          break;
65                  case '1':
66                          if (ipl_phase != RUNTIME_CFG)
67                                  errflg++;
68                          ipl_phase = PHASE1;
69                          break;
70                  case '2':
71                          if (ipl_phase != RUNTIME_CFG)
72                                  errflg++;
73                          ipl_phase = PHASE2;
74                          break;
75                  default:
76                          errflg++;
77                  }
78          }
79          if (errflg) {
80                  /* error parsing parameters */
81                  DEBUG_0("cfgrica: command line error\n");
82                  exit(E_ARGS);
83          }
84
85          /*****                                                          */
86          /***** Validate Parameters                                      */
87          /*****                                                          */
88          /* logical name must be specified */
89          if (logical_name == NULL) {
90                  DEBUG_0("cfgrica: logical name must be specified\n");
91                  exit(E_LNAME);
92          }
93
94          DEBUG_1 ("Configuring device: %s\n",logical_name)
```

```
 95         /* start up odm */
 96         if (odm_initialize() == -1) {
 97                 /* initialization failed */
 98                 DEBUG_0("cfgrica: odm_initialize() failed\n")
 99                 exit(E_ODMINIT);
100         }
101
102         /* lock the database */
103         if (odm_lock("/etc/objrepos/config_lock",0) == -1) {
104                 DEBUG_0("cfgrica: odm_lock() failed\n")
105                 err_exit(E_ODMLOCK);
106         }
107         DEBUG_0 ("ODM initialized and locked\n")
108
109         /* open customized devices object class */
110         if ((int)(cusdev = odm_open_class(CuDv_CLASS)) == -1) {
111                 DEBUG_0("cfgrica: open class CuDv failed\n");
112                 err_exit(E_ODMOPEN);
113         }
114
115         /* search for customized object with this logical name */
116         sprintf(sstring, "name = '%s'", logical_name);
117         rc = (int)odm_get_first(cusdev,sstring,&cusobj);
118         if (rc==0) {
119                 /* No CuDv object with this name */
120                 DEBUG_1("cfgrica: failed to find CuDv object for %s\n", logical_name);
121                 err_exit(E_NOCuDv);
122         }
123         else if (rc==-1) {
124                 /* ODM failure */
125                 DEBUG_0("cfgrica: ODM failure getting CuDv object");
126                 err_exit(E_ODMGET);
127         }
128
129         /* open predefined devices object class */
130         if ((int)(predev = odm_open_class(PdDv_CLASS)) == -1) {
131                 DEBUG_0("cfgrica: open class PdDv failed\n");
132                 err_exit(E_ODMOPEN);
133         }
134
135         /* get predefined device object for this logical name */
136         sprintf(sstring, "uniquetype = '%s'", cusobj.PdDvLn_Lvalue);
137         rc = (int)odm_get_first(predev, sstring, &preobj);
138         if (rc==0) {
139                 /* No PdDv object for this device */
140                 DEBUG_0("cfgrica: failed to find PdDv object for this device\n");
141                 err_exit(E_NOPdDv);
142         }
143         else if (rc==-1) {
144                 /* ODM failure */
145                 DEBUG_0("cfgrica: ODM failure getting PdDv object");
146                 err_exit(E_ODMGET);
147         }
148
149         /* close predefined device object class */
150         if (odm_close_class(predev) == -1) {
151                 DEBUG_0("cfgrica: close object class PdDv failed");
152                 err_exit(E_ODMCLOSE);
153         }
154
155         /*****************************************************************
156           If ric adapter is being configured during an ipl phase, then
157           display this device's LED value on the system LEDs.
158           *****************************************************************/
159         if (ipl_phase != RUNTIME_CFG)
160                 setleds(preobj.led);
161         /*****************************************************************
162           Check to see if the device is already configured (AVAILABLE).
163           We actually go about the business of configuring the device
164           only if the device is not configured yet. Configuring the
165           device in this case refers to the process of checking parent
166           and sibling status, checking for attribute consistency
167           *****************************************************************/
168
169         if (cusobj.status == DEFINED) {
```

```
170
171                 /******************************************************
172                   The device is not available to the system yet. Now
173                   check to make sure that the device's relations will
174                   allow it to be configured. In particular, make sure
175                   that the parent is configured (AVAILABLE), and that
176                   no other devices are configured at the same location.
177                   ******************************************************/
178
179                 /* get the device's parent object */
180                 sprintf(sstring, "name = '%s'", cusobj.parent);
181                 rc = (int)odm_get_first(cusdev,sstring,&parobj);
182                 if (rc==0) {
183                         /* Parent device not in CuDv */
184                         DEBUG_0("cfgrica: no parent CuDv object\n")
185                         err_exit(E_NOCuDvPARENT);
186                 }
187                 else if (rc==-1) {
188                         /* ODM failure */
189                         DEBUG_0("cfgrica: ODM failure getting parent CuDv object\n")
190                         err_exit(E_ODMGET);
191                 }
192
193                 /* parent must be available to continue */
194                 if (parobj.status != AVAILABLE) {
195                         DEBUG_0("cfgrica: parent is not AVAILABLE")
196                         err_exit(E_PARENTSTATE);
197                 }
198
199                 /* make sure that no other devices are configured     */
200                 /* at this location                                   */
201                 sprintf(sstring, "name = '%s' AND parent = '%s' AND
202                                 connwhere = '%s' AND status = %d",
203                         cusobj.name, cusobj.parent, cusobj.connwhere, AVAILABLE);
204                 rc = (int)odm_get_first(cusdev,sstring,&dmyobj);
205                 if (rc == -1) {
206                         /* odm failure */
207                         err_exit(E_ODMGET);
208                 } else if (rc) {
209                         /* Error: device config'd at this location */
210                         DEBUG_0("cfgrica: device already AVAILABLE at this connection\n")
211                         err_exit(E_AVAILCONNECT);
212                 }
213
214                 /**************************************************
215                   If ric adapter is being configured at RUN TIME,
216                   then we must resolve any bus attribute conflicts
217                   before configuring device to the driver.
218                   If being configured at boot time, the bus
219                   configurator will have already resolved conflicts.
220                   **************************************************/
221                 if (ipl_phase == RUNTIME_CFG) {
222                         if (!strcmp(preobj.subclass,"mca")) {
223                                 /* Make sure card is in specified slot */
224                                 slot = atoi(cusobj.connwhere);
225                                 DEBUG_1("cfgrica: slot = %d\n", slot)
226                                 devid = (ushort) strtol(preobj.devid,(char **) NULL,0);
227                                 sprintf (sstring,"/dev/%s",cusobj.parent);
228                                 rc = chkslot(sstring,slot,devid);
229                                 if (rc != 0) {
230                                         DEBUG_2("cfgrica: card %s not found in slot %d\n",
231                                                 logical_name,slot);
232                                         err_exit(rc);
233                                 }
234                                         DEBUG_2("cfgrica: card %s found in slot %d\n",
235                                         logical_name,slot);
236                         }
237
238                         /* Invoke Bus Resolve  */
239                         rc = busresolve(logical_name,(int)0,conflist,
240                                 not_resolved, cusobj.parent);
241                         if (rc != 0) {
242                                 DEBUG_0("cfgrica: bus resources could not be resolved\n")
243                                 err_exit(rc);
244                         }
```

```
245                 }
246
247                 /* update customized device object with a change operation */
248                 cusobj.status = AVAILABLE;
249                 if (odm_change_obj(cusdev, &cusobj) == -1) {
250                         /* ODM failure */
251                         DEBUG_0("cfgrica: ODM failure updating CuDv object\n");
252                         err_exit(E_ODMUPDATE);
253                 }
254         } /* end if (device is not AVAILABLE) then ... */
255
256         /* call device specific routine to detect/manage child devices */
257         DEBUG_0("cfgrica: Calling define_children()\n")
258         if (define_children(logical_name, ipl_phase) != 0) {
259                 /* error defining children */
260                 DEBUG_0("cfgrica: error defining children\n");
261                 err_exit(E_FINDCHILD);
262         }
263         DEBUG_0("cfgrica: Returned from define_children()\n")
264
265         /* close customized device object class */
266         if (odm_close_class(cusdev) == -1) {
267                 DEBUG_0("cfgrica: error closing CuDv object class\n");
268                 err_exit(E_ODMCLOSE);
269         }
270
271         odm_terminate();
272         exit(0);
273
```

```
274     }
275     /*
276      * NAME: err_exit
277      *
278      * FUNCTION: Closes any open object classes and terminates ODM.  Used to
279      *           back out on an error.
280      *
281      * NOTES:
282      *
283      *   err_exit( exitcode )
284      *       exitcode = The error exit code.
285      *
286      * RETURNS:
287      *               None
288      */
289
290     err_exit(exitcode)
291     char    exitcode;
292     {
293             /* Close any open object class */
294             odm_close_class(CuDv_CLASS);
295             odm_close_class(PdDv_CLASS);
296             odm_close_class(CuAt_CLASS);
297
298             /* Terminate the ODM */
299             odm_terminate();
300             exit(exitcode);
301     }
```

```
302  /*
303   *
304   * NAME: define_children
305   *
306   * FUNCTION: To invoke the generic define method for each child device not
307   *           already in the customized database. This will result in all
308   *           children that are not AVAILABLE being created in the customized
309   *           data base with status DEFINED. To then output on stdout the name
310   *           of each DEFINED child device in order to cause that child's
311   *           configuration.
312   *
313   *
314   * NOTES: For the RIC adapter, there are eight children (ports 0 through 7)
315   *        which are specifically looped over.
316   *
317   * RETURNS: 0 - Success
318   *         <0 - Failure :
319   *               E_ODMOPE    - Open of PdDv or CuDv failed.
320   *               E_ODMGET    - Get of an object from ODM failed.
321   *               E_FINDCHILD - The Define method for a child returned in error.
322   *               E_ODMUPDATE - Update of an object in ODM failed.
323   *               E_ODMCLOSE  - Close of an object class failed.
324   *
325   */
326
327  int  define_children( lognam, iplphs )
328  char  *lognam;                      /* logical name of device instance */
329  int    iplphs;                      /* phase if ipl */
330  {
331        long    rc, port;             /* return code, port counter */
332        char    string[512];          /* working string */
333        char    *out_p;
334        long    objtest = 0;          /* existance test, 1-test, 0-don't */
335        struct  CuDv   cudvport;      /* Object structures */
336        struct  PdDv   pddvport;
337
338
339          /* read Predef object for RICP ports */
340
341          if(( rc = odm_get_obj( PdDv_CLASS,"uniquetype = 'ricport/ricp/port'",
342                  &pddvport, ODM_FIRST )) == 0 ) {
343                  DEBUG_2("def_chil: get failed lname=%s rc=%d\n",
344                          lognam,rc)
345                  return E_NOPdDv;
346          } else if ( rc == -1 ) {
347                  DEBUG_1("def_chil: odmget error lnmae=%s\n",lognam)
348                  return E_ODMGET;
349          }
350
351          DEBUG_0("def_chil: got pddv object\n")
352
353          for (port=0; port<8; ++port)    /* for each port on the adapter */
354          {
355                  objtest = 1;
356
357                  /* retrieve current CuDv port object */
358                  sprintf( string, "parent = '%s' AND connwhere = '%d'", lognam,
359                          port );
360                  DEBUG_1("def_chil: str=*%s*\n",string)
361                  rc =(long)odm_get_obj(CuDv_CLASS,string,&cudvport,ODM_FIRST);
362                  DEBUG_1("def_chil: after get rc=%d\n",rc)
363
364                  if (rc == 0)    /* if current port is not defined ... */
365                  /* invoke the generic define method */
366                  /* retrieve current CuDv port object */
367                  {
368                          sprintf( string,
369                                  "-c ricport -s ricp -t %s -p %s -w %d",
370                                  "port", lognam, port );
371                          DEBUG_2("def_chil: calling %s %s\n",pddvport.Define,
372                                  string)
373                          if(odm_run_method(pddvport.Define,string,&out_p,NULL)){
374                                  fprintf(stderr,"cfgrica: can't run %s\n",
375                                          pddvport.Define);
376                                  return E_ODMRUNMETHOD;
```

```
377                         }
378                         fprintf( stdout, "%s\n", out_p );
379                         objtest = 0;
380                 }
381                 else if (rc > 0) {
382                         cudvport.chgstatus = SAME;
383 
384                         /*
385                         * Change Customized Device Object Class
386                         */
387                         if ((rc = odm_change_obj(CuDv_CLASS,&cudvport)) < 0) {
388                                 fprintf(stderr,"def_chil: change failed\n");
389                                 return E_ODMUPDATE;
390                         }
391                         fprintf (stdout, "%s\n", cudvport.name);
392 
393                 }
394         }
395         return 0;
396 }
```

```
397  /*
398   *  NAME: chkslot
399   *
400   *  FUNCTION:
401   *      Return zero value if cardid is in desired slot.
402   *
403   *  INPUTS:
404   *      bus     - The name of the bus device, for example, bus0.
405   *      slot    - The slot number from the parent connection descriptor.
406   *                It should be a value of 1 through 8, with 0 being used
407   *                for the Standard I/O Planar.
408   *      cardid  - The card Id composed as ((POS0<<8) || POS1).
409   *
410   * RETURNS: Returns  0 on success, >0 Error code.
411   */
412
413  int
414  chkslot(bus, slot, cardid)
415  char *bus;
416  int slot;
417  ushort cardid;
418  {
419          MACH_DD_IO mddRecord;
420          uchar pos[2];
421          int fd;
422          int i;
423
424          pos[0] = 0xff;
425          pos[1] = 0xff;
426
427          /* decrement slot number found in database */
428          if (slot == 0)
429                  /* checking standard I/O planar */
430                  slot = 15;
431          else
432                  slot--;
433
434          if (0 > (fd = open(bus, O_RDWR))) {
435                  DEBUG_1("cfgrica: open %s failed\n", bus)
436                  return(E_NODETECT);
437          }
438
439          mddRecord.md_size = 2;
440          mddRecord.md_incr = MV_BYTE;
441          mddRecord.md_data = pos;
442          mddRecord.md_addr = POSREG(0, slot);
443
444          if (0 > ioctl(fd, MIOCCGET, &mddRecord)) {
445                  DEBUG_0("cfgrica: ioctl failed\n")
446                  return(E_NODETECT);
447          }
448
449          close(fd);
450
451          if (cardid == ((pos[0] << 8) | pos[1]))
452                  return(0);
453          else
454                  DEBUG_2("cfgrica: cardid = 0x%x pos = 0x%x\n", cardid,
455                          ((pos[0] << 8) | pos[1]))
456                  return(E_NODETECT);
457  }
```

B.5.4 Ric Port Configuration Method (cfgricp.c)

```
1    /*
2     *
3     * FUNCTION:    Configure method for a RIC Adapter Port
4     *
5     * INTERFACE:   cfgricp -l <logical_name> [-<1|2>]
6     *
7     */
8
9    /* header files needed for compilation */
10   #include <stdio.h>
11   #include <sys/types.h>
12   #include <sys/cfgdb.h>
13   #include <sys/cfgodm.h>
14   #include <cf.h>
15   #include <sys/sysconfig.h>
16   #include <sys/sysmacros.h>
17   #include <sys/device.h>
18   #include <sys/stat.h>
19   #include <sys/errno.h>
20
21   /* Local header files */
22   #include "debug.h"
23   #include "ric.h"
24   #include "ricmisc.h"
25   #include "ricstruct.h"
26
27   /* external functions */
28   extern long     genmajor();
29
30   /* main function code */
31   main(argc, argv, envp)
32   int     argc;
33   char    *argv[];
34   char    *envp[];
35   {
36        char     *logical_name;          /* logical name to configure */
37        char     *phase1, *phase2;       /* ipl phase flags */
38        char     sstring[256];           /* search criteria pointer */
39        char     conflist[1024];         /* busresolve() configured devices */
40        char     not_resolved[1024];     /* busresolve() not resolved devices */
41
42        struct cfg_dd cfg;               /* sysconfig command structure */
43        struct Class *cusdev;            /* customized devices class ptr */
44        struct Class *predev;            /* predefined devices class ptr */
45        struct CuDv cusobj;              /* customized device object storage */
46        struct PdDv preobj;              /* predefined device object storage */
47        struct CuDv parobj;              /* customized device object storage */
48        struct CuDv dmyobj;              /* customized device object storage */
```

```
 49         int     majorno;                  /* major number assigned to device */
 50         int     minorno;                  /* minor number assigned to device */
 51         long    *minor_list;              /* list returned by getminor */
 52         int     how_many;                 /* number of minors in list */
 53         ushort     devid;                 /* Device id - used at run-time */
 54         int     ipl_phase;                /* ipl phase: 0=run,1=phase1,2=phase2 */
 55         int     slot;                     /* slot of adapters */
 56         int     rc;                       /* return codes go here */
 57         int     errflg,c;                 /* used in parsing parameters   */
 58
 59         extern  int     optind;           /* for getopt function */
 60         extern  char    *optarg;          /* for getopt function */
 61
 62            /*****                                                          */
 63            /***** Parse Parameters                                         */
 64            /*****                                                          */
 65            ipl_phase = RUNTIME_CFG;
 66            errflg = 0;
 67            logical_name = NULL;
 68
 69            while ((c = getopt(argc,argv,"l:12")) != EOF) {
 70                    switch (c) {
 71                    case 'l':
 72                            if (logical_name != NULL)
 73                                    errflg++;
 74                            logical_name = optarg;
 75                            break;
 76                    case '1':
 77                            if (ipl_phase != RUNTIME_CFG)
 78                                    errflg++;
 79                            ipl_phase = PHASE1;
 80                            break;
 81                    case '2':
 82                            if (ipl_phase != RUNTIME_CFG)
 83                                    errflg++;
 84                            ipl_phase = PHASE2;
 85                            break;
 86                    default:
 87                            errflg++;
 88                    }
 89            }
 90            if (errflg) {
 91                    /* error parsing parameters */
 92                    DEBUG_0("cfgricp: command line error\n");
 93                    exit(E_ARGS);
 94            }
 95
 96            /*****                                                          */
 97            /***** Validate Parameters                                      */
 98            /*****                                                          */
 99            /* logical name must be specified */
100            if (logical_name == NULL) {
101                    DEBUG_0("cfgricp: logical name must be specified\n");
102                    exit(E_LNAME);
103            }
104
105            DEBUG_1 ("Configuring device: %s\n",logical_name)
106
107            /* start up odm */
108            if (odm_initialize() == -1) {
109                    /* initialization failed */
110                    DEBUG_0("cfgricp: odm_initialize() failed\n")
111                    exit(E_ODMINIT);
112            }
113
114            /* lock the database */
115            if (odm_lock("/etc/objrepos/config_lock",0) == -1) {
116                    DEBUG_0("cfgricp: odm_lock() failed\n")
117                    err_exit(E_ODMLOCK);
118            }
119
120            DEBUG_0 ("ODM initialized and locked\n")
121
122            /* open customized devices object class */
123            if ((int)(cusdev = odm_open_class(CuDv_CLASS)) == -1) {
```

```
124                 DEBUG_0("cfgricp: open class CuDv failed\n");
125                 err_exit(E_ODMOPEN);
126         }
127
128         /* search for customized object with this logical name */
129         sprintf(sstring, "name = '%s'", logical_name);
130         rc = (int)odm_get_first(cusdev,sstring,&cusobj);
131         if (rc==0) {
132                 /* No CuDv object with this name */
133                 DEBUG_1("cfgricp: failed to find CuDv object for %s\n", logical_name);
134                 err_exit(E_NOCuDv);
135         }
136         else if (rc==-1) {
137                 /* ODM failure */
138                 DEBUG_0("cfgricp: ODM failure getting CuDv object");
139                 err_exit(E_ODMGET);
140         }
141
142         /* open predefined devices object class */
143         if ((int)(predev = odm_open_class(PdDv_CLASS)) == -1) {
144                 DEBUG_0("cfgricp: open class PdDv failed\n");
145                 err_exit(E_ODMOPEN);
146         }
147
148         /* get predefined device object for this logical name */
149         sprintf(sstring, "uniquetype = '%s'", cusobj.PdDvLn_Lvalue);
150         rc = (int)odm_get_first(predev, sstring, &preobj);
151         if (rc==0) {
152                 /* No PdDv object for this device */
153                 DEBUG_0("cfgricp: failed to find PdDv object for this device\n");
154                 err_exit(E_NOPdDv);
155         }
156         else if (rc==-1) {
157                 /* ODM failure */
158                 DEBUG_0("cfgricp: ODM failure getting PdDv object");
159                 err_exit(E_ODMGET);
160         }
161
162         /* close predefined device object class */
163         if (odm_close_class(predev) == -1) {
164                 DEBUG_0("cfgricp: close object class PdDv failed");
165                 err_exit(E_ODMCLOSE);
166         }
167
168         /****************************************************************
169           If this device is being configured during an ipl phase, then
170           display this device's LED value on the system LEDs.
171           ****************************************************************/
172         if (ipl_phase != RUNTIME_CFG)
173                 setleds(preobj.led);
174
175         /*****************************************************************
176           Check to see if the device is already configured (AVAILABLE).
177           We actually go about the business of configuring the device
178           only if the device is not configured yet. Configuring the
179           device in this case refers to the process of checking parent
180           and sibling status, checking for attribute consistency, building
181           a DDS, loading the driver, etc...
182           ****************************************************************/
183
184         if (cusobj.status == DEFINED) {
185
186                 /*******************************************************
187                   The device is not available to the system yet. Now
188                   check to make sure that the device's relations will
189                   allow it to be configured. In particular, make sure
190                   that the parent is configured (AVAILABLE), and that
191                   no other devices are configured at the same location.
192                   *****************************************************/
193
194                 /* get the device's parent object */
195                 sprintf(sstring, "name = '%s'", cusobj.parent);
196                 rc = (int)odm_get_first(cusdev,sstring,&parobj);
197                 if (rc==0) {
198                         /* Parent device not in CuDv */
```

```
199                     DEBUG_0("cfgricp: no parent CuDv object\n");
200                     err_exit(E_NOCuDvPARENT);
201             }
202             else if (rc==-1) {
203                     /* ODM failure */
204                     DEBUG_0("cfgricp: ODM failure getting parent CuDv object\n")
205                     err_exit(E_ODMGET);
206             }
207
208             if (parobj.status != AVAILABLE) {
209                     DEBUG_0("cfgricp: parent is not AVAILABLE")
210                     err_exit(E_PARENTSTATE);
211             }
212
213             /* make sure that no other devices are configured     */
214             /* at this location                                    */
215             sprintf(sstring, "parent = '%s' AND connwhere = '%s' AND status = %d",
216                     cusobj.parent, cusobj.connwhere, AVAILABLE);
217             rc = (int)odm_get_first(cusdev,sstring,&dmyobj);
218             if (rc == -1) {
219                     /* odm failure */
220                     err_exit(E_ODMGET);
221             } else if (rc) {
222                     /* Error: device config'd at this location */
223                     DEBUG_0("cfgricp: device already AVAILABLE at this connection\n")
224                     err_exit(E_AVAILCONNECT);
225             }
226
227             /****************************************************
228               Load device driver, get major number, and call
229               device dependent routines to get minor number,
230               make special files, and build DDS.
231               This code then passes the DDS to the driver.
232              ****************************************************/
233             /* call loadext to load the device driver */
234             if ((cfg.kmid = loadext(preobj.DvDr, TRUE, FALSE)) == NULL) {
235                     /* error loading device driver */
236                     DEBUG_1("cfgricp: error loading driver %s\n", preobj.DvDr)
237                     err_exit(E_LOADEXT);
238             }
239
240             /* get major number      */
241             DEBUG_0("cfgricp: Calling genmajor()\n")
242             if ((majorno = genmajor(preobj.DvDr)) == -1) {
243                     DEBUG_0("cfgricp: error generating major number");
244                     err_undo(preobj.DvDr);
245                     err_exit(E_MAJORNO);
246             }
247
248             DEBUG_1("cfgricp: Returned major number: %d\n",majorno)
249
250             /* get minor number      */
251             DEBUG_0("cfgricp: Calling getminor()\n")
252             minor_list = getminor(majorno,&how_many,logical_name);
253             if (minor_list == NULL || how_many == 0) {
254                     DEBUG_0("cfgricp: Calling generate_minor()\n")
255                     rc = generate_minor(logical_name, majorno, &minorno);
256                     if (rc) {
257                             DEBUG_1("cfgricp: error generating minor number, rc=%d\n",rc)
258                             /* First make sure any minors that might have */
259                             /* been assigned are cleaned up */
260                             reldevno(logical_name, TRUE);
261                             err_undo(preobj.DvDr);
262                             if ( rc < 0 || rc > 255)
263                                     rc = E_MINORNO;
264                             err_exit(rc);
265                     }
266                     DEBUG_0("cfgricp: Returned from generate_minor()\n")
267             }
268             else
269                     minorno = *minor_list;
270             DEBUG_1("cfgricp: minor number: %d\n",minorno)
271
272             /* create devno for this device */
273             cfg.devno = makedev(majorno, minorno);
```

```
274
275                 /* make special files      */
276                 DEBUG_0("cfgricp: Calling make_special_files()\n")
277                 rc = make_special_files(logical_name, cfg.devno);
278                 if (rc) {
279                         /* error making special files */
280                         DEBUG_1("cfgricp: error making special file(s), rc=%d\n",rc)
281                         err_undo(preobj.DvDr);
282                         if ( rc < 0 || rc > 255)
283                                 rc = E_MKSPECIAL;
284                         err_exit(rc);
285                 }
286
287                 DEBUG_0("cfgricp: Returned from make_special_files()\n")
288
289                 /* build the DDS  */
290                 DEBUG_0("cfgricp: Calling build_dds()\n")
291                 rc = build_dds(logical_name, &cfg.ddsptr, &cfg.ddslen);
292                 if (rc) {
293                         /* error building dds */
294                         DEBUG_1("cfgricp: error building dds, rc=%d\n",rc)
295                         err_undo(preobj.DvDr);
296                         if ( rc < 0 || rc > 255)
297                                 rc = E_DDS;
298                         err_exit(rc);
299                 }
300
301                 DEBUG_0("cfgricp: Returned from build_dds()\n")
302
303                 /* call sysconfig to pass DDS to driver */
304                 DEBUG_0("cfgricp: Pass DDS to driver via sysconfig()\n")
305                 cfg.cmd = CFG_INIT;
306                 if (sysconfig(SYS_CFGDD, &cfg, sizeof(struct cfg_dd )) == -1) {
307                         /* error configuring device */
308                         DEBUG_0("cfgricp: error configuring device\n")
309                         err_undo(preobj.DvDr);
310                         err_exit(E_CFGINIT);
311                 }
312
313                 /* download microcode if necessary */
314                 DEBUG_0("cfgdevice: Calling download_microcode()\n")
315                 rc = download_microcode(logical_name);
316                 if (rc) {
317                         /* error downloading microcode */
318                         DEBUG_1("cfgdevice: error downloading microcode, rc=%d\n",rc)
319                         err_undo2(cfg.devno);
320                         err_undo(preobj.DvDr);
321                         if ( rc < 0 || rc > 255)
322                                 rc = E_UCODE;
323                         err_exit(rc);
324                 }
325                 DEBUG_0("cfgdevice: Returned from download_microcode()\n")
326
327                 /* update customized device object with a change operation */
328                 cusobj.status = AVAILABLE;
329                 if (odm_change_obj(cusdev, &cusobj) == -1) {
330                         /* ODM failure */
331                         DEBUG_0("cfgricp: ODM failure updating CuDv object\n");
332                         err_exit(E_ODMUPDATE);
333                 }
334         } /* end if (device is not AVAILABLE) then ... */
335
336         /* close customized device object class */
337         if (odm_close_class(cusdev) == -1) {
338                 DEBUG_0("cfgricp: error closing CuDv object class\n");
339                 err_exit(E_ODMCLOSE);
340         }
341         odm_terminate();
342         exit(0);
343 }
```

```
344    /*
345     * NAME: err_exit
346     *
347     * FUNCTION: Closes any open object classes and terminates ODM.  Used to
348     *           back out on an error.
349     *
350     * NOTES:
351     *
352     *   err_exit( exitcode )
353     *      exitcode = The error exit code.
354     *
355     * RETURNS:
356     *               None
357     */
358
359    err_exit(exitcode)
360    char    exitcode;
361    {
362            /* Close any open object class */
363            odm_close_class(CuDv_CLASS);
364            odm_close_class(PdDv_CLASS);
365            odm_close_class(CuAt_CLASS);
366
367            /* Terminate the ODM */
368            odm_terminate();
369            exit(exitcode);
370    }
```

```
371     /*
372      * NAME: err_undo
373      *
374      * FUNCTION: Unloads the device's device driver.  Used to back out on an
375      *           error.
376      *
377      *   err_undo( DvDr )
378      *      DvDr = pointer to device driver name.
379      *
380      * RETURNS:
381      *               None
382      */
383
384     err_undo(DvDr)
385     char    *DvDr;                  /* pointer to device driver name */
386     {
387             /* unload driver */
388             if (loadext(DvDr,FALSE,FALSE) == NULL) {
389                     DEBUG_0("cfgricp: error unloading driver\n");
390             }
391             return;
392     }
```

```
393   /*
394    * NAME: err_undo2
395    *
396    * FUNCTION: Terminates the device.  Used to back out on an error.
397    *
398    *
399    *   err_undo2( devno )
400    *       devno = The device's devno.
401    *
402    * RETURNS:
403    *              None
404    */
405
406   err_undo2(devno)
407   dev_t   devno;                          /* The device's devno */
408   {
409           struct  cfg_dd cfg;             /* sysconfig command structure */
410
411           /* terminate device */
412           cfg.devno = devno;
413           cfg.kmid = (mid_t)0;
414           cfg.ddsptr = (caddr_t) NULL;
415           cfg.ddslen = (int)0;
416           cfg.cmd = CFG_TERM;
417
418           if (sysconfig(SYS_CFGDD,&cfg,sizeof(struct cfg_dd)) == -1) {
419                   DEBUG_0("cfgdevice: error unconfiguring device\n");
420           }
421           return;
422   }
```

```
423    /*
424     * NAME: build_dds
425     *
426     * FUNCTION: Builds the DDS (Defined Data Structure) for the
427     *           Realtime Interface Co-Processor Portmaster adapter
428     *
429     * RETURNS: 0 - success
430     *         >0 - failure
431     *
432     */
433
434    int  build_dds( lognam, addr, len )
435    char  *lognam;                         /* logical name of device */
436    uchar **addr;                          /* receiving pointer for DDS address */
437    int  *len;                             /* receiving variable for DDS length */
438    {
439           int     rc,                     /* return code                    */
440                   result;                 /* work variable                  */
441           long    objtest = 0;            /* existance test, 1-test, 0-don't */
442           char    sstring[512];           /* working string                 */
443           struct  CuDv    cudvport,
444                           cudvadap;       /* object class records           */
445           t_ric_dds      *dds;            /* pointer to DDS structure       */
446
447           dds = (t_ric_dds *) malloc( sizeof(t_ric_dds) );
448           if (dds == NULL)
449                   return(E_MALLOC);       /* report allocation error */
450
451           /* Driver requires dds be cleared: */
452           memset( dds, 0, sizeof(t_ric_dds) );
453
454           sprintf( sstring, "name = '%s'", lognam );
455           if(( rc = odm_get_obj( CuDv_CLASS, sstring, &cudvport, ODM_FIRST ))==0)
456                   return( E_NOCuDv );
457           else if ( rc == -1 )
458                   return( E_ODMGET );
459
460           /* fill in resource name fields */
461           strcpy(dds->dds_vpd.devname,lognam);
462           strcpy(dds->dds_vpd.adpt_name,cudvport.parent);
463
464
465           /* scan field to extract port number */
466           if (sscanf( cudvport.connwhere, "%d", &result ) != 1 )
467                   return(E_INVCONNECT);
468
469           dds->dds_dvc.port_num = result;
470
471           DEBUG_1("PORT NUMBER IS %d\n", dds->dds_dvc.port_num )
472
473           /* read the parent adapter object */
474           sprintf( sstring, "name = '%s'", cudvport.parent );
475           if(( rc = odm_get_obj( CuDv_CLASS, sstring, &cudvadap, ODM_FIRST ))==0)
476                   return( E_NOCuDv );
477           else if ( rc == -1 )
478                   return( E_ODMGET );
479
480           /* scan field to extract slot number */
481           if (sscanf( cudvadap.connwhere, "%d", &dds->dds_hdw.slot_num ) != 1 )
482                   return(E_INVCONNECT);
483           dds->dds_hdw.slot_num--; /* connwhere = 1,2,.. slot_num = 0,1,.. */
484
485           DEBUG_1( "Slot number is %d after decrement\n", dds->dds_hdw.slot_num )
486
487           /* the following values should be obtained from the PdAt and CuAt */
488           /* object classes, these are just the default values...         */
489           dds->dds_hdw.bus_mem_addr = 0x10000;
490           dds->dds_hdw.tcw_bus_mem_addr = 0x100000;
491           dds->dds_hdw.bus_intr_lvl = 3;
492           dds->dds_hdw.bus_io_addr = 0x2a0;
493           dds->dds_hdw.dma_lvl = 0;
494           dds->dds_dvc.rdto = 92;
495           dds->dds_hdw.intr_priority = 3;
496
497           *addr = (caddr_t)(dds);         /* output the DDS address and length */
```

```
498         *len  = sizeof(t_ric_dds);
499
500         return(E_OK);
501     }
```

```
502   /*
503    * NAME: generate_minor
504    *
505    * FUNCTION: Routine for generating the device minor number
506    *
507    * RETURNS:
508    *   minor number success
509    *   E_MINORNO on failure
510    */
511
512   int
513   generate_minor(lname, majno, minorno)
514   char    *lname;     /* logical device name */
515   long    majno;      /* device major number */
516   long    *minorno;   /* device minor number */
517   {
518           long    *minorptr;
519
520           /*
521            * use genminor() to create and reserve the minor
522            * numbers used by this device.
523            */
524
525           minorptr = genminor(lname, majno, -1, 1, 1, 1);
526
527           if ( minorptr == (long *)NULL )
528                   /* error generating minor number */
529                   return(E_MINORNO);
530
531           *minorno = *minorptr;
532           return(E_OK);
533   }
```

```
534  /*
535   *
536   * NAME: download_microcode
537   *
538   * FUNCTION: To download the micro code for the RIC adapter.
539   *
540   * NOTES: If this is the FIRST port configured on an adapter, the microcode
541   *        file named by the "microcode_file" attribute of the CuAt object
542   *        class will be opened, read into a buffer and then ioctl'd down to
543   *        the adapter through the current port.
544   *
545   * RETURNS: 0 - success
546   *         >0 - failure
547   *
548   */
549
550  int     download_microcode( lognam )
551  char    *lognam;
552  {
553          char        port_fn[PATH_MAX]; /* port file name */
554          char        *mcode_fn;         /* microcode file name */
555          char        sstring[512];      /* working string */
556          uchar       *buffer;           /* microcode buffer */
557          int         rc;                /* return code */
558          long        length;            /* microcode length */
559          long        objtest = 0;       /* existance test, 1-test, 0-don't */
560          int         port_fd = 0,       /* port file descriptor */
561                      mcode_fd = 0;      /* uCode file descriptor */
562          struct CuDv cudvport,          /* CuDv record for port */
563                      cudvsib;           /* CuDv record for adapter */
564          struct CuAt *cuatptr;
565          t_rw_cmd    iocb;              /* communications block for ioctl */
566          int         how_many;
567
568          /* read CuDv record for current port */
569          sprintf( sstring, "name = '%s'", lognam );
570          if(( rc = odm_get_obj( CuDv_CLASS, sstring, &cudvport, ODM_FIRST ))==0)
571                  return( E_NOCuDv );
572          else if ( rc == -1 )
573                  return( E_ODMGET );
574          /* if 1 or more sibling ports exist, then the microcode for the */
575          /* adapter has already been loaded */
576          sprintf( sstring, "parent = '%s' AND status = %d", cudvport.parent,
577                  AVAILABLE );
578          if(( rc = odm_get_obj( CuDv_CLASS, sstring, &cudvsib, ODM_FIRST ))==-1)
579                  return( E_ODMGET );
580          else if ( rc != 0 )
581                  return 0;
582          objtest = 0;
583
584          cuatptr = getattr( cudvport.parent, "ucode", FALSE, &how_many );
585
586          if( cuatptr == ( struct CuAt * )NULL )
587                  return E_NOATTR;
588          if(( mcode_fd = open( cuatptr->value, O_RDONLY ) ) == -1 )
589                  return E_NOUCODE;
590          if((length = lseek( mcode_fd, 0L, 2 ))==-1)
591                  return E_NOUCODE;
592
593          DEBUG_1("Microcode length is %d\n", length )
594          if(lseek( mcode_fd, 0L, 0 )==-1)
595                  return E_NOUCODE;
596          if((buffer = malloc( length ))==NULL)
597                  return E_MALLOC;
598          if(read( mcode_fd, buffer, length )==-1)
599                  return E_NOUCODE;
600          if(close( mcode_fd )==-1)
601                  return E_NOUCODE;
602          sprintf( port_fn, "/dev/%s", lognam );
603          DEBUG_1("OPENING %s\n", port_fn )
604          if((port_fd = open( port_fn, O_RDWR )) == -1 )
605          {
606                  DEBUG_0("OPEN FAILED\n")
607                  return E_DEVACCESS;
608          }
```

```
609         iocb.length  = length;          /* set control block for ioctl */
610         iocb.mem_off = 0x10100;
611         iocb.usr_buf = buffer;
612         /* call ioctl to download micro code */
613         if( ioctl( port_fd, Q_RASW, &iocb ) <0 )
614                 return E_UCODE;
615         free( buffer );                 /* free the buffer area */
616         return 0;
617     }
```

```
618   /*
619    * NAME: make_special_file
620    *
621    * FUNCTION: Creates, or alters a special file as required
622    *
623    *
624    * NOTES:
625    *       make_special_file(suffix,devno)
626    *
627    *       suffix  = suffix part of the special file name.  For most cases
628    *               this will be the device logical name.  For devices with
629    *               more than one special file, this routine will be called
630    *               one time for each special file needed, passing the file
631    *               name required for the special file.
632    *
633    *       If the special file already exists, then the major/minor numbers
634    *       are checked. If they are incorrect, the old file is deleted, and
635    *       a new one created. If the numbers were correct, no action is
636    *       taken, and 0 is returned.
637    *
638    * RETURNS: 0 For success, errno for failure.
639    */
640   extern  int     errno;
641
642   int make_special_files(suffix,devno)
643   char        *suffix;                  /* suffix for special file name */
644   dev_t        devno;                   /* major and minor numbers */
645   {
646           long        cflags;           /* create flag / mode & type indicator */
647           struct  stat    buf;
648           char    spfilename[128];
649           int     rc;
650           long    filetype;             /* character or block device */
651
652           cflags =  S_IRUSR | S_IFCHR   /* set type for char special file */
653                   | S_IRGRP | S_IROTH   /* set mode for rw------- permisions */
654                   | S_IWUSR | S_IFMPX
655                   | S_IWGRP | S_IWOTH;
656
657           filetype=cflags&(S_IFBLK|S_IFCHR);
658
659           if(devno<0 | *suffix=='\0' | !filetype)
660                   return(E_MKSPECIAL);   /* error in parameters */
661
662           sprintf(spfilename,"/dev/%s",suffix);   /* file name =/dev/[suffix] */
663
664
665           if(stat(spfilename,&buf)) {
666
667                   /* stat failed, check that reason is ok */
668
669                   if( errno != ENOENT ) {
670                           DEBUG_0("stat failed\n")
671                           return(E_MKSPECIAL);
672                   }
673
674                   /* file does not exist, so make it */
675
676                   if(mknod(spfilename,filetype,devno)) {
677                           DEBUG_0("mknod failed\n")
678                           return(E_MKSPECIAL);
679                   }
680
681           } else {
682
683                   /* stat succeeded, so file already exists */
684
685                   if(buf.st_rdev==devno)          /* major/minor #s are same, */
686                           return(0);              /* leave special file alone */
687                   if(unlink(spfilename)) {        /* unlink special file name */
688                           DEBUG_0("unlink failed\n")
689                           return(E_MKSPECIAL);
690                   }
691                   if(mknod(spfilename,filetype,devno)) {
692                           /* create special file */
```

```
693                             DEBUG_0("mknod failed\n")
694                             return(E_MKSPECIAL);
695                     }
696             }
697
698             /* change mode of special file.  This is not in the same step as   */
698             /* creating the special file because of an error in mknod().       */
700             if(chmod(spfilename, (cflags&( filetype)))) {
701                     DEBUG_0("chmod failed\n")
702                     return (E_MKSPECIAL);
703             }
704
705             return (0);
706     }
```

B.5.5 Header File for Configuration Methods (debug.h)

```
1     /*
2      * DEBUGGING AIDS
3      */
4
5     #ifdef DEBUG
6     #include <stdio.h>
7     #define DEBUG_0(A)              {fprintf(stderr,A);fflush(stderr);}
8     #define DEBUG_1(A,B)            {fprintf(stderr,A,B);fflush(stderr);}
9     #define DEBUG_2(A,B,C)          {fprintf(stderr,A,B,C);fflush(stderr);}
10    #define DEBUG_3(A,B,C,D)        {fprintf(stderr,A,B,C,D);fflush(stderr);}
11    #define DEBUG_4(A,B,C,D,E)      {fprintf(stderr,A,B,C,D,E);fflush(stderr);}
12    #define DEBUG_5(A,B,C,D,E,F)    {fprintf(stderr,A,B,C,D,E,F);fflush(stderr);}
13    #define DEBUG_6(A,B,C,D,E,F,G)  {fprintf(stderr,A,B,C,D,E,F,G);fflush(stderr);}
14    #define DEBUG_7(A)              {printf(A);}
15    #define DEBUG_8(A,B)            {printf(A,B);}
16    #define DEBUG_9(A,B,C)          {printf(A,B,C);}
17    #define DEBUGELSE               else
18    #else
19    #define DEBUG_0(A)
20    #define DEBUG_1(A,B)
21    #define DEBUG_2(A,B,C)
22    #define DEBUG_3(A,B,C,D)
23    #define DEBUG_4(A,B,C,D,E)
24    #define DEBUG_5(A,B,C,D,E,F)
25    #define DEBUG_6(A,B,C,D,E,F,G)
26    #define DEBUG_7(A)
27    #define DEBUG_8(A,B)
28    #define DEBUG_9(A,B,C)
29    #define DEBUGELSE
30    #endif
```

B.5.6 Makefile for Configuration Methods and Message Catalog

```
#
# Makefile for cfgrica and cfgricp
#

CC= cc
CFLAGS= -DDEBUG
LIBS= -lodm -lcfg
DEST=/etc/methods

all: cfgrica cfgricp ric.cat

ric.cat: ric.msg
	gencat ric.cat ric.msg

cfgrica: cfgrica.c
	$(CC) $(CFLAGS) -o cfgrica cfgrica.c $(LIBS)

cfgricp: cfgricp.c
	$(CC) $(CFLAGS) -o cfgricp cfgricp.c $(LIBS)

install: all
	mv cfgrica $(DEST)
	mv cfgricp $(DEST)
	cp ric.cat /usr/lpp/msg/C
	mv ric.cat /usr/lpp/msg/En_US

clean:
	rm -f *.o *.map core
	rm -f $(DEST)/cfgric*
	rm -f /usr/lpp/msg/*/ric.cat
```

Appendix C. SMIT

C.1 Object Classes

C.1.1 Menu Object Class (sm_menu_opt)

Each item on a menu is specified by an sm_menu_opt object. The displayed menu represents the set of objects that have the same value for id plus the sm_menu_opt object used for the title, which has a next_id value equal to the ID value of the other objects.

The descriptors for **sm_menu_opt** objects are [1] :

id The ID or name of the object. The value of id is a string with a maximum length of 64 characters. IDs should be unique both to your application and unique within the particular SMIT database used.

id_seq_num The position of this item in relation to other items on the menu. Non-title sm_menu_opt objects are sorted on this string field. The value of id_seq_num is a string with a maximum length of 16 characters.

next_id The fast path name of the next menu, if the value for the next_type descriptor of this object is "m" (menu). All non-alias sm_menu_opt objects with id values matching the value of next_id form the set of selections for that menu. The value of next_id is a string with a maximum length of 64 characters.

text The description of the task that is displayed as the menu item. The value of text is a string with a maximum length of 1024 characters. This string can be formatted with embedded \n (newline) characters.

text_msg_file The file name (not the full path name) that is the Message Facility catalog for the string, text. The value of text_msg_file is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility. Set to "" if you are not using the Message Facility.

text_msg_set The Message Facility set ID for the string, text. Set IDs can be used to indicate subsets of a single catalog. The value of text_msg_set is an integer. Set to 0 (zero) if you are not using the Message Facility.

text_msg_id The Message Facility ID for the string, text. The value of text_msg_id is an integer. Set to 0 (zero) if you are not using the Message Facility.

[1] Note: when coding an object in this object class, set unused empty strings to "" (two adjacent double quotation marks) and unused integer fields to 0 (zero).

© Copyright IBM Corp. 1991

next_type The type of the next object if this item is selected. Valid values are:

"m" Menu: the next object is a menu.

"d" Dialog: the next object is a dialog.

"n" Name: the next object is a selector.

"i" Info: this object is used to put blank or other separator lines in a menu, or to present a topic that does not lead to an executable task but about which help is provided via the help_msg_loc descriptor of this object.

alias Defines whether or not the value of the id descriptor for this menu object is an alias for another existing fast path specified in the next_id field of this object. The value of the alias descriptor must be "n" for a menu object.

help_msg_id For internal use only; set to "" (empty string).

help_msg_loc The file name sent as a parameter to the man command for retrieval of help text. The output of the man command is displayed by SMIT as the help message. The value of help_msg_loc is a string with a maximum length of 1024 characters. Set to "" (empty string) means no help is provided.

C.1.2 Menu Object Class (sm_menu_opt) Used for Aliases

A SMIT alias is specified by an sm_menu_opt object.

The descriptors for the **sm_menu_opt** object class and their settings to specify an alias are:

id The ID or name of the new or alias fast path. The value of id is a string with a maximum length of 64 characters. IDs should be unique to your application and unique within the SMIT database in which they are used.

id_seq_num Set to "" (empty string).

next_id Specifies the id_seq_num of the menu object pointed to by the alias. The value of next_id is a string with a maximum length of 64 characters.

text Set to "" (empty string).

text_msg_file Set to "" (empty string).

text_msg_set Set to 0 (zero).

text_msg_id Set to 0 (zero).

next_type The fast path screen type. The value of next_type is a string. Valid values are:

"m" Menu: the next_id field specifies a menu screen fast path.

"d" Dialog: the next_id field specifies a dialog screen fast path.

"n" Name: the next_id field specifies a selector screen fastpath.

alias Defines this object as an alias fast path. The alias descriptor for an alias must be set to "y" (yes).

help_msg_id Set to "" (empty string).

help_msg_loc Set to "" (empty string).

C.1.3 Selector Header Object Class (sm_name_hdr)

A selector screen is specified by two objects: an **sm_name_hdr** object that specifies the screen title and other information, and an **sm_cmd_opt** object that specifies what type of data item is to be obtained.

The descriptors for the **sm_name_hdr** object class are [2] :

id The ID or name of the object. The id field can be externalized as a fast path ID unless has_name_select is set to "y" (yes). The value of id is a string with a maximum length of 64 characters. IDs should be unique to your application and unique within your system.

next_id Specifies the header object for the subsequent screen; set to the value of the id field of the sm_cmd_hdr object or the sm_name_hdr object that follows this selector. The next_type field described below specifies which object class is indicated. The value of next_id is a string with a maximum length of 64 characters.

option_id Specifies the body of this selector; set to the id field of the sm_cmd_opt object. The value of option_id is a string with a maximum length of 64 characters.

has_name_select
Specifies whether this screen must be preceded by a selector screen. Valid values are:

- **"" or "n"** No; this is the default case. The ID of this object can be used as a fast path, even if preceded by a selector screen.
- **"y"** Yes; a selector must precede this object. This setting prevents the ID of this object from being used as a fast path to the corresponding dialog screen.

name The text displayed as the title of the selector screen. The value of name is a string with a maximum length of 1024 characters. The string can be formatted with embedded \n (newline) characters.

name_msg_file
The file name (not the full path name) that is the Message Facility catalog for the string name. The value of name_msg_file is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility.

name_msg_set
The Message Facility set ID for the string name. Set IDs can be used to indicate subsets of a single catalog. The value of name_msg_set is an integer.

---

[2] Note: when coding an object in this object class, set unused empty strings to "" (two adjacent double quotation marks) and unused integer fields to 0 (zero).

name_msg_id
The Message Facility ID for the string name. The value of name_msg_id is an integer.

type
The method to be used to process the selector. The value of type is a string with a maximum length of one character. Valid values are:

"" or "j" Just next ID: the object following this object is always the object specified by the value of the next_id descriptor. The next_id descriptor is a fully-defined string initialized at development time.

"r" Cat raw name: in this case, the next_id descriptor is defined partially at development time and partially at run time by user input. The value of the next_id descriptor defined at development time is concatenated with the value selected by the user to create the ID value to search for next (that of the dialog or selector to display).

"c" Cat cooked name: the value selected by the user requires processing for more information. This value is passed to the command named in the cmd_to_classify descriptor, and then output from the command is concatenated with the value of the next_id descriptor to create the ID descriptor to search for next (that of the dialog or selector to display).

ghost
Specifies whether to display this selector screen or only the list pop-up panel produced by the command in the cmd_to_list field. The value of ghost is a string. Valid values are:

"" or "n" No; display this selector screen.

"y" Yes; display only the pop-up panel produced by the command string constructed using the cmd_to_list and cmd_to_list_postfix fields in the associated sm_cmd_opt object. If there is no cmd_to_list, then SMIT assumes this is a "super ghost" (nothing is displayed), runs the cmd_to_classify command and proceeds.

cmd_to_classify
The command string to be used, if needed, to classify the value of the name field of the sm_cmd_opt object associated with this selector. The value of cmd_to_classify is a string with a maximum length of 1024 characters. The input to the cmd_to_classify taken from the entry field is called the "raw name" and the output of the cmd_to_classify is called the "cooked name".

cmd_to_classify_postfix
The postfix to interpret and add to the command string in the cmd_to_classify field. The value of cmd_to_classify_postfix is a string with a maximum length of 1024 characters.

raw_field_name
The alternate name for the raw value. The value of raw_field_name is a string with a maximum length of 1024 characters. The default value is "_rawname".

cooked_field_name
The alternate name for the cooked value. The value of cooked_field_name is a string with a maximum length of 1024 characters. The default value is "_cookedname".

next_type The type of screen that follows this selector. Valid values are:

"n" Name: a selector screen follows. See the description of next_id above for related information.

"d" Dialog: a dialog screen follows. See the description of next_id above for related information.

help_msg_id
For internal use only; set to "" (empty string).

help_msg_loc
The file name sent as a parameter to the man command for retrieval of help text. The value of help_msg_loc is a string with a maximum length of 1024 characters. The output of the man command is displayed by SMIT as the help message. Set to "" (empty string) if no help is provided.

C.1.4 Dialog Header Object Class (sm_cmd_hdr)

A dialog header object is an **sm_cmd_hdr** object. A dialog header object is required for each dialog, and points to the dialog command option objects associated with the dialog.

The descriptors for the **sm_cmd_hdr** object class are [3] :

id The ID or name of the object. The value of id is a string with a maximum length of 64 characters. The id field can be used as a fast path ID unless there is a selector associated with the dialog. IDs should be unique to your application and unique within your system.

option_id The ID of the sm_cmd_opt objects (the dialog fields) to which this header refers. The value of option_id is a string with a maximum length of 64 characters.

has_name_select
Specifies whether this screen must be preceded by a selector screen or a menu screen. Valid values are:

"" or "n" No; this is the default case.

"y" Yes; a selector precedes this object. This setting prevents the ID of this object from being used as a fast path to the corresponding dialog screen.

name The text displayed as the title of the dialog screen. The value of name is a string with a maximum length of 1024 characters. The text describes the task performed by the dialog. The string can be formatted with embedded \n (newline) characters.

[3] Note: when coding an object in this object class, set unused empty strings to "" (two adjacent double quotation marks) and unused integer fields to 0 (zero).

name_msg_file
The file name (not the full path name) that is the Message Facility catalog for the string, name. The value of name_msg_file is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility.

name_msg_set
The Message Facility set ID for the string, name. Set IDs can be used to indicate subsets of a single catalog. The value of name_msg_set is an integer.

name_msg_id
The Message Facility ID for the string, name. Message IDs can be created by the message extractor tools owned by the Message Facility. The value of name_msg_id is an integer.

cmd_to_exec
The initial part of the command string, which can be the command or the command and any fixed options that execute the task of the dialog. Other options are automatically appended through user interaction with the command option objects (sm_cmd_opt) associated with the dialog screen. The value of cmd_to_exec is a string with a maximum length of 1024 characters.

ask
Defines whether or not the user is prompted to reconsider the choice to execute the task. Valid values are:

"" or "n" No; the user is not prompted for confirmation; the task is performed when the dialog is committed. This is the default setting for the ask descriptor.

"y" Yes; the user is prompted to confirm that the task be performed; the task is performed only after user confirmation. Prompting the user for execution confirmation is especially useful prior to performance of deletion tasks, where the deleted resource is either difficult or impossible to recover, or when there is no displayable dialog associated with the task (when the ghost field is set to "y").

exec_mode
Defines the handling of stdin, stdout, and stderr during task execution. The value of exec_mode is a string. Valid values are:

"" or "p" Pipe mode: the default setting for the exec_mode descriptor. The command executes with stdout and stderr redirected through pipes to SMIT. SMIT manages output from the command. The output is saved and is scrollable by the user after the task finishes running. While the task runs, output is scrolled as needed.

"n" No scroll pipe mode: works like the "p" mode, except that the output is not scrolled while the task runs. The first screen of output will be shown as it is generated and then remains there while the task runs. The output is saved and is scrollable by the user after the task finishes running.

"i" Inherit mode: the command executes with stdin, stdout, and stderr inherited by the child process in which the task runs. This gives input and output control to the executed command.

"e" Exit/exec mode: causes SMIT to run (do an execl subroutine call on) the specified command string in the current process, which effectively terminates SMIT.

ghost Indicates if the normally displayed dialog should not be shown. The value of ghost is a string. Valid values are:

"" or "n" No; the dialog associated with the task is displayed. This is the default setting.

"y" Yes; the dialog associated with the task is not displayed because no further information is required from the user. The command specified in the cmd_to_exec descriptor is executed as soon as the user selects the task.

cmd_to_discover
The command string used to discover the default or current values of the object being manipulated. The value of cmd_to_discover is a string with a maximum length of 1024 characters. The command is executed before the dialog is displayed, and its output is retrieved. Output of the command must be in colon format.

cmd_to_discover_postfix
The postfix to interpret and add to the command string in the cmd_to_discover field. The value of cmd_to_discover_postfix is a string with a maximum length of 1024 characters.

help_msg_id
For internal use only; set to "" (empty string).

help_msg_loc
The file name sent as a parameter to the AIX man command for retrieval of help text. The value of help_msg_loc is a string with a maximum length of 1024 characters. The output of the man command is displayed by SMIT as the help message. Set to "" (empty string) if no help is availabe.

C.1.5 Dialog/Selector Command Option Object Class (sm_cmd_opt)

Each object in a dialog except the dialog header object normally corresponds to a flag, option, or attribute of the command that the dialog performs. One or more of these objects is created for each SMIT dialog; a ghost dialog can have no associated dialog command option objects.

Each selector screen is composed of one selector header object and one selector command option object.

The dialog command option object and the selector command option object are both **sm_cmd_opt** objects. The descriptors for the **sm_cmd_opt** object class and their functions are [4] :

[4] Note: when coding an object in this object class, set unused empty strings to "" (two adjacent double quotation marks) and unused integer fields to 0 (zero).

id The ID or name of the object. The ID of the associated dialog or selector header object can be used as a fast path to this and other dialog objects in the dialog. The value of id is a string with a maximum length of 64 characters. All dialog objects that appear in one dialog must have the same ID. Also, IDs should be unique to your application and unique within the particular SMIT database used.

id_seq_num
The position of this item in relation to other items on the dialog; sm_cmd_opt objects in a dialog are sorted on this string field. The value of id_seq_num is a string with a maximum length of 16 characters. When this object is part of a dialog screen, the string "0" is not a valid value for this field. When this object is part of a selector screen, the id_seq_num descriptor must be set to "0" (zero).

disc_field_name
A string that should match one of the name fields in the output of the cmd_to_discover command in the associated dialog header. The value of disc_field_name is a string with a maximum length of 64 characters.

The value of the disc_field_name descriptor can be defined using the raw or cooked name from a preceding selector instead of the cmd_to_discover command in the associated header object. If the descriptor is defined with input from a preceding selector, it must be set to either "_rawname" or "_cookedname", or to the corresponding sm_name_hdr.cooked_field_name value or sm_name_hdr.raw_field_name value if this was used to redefine the default name.

name The string that appears on the dialog or selector screen as the field name. It is the visual questioning or prompting part of the object, a natural language description of a flag, option or parameter of the command specified in the cmd_to_exec field of the associated dialog header object. The value of name is a string with a maximum length of 1024 characters.

name_msg_file
The file name (not the full path name) that is the Message Facility catalog for the string, name. The value of name_msg_file is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility. Set to "" (empty string) if not used.

name_msg_set
The Message Facility set ID for the string, name. The value of name_msg_set is an integer. Set to 0 (zero) if not used.

name_msg_id
The Message Facility message ID for the string, name. The value of name_msg_id is an integer. Set to 0 (zero) if not used.

op_type The type of auxiliary operation supported for this field. The value of op_type is a string. Valid values are:

"" or "n" This is the default case. No auxiliary operations (list or ring selection) are supported for this field.

"l" List selection operation provided. A pop-up window displays a list of items produced by running the command in the cmd_to_list field of this object when the user selects the F4=List function of the SMIT interface.

"r" Ring selection operation provided. The string in the disp_values or aix_values field is interpreted as a comma-delimited set of valid entries. The user can tab or backtab through these values to make a selection. Also, the F4=List interface function can be used in this case, since SMIT will transform the ring into a list as needed.

The values "N", "L", and "R" can be used as op_type values just as the lowercase values "n", "l", and "r". However, with the uppercase values, if the cmd_to_exec command is run and returns with an exit status of 0 (zero), then the corresponding entry field will be cleared to an empty string.

entry_type
The type of value required by the entry field. The value of entry_type is a string. Valid values are:

"" or "n" No entry: the current value cannot be modified via direct type-in. The field is informational only.

"t" Text entry: alphanumeric input can be entered.

"#" Numeric entry: numeric input only can be entered.

"x" Hex entry: hexidecimal input only can be entered.

"f" File entry: a file name only should be entered. SMIT checks to make sure the file exists and is accessable by the user. SMIT will not allow the user to run the task until this condition is satisified.

"r" Alphanumeric input can be entered. Leading and trailing spaces are considered significant and are not stripped off the field.

entry_size
Limits the number of characters the user can type in the entry field. The value of entry_size is an integer. A value of 0 defaults to the maximum allowed value size.

required Defines if a command field must be sent to the cmd_to_exec command defined in the associated dialog header object. The value of required is a string. If the object is part of a selector screen, the required field should normally be set to "" (empty string). If the object is part of a dialog screen, valid values are:

"" or "n" No; the option is added to the command string in the cmd_to_exec field only if the user changes the initially-displayed value. This is the default case.

"y" Yes; the value of the prefix field and the value of the entry field are always sent to the cmd_to_exec command.

"+" The command field is always sent to the cmd_to_exec command and must contain at least one non-blank character. SMIT will not allow the user to run the task until this condition is satisified.

prefix In the simplest case, it defines the flag to send with the entry field value to the cmd_to_exec command defined in the associated dialog header object. The value of prefix is a string with a maximum length of 1024 characters. The use of this field depends on the setting of the required field, the contents of the prefix field, and the contents of the associated entry field.

cmd_to_list_mode
Defines how much of an item from a list should be used. The list is produced by the command specified in this object's cmd_to_list field. The value of cmd_to_list_mode is a string with a maximum length of one character. Valid values are:

"" or "a" Get all fields. This is the default case.

"1" Get the first field.

"2" Get the second field.

"r" Range: running the command string in the cmd_to_list field returns a range (such as 1..99) instead of a list. Ranges are for information only; they are displayed in a list pop-up, but do not change the associated entry field.

cmd_to_list
The command string used to get a list of valid values for the value field. The value of cmd_to_list is a string with a maximum length of 1024 characters. This command should output values that are separated by \n (newline) characters.

cmd_to_list_postfix
The postfix to interpret and add to the command string specified in the cmd_to_list field of the dialog object. The value of cmd_to_list_postfix is a string with a maximum length of 1024 characters. Each line should not exceed 70 characters. If the first line starts with "# " (pound sign, space), that entry will be made non-selectable. This is useful for column headings. Subsequent lines that start with a "#", optionally preceded by spaces, are treated as a comment and as a continuation of the preceding entry.

multi_select
Defines if the user can make multiple selections from a list of valid values produced by the command in the cmd_to_list field of the dialog object. The value of multi_select is a string. Valid values are:

"" No; a user can select only one value from a list. This is the default case.

"y" Yes; a user can select multiple values from the list or option ring. When the command is built, the option prefix is inserted once before the string of selected items.

"m" Yes; a user can select multiple items from the list or option ring. When the command is built, the option prefix is inserted before each selected item.

value_index
For an option ring, the zero-origin index to the array of disp_value fields. The value_index number indicates the value that is displayed as the default in the entry field to the user. The value of entry_size is an integer.

disp_values
The array of valid values in an option ring to be presented to the user. The value of disp_values is a string with a maximum length of 1024 characters. The field values are separated by , (commas) with no spaces preceding or following the commas.

values_msg_file
The file name (not the full path name) that is the Message Facility catalog for the values in the disp_values fields, if the values are initialized at development time. The value of values_msg_file is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility.

values_msg_set
The Message Facility set ID for the values in the disp_values fields. Set to 0 (zero) if not used.

values_msg_id
The Message Facility message ID for the values in the disp_values fields. Set to 0 (zero) if not used.

aix_values
If for an option ring, an array of AIX values specified so that each element corresponds to the element in the disp_values array in the same position; use if the natural language values in disp_values are not the actual options to be used for the command. The value of aix_values is a string with a maximum length of 1024 characters.

help_msg_id
For internal use only; set to "" (empty string).

help_msg_loc
The file name sent as a parameter to the AIX man command for retrieval of help text. The value of help_msg_loc is a string with a maximum length of 1024 characters. The output of the man command is displayed by SMIT as the help message. Set to "" (empty string) if no help is available.

C.2 Additional Information

C.2.1 Information Retrieval

SMIT can use several descriptors defined in its objects to get information, such as current run time values, that is required for continuation through the SMIT interface structure. Each of these descriptors is assigned some form of command string to run and retrieve the needed data.

The descriptors that can be set to a command for discovery of required information are:

- The **cmd_to_discover** descriptor that is part of the **sm_cmd_hdr** object class used to define a dialog header
- The **cmd_to_classify** descriptor that is part of the **sm_name_hdr** object class used to define a selector header
- The **cmd_to_list** descriptor that is part of the **sm_cmd_opt** object class used to define a selector option list associated with a selector or a dialog command option list associated with a dialog entry field.

SMIT executes a command string specified by a **cmd_to_list**, **cmd_to_classify**, or **cmd_to_discover** descriptor by first creating a child process. The *stderr* (standard error) and *stdout* (standard output) of the child process are redirected to SMIT via pipes. SMIT next executes a setenv("ENV=") subroutine in the child process to prevent commands specified in the $HOME/.env file of the user from being run automatically when a new shell is invoked. Finally, SMIT calls the execl system subroutine to start a new ksh shell, using the command string as the ksh -c parameter value. If the exit status is not 0 (zero), SMIT notifies the user that the command failed.

C.2.2 Default Values Setting

When SMIT puts up a dialog, it gets the dialog header (the **sm_cmd_hdr** object) and its associated dialog body (one or more **sm_cmd_opt** bjects) from the object repository. However, the **sm_cmd_opt** objects can also be initialized with current run time values. If the **sm_cmd_hdr**.*cmd_to_discover* field is not empty (""), SMIT runs the command specified in the field to obtain current run time values.

Any valid ksh command string can be used as a cmd_to_discover descriptor value. The command should generate the following output format as its stdout (standard output):

```
#name_1:name_2: ... :name_n\n
value_1:value_2: ... :value_n
```

In the stdout of a command, the first character is always a # (pound sign), a **\n** (newline character) is always present to separate the name line from the value line; multiple names and values are separated by : (colons), and any name or value can be an empty string (which in the output format appears as two colons with no space between them). SMIT maintains an internal current value set in this format that is used to pass name-value pairs from one screen to the next.

When SMIT runs a command specified in a **cmd_to_discover** field, it captures the stdout of the command and loads these name-value pairs (*name_1* and *value_1*, *name_2* and *value_2*, and so on) into the *disp_values* and *aix_values*

descriptors of the dialog command option (**sm_cmd_opt**) objects by matching each name to a **sm_cmd_opt**.*disc_field_name* descriptor in each **sm_cmd_opt** object.

For a dialog command option (**sm_cmd_opt**) object that displays a value from a preceding selector, the *disc_field_name* descriptor for the dialog command option object must be set to *"_rawname"* or *"_cookedname"* (or whatever alternate name was used to override the default name) to indicate which value to use. In this case, the *disc_field_name* descriptor of the dialog command option (**sm_cmd_opt**) object should normally be a no-entry field. If a particular value should always be passed to the command, the required descriptor for the dialog command option (**sm_cmd_opt**) object must be set to "y" (yes), or one of the other alternatives.

A special case of option ring field initialization permits the current value for a *cmd_to_discover* descriptor (i.e., any name-value pair from the current value set of a dialog) of a ring entry field to specify which pre-defined ring value to use as the default or initial value for the corresponding entry field. At dialog initialization time, when a dialog entry field matches a name in the current value set of the dialog (via **sm_cmd_opt**.*disc_field_name*), a check is made to determine if it is an option ring field (**sm_cmd_opt**.*op_type* = "r") and if it has predefined ring values (**sm_cmd_opt**.*aix_values* != ""). If so, this set of option ring values is compared with the current value for *disc_field_name* from the current value set. If a match is found, the matched option ring value becomes the default ring value (**sm_smd_opt**.*value_index* is set to its index). The corresponding translated value (**sm_cmd_opt**.*disp_values*), if available, is displayed. If no match is found, the error is reported and the current value becomes the default and only value for the ring.

In many cases, discovery commands already exist. In the devices and storage areas, the general paradigms of add, remove, change, and show exist. For example, to *add* (**mk**), a dialog is needed to solicit characteristics. The dialog can have as its discovery command the *show* (**ls**) command with a parameter that requests default values. SMIT uses the standard output of the *show* (**ls**) command to fill in the suggested defaults. However, for objects with default values that are constants known at development time (i.e., that are not based on the current state of a given machine), the defaults can be initialized in the dialog records themselves; in this case, no *cmd_to_discover* is needed. The dialog is then displayed. When all fields are filled in and the dialog is committed, the *add* (**mk**) command is executed.

As another example, a *change* (**ch**) dialog can have as its discovery command a *show* (**ls**) command to get current values for a given instance such as particular device. SMIT uses the standard output of the *show* (**ls**) command to fill in the values before displaying the dialog. The *show* (**ls**) command used for discovery in this instance can be the same as the one used for discovery in the *add* (**mk**) example, except with a slightly different set of options.

C.2.3 Flags and Parameters Setting

Associated with each occurrence of a *cmd_to_discover*, *cmd_to_classify*, or *cmd_to_list* descriptor is a second descriptor that defines the postfix for the command string defined by the *cmd_to_discover*, *cmd_to_classify*, or *cmd_to_list* descriptor. The postfix is a character string defining the flags and parameters that are appended to the command before it is executed.

The descriptors that can be used to define a postfix to be appended to a command are:

- The *cmd_to_discover_postfix* descriptor that defines the postfix for the *cmd_to_discover* descriptor in an **sm_cmd_hdr** object defining a dialog header.
- The *cmd_to_classify_postfix* descriptor that defines the postfix for the *cmd_to_classify* descriptor in an **sm_name_hdr** object defining a selector header.
- The *cmd_to_list_postfix* descriptor that defines the postfix for the *cmd_to_list* descriptor in an **sm_cmd_opt** object defining a selector entry field associated with a selector or an dialog entry field associated with a dialog.

The following is an example of how the postfix descriptors are used to specify parameter flags and values. The * (asterisk) in the example can be list, classify, or discover.

Assume that:

```
cmd_to_* equals "DEMO -a"
```

```
cmd_to_*_postfix equals "-l _rawname -n stuff -R _cookedname"
```

and the current value set is:

```
#name1:_rawname:_cookedname::stuff\n
value1:gigatronicundulator:parallel:xxx:47
```

Then the constructed command string would be:

```
DEMO -a -l 'gigatronicundulator' -n '47' -R 'parallel'
```

Surrounding ' ' (single quotation marks) can be added around postfix descriptor values to permit handling of parameter values with embedded spaces.

C.2.4 Aliases and Fast Paths

A SMIT **sm_menu_opt** object can be used to define a fast path that, when entered with the smit command to start SMIT, can get a user directly to a specific menu, selector, or dialog; the alias itself is never displayed. Use of a fast path allows a user to bypass the main SMIT menu and other objects in the SMIT interface path to that menu, selector, or dialog. Any number of fast paths can point to the same menu, selector, or dialog.

An **sm_menu_opt** object is used to define a fast path by setting the **sm_menu_opt**.*alias* field to "y". In this case, the **sm_menu_opt** object is used exclusively to define a fast path. The new fast path or alias name is specified by the value in the **sm_menu_opt**.*id* field. The contents of the **sm_menu_opt**.*next_id* field points to another menu object,i selector header object, or dialog header object, depending on whether the value of the **sm_menu_opt**.*next_type* field is "m" (menu), "n" (selector), or "d" (dialog).

C.3 Examples

C.3.1 ODM Stanzas for Ric Dialogs (sm_ric.add file)

```
#  sm_ric.add
#
# The seven first stanzas (sm_menu_opt) attach the Realtime Interface
# Co-Processor Portmaster Adapter dialogs to the
# "system management"/"devices"/"communication devices" menu
#
# The next stanzas are supporting the following SMIT functions :
#
#       List All Defined Ric Ports
#       Add a Ric Port
#       Move a Ric Port Definition to Another Port
#       Change/Show Characteristics of a Ric Port
#       Remove a Ric Port
#       Configure a Defined Ric Port
#
sm_menu_opt:
    id                        = "commodev"
    id_seq_num                = "070"
    next_id                   = "ric"
    text                      = "Realtime Interface Co-Processor Portmaster Adapter"
    text_msg_file             = ""
    text_msg_set              = 0
    text_msg_id               = 0
    next_type                 = "m"
    alias                     = ""
    help_msg_id               = ""
    help_msg_loc              = ""

sm_menu_opt:
    id                        = "ric"
    id_seq_num                = "010"
    next_id                   = "lsdric"
    text                      = "List All Defined Ric Ports"
    text_msg_file             = ""
    text_msg_set              = 0
    text_msg_id               = 0
    next_type                 = "d"
    alias                     = ""
    help_msg_id               = ""
    help_msg_loc              = ""

sm_menu_opt:
    id                        = "ric"
    id_seq_num                = "020"
    next_id                   = "makric"
    text                      = "Add a Ric Ports"
    text_msg_file             = ""
    text_msg_set              = 0
    text_msg_id               = 0
    next_type                 = "n"
    alias                     = ""
    help_msg_id               = ""
    help_msg_loc              = ""
```

```
sm_menu_opt:
    id                      = "ric"
    id_seq_num              = "030"
    next_id                 = "movric"
    text                    = "Move a Ric Port Definition to Another Port"
    text_msg_file           = ""
    text_msg_set            = 0
    text_msg_id             = 0
    next_type               = "n"
    alias                   = ""
    help_msg_id             = ""
    help_msg_loc            = ""

sm_menu_opt:
    id                      = "ric"
    id_seq_num              = "040"
    next_id                 = "chgric"
    text                    = "Change/Show Characteristics of a Ric Port"
    text_msg_file           = ""
    text_msg_set            = 0
    text_msg_id             = 0
    next_type               = "n"
    alias                   = ""
    help_msg_id             = ""
    help_msg_loc            = ""

sm_menu_opt:
    id                      = "ric"
    id_seq_num              = "050"
    next_id                 = "rmvric"
    text                    = "Remove a Ric Port"
    text_msg_file           = ""
    text_msg_set            = 0
    text_msg_id             = 0
    next_type               = "n"
    alias                   = ""
    help_msg_id             = ""
    help_msg_loc            = ""

sm_menu_opt:
    id                      = "ric"
    id_seq_num              = "060"
    next_id                 = "cfgric"
    text                    = "Configure a Defined Ric Port"
    text_msg_file           = ""
    text_msg_set            = 0
    text_msg_id             = 0
    next_type               = "n"
    alias                   = ""
    help_msg_id             = ""
    help_msg_loc            = ""
```

```
#
# List All Defined Ric Ports
# This uses a ghost dialogue to list all the defined Ric ports.
# The lsdev command is executed from this dialogue.
#

sm_cmd_hdr:
    id                          = "lsdric"
    option_id                   = ""
    has_name_select             = "n"
    name                        = "List All Defined Ric Ports"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 0
    cmd_to_exec                 = "lsdev -C -c ricport -H"
    ask                         = ""
    exec_mode                   = ""
    ghost                       = "y"
    cmd_to_discover             = ""
    cmd_to_discover_postfix     = ""
    name_size                   = 0
    value_size                  = 0
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
#
# Add a Ric Port
# This allows a ric port to be added by defining and configuring
# it.  There is one name selector followed by the dialogue.  The name
# selector is used to put up a list of defined ric adapters
# for the user to select from.  The dialogue then puts up a list of the
# user configurable attributes.

# Select parent adapter
sm_name_hdr:
    id                          = "makric"
    next_id                     = "makric_hdr"
    option_id                   = "ric_mk_parent"
    has_name_select             = "n"
    name                        = "Add a Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 1
    type                        = ""
    ghost                       = "y"
    cmd_to_classify             = ""
    cmd_to_classify_postfix     = ""
    raw_field_name              = "parent"
    cooked_field_name           = ""
    next_type                   = "d"
    help_msg_id                 = ""
    help_msg_loc                = ""

# Name selector command option for parent adapter
sm_cmd_opt:
    id                          = "ric_mk_parent"
    id_seq_num                  = "0"
    disc_field_name             = ""
    name                        = "Parent Adapter"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 8
    op_type                     = "l"
    entry_type                  = "t"
    entry_size                  = 0
    required                    = "y"
    prefix                      = ""
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "lsparent -C -k ricp"
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""

# The dialogue header.
sm_cmd_hdr:
    id                          = "makric_hdr"
    option_id                   = "ric_add,ric_common"
```

```
has_name_select           = "y"
name                      = "Add a Ric Port"
name_msg_file             = "ric.cat"
name_msg_set              = 3
name_msg_id               = 1
cmd_to_exec               = "mkdev -c ricport -s ricp -t port "
ask                       = ""
exec_mode                 = ""
ghost                     = "n"
cmd_to_discover           = "lsattr -c ricport -s ricp -t port -D -O"
cmd_to_discover_postfix   = ""
name_size                 = 0
value_size                = 0
help_msg_id               = ""
help_msg_loc              = ""
```

```
#
# Move a Ric Port Definition to Another Port
# This allows a ric port definition to be moved to another port
# or another adapter.  There are two name selectors followed by the
# dialogue.  The first name selector is used to put up a list of defined
# ric ports for the user to select from.  The second name selector
# puts up a list of defined ric adapters which can have the
# selected port definition moved to.  The dialogue then allows for a new
# port to be selected.

# Select ric port by logical name
sm_name_hdr:
    id                          = "movric"
    next_id                     = "movric_parent"
    option_id                   = "ric_ln_opt"
    has_name_select             = "n"
    name                        = "Move a Ric Port Definition to Another Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 2
    type                        = ""
    ghost                       = "y"
    cmd_to_classify             = ""
    cmd_to_classify_postfix     = ""
    raw_field_name              = "logicname"
    cooked_field_name           = ""
    next_type                   = "n"
    help_msg_id                 = ""
    help_msg_loc                = ""

# Select parent adapter
sm_name_hdr:
    id                          = "movric_parent"
    next_id                     = "movric_hdr"
    option_id                   = "ric_mv_parent"
    has_name_select             = "y"
    name                        = "Move a Ric Port Definition to Another Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 2
    type                        = ""
    ghost                       = "y"
    cmd_to_classify             = ""
    cmd_to_classify_postfix     = ""
    raw_field_name              = "parent"
    cooked_field_name           = ""
    next_type                   = "d"
    help_msg_id                 = ""
    help_msg_loc                = ""

# Name selector command option for parent adapter
sm_cmd_opt:
    id                          = "ric_mv_parent"
    id_seq_num                  = "0"
    disc_field_name             = ""
    name                        = "Parent Adapter"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 8
```

```
    op_type                    = "l"
    entry_type                 = "t"
    entry_size                 = 0
    required                   = "y"
    prefix                     = ""
    cmd_to_list_mode           = "1"
    cmd_to_list                = "lsparent -C "
    cmd_to_list_postfix        = "-l logicname"
    multi_select               = ""
    value_index                = 0
    disp_values                = ""
    values_msg_file            = ""
    values_msg_set             = 0
    values_msg_id              = 0
    aix_values                 = ""
    help_msg_id                = ""
    help_msg_loc               = ""

# Dialogue header
sm_cmd_hdr:
    id                         = "movric_hdr"
    option_id                  = "ric_mv"
    has_name_select            = "y"
    name                       = "Move a Ric Port Definition to Another Port"
    name_msg_file              = "ric.cat"
    name_msg_set               = 3
    name_msg_id                = 2
    cmd_to_exec                = "chdev "
    ask                        = ""
    exec_mode                  = ""
    ghost                      = "n"
    cmd_to_discover            = ""
    cmd_to_discover_postfix    = ""
    name_size                  = 0
    value_size                 = 0
    help_msg_id                = ""
    help_msg_loc               = ""
```

```
#
# Change/Show Characteristics of a Ric Port
# This allows a ric port's characteristics to be shown and,
# if desired, changed.  First, there is a name selector used to put up a
# list of the defined ric ports for the user to select from.  The
# dialogue then shows all of the characteristics.

# Select ric port by logical name
sm_name_hdr:
    id                          = "chgric"
    next_id                     = "chgric_hdr"
    option_id                   = "ric_ln_opt"
    has_name_select             = "n"
    name                        = "Change/Show Characteristics of a Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 3
    type                        = ""
    ghost                       = "y"
    cmd_to_classify             = "lsdev -C -F \"parent:connwhere:location:status\" -l "
    cmd_to_classify_postfix     = " logicname "
    raw_field_name              = "logicname"
    cooked_field_name           = "parent:port:loc:state"
    next_type                   = "d"
    help_msg_id                 = ""
    help_msg_loc                = ""

# The dialogue header.
sm_cmd_hdr:
    id                          = "chgric_hdr"
    option_id                   = "ric_chg,ric_common"
    has_name_select             = "y"
    name                        = "Change/Show Characteristics of a Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 3
    cmd_to_exec                 = "chdev "
    ask                         = ""
    exec_mode                   = "P"
    ghost                       = "n"
    cmd_to_discover             = "lsattr "
    cmd_to_discover_postfix     = "-l logicname -E -O"
    name_size                   = 0
    value_size                  = 0
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
#
# Remove a Ric Port
# This allows a ric port to be removed, including its definition
# in the database, from the system.  A name selector is used to put up a
# list of the "defined" and "configured" ric ports for the user
# to select from.  The dialogue then uses the rmdev command to remove the
# selected device.

# Select ric port by logical name
sm_name_hdr:
    id                          = "rmvric"
    next_id                     = "rmvric_hdr"
    option_id                   = "ric_ln_opt"
    has_name_select             = "n"
    name                        = "Remove a Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 5
    type                        = ""
    ghost                       = "y"
    cmd_to_classify             = ""
    cmd_to_classify_postfix     = ""
    raw_field_name              = "logicname"
    cooked_field_name           = ""
    next_type                   = "d"
    help_msg_id                 = ""
    help_msg_loc                = ""

# Dialogue header
sm_cmd_hdr:
    id                          = "rmvric_hdr"
    option_id                   = "rmvric_opt"
    has_name_select             = "y"
    name                        = "Remove a Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 5
    cmd_to_exec                 = "rmdev "
    ask                         = "y"
    exec_mode                   = ""
    ghost                       = "n"
    cmd_to_discover             = ""
    cmd_to_discover_postfix     = ""
    name_size                   = 0
    value_size                  = 0
    help_msg_id                 = ""
    help_msg_loc                = ""

# Command options
sm_cmd_opt:
    id                          = "rmvric_opt"
    id_seq_num                  = "010"
    disc_field_name             = "logicname"
    name                        = "Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 7
    op_type                     = ""
    entry_type                  = ""
```

```
    entry_size              = 0
    required                = "y"
    prefix                  = "-l "
    cmd_to_list_mode        = ""
    cmd_to_list             = ""
    cmd_to_list_postfix     = ""
    multi_select            = ""
    value_index             = 0
    disp_values             = ""
    values_msg_file         = ""
    values_msg_set          = 0
    values_msg_id           = 0
    aix_values              = ""
    help_msg_id             = ""
    help_msg_loc            = ""

sm_cmd_opt:
    id                      = "rmvric_opt"
    id_seq_num              = "020"
    disc_field_name         = ""
    name                    = "KEEP definition in database"
    name_msg_file           = "ric.cat"
    name_msg_set            = 3
    name_msg_id             = 4
    op_type                 = "r"
    entry_type              = "n"
    entry_size              = 0
    required                = "n"
    prefix                  = ""
    cmd_to_list_mode        = ""
    cmd_to_list             = ""
    cmd_to_list_postfix     = ""
    multi_select            = ""
    value_index             = 0
    disp_values             = "yes,no"
    values_msg_file         = "ric.cat"
    values_msg_set          = 3
    values_msg_id           = 12
    aix_values              = ",-d "
    help_msg_id             = ""
    help_msg_loc            = ""
```

```
#
# Configure a Defined Ric Port
# This allows a ric port that is defined but not configured to
# be configured.  A name selector is used to put up a list of the "defined"
# ric ports for the user to select from.  The dialogue then uses
# the mkdev command to configure the selected device.

# Select ric port by logical name
sm_name_hdr:
    id                          = "cfgric"
    next_id                     = "cfgric_hdr"
    option_id                   = "ric_ln_opt"
    has_name_select             = "n"
    name                        = "Configure a Defined Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 6
    type                        = ""
    ghost                       = "y"
    cmd_to_classify             = ""
    cmd_to_classify_postfix     = ""
    raw_field_name              = "logicname"
    cooked_field_name           = ""
    next_type                   = "d"
    help_msg_id                 = ""
    help_msg_loc                = ""

# Dialogue header
sm_cmd_hdr:
    id                          = "cfgric_hdr"
    option_id                   = "cfgric_opt"
    has_name_select             = "y"
    name                        = "Configure a Defined Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 6
    cmd_to_exec                 = "mkdev "
    ask                         = ""
    exec_mode                   = ""
    ghost                       = "y"
    cmd_to_discover             = ""
    cmd_to_discover_postfix     = ""
    name_size                   = 0
    value_size                  = 0
    help_msg_id                 = ""
    help_msg_loc                = ""

# Command option
sm_cmd_opt:
    id                          = "cfgric_opt"
    id_seq_num                  = "010"
    disc_field_name             = "logicname"
    name                        = "Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 7
    op_type                     = ""
    entry_type                  = ""
    entry_size                  = 0
```

```
required                  = "y"
prefix                    = "-l "
cmd_to_list_mode          = ""
cmd_to_list               = ""
cmd_to_list_postfix       = ""
multi_select              = ""
value_index               = 0
disp_values               = ""
values_msg_file           = ""
values_msg_set            = 0
values_msg_id             = 0
aix_values                = ""
help_msg_id               = ""
help_msg_loc              = ""
```

```
#
# Name selector command option for listing ric ports by
# logical name.
# Used by: move, change/show, remove, and configure functions.
sm_cmd_opt:
    id                          = "ric_ln_opt"
    id_seq_num                  = "0"
    disc_field_name             = ""
    name                        = "Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 7
    op_type                     = "l"
    entry_type                  = "t"
    entry_size                  = 0
    required                    = "y"
    prefix                      = ""
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "lsdev -C -c ricport"
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
#
# Dialog header command options.  Specific to add.
# Used by: add function.

# Displays ric port's parent adapter.
sm_cmd_opt:
    id                          = "ric_add"
    id_seq_num                  = "001"
    disc_field_name             = "parent"
    name                        = "Parent Adapter"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 8
    op_type                     = ""
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "y"
    prefix                      = "-p "
    cmd_to_list_mode            = ""
    cmd_to_list                 = ""
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""

# Displays physical port number being defined.
sm_cmd_opt:
    id                          = "ric_add"
    id_seq_num                  = "002"
    disc_field_name             = ""
    name                        = "PORT number"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 9
    op_type                     = "l"
    entry_type                  = "t"
    entry_size                  = 0
    required                    = "+"
    prefix                      = "-w "
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "lsconn -k ricp "
    cmd_to_list_postfix         = "-p parent"
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
#
# Dialog header command options.  Specific to show/change.
# Used by: show/change function.

# Displays the ric port's logical name.
sm_cmd_opt:
    id                      = "ric_chg"
    id_seq_num              = "001"
    disc_field_name         = "logicname"
    name                    = "Ric Port"
    name_msg_file           = "ric.cat"
    name_msg_set            = 3
    name_msg_id             = 7
    op_type                 = ""
    entry_type              = "n"
    entry_size              = 0
    required                = "y"
    prefix                  = "-l "
    cmd_to_list_mode        = ""
    cmd_to_list             = ""
    cmd_to_list_postfix     = ""
    multi_select            = ""
    value_index             = 0
    disp_values             = ""
    values_msg_file         = ""
    values_msg_set          = 0
    values_msg_id           = 0
    aix_values              = ""
    help_msg_id             = ""
    help_msg_loc            = ""

# Displays device's state.
sm_cmd_opt:
    id                      = "ric_chg"
    id_seq_num              = "002"
    disc_field_name         = "state"
    name                    = "Status"
    name_msg_file           = "ric.cat"
    name_msg_set            = 3
    name_msg_id             = 10
    op_type                 = ""
    entry_type              = "n"
    entry_size              = 0
    required                = "n"
    prefix                  = ""
    cmd_to_list_mode        = ""
    cmd_to_list             = ""
    cmd_to_list_postfix     = ""
    multi_select            = ""
    value_index             = 0
    disp_values             = ""
    values_msg_file         = ""
    values_msg_set          = 0
    values_msg_id           = 0
    aix_values              = ""
    help_msg_id             = ""
    help_msg_loc            = ""
```

```
# Displays device's location.
sm_cmd_opt:
    id                          = "ric_chg"
    id_seq_num                  = "003"
    disc_field_name             = "loc"
    name                        = "Location"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 11
    op_type                     = ""
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "n"
    prefix                      = ""
    cmd_to_list_mode            = ""
    cmd_to_list                 = ""
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""

# Displays parent adapter.
sm_cmd_opt:
    id                          = "ric_chg"
    id_seq_num                  = "004"
    disc_field_name             = "parent"
    name                        = "Parent adapter"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 8
    op_type                     = ""
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "n"
    prefix                      = "-p"
    cmd_to_list_mode            = ""
    cmd_to_list                 = ""
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""

# Displays physical port number being defined.
sm_cmd_opt:
    id                          = "ric_chg"
    id_seq_num                  = "005"
    disc_field_name             = "port"
```

```
    name                      = "PORT number"
    name_msg_file             = "ric.cat"
    name_msg_set              = 3
    name_msg_id               = 9
    op_type                   = "l"
    entry_type                = "t"
    entry_size                = 0
    required                  = "n"
    prefix                    = "-w "
    cmd_to_list_mode          = "1"
    cmd_to_list               = "lsconn "
    cmd_to_list_postfix       = "-p parent -l logicname"
    multi_select              = ""
    value_index               = 0
    disp_values               = ""
    values_msg_file           = ""
    values_msg_set            = 0
    values_msg_id             = 0
    aix_values                = ""
    help_msg_id               = ""
    help_msg_loc              = ""

# Display database only question (last item on screen).
sm_cmd_opt:
    id                        = "ric_chg"
    id_seq_num                = "050"
    disc_field_name           = ""
    name                      = "Apply change to DATABASE only"
    name_msg_file             = "ric.cat"
    name_msg_set              = 3
    name_msg_id               = 13
    op_type                   = "r"
    entry_type                = "n"
    entry_size                = 0
    required                  = "n"
    prefix                    = ""
    cmd_to_list_mode          = ""
    cmd_to_list               = ""
    cmd_to_list_postfix       = ""
    multi_select              = "n"
    value_index               = 1
    disp_values               = "yes,no"
    values_msg_file           = "ric.cat"
    values_msg_set            = 3
    values_msg_id             = 12
    aix_values                = "-P ,"
    help_msg_id               = ""
    help_msg_loc              = ""
```

```
#
# Dialog header command options.  Common to add and show/change.
# Used by: add and show/change functions.

# Displays rdto attribute.
sm_cmd_opt:
    id                          = "ric_common"
    id_seq_num                  = "010"
    disc_field_name             = "rdto"
    name                        = "RECEIVE DATA TRANSFER OFFSET"
    name_msg_file               = "ric.cat"
    name_msg_set                = 2
    name_msg_id                 = 2
    op_type                     = "l"
    entry_type                  = "#"
    entry_size                  = 0
    required                    = "n"
    prefix                      = "-a rdto="
    cmd_to_list_mode            = "r"
    cmd_to_list                 = "lsattr -c ricport -s ricp -t port -a rdto -R"
    cmd_to_list_postfix         = ""
    multi_select                = "n"
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""

# Displays autoconfig attribute.
sm_cmd_opt:
    id                          = "ric_common"
    id_seq_num                  = "020"
    disc_field_name             = "autoconfig"
    name                        = "STATE to be configured at boot time"
    name_msg_file               = "ric.cat"
    name_msg_set                = 2
    name_msg_id                 = 3
    op_type                     = "l"
    entry_type                  = "t"
    entry_size                  = 0
    required                    = "n"
    prefix                      = "-a autoconfig="
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "lsattr -c ricport -s ricp -t port -a autoconfig -R'
    cmd_to_list_postfix         = ""
    multi_select                = "n"
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
#
# Dialog header command options.  Specific to move.
# Used by: move function.

# Displays the ric port's logical name.
sm_cmd_opt:
    id                          = "ric_mv"
    id_seq_num                  = "001"
    disc_field_name             = "logicname"
    name                        = "Ric Port"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 7
    op_type                     = ""
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "y"
    prefix                      = "-l "
    cmd_to_list_mode            = ""
    cmd_to_list                 = ""
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""

# Displays parent adapter.
sm_cmd_opt:
    id                          = "ric_mv"
    id_seq_num                  = "002"
    disc_field_name             = "parent"
    name                        = "Parent Adapter"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 8
    op_type                     = ""
    entry_type                  = "n"
    entry_size                  = 0
    required                    = "y"
    prefix                      = "-p "
    cmd_to_list_mode            = ""
    cmd_to_list                 = ""
    cmd_to_list_postfix         = ""
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

```
# Displays physical port number being defined.
sm_cmd_opt:
    id                          = "ric_mv"
    id_seq_num                  = "003"
    disc_field_name             = "port"
    name                        = "PORT number"
    name_msg_file               = "ric.cat"
    name_msg_set                = 3
    name_msg_id                 = 9
    op_type                     = "l"
    entry_type                  = "t"
    entry_size                  = 0
    required                    = "+"
    prefix                      = "-w "
    cmd_to_list_mode            = "1"
    cmd_to_list                 = "lsconn "
    cmd_to_list_postfix         = "-p parent -l logicname"
    multi_select                = ""
    value_index                 = 0
    disp_values                 = ""
    values_msg_file             = ""
    values_msg_set              = 0
    values_msg_id               = 0
    aix_values                  = ""
    help_msg_id                 = ""
    help_msg_loc                = ""
```

C.3.2 SMIT Log File

```
    Starting SMIT

(Menu screen selected as FastPath,
     id          = "__ROOT__",
     id_seq_num  = "0",
     next_id     = "top_menu",
     title       = "System Management".)
    Object class: sm_menu_opt,
     id   = "__ROOT__", id_seq_num = "0", next_id = "top_menu",
        text = "System Management"


(Menu screen selected,
     FastPath    = "top_menu",
     id_seq_num  = "0",
     next_id     = "top_menu",
     title       = "System Management".)
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "010", next_id = "install",
        text = "Installation and Maintenance"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "020", next_id = "dev",
        text = "Devices"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "030", next_id = "storage",
        text = "Physical and Logical Storage"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "040", next_id = "security",
        text = "Security and Users"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "050", next_id = "commo",
        text = "Communications Applications and Services"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "060", next_id = "spooler",
        text = "Spooler (Print Jobs and Printer)"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "070", next_id = "problem",
        text = "Problem Determination"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "080", next_id = "performance",
        text = "Performance and Resource Scheduling"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "090", next_id = "system",
        text = "System Environments and Processes"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "100", next_id = "apps",
        text = "Applications"
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "999", next_id = "",
        text = "Using SMIT (information only)"

1 asl_screen() returned action/index = 7 / 2
    Object class: sm_menu_opt,
     id   = "top_menu", id_seq_num = "020", next_id = "dev",
        text = "Devices"
```

```
(Menu screen selected,
     FastPath    = "dev",
     id_seq_num  = "020",
     next_id     = "dev",
     title       = "Devices".)
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "010", next_id = "printer",
        text = "Printer/Plotter Devices"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "030", next_id = "tty",
        text = "TTY"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "040", next_id = "pty",
        text = "PTY"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "050", next_id = "disk",
        text = "Fixed Disk"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "060", next_id = "cdrom",
        text = "CD ROM Drive"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "070", next_id = "diskette",
        text = "Diskette Drive"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "080", next_id = "tape",
        text = "Tape Drive"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "100", next_id = "commodev",
        text = "Communication Devices"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "110", next_id = "hft",
        text = "High Function Terminal (HFT)"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "120", next_id = "lssdev",
        text = "List All Supported Devices"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "130", next_id = "lsddev",
        text = "List All Defined Devices"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "135", next_id = "cfgmgr",
        text = "Configure Devices Added After IPL"
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "140", next_id = "_dump_link",
        text = "System Dump"

1 asl_screen() returned action/index = 7 / 8
    Object class: sm_menu_opt,
     id   = "dev", id_seq_num = "100", next_id = "commodev",
        text = "Communication Devices"
```

```
(Menu screen selected,
     FastPath    = "commodev",
     id_seq_num  = "100",
     next_id     = "commodev",
     title       = "Communication Devices".)
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "010", next_id = "ethernet",
        text = "Ethernet Adapter"
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "020", next_id = "token",
        text = "Token Ring Adapter"
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "030", next_id = "multiprotocol",
        text = "Multiprotocol Adapter"
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "040", next_id = "3270con",
        text = "3270 Connection Adapter"
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "050", next_id = "5085",
        text = "5085/86/88 Attachment Adapter"
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "060", next_id = "x25",
        text = "X.25 Adapter"
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "070", next_id = "ric",
        text = "Realtime Interface Co-Processor Portmaster Adapter"

1 asl_screen() returned action/index = 7 / 7
    Object class: sm_menu_opt,
     id   = "commodev", id_seq_num = "070", next_id = "ric",
        text = "Realtime Interface Co-Processor Portmaster Adapter"
```

```
(Menu screen selected,
     FastPath    = "ric",
     id_seq_num  = "070",
     next_id     = "ric",
     title       = "Realtime Interface Co-Processor Portmaster Adapter".)
    Object class: sm_menu_opt,
     id   = "ric", id_seq_num = "010", next_id = "lsdric",
       text = "List All Defined Ric Ports"
    Object class: sm_menu_opt,
     id   = "ric", id_seq_num = "020", next_id = "makric",
       text = "Add a Ric Ports"
    Object class: sm_menu_opt,
     id   = "ric", id_seq_num = "030", next_id = "movric",
       text = "Move a Ric Port Definition to Another Port"
    Object class: sm_menu_opt,
     id   = "ric", id_seq_num = "040", next_id = "chgric",
       text = "Change/Show Characteristics of a Ric Port"
    Object class: sm_menu_opt,
     id   = "ric", id_seq_num = "050", next_id = "rmvric",
       text = "Remove a Ric Port"
    Object class: sm_menu_opt,
     id   = "ric", id_seq_num = "060", next_id = "cfgric",
       text = "Configure a Defined Ric Port"

1 asl_screen() returned action/index = 7 / 1
----entering smit_dialog (lsdric, #
, 0)
    Object class: sm_cmd_hdr,
     id   = "lsdric", option_id = "  ",
     name = "List All Defined Ric Ports"
(Dialogue screen selected,
     FastPath    = "lsdric",
     id          = "lsdric",
     title       = "List All Defined Ric Ports".)
    Command_to_Execute follows below:
>> lsdev -C -c ricport -H
```

Appendix D. Installp/Updatep Files

D.1 Required Files for Creating Compatible Application Programs

The files described in the following paragraphs are required for the installation or update of an application program.

D.1.1 The lpp_name File

The application program name file, ./lpp_name, provides the installp command and the updatep command with information about each program and program option contained on the installation or update medium. This file is a variable record length file containing one record (line of text) for each program or program option in the same order as the programs and options appear on the distribution medium. This list is enclosed by { } (curly braces). Information that is not contained within the braces applies to the medium and the platform.

The first line contains three tokens that specify the format, platform, and type of media followed by the left brace delimiter for the list of programs. The tokens are delimited by one or more blank characters. The current format token is 1 (one), the platform token is R, and the media type token is I for an installation or M for multiple updates.

Each line within the braces is made up of information about a single program option. With the exception of the description and comment, all of the items of information are separated by one or more blank characters. Each line contains the following items:

Program Identifies the program or program option. This value is a character string.

Level Identifies the version, release, modification, and fix levels at which the program will be when installed or updated. This value is a character string.

Location Indicates the volume location where the files belonging to this option can be found on the media. This value is a character an integer.

QuiesceValue Indicates whether or not the subsystem should be stopped prior to installation or update of the program or option. This value is either Y or N.

Type Indicates the type of information contained on the media, such as object code or documentation.

Language The NLS language token that indicates the language. This value is US_ENG, to indicate U.S. English.

Description Describes the program. This value is a character string that extends to the end of the line or until a pound symbol (#) is encountered.

#Comment All text from a pound symbol until the end of the line is considered to be a comment.

© Copyright IBM Corp. 1991

For example, the following application program name file specifies information about the bosdev program, which contains three options, sendmail, ftp, and tn. All of the options will be at the same level when installed; all are on the same volume of media; none require the system to be stopped before installation; all contain object code; and all are in U.S. English.

```
1 R I {
bosdev.tcpip.sendmail 03.01.0010.0000 1 N O US_ENG Sendmail progrm
bosdev.tcpip.ftp 03.01.0010.0000 1 N O US_ENG FTP program#Bug fix
bosdev.tcpip.tn 03.01.0010.0000 1 N O US_ENG Telnet program
}
```

The application program name file is displayed when the installp -l command or updatep -l command is invoked.

D.1.2 The liblpp.a File

The application program installation library file, /usr/lpp/Program/liblpp.a, contains the library of the program's installation related files. The application program installation files are extracted from a single file on the distribution medium (in the backup -i format), which is initially restored into the application program installation library file by the installp command. Program is the name of the program.

D.1.3 The instal Script File

An application program to be installed must provide an installation script named instal. The installation script can be either a shell script or an executable file in the a.out file format. The installp command invokes the installation script and waits for a return code from it.

The installation script is required for an application program even if the program has separately installable options. The program installation script should know how to process the program's options. For example, bosext has one top-level installation script that processes the installation for all of its options based upon the installation files associated with each option. It is also possible to have a top-level installation script that invokes a different installation script for each option.

The installation script for an application program should do any processing necessary to facilitate the installation of the program, such as checking prerequisites, restoring all files associated with the program, performing any configuration necessary for the program, and updating the error and trace templates. The installp and updatep subcommands as well as any of the AIX commands are valid for use in the installation script. The following commands are available to help with installation processing:

inusave Saves all files specified by the apply list (al) into the save directory belonging to the application program. This is not required for an installation.

inurest Restores files from the distribution media onto the system using the apply list as input.

inuumsg Issues translated messages for the application program being installed.

ckprereq Verifies compatibility of the application program with any dependencies using the history information found in the VPD and the supplied prereq file.

The path to these commands is /etc/command.

The installp command invokes the installation script and passes two arguments to it. The first argument is the name of the device from which the installation is being done or the path name of a backup format file on the distribution medium. The installation script should pass this argument to the inurest command in order to identify the device to be used for the restore. The second argument is the name of the file that contains a list of the options to install.

For example, a user chooses to install the acct and ate options of the bosext program from the /dev/rfd0 device. The installp command invokes the installation script, passing to it the device name, /dev/rfd0, and the name of a file that contains the following option names:

```
bosext.acct.obj
bosext.ate.obj
```

The installation script reads the file to determine which options are being installed. An installation script must perform the following procedures:

- Use the ckprereq command to verify compatibility of the application program with other installed programs using the history information of the other programs The history information can be found in the Vital Product Data (VPD) database and the prereq file.
- Verify that the required files exist and that information contained in them is in the expected format.
- Use the inurest command to restore all of the required files from the distribution medium.
- Execute the configuration procedure if it exists.
- Return an exit code indicating the status of the installation. The installp command issues an appropriate error message based on this exit code.

The installp command can continue the installation of an application program even if some of the program options do not install successfully. If no information other than the error return code is given, the installp command assumes that the entire application program installation was unsuccessful. If the installation procedure returns a nonzero return code to the installp command, the installp command looks for a file called status. If none is found, then all options are considered to have failed.

The status file informs the installp command when certain application program options install successfully and certain options do not. The installation script must create the status file to specify this information. The format of the status file is two items per line. The first item is a single character, either s (success) or f (failure). The second item is the name of the program or option.

When an application program or option installs successfully, information about it is entered into the Vital Product Data (VPD) database. If an application program or option does not install successfully, the cleanup procedure is carried out for it and no information is entered into the VPD database.

The following example of a status file indicates to the installp command that the installations for the sendmail option and the ftp option are successful and the installation for the telnet option is not successful. VPD entries for the sendmail option and the ftp option are created. The cleanup procedure is carried out for the telnet option, for which no VPD entries are created.

```
s bosdev.tcpip.sendmail
f bosdev.tcpip.telnet
s bosdev.tcpip.ftp
```

Refer to the sample installation script for the extended base operating system program that is provided as an example of an installation script.

D.1.4 The al File

The installation apply list file, al, contains an apply list for the entire application program. The installation apply list contains the names of all of the files to be restored from the distribution medium during the installation of an application program. The files must be listed as relative file specifications from the root directory (for example, ./usr/lib/sendmail). The inurest command, which archives and restores files for the installp command, restores these files to the appropriate system directories. The current directory is always set to root by the inurest command

The entries in the installation apply list file(s) can be used to locate the corresponding entries in an archive control file (/usr/lpp/Program/lpp.acf) when a restored file is being archived.

If the application program has separately installable options, an installation apply list file (Option.al) can be supplied for each program option. The apply list files for each of the individual program options can be used instead of the al file. The installation script must know how to call the inurest command for each of the apply list files. If the program does not require a more specific method of specifying file names, it is not necessary to provide an apply list file for each option that is being installed.

If only an al file is present, the information applies to the application program as a whole. If various Option.al files are present, they each apply to a specific program option.

D.1.5 The copyright File

The copyright file contains the copyright information associated with each application program included on the installation or update distribution medium. This file can be empty, but it must be present. If copyright information is included for an application program, it contains the full name of the program followed by the copyright notice. For a program update, the copyright file must be the first file on the update distribution medium. If more than one program update is contained on the medium, the copyright file contains a copy of the copyright information for each of the programs.

D.1.6 The size File

The application program size file, size, contains size specifications for the installation or update of the entire program. If the program has separately installable options, supply a size file (Option.size) for each option. The size files for each of the individual program options can be used instead of the size file.

If only a size file is present, the information applies to the application program as a whole. If various Option.size files are present, they each apply to a specific program option.

The installp command uses the information in the size file to ensure that the system as currently configured has enough mass storage space to allow installation or update of the program or option. It is important to determine how much space is required in the user's system with the file systems that are defined. Each major directory is listed along with how many 512-byte blocks the program requires in the directory. A major directory refers to any directory that is not split across logical volumes.

Each separate directory that functions as a unit or is large enough that it might be split must be specified in the size file. The size file should specify the lowest level of directories that is feasible. For example, specify /usr/lpp/Program rather than /usr/lpp; specify /usr/lib and /usr/bin rather than simply /usr.

The format of the size file is one record (ASCII text line) for each directory. The full path name of the directory is followed by the amount o information (in 512-byte blocks) that the installation will attempt to put there. The directory name and the size are separated by one or more blank spaces. It is not necessary for the directories to exist already; they can be directories that are to be added during the installation or update.

For example, the size file can include the following entries to specify that the /usr/lib directory being installed or updated will require 200 512-KB blocks (102,400 KB or might look something like this:

```
/usr/lib     200
/usr/bin     30
/lib         40
```

The size file should also include any old files (to be replaced) that will be moved into the /usr/lpp/Program/inst_updt.save file so that these files can be restored if the update is later rejected.

D.1.7 The lpp.cleanup File

The application program cleanup file, lpp.cleanup, provides a procedure to recover from a failed program installation attempt. The installp command carries out this procedure when an error occurs or when the -C flag is specified for the installp command. The cleanup procedure can be either a shell script or an executable file in the a.out file format.

The cleanup procedure can call the installp and updatep subcommands as well as AIX commands. The contents of the lpp.cleanup file depend upon what procedures were implemented by the program during installation. At the very minimum, the cleanup procedure must remove any files that were restored. If

the program has a configuration procedure, then the cleanup procedure must be able to undo the configuration procedures. If the program has separately installable options, the cleanup procedure must be able to clean up separate options or carry out a separate cleanup procedure for each option.

The installp command calls the program cleanup procedure if the installation of the program or one or more of the program's installable options fails. Two arguments are passed to the cleanup procedure. The first argument is the device name, which can generally be ignored. The second argument is the name of the file that contains a list of the options that failed to install. These arguments are the same as those passed into the installation script.

For example, a user attempts to install the acct and ate options in bosext from /dev/rfd0. The acct option installs successfully but ate fails. The installp command then invokes the lpp.cleanup script, passing in /dev/rfd0 and the name of a file whose contents are: bosext.ate.obj.

If this application program is an option of a larger program, the cleanup file must be put into a directory of its own so that other options of the larger program do not write over it with their own lpp.cleanup files. In this case, the primary cleanup procedure should know that other cleanup procedures exist and how to access them.

D.1.8 The special File

The application update program special file, ./special, describes any special update requirements of an program. This file can be empty if there are no special requirements, but it must exist on the update distribution medium. The following two types of special requirements are included in this file:

ras RAS template updates are included.

config Configuration updates are included.

The formats for each of the respective types of records are single lines of text containing the following information delimited by one or more spaces or tabs :

```
ras Program  Level
```

A ras entry indicates that the updates for the program or program option specified by Program at the level indicated by Level include changes for a ras trace or error template. If a program or option includes a ras entry in the special file, the updates for the program or option must be applied and committed or rejected without being grouped with any other program. A specific update instruction must be included with any ras update to inform the user how to recover the ras templates that were saved during the installation or update.

```
config  Program Level
```

A config entry indicates that the updates for the program or program option specified by Program at the level indicated by level include configuration changes. If an program or option includes a config entry in the special file, the updates for the program or option must be applied and committed or rejected without being grouped with any other program. A config entry supercedes any ras requirements.

D.1.9 The service_num File

The service number file, ./service_num, contains the corrective service number of the update distribution medium. The service number can be as many as 10 characters. This information is entered into the Vital Product Data (VPD) database.

D.1.10 The arp File

The update archive file, ./usr/lpp/Program/inst_updt/arp, contains the procedures to be used to perform the update. Each program named in the update control list requires a separate update archive file. The update archive file is in the archive format.

D.1.11 The update Script File

An update to an application program must provide an update script named update. The update script can be either a shell script or an executable file in the a.out file format. The updatep command invokes the update script and waits for a return code from it.

The update script is required to update an application program even if the program has separately installable options. The program update script should know how to process the program's options.

The update script for an application program should do any processing necessary to facilitate the update of the program, such as checking prerequisites, restoring all files associated with the program, performing any configuration necessary for the program, and updating the error and trace templates. The installp and updatep subcommands as well as any of the AIX commands are valid for use in the update script. The following commands are available to help with update processing:

inusave Saves all files specified by the apply list (al) into the save directory belonging to the program. This must be used for an update.

inurest Restores files from the distribution media onto the system using the apply list as input.

inuumsg Issues translated messages for the program being updated.

ckprereq Verifies compatibility of the program with any dependencies using the history information found in the VPD and the supplied prereq file.

The path to these commands is /etc/command.

The updatep command invokes the update script and passes three arguments to it. The first argument is the name of the device from which the update is being done or the path name of a backup format file on the distribution medium. The update script should pass this argument to the inurest command in order to identify the device to be used for the restore. The second argument is the name of the file that contains a list of the options to update. The list contains the option name and the level that is currently installed. The third argument is the name of the apply list generated by the updatep command.

For example, a user chooses to update the acct and ate options of the bosext program from the /dev/rfd0 device. The updatep command invokes the update script, passing to it the device name, /dev/rfd0, and the name of a file that contains the following option names:

```
bosext.acct.obj 03.01.0010.0000
bosext.ate.obj 03.01.0020.0000
```

The update script reads the file to determine which options are being updated. An update script must perform the following procedures:

- Use the ckprereq command to verify compatibility of the program with other installed programs using the history information of the other programs. The history information can be found in the Vital Product Data (VPD) database and the prereq file.
- Verify that the required files exist and that information contained in them is in the expected format.
- Use the inusave command to back up any files that will be changed or replaced.
- Use the inurest command to restore all of the required files (according to the apply list) from the distribution medium.
- Execute the configuration procedure if it exists.
- Return an exit code indicating status of the update. The updatep command issues an appropriate error message based on this exit code.

The updatep command can continue the update of an application program even if some of the program options do not update successfully. If no information other than the error return code is given, the updatep command assumes that the entire program update was unsuccessful. If the update procedure returns a nonzero return code to the updatep command, the updatep command looks for a file called status. If none is found, then all options are considered to have failed.

The status file informs the updatep command when certain program options update successfully and certain options do not. The update script must create the status file to specify this information. The format of the status file is two items per line. The first item is a single character, either s (success) or f (failure). The second item is the name of the program or option.

When an program or option updates successfully, information about it is entered into the Vital Product Data (VPD) database. If an program or option does not update successfully, the cleanup procedure is carried out for it and no information is entered into the VPD database.

The following example of a status file indicates to the updatep command that the updates to the sendmail option and the ftp option are successful and the update to the telnet option is not successful. VPD entries for the sendmail option and the ftp option are created. The cleanup procedure is carried out for the telnet option, for which no VPD entries are created.

```
s bosdev.tcpip.sendmail
f bosdev.tcpip.telnet
s bosdev.tcpip.ftp
```

D.1.12 The al_Level File

The update apply list file, al_Level (where Level refers to the program version level in vv.rr.mmmm.ffff format), contains an apply list for this level of the application program. The update apply list contains the names of all of the files to be restored from the distribution medium during the update of an application program. The files must be listed as relative file specifications from the root directory (for example, ./usr/lib/sendmail). The inurest command, which archives and restores files for the updatep command, restores these files to the appropriate system directories. The current directory is always set to root by the inurest command.

The entries in the update apply list file(s) can be used to locate the corresponding entries in an archive control file (/usr/lpp/Program/lpp.acf) to determine if a restored file is to be archived.

If the program has separately installable options, an update apply list file (Option.al_Level) can be supplied for each program option. The apply list files for each of the individual program options can be used instead of the al_Level file. The update script must know how to call the inurest command for each of the apply list files. If the program does not require a more specific method of specifying file names, it is not necessary to provide an apply list file for each option that is being updated.

If only an al_Level file is present, the information applies to the program as a whole. If various Option.al_Level files are present, they each apply to a specific program option.

It is customary to include on the distribution medium an apply list for all previous updates that are released for an application program. There can be several files for different versions, releases, modifications, or fixes of the program. For example, if a program will be at level 01.01.0030.0000 after application of the current update, then the following three apply lists might be included on the distribution medium:

al_01.01.0010.0000 The first set of updates.

al_01.01.0020.0000 The second set of updates.

al_01.01.0030.0000 The current updates.

In order to apply the following fix to a particular program, the program must be at the appropriate level. For example, the program must be at level 30 in order to apply the following fix.

al_01.01.0030.0010 Fixes to the current updates.

Each apply list contains the target paths for all of the files that must be restored by the update to bring this program to the level shown.

The updatep command concatenates all of the apply lists from the current level to the new level and passes the name of the concatenated file to the update script. If an Option.al_Level file exists for an option, the updatep command creates an Option.al file, which contains all of the files from the current level to the new level for each option.

D.2 Optional Files for Creating Compatible Application Programs

The files described in the following paragraphs are optional for each program.

D.2.1 The config File and Option.config File

The configuration file is a shell procedure or C program that performs special configuration actions needed by the program (the config file) or a program option (the Option.config file) to complete the installation or update. If only a config file is present, it is assumed to apply to the entire program. The installation or update script should execute the configuration procedure at the end of installation or update processing (after all the files have been restored, the error templates updated, the trace templates updated, and so forth). Some steps that might be included in a configuration file are:

- Add a user ID or group ID that your program requires to the system
- Adding ODM objects
- Add SMIT dialogs.

If the program has separately installable options, then supply a configuration script file (Option.config) for each option. The installation (instal file) or update (update file) script must know how to carry out an option's configuration procedure.

D.2.2 The prereq and Option.prereq File

The prerequisites file, prereq, contains prerequisite information for the installation or update of the entire application program. If a program has any dependencies on other programs being installed or at a certain level, you must supply a prerequisites file. An update prerequisite file can reference a particular fix level. The installation (instal file) or update (update file) script should call the ckprereq command prior to restoring the files associated with the program. The installation or update script must know how to call the ckprereq command, which uses the prerequisites files to verify that the program dependencies are met.

If the application program has separately installable options, supply a prerequisites file (Option.prereq) for each option whose prerequisites differ from those of the program. The individual program options prerequisites files can be used instead of the prereq file. The installation or update script must know how to call the ckprereq command for each of the prerequisites files. If all of the program's options have the same prerequisites, it is not necessary to provide a prerequisites file for each option.

If only a prereq file is present, the information applies to the program as a whole. If various Option.prereq files are present, they each apply to a specific program option.

Note that the entry for the prerequisites string in the Vital Product Data (VPD) database is limited to 1020 characters. Therefore, because the exact contents of the prerequisites file are entered into the VPD, a prerequisites file cannot contain more than 1020 characters.

Each record (line of text) in a prerequisites file is line of text that describes a single prerequisite. There are two types of prerequisite records. The first type

describes the required program level prerequisites. The second type describes the relational prerequisites between programs.

The first record type describes the specific version (v), release (r), modification (m), and fix (f) levels that are required of a specified program. A program level record lists the name of the program required, followed by one or more expressions of equality or inequality between the letters v, r, m, and f and a numeric expression of those values. The fields in each record are separated by one or more spaces or tabs.

Note that the l (level) value in an AIX Version 2 prerequisites file is now regarded as the m (modification) value. If the l key letter exists in an old prerequisites file, it should be updated to the m key letter.

For example, the following record indicates that Version 1, Release 3 or later of an application program named database is required.

```
database v=1  r>2
```

In the following example, the record indicates that Version 2 or later of an application program named spreadsheet is required.

```
spreadsheet v>1
```

It is also possible to specify more than one version, release, modification, or fix level for a certain program by specifying the or option (o) between the values for a particular specification. For example, the following record indicates that Version 1, Release 3, and either Modification 1 or 2 of the application program named timemgr is required.

```
timemgr  v=1 r=3 m=1 o=2
```

The second record type allows programs to be specified in relation to each other. A relational prerequisites record begins with a logical expression (> followed by an integer) and then a left brace. One or more program level records, each on a new line, follows. A } (right brace) ends the relational prerequisites record. In this way, it is possible to declare how many of the programs listed are required as well as what logical relationships are required between the programs.

For example, the following relational prerequisites record specifies that more than one of the programs specified must be installed. Because only two are listed, both the program named spreadsheet and the program named database must be installed. In addition, the spreadsheet program must be Version 1, Release 2, and the database program version must be later than Version 1.

```
>1 {
spreadsheet v=1  r=2
database v>1
}
```

In the following example, the relational prerequisites record specifies that more than none (at least one) of the programs specified must be installed. Any the specified programs can be installed. However, any program that is installed must meet the specified version, release, modification, and fix level requirements.

```
>0 {
custmenu v>2
menus v=1  r=2  m=2
scrnmgr v=1
grphint v>2
}
```

When a relational prerequisites expression fails, the ckprereq command enters the appropriate error code before the > symbol in the first column of the first line.

It is possible to state any sort of relationship with this format, however, a complex relationship will require a complex set of statements and checking. Note that it is perfectly reasonable to include statements in your prerequisites file that are mutually exclusive and check for a certain number of failures. It is not necessary that an program specifically look for a return of zero from the ckprereq command. A combination of statements can be written such that a return code less than some value n is acceptable.

Note that while this version of AIX is backwardly compatible with Version 2 prerequisites files, additional features have been added in the current version. Further, error codes are now entered into column 1 of a prerequisites file (rather than column 18), making the output format different. It is recommended that any old program prerequisites files be updated to the current file format.

D.2.3 The lpp.doc File

The documentation file, lpp.doc, contains any document pages for the program that have changed from a previous version or that should be installed.

If this program is an option of a larger application program, the documentation file must be put into a directory of its own so that other options of the larger program do not write over it with their own lpp.doc files.

D.2.4 The Filename.err File

The Filename.err file contains error report format templates for the program. If an program has to add error report format templates during installation, a Filename.err file must be supplied. The program installation script must call the errinstall command or the errupdate command and pass the Filename.err file to it.

The name of a file that contains information to undo what is changed by this file (to be used if an update is rejected) should be begin with lpp. so that the updatep command leaves it in the /usr/lpp/Program directory and does not delete it.

D.2.5 The Filename.trc File

The Filename.trc file contains trace report format templates and event numbers for the program. If an application program has to add trace report format templates during installation, a Filename.trc file must be supplied. The program installation script must call the trcupdate command and pass the Filename.trc file to it.

D.2.6 The Filename.evt File

The Filename.evt file contains trace event types and event their hook ID relationships. If an application program has to add trace report format templates during installation, a Filename.evt file must be supplied. The program installation script must call the trcupdate command and pass the Filename.evt file to it.

D.2.7 The lpp.acf File

The archive control file, lpp.acf, describes library archive requirements for an application program. This file is necessary only if the program is adding or updating one or more files to a library that is owned by some other program. The archive control file consists of one or more entries, each identifying a file that is to be archived into a particular library. Each entry must be in the following format:

```
Filename ArchiveFilename
```

Filename Refers to the complete path name, relative to root, for the file that is owned by the other program when it is restored by the installation or update procedure. Any unique path name is valid; for consistency however, the file should conform to the following format:

```
./LibPath/inst_updt/LibName/Filename
```

In this format, LibPath refers to the full path to the directory that normally contains the library; inst_updt is a special directory used by the installp command and the updatep command; LibName refers to the library (the archive file) where the file that is owned by the other program is archived; and Filename is the name of the file that is owned by the other program.

ArchiveFilename
Refers to the full path name of the target archive file into which this file will be archived by the installation or update process. The two file names must be separated by one or more spaces or tabs.

For example, a file named doprnt.o that is archived in the /lib/libc.a library would be referred to in the archive control file by the following entry:

```
./lib/inst_updt/libc.a/doprnt.o /lib/libc.a
```

The archive control file entry for a file named doprnt.o that is archived in the /usr/lib/libcurses library would be as follows:

```
./usr/lib/inst_updt/libcurses/doprnt.o /usr/lib/libcurses
```

If this program is an option of a larger application program, the archive control file must be put into a directory of its own so that other options of the larger program do not write over it with their own lpp.acf files.

If an existing archive control file is present, the existing file is moved from /usr/lpp/Program/inst_updt/lpp.acf to /usr/lpp/Program/lpp.acf. If there is also an old archive control file with that name, it is replaced.

D.2.8 The productid File

The product ID file, productid, contains the part number of the program. This file contains one line of text that is entered into the Vital Product Data (VPD) database at installation time.

D.2.9 The inventory File

The inventory file contains specific information about each file that remains in an application program or program option after the installation or update is complete. Much of the inventory information is entered into the Vital Product Data (VPD) database and the /etc/security/sysck.cfg file by the sysck command.

If the program has separately installable options, an inventory file (Option.inventory) can be supplied for each program option. The inventory files for each of the individual program options can be used instead of the inventory file. The update script must know how to call the sysck command for each of the inventory files. Because the installp command automatically calls the sysck command, this is not required of an installation script.

If only an inventory file is present, the information applies to the program as a whole. If various Option.inventory files are present, they each apply to a specific program option.

The inventory file consists of ASCII text in a stanza format.

Note that while the inventory file is not strictly required, it is recommended so that commands such as lslpp and lppchk (which provide information about the program and check consistency) function properly.

D.2.10 The lpp.deinst File

The application program deinstall file, lpp.deinst, contains a procedure for manually deinstalling an program.

If this program is an option of a larger application program, the deinstall file must be put into a directory of its own so that other options of the larger program do not write over it with their own lpp.deinst files.

D.2.11 The rename File

The rename file, rename, contains the mv commands necessary to rename abbreviated file names to the appropriate names for the program installation or update. Because the file names of many of the installation and update files include an application program option name, the file names are frequently longer than 14 characters, which was the limit for a file name length in Version 2 of the AIX operating system. During the program installation, the installp command carries out the rename file if it exists and renames abbreviated file names using the current installation naming conventions.

When the liblpp.a file is extracted on the target machine during an installation, the rename script file is carried out before the instal script file is carried out. Similarly, when the arp file is extracted on the target machine during an update, the rename script file is carried out before the update script file is carried out.

If this file is not present, the installp command expects the file names to be correct.

D.2.12 The lpp.instr File

The application program update instructions file, lpp.instr, describes specific instructions that must be followed in order to apply an update. The file must be written in a simple text format so that it can be printed by a standard system print command. The instructions should include specific update information for each version, release, modification, and fix of the program. This file was referred to as the Program_instr file in AIX Version 2.

D.3 Real Time Interface Co-Processor Device Driver Package

D.3.1 Makefile

```
# Makefile to create installp compatible package for ricdd
#
#
# A. Required files :
#
#    --> The first three are a must -- make will fail otherwise.
#
#         1.  lpp_name (the list of options) (must be in directory $(ROOT)),
#         2.  instal (installation script),
#         3.  lpp.cleanup (cleanup script if installation fails),
#         4a. Option.al (apply list for each option),
#         4b. al (apply list),
#         5a. Option.size (size file for each option);
#         5b. size (size file),
#
#    --> NOTES : - If 4a and 5a are used, a file named Options is required,
#                  it contains the list of options.
#
#                - In this case, there is an Options file containing :
#
#                     ricdd.driver
#                     ricdd.src
#
#                - Makefile knows how to make al(everything under $(ROOT)),
#                  and can calculate size or Option.size from al or Option.al.
#
#
#    --> These two files are required, but Makefile knows how to supply them :
#
#         6.  liblpp.a (archive containing all installp files except lpp_name)
#             (Makefile will be create or remake it if it is out of date),
#         7.  copyright (if missing, Makefile will supply a default file).
#
# B. Optional files:
#
#         - config, prereq, lpp.acf -- various requirements for installation;
#         - inventory, productid, lpp.doc -- documentary files;
#         - *.err, *.trc, *.evt -- for error recovery;
#         - lpp.deinst -- to remove an installation;
#         - rename -- for renaming files if names are too long.
#
#    -> ALL THE OPTIONAL FILES to be included in distribution must be
#       listed in the file "optionalfiles", one per line.
#

# destination of image of distribution medium
#DEV = /u/luc/devdriver/installp/TAPE
#DEV = /usr/lpp.install
DEV = /dev/fd0

# Name of application program
PROG = ricdd
```

```
# File containing list of LPP options
LPPOPTIONS = Options

# The root of the application files directory is $(ROOT); it is assumed
# that every file below $ROOT other than $ROOT/lpp_name and
# $ROOT/usr/lpp/$PROG/liblpp.a is part of the application program package.
ROOT = /u/luc/devdriver/installp/root

# files to be backed up
BACKFILELIST = backup_files

# list of optional files
OPTIONALS = `if [ -f optionalfiles ] ; then cat optionalfiles ; fi`

# The files al_files, size_files and cleanup_files contain list of
# the al, size and cleanup files.
# Makefile will make al_files, size_files and cleanup_files if missing.
AL = `cat al_files`
SIZE = `cat size_files`
CLEANUP= `cat cleanup_files`

#
# create the installp package
#
$(DEV): $(ROOT)/lpp_name al_files size_files cleanup_files \
        $(ROOT)/usr/lpp/$(PROG)/liblpp.a $(BACKFILELIST) chk_inv
      @echo Making installp format distribution on $(DEV)
      @cd $(ROOT) ; backup -vi -f $(DEV) < ../$(BACKFILELIST)

#
# the lpp archive file contains the required and optional files for PROG
#
$(ROOT)/usr/lpp/$(PROG)/liblpp.a: instal $(AL) $(SIZE) $(CLEANUP) \
                                  lpp.cleanup copyright $(OPTIONALS)
      @if [ -f $@ ] ; then echo Updating $@ archive; ar ru $@ $? ;\
                      else echo Making $@ archive; ar cq $@ $? ; fi

#
# check if al_files, size_files and cleanup_files are up-to-date
#
al_files: \
  `if [ -f $(LPPOPTIONS) ] ; then echo $(LPPOPTIONS) ; else echo size ; fi`
      @echo al_files is out-of-date, recreating it
      @if [ -f $(LPPOPTIONS) ] ;\
         then cat $(LPPOPTIONS)|awk '{ printf "%s.al\n", $$1 }' >al_files ;\
         else echo al > al_files ;fi

size_files:\
  `if [ -f $(LPPOPTIONS) ] ; then echo $(LPPOPTIONS) ; else echo size ; fi`
      @echo size_files is out-of-date, recreating it
      @if [ -f $(LPPOPTIONS) ] ;\
         then cat $(LPPOPTIONS)|awk '{ printf "%s.size\n", $$1 }' >size_files ;\
         else echo size > size_files ;fi

cleanup_files:\
  `if [ -f $(LPPOPTIONS) ] ; then echo $(LPPOPTIONS);else echo lpp.cleanup ; fi`
      @echo cleanup_files is out-of-date, recreating it
      @if [ -f $(LPPOPTIONS) ] ;\
         then cat $(LPPOPTIONS)|awk '{ printf "%s.cleanup\n", $$1 }' >cleanup_files ;\
```

```
	else echo lpp.cleanup > cleanup_files ;fi

#
# If al file is missing, assume that everything below $(ROOT) is in.
#
al:
	@echo No apply list, making one
	@cd $(ROOT) ; find . \( ! -name lpp_name \)\
		\( ! -name usr/lpp/$(PROG)/liblpp.a \) -type f -print > ../al

#
# The same for size file; cheat a little: includes size of lpp_name, liblpp.a
#
size:
	@echo No size file, making one
	@cd $(ROOT) ; find . -type d -exec du {} \; > ../size

#
# If any Option.size file is missing, Makefile will create it
# from the corresponding Option.al.
#
.SUFFIXES: .size .al
.al.size :
	@echo $@ is missing, making it
	@cd $(ROOT) ; for fl in `cat ../$<` ;do du -a $$fl |\
		awk '{ printf "%s %d\n", $$2, $$1*2 }' >> ../$@ ;done


#
# copyright file is required -- if not present, make an empty one
#
copyright:
	@echo No copyright file, making an empty one
	@echo > copyright

#
# list of files to backup
#
$(BACKFILELIST): $(AL)
	@echo Making list of files to backup
	@echo ./lpp_name > $(BACKFILELIST)
	@echo ./usr/lpp/$(PROG)/liblpp.a >> $(BACKFILELIST)
	@for alf in $(AL) ;do cat $$alf >> $(BACKFILELIST) ;done

#
# Check that the application files specified are actually present.
#
chk_inv:
	@echo Checking inventory
	@for file in `cat $(BACKFILELIST)` ;do if [ ! -f $(ROOT)/$$file ] ;\
		then echo $$file is missing! ;exit 1; fi ;done
	@echo All files present!
```

D.3.2 Instal

```
#!/bin/sh
#
# instal script to install ricdd
#

# set the device and lpp name
DEVICE=$1
OPTIONLIST=$2
CWD=`pwd`
LPPNAME=`basename $CWD`

# setup error codes
RC_PREREQ=3     # prereq error number for inuumsg
RC_INUREST=25   # restore error number for inuumsg
RC=0            # return code
FAILED="false"  # true if installation fails

# remove the status file
rm -f ./_status

#
# Check prereqs for LPP if prereq file exists
#
if [ -s prereq ]
then
 /etc/ckprereq -v -f prereq
 RC=$?

        # if prereqs for LPP are not met exit with return code from chkprereq
        if [ $RC -ne 0 ] ; then
                /etc/inuumsg $RC_PREREQ  # "prereqs are not met"
                exit $RC
        fi
fi
```

```
# Now, loop through the option names: for each option, do any
# processing that needs to take place before the files are restored; if
# there are no errors add that option to the $OPTIONLIST.new file.

rm -f $OPTIONLIST.new
exec < $OPTIONLIST
while read LPPOPTION
do
        #
        # Check for necessary prerequisites, if the prereqs are not met
        # call inuumsg to issue an error message, set failed to true,
        # echo the option name and status to the status file
        #
        echo installing $LPPOPTION ...
        if [ -s $LPPOPTION.prereq ]
        then
                /etc/ckprereq -v -f $LPPOPTION.prereq
                if [ $? -ne 0 ] ; then
                        /etc/inuumsg $RC_PREREQ  # "prereqs are not met"
                        FAILED="true"
                        echo "$LPPOPTION f" >>./_status
                        continue
                fi
        fi

        # Execute the options pre_instal procedure (if it exists)
        #
        if [ -x $LPPOPTION.pre_i ] ; then
                $LPPOPTION.pre_i "$1"
                if [ $? -ne 0 ] ; then
                        FAILED="true"
                        echo "$LPPOPTION f" >>./_status
                        continue
                fi
        fi

        # if the option doesn't have an apply list, and al file doesn't
        # exist, set failed to true and mark the option's status as failed.
        #
        if [ ! -s $LPPOPTION.al ] ; then
                FAILED="true"
                echo "$LPPOPTION f" >>./_status
                continue
        fi

        # restore files from apply list.
        echo $LPPOPTION >./.option_list
        /etc/inurest -d $DEVICE -q `pwd`/$LPPOPTION.al $LPPNAME `pwd`/.option_list
        RC=$?
        rm -f ./.option_list

        # if inurest failed exit with the return code from inurest
        if [ $RC -ne 0 ]
        then
                /etc/inuumsg $RC_INUREST
                exit $RC
        fi

        # if no errors so far, add this option to the new option list
```

```
        echo $LPPOPTION >> $OPTIONLIST.new

done    # end while read LPPOPTION #

exec < $OPTIONLIST.new
while read LPPOPTION
do
        #
        # Execute the option's post_instal procedure (if it exists)
        #
        if [ -x $LPPOPTION.post_i ] ; then
                $LPPOPTION.post_i "$1" "$2"
                if [ $? -ne 0 ] ; then
                        FAILED="true"
                        echo "$LPPOPTION f" >>./_status
                        continue
                fi
        fi

        # Execute the errupdate command if the $LPPOPTION.err file exists
        #
        if [ -s $LPPOPTION.err ] ; then
                /usr/bin/errupdate -f $LPPOPTION
                if [ $? -ne 0 ] ; then
                        FAILED="true"
                        echo "$LPPOPTION f" >>./_status
                        continue
                fi
        fi

        # Execute the trcupdate command if the $LPPOPTION.trc file exists
        #
        if [ -s $LPPOPTION.trc ] ; then
                /usr/bin/trcupdate -o $LPPOPTION
                if [ $? -ne 0 ] ; then
                        FAILED="true"
                        echo "$LPPOPTION f" >>./_status
                        continue
                fi
        fi

        # Execute the option's config procedure (if it exists)
        #
        if [ -x $LPPOPTION.config ] ; then
                $LPPOPTION.config
                if [ $? -ne 0 ] ; then
                        FAILED="true"
                        echo "$LPPOPTION f" >>./_status
                        continue
                fi
        fi

        # update the status file
        #
        echo "$LPPOPTION s" >>./_status

done # end while read OPTION ##
```

```
# Execute the lpp's config procedure (if it exists)
#
if [ -x config ] ; then
        config
        if [ $? -ne 0 ] ; then
                FAILED="true"
                echo "$LPPNAME f" >>./_status
        fi
fi

# if any of the options failed, exit with a non-zero return code
#
if [ "$FAILED" = "true" ]
then
        exit 100
else
        exit 0
fi
```

D.3.3 ricdd.driver.config

```
#!/bin/sh
#
# ric device driver installation configuration script
#

#
# set the ODMDIR var just to make sure
#
RC=0
ODMDIR=/etc/objrepos

echo configuring ricdd.driver...

#
# Add ricdd definition in PdDv, PdAt and PdCn object classes
#
/usr/bin/odmadd /usr/lpp/ricdd/ric.add
RC=$?
if [ $RC -ne 0 ] ; then
    exit $RC
fi

#
# Add SMIT dialogs for ricdd in ODM
#
/usr/bin/odmadd /usr/lpp/ricdd/sm_ric.add
RC=$?
if [ $RC -ne 0 ] ; then
    exit $RC
fi

#
# First generate ricdd message catalog
#
/usr/bin/gencat ric.cat ric.msg
RC=$?
if [ $RC -ne 0 ] ; then
    exit $RC
fi
#
# Then put the catalog in C and $LANG catalog repositories
#
/bin/cp ric.cat /usr/lpp/msg/C
/bin/cp ric.cat /usr/lpp/msg/$LANG

#
# Cleanup
#
/bin/rm ric.cat ric.add sm_ric.add
```

D.3.4 Lpp.cleanup

```
#!/bin/bsh
#
# script called from installp to cleanup ricdd
#

# set option list and lpp name
OPTIONLIST=$2
LPPNAME=`pwd`
LPPNAME=`basename $LPPNAME`

# setup global vars
RC=0                # return code
ERR_INUCLEN=18      # cleanup error number for inuumsg

#
# loop thru the option names, for each option, invoke the option.cleanup
# provided for that option
#
exec < $OPTIONLIST
while read LPPOPTION
do
     if [ -s $LPPOPTION.cleanup ]
     then
          ./$LPPOPTION.cleanup
          if [ $? -ne 0 ] ; then
               /etc/inuumsg $ERR_INUCLN $LPPOPTION
               RC=1
               continue
          fi
     else /etc/inuumsg $LPPOPTION.cleanup is missing !
     fi

done

# erase LPP directory
echo erasing /usr/lpp/$LPPNAME
rm -rf /usr/lpp/$LPPNAME

exit $RC
```

D.3.5 Ricc.src.cleanup

```
#!/bin/bsh
#
# script to cleanup ricdd.src
#

echo cleaning up ricdd.src...

cd /usr/lpp/ricdd

# remove the src dir
rm -rf src

# remove odm entry
odmdelete -o lpp -q "name = 'ricdd.src'" 2>/dev/null 1>/dev/null

# always succeed
exit 0
```

D.3.6 Ricc.driver.cleanup

```
#!/bin/bsh
#
# script to cleanup ricdd.driver
#

echo cleaning up ricdd.driver...

cd /usr/lpp

trap ":" 0 1 2 3

# remove the device driver
rm -f /etc/drivers/ricdd

# remove the two configuration methods
rm -f /etc/methods/cfgrica
rm -f /etc/methods/cfgricp

# remove the catalogue
rm -f /usr/lpp/msg/$LANG/ric.cat
rm -f /usr/lpp/msg/C/ric.cat

#
# remove odm entries
#

# in lpp object class
odmdelete -o lpp -q "name = 'ricdd.driver'"  2>/dev/null 1>/dev/null

# in Predefined Devices object class
odmdelete -o PdDv -q "uniquetype = 'adapter/mca/ric'" 2>/dev/null 1>/dev/null
odmdelete -o PdDv -q "uniquetype = 'ricport/ricp/port'" 2>/dev/null 1>/dev/null

# in Predefined Attributes object class
odmdelete -o PdAt -q "uniquetype = 'adapter/mca/ric'" 2>/dev/null 1>/dev/null
odmdelete -o PdAt -q "uniquetype = 'ricport/ricp/port'" 2>/dev/null 1>/dev/null

# in Predefined Connections object class
odmdelete -o PdCn -q "uniquetype = 'adapter/mca/ric'" 2>/dev/null 1>/dev/null
```

```
# in SMIT object classes
odmdelete -o sm_menu_opt -q "next_id = 'ric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_menu_opt -q "next_id = 'lsdric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_menu_opt -q "next_id = 'makric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_menu_opt -q "next_id = 'movric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_menu_opt -q "next_id = 'chgric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_menu_opt -q "next_id = 'rmvric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_menu_opt -q "next_id = 'cfgric'" 2>/dev/null 1>/dev/null

odmdelete -o sm_name_hdr -q "id = 'cfgric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_name_hdr -q "id = 'chgric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_name_hdr -q "id = 'makric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_name_hdr -q "id = 'movric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_name_hdr -q "id = 'movric_parent'" 2>/dev/null 1>/dev/null
odmdelete -o sm_name_hdr -q "id = 'rmvric'" 2>/dev/null 1>/dev/null

odmdelete -o sm_cmd_hdr -q "id = 'cfgric_hdr'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_hdr -q "id = 'chgric_hdr'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_hdr -q "id = 'lsdric'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_hdr -q "id = 'makric_hdr'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_hdr -q "id = 'movric_hdr'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_hdr -q "id = 'rmvric_hdr'" 2>/dev/null 1>/dev/null

odmdelete -o sm_cmd_opt -q "id = 'cfgric_opt'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_add'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_chg'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_common'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_ln_opt'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_mk_parent'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_mv'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'ric_mv_parent'" 2>/dev/null 1>/dev/null
odmdelete -o sm_cmd_opt -q "id = 'rmvric_opt'" 2>/dev/null 1>/dev/null

# always succeed
exit 0
```

D.3.7 Lpp.deinst

```
#!/bin/ksh
#
# script to de-install ricdd
#

# Make sure the user wants this !
dspmsg Program.cat -s 6 6 "Are you sure? (y/n)[n]"
read x < /dev/tty
case $x in
        y*|Y*) dspmsg Program.cat -s 6 7 "Deinstalling Program ...\n"
                continue ;;
        *) dspmsg Program.cat -s 6 8 "Quitting without de-installing Program.\n"
           exit 1 ;;
esac

final_msg= dspmsg Program.cat -s 6 9 "Program deinstalled successfully.\n"

trap ":" 0 1 2 3

#
# First try to de-install ricdd.src
#

# check if ricdd.src is installed
odmget -q "name = 'ricdd.src'" lpp | grep name > /dev/null
RC=$?
if [ $RC -ne 0 ]
then
   # ricdd.src not installed.
   echo ricdd.src not installed.
   continue
else
   # OK let's de-install ricdd.src
   echo De-installing ricdd.src...

   cd /usr/lpp/ricdd

   # remove the src dir
   rm -rf src

   # remove odm entry
   odmdelete -o lpp -q "name = 'ricdd.src'" 2>/dev/null 1>/dev/null
fi
```

```
#
# Then try to de-install ricdd.driver
#

# check if ricdd.driver is installed
odmget -q "name = 'ricdd.driver'" lpp | grep name > /dev/null
RC=$?
if [ $RC -ne 0 ]
then
   # ricdd.driver not installed.
   echo ricdd.driver not installed.
   continue
else
   # OK let's de-install ricdd.driver
   echo De-installing ricdd.driver...

   cd /usr/lpp

   # remove the device driver
   rm -f /etc/drivers/ricdd

   # remove the two configuration methods
   rm -f /etc/methods/cfgrica
   rm -f /etc/methods/cfgricp

   # remove the catalogue
   rm -f /usr/lpp/msg/$LANG/ric.cat
   rm -f /usr/lpp/msg/C/ric.cat

   #
   # remove odm entries
   #

   # in lpp object class
   odmdelete -o lpp -q "name = 'ricdd.driver'"  2>/dev/null 1>/dev/null

   # in Predefined Devices object class
   odmdelete -o PdDv -q "uniquetype = 'adapter/mca/ric'" 2>/dev/null 1>/dev/null
   odmdelete -o PdDv -q "uniquetype = 'ricport/ricp/port'" 2>/dev/null 1>/dev/null

   # in Predefined Attributes object class
   odmdelete -o PdAt -q "uniquetype = 'adapter/mca/ric'" 2>/dev/null 1>/dev/null
   odmdelete -o PdAt -q "uniquetype = 'ricport/ricp/port'" 2>/dev/null 1>/dev/null

   # in Predefined Connections object class
   odmdelete -o PdCn -q "uniquetype = 'adapter/mca/ric'" 2>/dev/null 1>/dev/null

   # in SMIT object classes
   odmdelete -o sm_menu_opt -q "next_id = 'ric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_menu_opt -q "next_id = 'lsdric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_menu_opt -q "next_id = 'makric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_menu_opt -q "next_id = 'movric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_menu_opt -q "next_id = 'chgric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_menu_opt -q "next_id = 'rmvric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_menu_opt -q "next_id = 'cfgric'" 2>/dev/null 1>/dev/null

   odmdelete -o sm_name_hdr -q "id = 'cfgric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_name_hdr -q "id = 'chgric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_name_hdr -q "id = 'makric'" 2>/dev/null 1>/dev/null
```

```
   odmdelete -o sm_name_hdr -q "id = 'movric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_name_hdr -q "id = 'movric_parent'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_name_hdr -q "id = 'rmvric'" 2>/dev/null 1>/dev/null

   odmdelete -o sm_cmd_hdr -q "id = 'cfgric_hdr'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_hdr -q "id = 'chgric_hdr'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_hdr -q "id = 'lsdric'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_hdr -q "id = 'makric_hdr'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_hdr -q "id = 'movric_hdr'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_hdr -q "id = 'rmvric_hdr'" 2>/dev/null 1>/dev/null

   odmdelete -o sm_cmd_opt -q "id = 'cfgric_opt'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_add'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_chg'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_common'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_ln_opt'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_mk_parent'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_mv'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'ric_mv_parent'" 2>/dev/null 1>/dev/null
   odmdelete -o sm_cmd_opt -q "id = 'rmvric_opt'" 2>/dev/null 1>/dev/null
fi

# now we can delete /usr/lpp/ricdd
rm -rf /usr/lpp/ricdd

echo $final_msg

# always succeed
exit 0
```

Appendix E. Sample Character Device Driver

E.1 Device Driver Main Body

```
1  /***************************************************************************/
2  /*  Sample Device Driver for the "Writing a Device Driver" Redbook          */
3  /*                                                                         */
4  /*      Name: ricdd.c                                                      */
5  /*                                                                         */
6  /*      Device Type: Realtime Interface Co-Processor                       */
7  /*                   with an optional interface board containing           */
8  /*                   eight serial communications ports, each capable       */
9  /*                   of operating at full duplex, independent of           */
10 /*                   each other.  Each of the ports can support            */
11 /*                   various electrical interfaces and protocols.          */
12 /*                                                                         */
13 /*      Entry Points: ricopen, ricclose, ricioctl, ricintr,                */
14 /*                    ricconfig, ricread, ricwrite, ricmpx,                */
15 /*                    ricselect                                            */
16 /*                                                                         */
17 /***************************************************************************/
18 #include <fcntl.h>              /* logical file system */
19 #include <sys/device.h>         /* dev switch table    */
20 #include <sys/uio.h>            /* Details of user's I/O request */
21 #include <sys/mdio.h>           /* Macros for accessing I/O hardware */
22 #include <sys/ioacc.h>          /* Macros for accessing I/O hardware */
23 #include <sys/sleep.h>          /* Post/wait and other routines */
24 #include <sys/ldr.h>            /* Kernel extension loader  */
25 #include <sys/pin.h>            /* Pin/unpin calls */
26 #include <sys/malloc.h>         /* Memory allocation routines */
27 #include <sys/pri.h>            /* Interrupt priorities */
28 #include <sys/poll.h>           /* Select */
29 #include <sys/lockl.h>          /* preemption synchronization */
30 #include <sys/errno.h>          /* error codes  */
31 #include <sys/dma.h>            /* dma external interface definition */
32 #include <sys/ioctl.h>          /* ioctl definitions    */
33 #include <sys/intr.h>           /* interrupt services interface def     */
34 #include "ric.h"                /* ric driver defines    */
35 #include "ricmisc.h"            /* miscellaneous structs and defs       */
36 #include "ricstruct.h"          /* device structs       */
37 #include <sys/syspest.h>        /* kernel debug macros  */
38 #include <sys/watchdog.h>       /* watchdog timers */
39 #include <sys/iocc.h>           /* IOCC register file mapping    */
40 #include <sys/adspace.h>        /* Address space manipulation */
41 #include <sys/comio.h>          /* common communications code */
42 #include <sys/devinfo.h>        /* IO structure definitions     */
43
44 int ricconfig();
45 int ricopen();
46 int ricclose();
47 int ricread();
48 int ricwrite();
49 int ricioctl();
50 int ricintr();
51 int ricmpx();
52 int ricselect();
53 extern int nodev();
54
55 t_ric_dds        *dds_dir[];       /* DDS directory        */
56 t_acb            *acb_dir[];       /* ACB directory        */
57 extern void rictimer(t_ric_dds *);         /* handles expired timers */
58 static struct devsw    ricsw;  /* dev switch table entry for ric driver */
59 static int act_adap = 0;       /* number of active adapters    */
60 extern int stop_port();
61 extern int flush_port();
62
63 /***************************************************************************
64
65     ricconfig: performs operations necessary for the intitialisation
```

```
 66            of an individual port on the adapter.  ricconfig will be
 67            called for each valid port during the bus/device config
 68            phase of the boot procedure.
 69
 70  ***************************************************************************/
 71  int ricconfig(devno, cmd, uiop)
 72  dev_t devno;
 73  int cmd;
 74  struct uio *uiop;
 75  {
 76         int port_num;           /* port number  */
 77         int adapt_num;          /* adapter number */
 78         int minor_num;          /* minor device number  */
 79         t_ric_dds       *dds_ptr;/* pointer to DDS      */
 80         t_acb           *acb_ptr;/* pointer to ACB      */
 81         int ret;                /* return values       */
 82         unsigned long   bus_sr; /* IO Seg Reg number mask */
 83         unsigned long   iob;    /* io base address     */
 84         unsigned long   memb;   /* bus memory base     */
 85
 86      /* get minor number. macro defined in /usr/include/sys/sysmacros.h */
 87         minor_num = minor(devno);
 88
 89      /* if the minor number is bad, return */
 90         if (minor_num >= (MAX_ADAP*NUM_PORTS))
 91         {
 92            return(EINVAL);
 93         }
 94
 95       /* get a DDS pointer    */
 96         dds_ptr = dds_dir[minor_num];
 97
 98     switch(cmd)             /* switch on command type */
 99     {
100      /* initialise device driver and internal data areas        */
101         case CFG_INIT:
102         {
103
104             /* first check whether dds exists      */
105                 if (dds_ptr != (t_ric_dds *)NULL)
106                 {
107                         return(EINVAL);
108                 }
109
110         /* now, if this is the first time through CFG_INIT, certain
111          * things must be done. no active adapters means first time
112          */
113                 if (act_adap == 0)
114                 {
115                      /* pin ric into memory  */
116                         if (( ret = pincode(ricconfig)) !=0)
117                         {
118                              /* return if pin fails     */
119                                 return(ret);
120                         }
121                      /* ok, so now it is pinned      */
122
123                  /* add entry points to the devsw table  */
124
125                         ricsw.d_open     = ricopen;
126                         ricsw.d_close    = ricclose;
127                         ricsw.d_read     = ricread;
128                         ricsw.d_write    = ricwrite;
129                         ricsw.d_ioctl    = ricioctl;
130                         ricsw.d_strategy = nodev;
131                         ricsw.d_ttys     = NULL;
132                         ricsw.d_select   = ricselect;
133                         ricsw.d_config   = ricconfig;
134                         ricsw.d_print    = nodev;
135                         ricsw.d_dump     = nodev;
136                         ricsw.d_mpx      = ricmpx;
137                         ricsw.d_revoke   = nodev;
138                         ricsw.d_dsdptr   = NULL;
139                         ricsw.d_selptr   = NULL;
140                         ricsw.d_opts     = 0;
```

```
141
142         /* if adding the entry points to devsw fails, return    */
143                     if((ret = devswadd(devno, &ricsw)) != 0)
144                     {
145                             unpincode(ricconfig);
146                             return(ret);
147                     }
148             } /* end first time through */
149          /* For this example we are allocating pinned space and  */
150          /* then we will copy the dds data structure */
151          /* allocate space for dds */
152                 dds_ptr = (t_ric_dds *)xmalloc (sizeof(t_ric_dds),
153                             2, pinned_heap);
154
155          /* if the xmalloc fails, return */
156                 if(dds_ptr == (t_ric_dds *)NULL)
157                 {
158                         free_it_up(act_adap, devno, NULL, NULL);
159                         return(ENOMEM);
160                 }
161
162          /* zero out dds */
163                 bzero((char *)dds_ptr, sizeof(t_ric_dds));
164
165          /* copy input struct into dds */
166                 ret = uiomove(dds_ptr, sizeof(t_ric_dds), UIO_WRITE,
167                                 uiop);
168
169          /* if uiomove is bad */
170                 if(ret)
171                 {
172                         free_it_up(act_adap, devno, dds_ptr, NULL);
173                         return(ret);
174                 }
175
176          /* set port number from dds */
177                 port_num = dds_ptr->dds_dvc.port_num;
178
179          /* adapter number is slot number */
180                 adapt_num = dds_ptr->dds_hdw.slot_num;
181                 acb_ptr = acb_dir[adapt_num];
182
183           /* if no ACB for this device */
184                 if(acb_ptr == (t_acb *)NULL)
185                 {
186                         /* allocate memory for the acb */
187                         acb_ptr = (t_acb *)xmalloc(sizeof(t_acb),
188                                     2, pinned_heap);
189
190                         /* if the allocation fails */
191                         if(acb_ptr == (t_acb *)NULL)
192                         {
193                                 free_it_up(act_adap, devno, dds_ptr,
194                                               NULL);
195                                 return(ENOMEM);
196                         }
197
198                         /* zero out acb */
199                         bzero((char *)acb_ptr, sizeof(t_acb));
200
201                         /* now fill it in */
202                         acb_ptr->p_port_dds[port_num] = dds_ptr;
203
204                 /* now set up the POS register settings */
205                         acb_ptr->int_lvl = dds_ptr->dds_hdw.bus_intr_lvl;
206                         acb_ptr->slot_num = (unsigned
207                                                 char)(dds_ptr->dds_hdw.slot_num);
208                         acb_ptr->arb_lvl = dds_ptr->dds_hdw.dma_lvl;
209                         acb_ptr->io_base = dds_ptr->dds_hdw.bus_io_addr;
210                         acb_ptr->mem_base = dds_ptr->dds_hdw.bus_mem_addr;
211                         acb_ptr->dma_base = dds_ptr->dds_hdw.tcw_bus_mem_addr;
212                         acb_ptr->io_segreg_val = IO_SEG_REG;
213                         acb_ptr->adapter_state = 0;
214                         acb_ptr->cpu_page = 0xFF;
215
```

```
216             /* invoke set_POS to set POS registers */
217             set_POS( acb_ptr );
218
219             /* set up segment register for next phase */
220             bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val, 0);
221
222             /* set up the busio and bus memory base address for the card */
223             iob = acb_ptr->io_base + bus_sr;
224             memb = acb_ptr->mem_base + bus_sr;
225             ret = reset_card ( acb_ptr, bus_sr, iob, memb);
226
227             /* free up segment register */
228             BUSIO_DET(bus_sr);
229
230             if /* reset failed... */
231                ( ret )
232             {
233                 free_it_up(act_adap, devno, dds_ptr, acb_ptr);
234                 return(EIO);
235             }
236
237             acb_ptr->c_intr_rcvd = 0;     /* zero interrupt count  */
238
239             /* now we set up our DMA channel by calling d_init */
240             acb_ptr->dma_channel_id =
241                     d_init((int)acb_ptr->arb_lvl, MICRO_CHANNEL_DMA,
242                             acb_ptr->io_segreg_val);
243
244             /* free up resources if d_init failed */
245             if (acb_ptr->dma_channel_id == DMA_FAIL)
246             {
247                 free_it_up(act_adap, devno, dds_ptr, acb_ptr);
248                 return(EIO);
249             }
250
251             /* enable DMA channel */
252             d_unmask( acb_ptr->dma_channel_id);
253
254             act_adap++;                         /* adapter is now active */
255             acb_dir[adapt_num] = acb_ptr;
256
257         } /* end of no existing acb if */
258
259         acb_ptr->n_cfg_ports++;
260         acb_ptr->p_port_dds[port_num] = dds_ptr;
261         dds_dir[minor_num] = dds_ptr;
262         break;
263
264     } /* end case CFG_INIT  */
265
266  /* terminate the device driver associated with the specified devno */
267     case CFG_TERM:
268     {
269         if (dds_ptr == NULL)
270                 return(EACCES);
271
272         if (dds_ptr->dds_dvc.port_state != CLOSED)
273                 return(EBUSY);
274
275         port_num = dds_ptr->dds_dvc.port_num;
276         adapt_num = dds_ptr->dds_hdw.slot_num;
277         acb_ptr = acb_dir[dds_ptr->dds_hdw.slot_num];
278
279         /* decrement number of configured ports on this adapter */
280         acb_ptr->n_cfg_ports--;
281
282         /* if last configured port on adapter, free adapter resources */
283         if (acb_ptr->n_cfg_ports == 0)
284         {
285                 /* Release the dma_channel */
286                 d_mask(acb_ptr->dma_channel_id);
287                 d_clear(acb_ptr->dma_channel_id);
288
289                 /* decrement number of active adapters */
290                 act_adap--;
```

```
291
292                     free_it_up(act_adap, devno, dds_ptr, acb_ptr);
293                     acb_dir[adapt_num] = (t_acb *)NULL;
294             }
295             else
296             {
297                     /* free up allocated resources. If number        */
298                     /* of active adapters now zero,                  */
299                     /* delete switch table entry and unpin the driver */
300                     free_it_up(act_adap, devno, dds_ptr, NULL);
301                     acb_ptr->p_port_dds[port_num] = NULL;
302             }
303
304             dds_dir[minor_num] = NULL;
305             break;
306         } /* end case CFG_TERM */
307
308       /*  query device specific VPD          */
309         case CFG_QVPD:
310                 break;
311
312         default:
313                 return(EINVAL);
314     } /* end switch statement */
315     return(0);
316 } /* end ricconfig */
317
318 /****************************************************************************
319  *
320  *      ricmpx is the mpx entry point to allocate or deallocate a
321  *      channel.
322  *
323  ***************************************************************************/
324 ricmpx(devno, chanp, channame)
325 dev_t devno;
326 int *chanp;
327 char *channame;
328 {
329     t_acb               *acb_ptr;         /* pointer to ACB */
330  /* ACB is the adapter control block. There is one ACB for each  */
331  /* adapter in the system */
332     t_ric_dds           *dds_ptr;         /* pointer to DDS */
333     int                 tmp_chan;       /* local chan storage */
334
335
336   /* if minor number is bad, return  */
337     if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
338     {
339         return(EINVAL);
340     }
341   /* Note: in our sample program, a port on the RIC will be allocated if */
342   /* the minor device number that is passed in has not been previously   */
343   /* allocated a port. (port 0 is always allocated here) Whatever process */
344   /* opens the port totally owns the port until a ricmpx call is made to */
345   /* deallocate that port. */
346
347   /* set up DDS pointer */
348     dds_ptr = dds_dir[minor(devno)];
349
350   /* if dds pointer is null, return error */
351     if (dds_ptr == NULL)
352         return(EINVAL);
353
354   /* get the acb pointer */
355     acb_ptr = acb_dir[dds_ptr->dds_hdw.slot_num];
356
357   /* see if we've been called to deallocate the channel */
358     if ( channame == (char *)NULL )
359     {
360       /* Deallocate the channel */
361         dds_ptr->dds_wrk.cur_chan_num = 0;
362
363       /* on a deallocate, always set diag flag to 0 */
364         acb_ptr->diag_flag = 0;
365     }
```

```
366         else
367         {
368           /* get channel allocated indicator */
369             tmp_chan = (int)dds_ptr->dds_wrk.cur_chan_num;
370
371           /* if channel number already allocated, return error */
372             if (tmp_chan > 0)
373             {
374                 return(ENXIO);
375             }
376
377               /* not diagnostics open */
378                 acb_ptr->diag_flag = 0;
379
380             dds_ptr->dds_wrk.cur_chan_num = 1;        /* allocate channel 0 */
381             *chanp = 0;                            /* channel returned is 0 */
382         }
383         return(0);
384     } /* end ricmpx */
385
386     /***************************************************************************
387      *
388      *      ricopen sets up the interrupt and dma services, as well as
389      *      checking that everything is in order for an open to occur
390      *
391      ***************************************************************************/
392     ricopen(devno, devflag, mpxchan, ext_ptr)
393     dev_t devno;
394     ulong devflag;
395     int mpxchan;
396     struct kopen_ext *ext_ptr;
397     {
398             int     ricintr();      /* interrupt handler    */
399             int     ricoffl();      /* offlevel     */
400             int port_num;           /* port number  */
401             int adapt_num;          /* adapter number */
402             int ilev;               /* adapter interrupt level      */
403             int old_pri;            /* interrupt level      */
404             int counter;            /* loop control counter */
405             struct intr     *intr_ptr;/* interrupt pointer   */
406             t_sel_que       *sqelm1_ptr;/* select queue element pointer     */
407             t_sel_que       *sqelm2_ptr;/* select queue element pointer     */
408             t_chan_info     *tmp_chnptr;/* temp channel info pointer       */
409             t_ric_dds       *dds_ptr;/* pointer to DDS      */
410             t_acb           *acb_ptr;/* pointer to ACB      */
411             int ret;                /* return values        */
412             unsigned long   bus_sr; /* IO Seg Reg number mask */
413             unsigned char   io_ptr; /* io base pointer      */
414             unsigned char   comreg; /* COMREG on Portmaster */
415
416           /* if minor number is bad, return  */
417             if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
418             {
419                     return(EINVAL);
420             }
421
422           /* if the channel number out of range, return */
423           /* Note that we are not really a multiplexed device  */
424             if ( mpxchan != 0 )
425             {
426                     return(ECHRNG);
427             }
428
429           /* get dds pointer from dds directory */
430             dds_ptr = dds_dir[minor(devno)];
431
432           /* if port not configured, return error */
433             if (dds_ptr == NULL)
434             {
435                     return(EINVAL);
436             }
437
438             adapt_num = dds_ptr->dds_hdw.slot_num;
439             acb_ptr = acb_dir[adapt_num];
440
```

```
441         port_num = dds_ptr->dds_dvc.port_num;
442
443      /* check to see whether any ports have been opened on
444       * the indicated adapter.  If not, register the
445       * interrupt handler and fill in the off level
446       * interrupt structures.
447       */
448      /* no registration has occured for this adapter */
449         if(acb_ptr->n_open_ports == 0)
450         {
451
452              /* first initialise the offlevel intr structures */
453                 acb_ptr->arq_sched = FALSE;
454                 acb_ptr->offl.p_acb_intr = (struct t_acb *)acb_ptr;
455                 intr_ptr = &(acb_ptr->offl.offl_intr);
456                 INIT_OFFL3(intr_ptr, ricoffl, IO_SEG_REG);
457
458                 acb_ptr->slih_intr.next = NULL;
459                 acb_ptr->slih_intr.handler = ricintr;
460                 acb_ptr->slih_intr.bus_type = BUS_MICRO_CHANNEL;
461                 acb_ptr->slih_intr.flags = 0;
462                 acb_ptr->slih_intr.level = acb_ptr->int_lvl;
463                 acb_ptr->slih_intr.priority = INTCLASS1;
464                 acb_ptr->slih_intr.bid = IO_SEG_REG;
465
466                 acb_ptr->cmd_queue_lock = LOCK_AVAIL;
467
468               /* registration of interrupt handler fails */
469                 if((ret = i_init(&acb_ptr->slih_intr)) != 0)
470                 {
471                         return(ENXIO);
472                 }
473
474
475             /* enable interrupts on the adapter */
476                 bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val, 0);
477
478                 io_ptr = (unsigned char *)( acb_ptr->io_base + bus_sr );
479
480                 comreg = PIO_GETC( io_ptr + COMREG );
481
482                 PIO_PUTC( io_ptr + COMREG, comreg | COM_IE );
483
484                 BUSIO_DET( bus_sr );
485
486         } /* end of no open ports loop  */
487
488       /* first time through successfully, allocate channel structure */
489         if(dds_ptr->dds_wrk.p_chan_info[mpxchan] == NULL)
490         {
491               /* allocate memory for channel related structures */
492                 dds_ptr->dds_wrk.p_chan_info[mpxchan] = tmp_chnptr =
493                     (t_chan_info *)xmalloc((uint)sizeof(t_chan_info),(uint)2,
494                                         pinned_heap);
495
496               /* memory allocation failed, return */
497                 if( tmp_chnptr == NULL)
498                 {
499                         return(ENOMEM);
500                 }
501
502                 bzero((void *)tmp_chnptr, (uint)sizeof(t_chan_info));
503
504              /* set major/minor device number */
505                 tmp_chnptr->devno = devno;
506                 tmp_chnptr->rcv_event_lst = EVENT_NULL;
507                 tmp_chnptr->xmt_event_lst = EVENT_NULL;
508                 acb_ptr->txfl_event_lst = EVENT_NULL;
509
510         }
511
512      /* now fetch the temporary channel info pointer */
513         tmp_chnptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
514
515      /* set common values for user and kernel llc calls */
```

```
516         tmp_chnptr->devflag = devflag; /* device flags opened with */
517
518     /* set port state variable to open */
519         dds_ptr->dds_dvc.port_state = OPEN;
520
521     /* increment number of open ports */
522         acb_ptr->n_open_ports++;
523
524         return(0);
525 } /* end ricopen */
526
527 /***************************************************************************
528  *
529  *      ricclose closes a single port.
530  *
531  ***************************************************************************/
532 ricclose(devno, mpxchan, ext)
533 dev_t devno;
534 int mpxchan;
535 int ext;
536 {
537         int             adapt_num;      /* adapter number */
538         int             port_num;       /* port number  */
539         t_acb           *acb_ptr;         /* pointer to ACB */
540         t_chan_info     *tmp_chanptr;  /* temp channel info pointer   */
541         t_ric_dds       *dds_ptr;         /* pointer to DDS */
542         unsigned int    ret;             /* return values */
543         int             old_pri;        /* interrupt level      */
544         unsigned long   bus_sr;         /* bus segment reg      */
545         unsigned char   *io_ptr;          /* pointer to io reg    */
546         unsigned char   comreg;         /* COMREG on ric     */
547         unsigned int    sleep_flag;     /* que_cmd sleep flag   */
548
549      /* if minor number is invalid, return error */
550         if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
551         {
552                 return(EINVAL);
553         }
554
555      /* if the channel number out of range, return */
556         if ( mpxchan != 0 )
557         {
558                 return(ECHRNG);
559         }
560
561      /* get dds pointer from dds directory */
562         dds_ptr = dds_dir[minor(devno)];
563
564      /* if port not configured, return error */
565         if (dds_ptr == NULL)
566         {
567                 return(EINVAL);
568         }
569
570         adapt_num = dds_ptr->dds_hdw.slot_num;
571         acb_ptr = acb_dir[adapt_num];
572
573         port_num = dds_ptr->dds_dvc.port_num;
574
575      /*   remove the select queue data structure, the channel
576       *   information data structure and zero out the dds pointer
577       *   to the channel ds
578       */
579
580         tmp_chanptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
581
582      /* remove device flags */
583         tmp_chanptr->devflag = 0;
584
585      /* last close for this adapter. notify kernel the adapter
586       *is no longer generating interrupts
587       */
588         if (--acb_ptr->n_open_ports == 0)
589         {
590              /* First disable interrupts from the adapter. */
```

```
591                 bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val,0);
592
593                 io_ptr = (unsigned char *)( acb_ptr->io_base + bus_sr);
594
595                 comreg = PIO_GETC( io_ptr + COMREG );
596
597                 PIO_PUTC(io_ptr + COMREG, comreg &  COM_IE );
598
599                 BUSIO_DET(bus_sr);
600
601                 i_clear(&acb_ptr->slih_intr);
602         }
603
604      /* set port state to closed */
605         dds_ptr->dds_dvc.port_state = CLOSED;
606
607         return(0);
608 } /* end ricclose */
609
610 /***************************************************************************
611  *
612  *      ricread reads the adapter
613  *
614  ***************************************************************************/
615 ricread(devno, uiop, mpxchan, rdext_ptr)
616 dev_t devno;
617 struct uio  *uiop;
618 int mpxchan;
619 struct read_extension  *rdext_ptr;
620 {
621         int  adapt_num;      /* adapter number  */
622         int  port_num;       /* port number  */
623         int  old_pri;                 /* interrupt level */
624         u_short         pkt_hdr_len;    /* packet header length        */
625         u_short         pkt_length;     /* receive data length */
626         u_short         pkt_status;     /* receive packet status */
627         t_acb           *acb_ptr;         /* pointer to ACB */
628         t_ric_dds      *dds_ptr;         /* pointer to DDS */
629         struct mbuf     *mbuf_ptr;        /* pointer to  mbuf */
630         caddr_t         p_pkt;          /* pointer to the received packet */
631         u_short         *p_shrt_pkt;    /* pointer to the received packet */
632         t_sel_que       *p_rcv_elem;    /* pointer to the receive entry */
633         volatile t_chan_info *tmp_chnptr; /* temp channel info pointer */
634         int             ret;             /* return code */
635         int             sleep_ret;       /* return code from e_sleep */
636
637     /* if minor number is invalid, return error */
638         if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
639         {
640                 return(EINVAL);
641         }
642
643     /* if the channel number out of range (only 0 is valid for now) */
644         if ( mpxchan != 0 )
645         {
646                 return (ECHRNG);
647         }
648
649     /* get dds pointer from dds directory */
650         dds_ptr = dds_dir[minor(devno)];
651
652     /* if port not configured, return error */
653         if (dds_ptr == NULL)
654         {
655                 return(ENXIO);
656         }
657
658         adapt_num = dds_ptr->dds_hdw.slot_num;
659         acb_ptr = acb_dir[adapt_num];
660
661         port_num = dds_ptr->dds_dvc.port_num;
662
663         /*
664          *  go get the channel information data struct pointer from
665          *  the DDS.
```

```
666              */
667             tmp_chnptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
668
669         /* disable interrupts to single thread */
670             old_pri = i_disable(INTOFFL3);
671
672         /* no packets are available on the queue */
673             while( tmp_chnptr->p_rcv_head == NULL)
674             {
675                 /* DNDELAY set, return at once */
676                     if( tmp_chnptr->devflag & DNDELAY )
677                     {
678                         /* end single thread */
679                             i_enable(old_pri);
680
681                         /* set length to zero */
682                             uiop->uio_resid = 0;
683
684                         /* no data, return zero */
685                             return(0);
686                     }
687                     else
688               /* NDELAY not set, wait until data is received */
689                     {
690                         /* do an e_sleep */
691                             sleep_ret = e_sleep(&(tmp_chnptr->rcv_event_lst),
692                                                    EVENT_SIGRET);
693
694                             if ( sleep_ret != EVENT_SUCC )
695                             {
696                                     i_enable( old_pri );
697                                     return( EINTR );
698                             }
699                     }
700             }
701         /*
702          * message waiting. deque it and copy to user's buffer
703          */
704         /* point to first element */
705             p_rcv_elem = tmp_chnptr->p_rcv_head;
706
707         /* copy the code field to the status field of read extension */
708             if ( rdext_ptr != NULL)
709             {
710                     rdext_ptr->status = (ulong) p_rcv_elem->stat_block.code;
711             }
712
713             tmp_chnptr->p_rcv_head = p_rcv_elem->p_sel_que; /* deque it */
714
715         /* get mbuf pointer */
716             mbuf_ptr = (struct mbuf *)p_rcv_elem->stat_block.option[0];
717
718         /* receive head ptr is null, make receive tail ptr null */
719             if(tmp_chnptr->p_rcv_head == NULL)
720             {
721                     tmp_chnptr->p_rcv_tail = NULL;
722             }
723
724         /*
725          * zero out the select queue element and add it back
726          * to the select queue available chain
727          */
728             p_rcv_elem->rqe_value = 0;
729             p_rcv_elem->stat_block.code = 0;
730             p_rcv_elem->stat_block.option[0] = 0;
731             p_rcv_elem->p_sel_que = tmp_chnptr->p_sel_avail;
732             tmp_chnptr->p_sel_avail = p_rcv_elem;
733
734             i_enable(old_pri);
735
736         /* if mbuf_ptr is NULL, there is a status, not a receive buffer */
737             if (mbuf_ptr ==  NULL)
738             {
739                     return (0);
740             }
```

```
741
742       /* get buffer address */
743          p_pkt = MTOD(mbuf_ptr, caddr_t);
744
745          p_shrt_pkt = (u_short *)p_pkt;
746
747       /* get information from packet header */
748          pkt_hdr_len = PIO_GETSR(p_shrt_pkt++);
749          pkt_length = PIO_GETSR(p_shrt_pkt++);
750          pkt_status = PIO_GETSR(p_shrt_pkt);
751
752       /* point packet address to start past header */
753          p_pkt = p_pkt + pkt_hdr_len;
754
755       /* attempt to move the packet contents to the user area */
756          ret = uiomove(p_pkt, (unsigned int)pkt_length, UIO_READ, uiop);
757
758       /* free the mbuf */
759          m_free( mbuf_ptr );
760
761          return(ret);
762
763   }  /* end ricread */
764
765   /*************************************************************************
766    *
767    *       ricwrite allows write or transmit for user level or kernel
768    *       level users of the ric.
769    *
770    *************************************************************************/
771   ricwrite(devno, uiop, mpxchan, ext_ptr, sleep_flag)
772   dev_t  devno;
773   struct uio *uiop;
774   int mpxchan;
775   t_write_ext *ext_ptr;
776   unsigned int sleep_flag;
777
778   {
779           int  adapt_num;       /* adapter number */
780           int  port_num;        /* port number    */
781           t_acb  *acb_ptr;      /* pointer to ACB */
782           t_ric_dds  *dds_ptr; /* pointer to DDS */
783           t_write_ext     lc_ext; /* local copy of write extension */
784           int  data_len;        /* total length of chained mbuf */
785           unsigned short  lc_flags;      /* local copy of flag bits */
786           unsigned short  lc_seq_num;
787           unsigned short  lc_xmt_length;
788           char            *lc_bus_buf;
789           char            *lc_bus_base;
790           char            *lc_host_buf;
791           struct mbuf     *lc_xmt_mbuf;
792           unsigned int    old_pri;   /* interrupt priority save element */
793           t_xmt_chain     *xchn_ptr; /* pointer to the xmit chain */
794           t_xmt_map       *xmap_ptr; /* pointer to current xmit map */
795           struct mbuf     *mbuf_ptr;         /* pointer to the mbuf */
796           struct mbuf     *freembuf_ptr;     /* pointer to mbuf to free */
797           struct mbuf     *freembufc_ptr; /* ptr to mbuf chain to free */
798           struct mbuf     *allocmbuf_ptr;   /* mbuf allocated by us */
799           unsigned char   *mbufdata_ptr;   /* pointer to mbuf data to be sent */
800           struct mbuf     *tmpmbuf_ptr;    /* temp pointer to mbuf */
801           int             ret;             /* return code */
802           struct xmem     xmd;            /* cross memory descriptor for dma */
803           t_adap_cmd      xmt_adap_cmd;   /* on stack adapter command buffer */
804           unsigned char   tmp_cntrl;      /* temp var for filling in cmd blk */
805
806    /* if minor number is bad, return error */
807       if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
808       {
809           return(EINVAL);
810       }
811
812    /* if the channel number out of range, return */
813       if ( mpxchan != 0 )
814       {
815           return (ECHRNG);
```

```
816        }
817
818     /* get dds pointer from dds directory */
819        dds_ptr = dds_dir[minor(devno)];
820
821     /* if port not configured, return error */
822        if (dds_ptr == NULL)
823        {
824            return(EINVAL);
825        }
826
827        adapt_num = dds_ptr->dds_hdw.slot_num;
828        acb_ptr = acb_dir[adapt_num];
829
830        port_num = dds_ptr->dds_dvc.port_num;
831
832     /* initialize local mbuf pointers */
833        freembuf_ptr = NULL;
834        freembufc_ptr = NULL;
835        allocmbuf_ptr = NULL;
836
837        bzero( ( char *)&xmt_adap_cmd , sizeof(t_adap_cmd));
838
839        /*  if write extension provided, copyin if from user space.
840         *  else copy directly (bcopy) if from kernel space.
841         */
842        bzero( &lc_ext, sizeof( t_write_ext ));
843        if ( ext_ptr )
844            if ( uiop->uio_segflg == UIO_USERSPACE )
845                copyin( ext_ptr, &lc_ext, sizeof( t_write_ext ));
846            else
847                bcopy ( ext_ptr, &lc_ext, sizeof( t_write_ext ));
848
849     /* initialize local flags */
850        if ( lc_ext.cio_write.flag & CIO_ACK_TX_DONE ) {
851            lc_flags = XMT_STAT_REQ;
852        }
853        else
854        {
855            lc_flags = 0;
856        }
857
858     /* get pointer to transmit chain */
859        xchn_ptr = dds_ptr->dds_wrk.p_xmt_chn;
860
861     /* if no available transmit map elements, then return */
862        if((xchn_ptr->elts_in_use +1 ) >= xchn_ptr->length)
863        {
864            return(EAGAIN);
865        }
866
867     /* a user process called the write */
868        if( uiop->uio_segflg == UIO_USERSPACE )
869        {
870            lc_xmt_length = (unsigned int)uiop->uio_resid;
871
872          /* data length is 48 bytes or less */
873            if( lc_xmt_length <= 48 )
874            {
875             /* do uiomove to get data into command block */
876                if((ret = uiomove(&(xmt_adap_cmd.u_data_area.d_ovl.data[0]),
877                               uiop->uio_resid, UIO_WRITE, uiop)) != 0)
878                {
879                 /* uiomove failed, return an error */
880                    return(ret);
881                }
882            } /* end of transmit <= 48 bytes */
883            else
884            {
885              /* if request for more than one page, return */
886                if( lc_xmt_length > PAGESIZE )
887                {
888                    return(EINVAL);
889                }
890
```

```
891             /* allocate an mbuf and copy the data into it */
892               mbuf_ptr = m_get( M_DONTWAIT, MT_DATA);
893
894              /* if no mbuf available, return */
895               if( mbuf_ptr == (struct mbuf *)NULL )
896               {
897                   return(ENOMEM);
898               }
899
900            /* try to get an mbuf cluster */
901               m_clget(mbuf_ptr);
902
903            /* no mbuf clusters available */
904               if(!M_HASCL(mbuf_ptr))
905               {
906                   m_free(mbuf_ptr);
907                   return(ENOMEM);
908               }
909
910            /* save pointer to mbuf */
911               allocmbuf_ptr = mbuf_ptr;
912
913               /* set local flags */
914               lc_flags |= (XMT_FREE_MBUF |        /* mbuf to be freed */
915                            XMT_DMA_REQ);          /* will be doing dma */
916
917            /* now get a pointer to the actual data */
918               mbufdata_ptr = MTOD(mbuf_ptr, char *);
919
920            /* now do uiomove to get data into mbuf or mbuf extension */
921             if((ret = uiomove(mbufdata_ptr, uiop->uio_resid, UIO_WRITE,
922                               uiop)) != 0)
923              {
924               /* uiomove failed, free the mbuf and return  */
925                   m_free(mbuf_ptr);
926                   return(ret);
927               }
928           }
929       }
930
931       if (lc_ext.transparent)
932               tmp_cntrl = (ADAP_TX_ACK | ADAP_TRANSP);
933       else
934               tmp_cntrl = ADAP_TX_ACK;
935
936       lc_seq_num = ++dds_ptr->dds_wrk.cmd_seq_num;
937
938    /* need to do a DMA */
939       if(lc_flags & XMT_DMA_REQ)
940       {
941        /* will be doing a XMIT_LONG command */
942
943        /* already running max number of dma's */
944           if(xchn_ptr->num_active_dma >= XMT_TCWS_PORT)
945           {
946               if (allocmbuf_ptr)
947                   m_free(allocmbuf_ptr);
948               return( EAGAIN );
949           }
950
951           lc_xmt_mbuf = mbuf_ptr;
952           lc_host_buf = MTOD(mbuf_ptr, char *);
953           lc_bus_base = reg_alloc ( dds_ptr->dds_wrk.p_reg_list, PAGESIZE);
954           lc_bus_buf =lc_bus_base + ((unsigned int)lc_host_buf % PAGESIZE);
955
956           /* make the buffer visible to the adapter */
957           xmd.aspace_id = XMEM_GLOBAL;
958           xmd.subspace_id = NULL;
959           d_master(acb_ptr->dma_channel_id, DMA_WRITE_ONLY, lc_host_buf,
960                   lc_xmt_length, &xmd, lc_bus_buf);
961
962           /* fill in command block */
963           xmt_adap_cmd.cmd_typ = XMIT_LONG;
964           xmt_adap_cmd.port_nmbr = (unsigned char)port_num;
965           xmt_adap_cmd.seq_num = SWAPSHORT(lc_seq_num);
```

```
966             xmt_adap_cmd.u_data_area.c_ovl.tst_length =
967                                             SWAPSHORT(lc_xmt_length);
968             xmt_adap_cmd.u_data_area.c_ovl.tst_addr =
969                                             SWAPLONG((unsigned int)lc_bus_buf);
970             xmt_adap_cmd.u_data_area.c_ovl.cntl = tmp_cntrl;
971         }
972         else
973         {
974          /* will be doing a XMIT_SHORT command */
975             lc_xmt_mbuf = NULL;
976             lc_host_buf = NULL;
977             lc_bus_base = NULL;
978             lc_bus_buf = NULL;
979
980          /* fill in command block */
981             xmt_adap_cmd.cmd_typ = XMIT_SHORT;
982             xmt_adap_cmd.port_nmbr = (unsigned char)port_num;
983             xmt_adap_cmd.seq_num = SWAPSHORT(lc_seq_num);
984             xmt_adap_cmd.lngth = (unsigned char)lc_xmt_length;
985             xmt_adap_cmd.cntrl = tmp_cntrl;
986         }
987
988     /* get pointer to next available transmit map element */
989         xmap_ptr = &(xchn_ptr->xmt_map_chn[(int)xchn_ptr->tail]);
990
991     /* fill it in */
992         xmap_ptr->seq_num = lc_seq_num;
993         xmap_ptr->xmt_elem_flags = lc_flags;
994         xmap_ptr->xmt_length = lc_xmt_length;
995         xmap_ptr->write_id = lc_ext.cio_write.write_id;
996         xmap_ptr->p_xmt_mbuf = lc_xmt_mbuf;
997         xmap_ptr->p_host_buf = lc_host_buf;
998         xmap_ptr->p_bus_base = lc_bus_base;
999         xmap_ptr->p_bus_buf = lc_bus_buf;
1000
1001      /* send the command down */
1002        old_pri = i_disable(INTOFFL3);
1003
1004      /* if unable to get available command block, return  */
1005        if((ret = que_command ( acb_ptr, &xmt_adap_cmd, sleep_flag)) < 0)
1006        {
1007            i_enable(old_pri);
1008        /* have d_mastered stuff here, d_complete it */
1009            if( lc_flags & XMT_DMA_REQ )
1010            {
1011             /* d_complete the transmit information */
1012                xmd.aspace_id = XMEM_GLOBAL;
1013                xmd.subspace_id = NULL;
1014                ret = d_complete(acb_ptr->dma_channel_id, 0, lc_host_buf,
1015                                    lc_xmt_length, &xmd, lc_bus_buf);
1016            }
1017
1018         /* free any mbuf allocated in this routine */
1019            if (allocmbuf_ptr)
1020                    m_free(allocmbuf_ptr);
1021
1022            return(EAGAIN);
1023        } /* cmd queued to adapter */
1024
1025        /* successfully started transmit */
1026
1027      /* increment number of outstanding active dma's */
1028        if (lc_flags & XMT_DMA_REQ)
1029            xchn_ptr->num_active_dma++;
1030
1031      /* incrment transmit map tail pointer */
1032        xchn_ptr->elts_in_use++;
1033        xchn_ptr->tail = (xchn_ptr->tail + 1) % XMT_CHN_ELEM;
1034
1035        i_enable(old_pri);
1036
1037      /* free any LLC mbufs that can be freed now */
1038        if (freembufc_ptr)
1039            m_free(freembufc_ptr);
1040        if (freembuf_ptr)
```

```
1041            m_free(freembuf_ptr);
1042
1043        /* accumulate the transmit stats here, and have a nice day ! */
1044            DDS_STAT.tx_port_cnt++;
1045            if (ULONG_MAX - xmt_adap_cmd.lngth < DDS_STAT.tx_byte_lcnt)
1046            {
1047                    DDS_STAT.tx_byte_mcnt++;
1048                    DDS_STAT.tx_byte_lcnt =
1049                            ULONG_MAX - DDS_STAT.tx_byte_lcnt;
1050                    DDS_STAT.tx_byte_lcnt =
1051                            xmt_adap_cmd.lngth - DDS_STAT.tx_byte_lcnt;
1052            }
1053            else
1054            {
1055                    DDS_STAT.tx_byte_lcnt += xmt_adap_cmd.lngth;
1056            }
1057            if (xmt_adap_cmd.cmd_typ == XMIT_SHORT)
1058            {
1059                    DDS_STAT.tx_short++;
1060                    DDS_STAT.tx_shortbytes += xmt_adap_cmd.lngth;
1061            }
1062            else
1063                    if ((xmt_adap_cmd.cmd_typ == XMIT_LONG) ||
1064                        (xmt_adap_cmd.cmd_typ == XMIT_GATHER))
1065                    {
1066                            DDS_STAT.tx_dma++;
1067                            DDS_STAT.tx_dmabytes += xmt_adap_cmd.lngth;
1068                    }
1069
1070        return(0);
1071    }  /* end ricwrite */
1072
1073    /***************************************************************************
1074     *
1075     *       ricioctl
1076     *
1077     ***************************************************************************/
1078    ricioctl(devno, cmd, arg, flag, mpxchan, ext)
1079    dev_t   devno;          /* major and minor device number */
1080    int     cmd;            /* command to be performed  */
1081    caddr_t arg;            /* address of parm block for ioctl system call*/
1082    int     flag;           /* flag from last open system call  */
1083    chan_t  mpxchan;        /* mpx channel number */
1084    caddr_t ext;            /* value of "ext" passed to WRITEX            */
1085    {
1086        int adapt_num;      /* adapter number */
1087        int port_num;       /* port number   */
1088        int ret;          /* return value  */
1089        t_ric_dds   *dds_ptr;         /* dds pointer */
1090        t_acb       *acb_ptr;         /* pointer to ACB struct */
1091        struct devinfo      *devinfo_ptr;
1092        volatile unsigned long       bus_sr;         /* IO Seg Reg number mask */
1093        int                          error;          /* return value  */
1094        unsigned long                iob;            /* adapter io base addr */
1095        unsigned long                memb;           /* adapter bus memory base */
1096        unsigned int                 sleep_flag;     /* sleep flag for que_command */
1097
1098        /* if minor number is invalid, return error */
1099        if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
1100        {
1101                return(EINVAL);
1102        }
1103
1104        /* if the channel number out of range (only 0 is valid for now) */
1105        if ( mpxchan != 0 )
1106        {
1107                return(ECHRNG);
1108        }
1109
1110        /* get dds pointer from dds directory */
1111        dds_ptr = dds_dir[minor(devno)];
1112
1113        /* if port not configured, return error */
1114        if (dds_ptr == NULL)
1115        {
```

```
1116            return(EINVAL);
1117    }
1118
1119    adapt_num = dds_ptr->dds_hdw.slot_num;
1120    acb_ptr = acb_dir[adapt_num];
1121
1122    port_num = dds_ptr->dds_dvc.port_num;
1123
1124    /* use the cmd parameter to switch for various operations */
1125
1126    ret = 0;
1127    switch (cmd)
1128    {
1129        case IOCINFO:/* Standard request for devinfo */
1130           devinfo_ptr = (struct devinfo*)arg;
1131           devinfo_ptr->devtype = DD_RIC;
1132           devinfo_ptr->flags = 0;
1133           break;
1134
1135        case RIC_RASW:   /* Reload adapter software */
1136        {
1137            /* invoke reload_asw to actually do adapter software */
1138            /* reload */
1139            sleep_flag = 0;
1140            error = reload_asw(acb_ptr, dds_ptr, mpxchan, arg, bus_sr, iob,
1141                      memb, sleep_flag);
1142
1143            break;
1144        }
1145
1146        default:
1147           return(EINVAL);
1148    }
1149
1150 } /* end ricioctl */
1151
1152 /***************************************************************************
1153  *
1154  *      ricselect
1155  *
1156  ***************************************************************************/
1157 ricselect(devno, events, revent_ptr, mpxchan)
1158 dev_t devno;
1159 unsigned short  events;
1160 unsigned short  *revent_ptr;
1161 int mpxchan;
1162 {
1163        int             adapt_num;      /* adapter number */
1164        int             port_num;       /* port number  */
1165        t_acb           *acb_ptr;         /* pointer to ACB */
1166        t_ric_dds       *dds_ptr;         /* pointer to DDS */
1167        t_chan_info     *tmp_chnptr;  /* temporary channel info pointer */
1168        unsigned char   done;
1169
1170     /* if minor number bad, return  */
1171        if (minor(devno) >= (MAX_ADAP*NUM_PORTS))
1172        {
1173               return(EINVAL);
1174        }
1175
1176     /* if the channel number out of range, return */
1177        if ( mpxchan != 0 )
1178        {
1179               return(ECHRNG);
1180        }
1181
1182     /* get dds pointer  */
1183        dds_ptr = dds_dir[minor(devno)];
1184
1185     /* if port not configured, return  */
1186        if (dds_ptr == NULL)
1187        {
1188               return(ENXIO);
1189        }
1190
```

```
1191            dds_ptr->dds_wrk.cmd_avail_flag = FALSE;
1192
1193            adapt_num = dds_ptr->dds_hdw.slot_num;
1194            acb_ptr = acb_dir[adapt_num];
1195
1196            port_num = dds_ptr->dds_dvc.port_num;
1197
1198        /*
1199         * get the channel information data structure
1200         * pointer from the dds for this channel.
1201         */
1202            tmp_chnptr = dds_ptr->dds_wrk.p_chan_info[mpxchan];
1203
1204            done = TRUE;
1205            while ( done == TRUE )
1206            {
1207                 /* check for requested selections, one at a time */
1208
1209                 /* select on receive data available */
1210                    if( events & POLLIN )
1211                    {
1212                          /* at least one event on the rcv queue */
1213                            if(tmp_chnptr->p_rcv_head != NULL)
1214                            {
1215                                    *revent_ptr |= POLLIN;
1216                            }
1217                            else
1218                            {
1219                                    if( !(events & POLLSYNC) )
1220                                    {
1221                                            tmp_chnptr->sync_flags |= POLLIN;
1222                                    }
1223                            }
1224                    } /* end check for POLLIN flag */
1225
1226                 /* select on status available */
1227                    if( events & POLLPRI )
1228                    {
1229                         /* at least one event on the status queue */
1230                            if(tmp_chnptr->p_stat_head != NULL)
1231                            {
1232                                    *revent_ptr |= POLLPRI;
1233                            }
1234                            else
1235                            {
1236                                    if( !(events & POLLSYNC) )
1237                                    {
1238                                            tmp_chnptr->sync_flags |= POLLPRI;
1239                                    }
1240                            }
1241                    }  /* end check for POLLPRI flag */
1242
1243            } /* end while */
1244
1245        /* return of zero tells poll/select to sleep if necessary */
1246            return(0);
1247
1248    } /* end ricselect */
1249
1250    /**************************************************************************
1251     *
1252     *      ricintr
1253     *
1254     **************************************************************************/
1255    ricintr(intr_ptr)
1256    struct intr  *intr_ptr;
1257    {
1258            unsigned long   bus_sr;
1259            t_acb           *acb_ptr;        /* adapter control block pointer */
1260            unsigned char   *io_ptr;
1261            unsigned char   serviced = 0;
1262            unsigned char   taskreg;
1263            unsigned char   comreg;
1264            unsigned int    old_pri;
1265
```

```
1266            acb_ptr = (t_acb *)intr_ptr;
1267
1268            serviced = 0;
1269
1270        /* set up bus access and get io base addr  */
1271            bus_sr = BUSIO_ATT(acb_ptr->io_segreg_val,0);
1272            io_ptr = (unsigned char *)( acb_ptr->io_base + bus_sr);
1273
1274        /* check if interrupt pending before reading taskreg */
1275            comreg = PIO_GETC( io_ptr + COMREG );
1276
1277            if ( comreg & COM_IP )
1278            {
1279                 /* read interrupt register, TASKREG, on adapter */
1280                    taskreg = PIO_GETC( io_ptr + TASKREG);
1281
1282                 /* switch based on the value of the interrupt register */
1283                    switch (taskreg)
1284                    {
1285                         case TR_WDE:/* watchdog timer expired */
1286                            serviced = 1;
1287                            acb_ptr->c_intr_rcvd++;   /* increment int. count */
1288                            break;          /* ignore this interrupt */
1289
1290                         case TR_NOI:            /* no int from this adapter */
1291                            break;
1292
1293                         case TR_DMA0:           /* Port 0 DMA Complete */
1294                         case TR_DMA1:           /* Port 1 DMA Complete */
1295                         case TR_DMA2:           /* Port 2 DMA Complete */
1296                         case TR_DMA3:           /* Port 3 DMA Complete */
1297                            serviced = 1;
1298                            acb_ptr->c_intr_rcvd++;   /* increment int. count */
1299
1300                            /* save taskreg val */
1301                            acb_ptr->cur_intr_val = taskreg;
1302
1303                            break;
1304
1305                         case TR_TXFL:   /* command blocks available     */
1306                            serviced = 1;
1307                            acb_ptr->c_intr_rcvd++;   /* increment int. count */
1308
1309                            /* save taskreg val */
1310                            acb_ptr->cur_intr_val = taskreg;
1311                            acb_ptr->adapter_state &=  SUSPENDED;
1312
1313                         /* wake up ports waiting for command blocks */
1314                            e_wakeup( &acb_ptr->txfl_event_lst );
1315                            break;
1316
1317                         default:        /* most real ints caught here */
1318                            serviced = 1;
1319                            acb_ptr->c_intr_rcvd++;   /* increment int. count */
1320
1321                         /* save taskreg val */
1322                            acb_ptr->cur_intr_val = taskreg;
1323
1324                         /* sched the offlevel */
1325                            if (acb_ptr->arq_sched != TRUE)
1326                            {
1327                                acb_ptr->arq_sched = TRUE;
1328                                i_sched(&acb_ptr->offl.offl_intr);
1329                            }
1330                            break;
1331
1332                    } /* end switch on taskreg value */
1333
1334            }
1335
1336        /* restore addressing on bus */
1337            BUSIO_DET( bus_sr );
1338
1339        /* if interrupt fielded, tell FLIH */
1340            if(serviced)
```

```
1341            {
1342                  /* reset to catch other interrupts */
1343                     i_reset( intr_ptr );
1344                     return (INTR_SUCC);
1345            }
1346            else
1347            {
1348                     return ( INTR_FAIL ); /* not our interrupt, tell the FLIH */
1349            }
1350   }  /* end ricintr */
1351
1352   ricoffl(t_offl_intr *offl_ptr)
1353   {
1354       volatile unsigned long      bus_sr;       /* IO Seg Reg number mask */
1355       volatile t_acb              *acb_ptr;       /* pointer to ACB */
1356       t_ric_dds                  *dds_ptr;       /*  pointer to DDS */
1357       unsigned int                resp_elem;    /* response queue element */
1358       unsigned int                rqe_cmd;      /* RQE command field */
1359       unsigned int                rqe_port;     /* RQE port field */
1360       unsigned short              rqe_seqno;    /* RQE status field */
1361       unsigned int                rqe_stat;     /* RQE status field */
1362       unsigned int                rqe_type;     /* RQE type field */
1363       unsigned int                sleep_flag;   /* que_cmd sleep flag */
1364
1365       bus_sr = BUSIO_ATT(IO_SEG_REG, 0);
1366
1367    /* get pointer acb for interrupting adapter */
1368       if (acb_ptr != NULL)
1369       {
1370           acb_ptr = (t_acb *)offl_ptr->p_acb_intr;
1371           acb_ptr->arq_sched = FALSE;
1372       }
1373       else
1374       {
1375        /* Spurrious interrupt?  acb_ptr not defined */
1376           BUSIO_DET(bus_sr);
1377           return;
1378       }
1379
1380       while ((resp_elem = get_rqe (acb_ptr, bus_sr)) != -2)
1381       {
1382
1383           if( resp_elem == -1 )
1384           {
1385                   BUSIO_DET(bus_sr);
1386                   return;
1387           }
1388
1389        /* isolate response type */
1390           rqe_type = RQE_TYPE( resp_elem );
1391
1392        /* isolate the port number */
1393           rqe_port = RQE_PORT( resp_elem );
1394
1395        /* isolate the status/command field */
1396           rqe_stat = RQE_XESTATUS( resp_elem );
1397           rqe_cmd  = RQE_COMMAND( resp_elem );
1398
1399        /* isolate the sequence number  */
1400           rqe_seqno = RQE_SEQUENCE( resp_elem );
1401
1402        /* get dds pointer for the port  */
1403
1404           /* port not configured ( dds pointer is null ) */
1405           if ((dds_ptr = acb_ptr->p_port_dds[rqe_port]) == (t_ric_dds *)NULL)
1406           {
1407                   BUSIO_DET(bus_sr);
1408                   return;
1409           }
1410
1411         /* invalid port number in the rqe */
1412           if( (rqe_cmd != STRT_CARD_RST) && (rqe_port >= NUM_PORTS) )
1413           {
1414                   BUSIO_DET(bus_sr);
1415                   return;
```

```
1416            }
1417
1418            if(dds_ptr->dds_dvc.port_state == CLOSED)
1419            {
1420                    BUSIO_DET(bus_sr);
1421                    return;
1422            }
1423
1424        /* flag set when there is a response q elem. cleared by mpqselect */
1425            dds_ptr->dds_wrk.cmd_avail_flag = TRUE;
1426
1427            switch (rqe_type)
1428            {
1429                case XMIT_COMPLETE:/* TX acknowledgement */
1430                {
1431                   ric_tx_ack( acb_ptr, dds_ptr, resp_elem, rqe_seqno );
1432                }
1433                break;
1434
1435                case RECV_COMPLETE_DMA:/* RX data ready */
1436                {
1437                    ric_rx_data( acb_ptr, dds_ptr, resp_elem, sleep_flag );
1438                }
1439                break;
1440
1441                case COMMAND_SUCCESS:/* Command complete */
1442                {
1443                    ric_cmd_cmplt( acb_ptr, dds_ptr, rqe_cmd, resp_elem );
1444                }
1445                break;
1446
1447                case SOL_STATUS:/* Solicited port status response */
1448                break;
1449
1450                case XMIT_ERROR:/* TX Error response */
1451                {
1452                    ric_tx_err( acb_ptr, dds_ptr, resp_elem, rqe_seqno );
1453                }
1454                break;
1455
1456                case RECV_COMPLETE:/* RX Error response */
1457                {
1458                    ric_rx_err( acb_ptr, dds_ptr, resp_elem );
1459                }
1460                break;
1461
1462                case COMMAND_FAILURE:/* Command Failure response */
1463                {
1464                    ric_cmd_fail( acb_ptr, dds_ptr, rqe_cmd, resp_elem );
1465                }
1466                break;
1467
1468                case UNSOL_STATUS:/* Unsolicited Port Status */
1469                {
1470                    ric_unsol_stat( acb_ptr, dds_ptr, rqe_seqno, resp_elem );
1471                }
1472                break;
1473
1474                default:/* invalid response */
1475                break;
1476
1477            } /* end switch based on response type */
1478        } /* end while more response queue elements queued */
1479
1480        BUSIO_DET(bus_sr);
1481        return;
1482    }  /* end ricoffl */
1483
```

E.2 Device Driver Header Files

E.2.1 ric.h

```
 1  /*
 2   *      Header file for the ric device driver
 3   */
 4  #define DD_RIC  '8'     /* driver for ric 8 port  */
 5  #define RIC_RASW         0x00ec  /* Reload adapter software      */
 6  #define Q_RASW          0x00ec  /* Reload adapter software      */
 7  #define MAX_ADAP        8       /* max number of adapters/machine       */
 8  #define NUM_PORTS       8       /* max number of ports/adapter          */
 9  #define IOCC_SEG_REG    0x02000000
10  #define IO_SEG_REG      0x820c0020
11  #define LOCK_LIST       (unsigned char)0xff
12  #define DDS_STAT        dds_ptr->dds_ras.cio_stats
13  #define ACMD_ELT_FREE   (unsigned char)0xf0
14  #define ACMD_ACQ        (unsigned short)0
15  #define ACMD_TXF        (unsigned short)1
16
17
18  #define INITREG1        0x10            /* Init. Register 1   (INITREG1) */
19  #define INITREG2        0x08            /* Init. Register 2   (INITREG2) */
20  #define CPUPAGE         0x05            /* CPU Page Register   (CPUPAGE) */
21  #define GAID            0x0F            /* Gate Array ID          (GAID) */
22  #define DREG            0x03            /* Data Register          (DREG) */
23  #define CAD_EN          0x15            /* Host Reset Enable    (CAD_EN) */
24  #define PCPAR0          0x0A            /* Parity Register 0    (PCPAR0) */
25  #define PCPAR1          0x0B            /* Parity Register 1    (PCPAR1) */
26  #define PCPAR2          0x11            /* Parity Register 2    (PCPAR2) */
27
28  #define PCP2_SYNC_CHCK  0x40            /* Synchronous IOCHCK   .PCPAR2  */
29  #define PCP2_EN_CHCK    0x20            /* Enable IOCHCK        .PCPAR2  */
30
31  #define N_RXFREE       48               /*                               */
32  #define N_TXFREE       52               /*                               */
33
34  #define POS0           0x100            /* POS Register 0 IOCC offset    */
35  #define P0_F            0x70            /* POS Card ID low, MPQP         */
36  #define POS1           0x101            /* POS Register 1 IOCC offset    */
37  #define P1_F            0x8F            /* POS1 Card ID high, MPQP       */
38
39  #define POS2           0x102            /* POS Register 2 IOCC offset    */
40  #define P2_ENABLE       0x01            /* -sleep/+ENABLE */
41  #define P2_INT3         0x00            /* interrupt level 3 mask */
42  #define P2_INT4         0x02            /* interrupt level 4 mask */
43  #define P2_INT7         0x04            /* interrupt level 7 mask */
44  #define P2_INT9         0x06            /* interrupt level 9 mask */
45  #define P2_INT10        0x08            /* interrupt level 10 mask */
46  #define P2_INT11        0x0A            /* interrupt level 11 mask */
47  #define P2_INT12        0x0C            /* interrupt level 12 mask */
48  #define P2_SYNC_CHCK    0x80            /* Channel Check Mode = Sync */
49
50  #define POS3           0x103            /* POS Register 3 IOCC offset    */
51
52  #define POS4           0x104            /* POS Register 4 IOCC offset    */
53  #define P4_WSIZ_8K      0x00            /* POS4 Window Size 8K */
54  #define P4_WSIZ_16K     0x20            /* POS4 Window Size 16K */
55  #define P4_WSIZ_32K     0x40            /* POS4 Window Size 32K */
56  #define P4_WSIZ_64K     0x60            /* POS4 Window Size 64K */
57  #define P4_WSIZ_128K    0x80            /* POS4 Window Size 128K */
58  #define P4_WSIZ_512K    0xA0            /* POS4 Window Size 512K */
59  #define P4_WSIZ_1M      0xC0            /* POS4 Window Size 1M */
60  #define P4_WSIZ_2M      0xE0            /* POS4 Window Size 2M */
61
62  #define POS5           0x105            /* POS Register 5 IOCC offset    */
63  #define P5_FAIRNESS     0x01            /* POS5 Fairness Enable */
64  #define P5_PAREN        0x20            /* POS5 Data Parity Enable */
65  #define P5_CHCKS        0x40            /* POS5 I/O Channel Check Status */
66  #define P5_CHCKI        0x80            /* POS5 I/O Channel Check Indicator */
67
68  #define POS6           0x106            /* POS Register 6 IOCC offset    */
```

```
69   #define POS7          0x107          /* POS Register 7 IOCC offset     */
70
71   #define RXREL_THRESH  16             /* Receive buffer release         */
72                                        /* threshold                      */
73   #define WINDOW_SIZE   0x10000        /* adapter window size, 64K       */
74
75   /*
76    *  Access DDS
77    */
78   # define DVC           dds_ptr->dds_dvc
79   # define HDW           dds_ptr->dds_hdw
80   # define WRK           dds_ptr->dds_wrk
81   # define XMITMAP       WRK.p_xmt_chn->xmt_map_chn
82
83   /*
84    *  internal port states for port_state variable in dds
85    */
86
87   #define DORMANT_STATE          0x00   /* initial state */
88   #define OPEN_REQUESTED         0x01   /* Open in progress */
89   #define OPEN                   0x02   /* Port opened */
90   #define START_REQUESTED        0x03   /* Start in progress */
91   #define STARTED                0x04   /* Port started */
92   #define DATA_XFER              0x04   /* Data tranfer state */
93   #define HALT_REQUESTED         0x05   /* Halt in progress */
94   #define HALTED                 0x02   /* Port halted */
95   #define CLOSE_REQUESTED        0x07   /* Close requested */
96   #define CLOSED                 0x00   /* Port closed */
97
98   #define COMREG        0x06           /* Command Register     (COMREG) */
99   #define PTRREG        0x02           /* Pointer Register     (PTRREG) */
100  #define INTCOM        0x09           /* Adapter Interrupt    (INTREG) */
101
102  #define  RIC_RDY_FOR_MAN_DIAL          0x2210
103  #define  RIC_ERR_THRESHLD_EXC          0x82
104
105
106  #define RIC_TX_FAILSAFE_TIMEOUT  0xb1    /*Transmit command did not complete*/
107  #define RIC_DSR_ON_TIMEOUT       0xa1    /*DSR fails to come on  */
108  #define RIC_X21_RETRIES_EXC      0xce    /* X21 Retries exceeded call
109                                             not completed  */
110  #define RIC_X21_TIMEOUT          0x21    /* X.21 timer expired     */
111  #define RIC_X21_CLEAR            0xD2    /* Unexpected Clear received from DCE*/
112
113  #define RIC_RCV_TIMEOUT          0xa7
114  #define RIC_AR_RCV_TIMEOUT       0xa8
115  #define RIC_DSR_DROPPED          0x41
116  #define RIC_ASY_LOST_RTS         0x42
117  #define RIC_TX_UNDERRUN          0x89
118  #define RIC_CTS_UNDERRUN         0x88
119  #define RIC_CTS_TIMEOUT          0x15
120  #define RIC_TX_FS_TIMEOUT        0x16
121  #define RIC_RX_OVERRUN           0x8001
122  #define RIC_RX_ABORT             0x18
123  #define RIC_BUF_STAT_OVFLW       0x1001
124  #define RIC_RX_FRAME_ERR         0xC001
125  #define RIC_RX_BSC_FRAME_ERR     0xA001
126  #define RIC_RX_BSC_PAD_ERR       0xA002
127  #define RIC_RX_PARITY_ERR        0x8002
128  #define RIC_FRAME_CRC            0x8003
129  #define RIC_LOST_SYNC            0x8004
130  #define RIC_RX_BAD_SYNC          0x1b
131  #define RIC_RX_DMA_BFR_ERR       0x1c
132  #define RIC_ADAP_NOT_FUNC        0x30    /* Adapter not functioning  */
133  #define RIC_TOTAL_TX_ERR         0x31
134  #define RIC_TOTAL_RX_ERR         0x32
135  #define RIC_TX_PERCENT           0x33
136  #define RIC_RX_PERCENT           0x34
137  #define RIC_DSR_ALRDY_ON         0x40
138  #define RIC_RESET_CMPL           0x20    /* Reset Completed */
139
140  #define RIC_XT1_TIMER            0xc1    /* X.21 Timer that expired */
141  #define RIC_X_DCE_READY_TIMER    0xcb    /* X.21 Timer that expired */
142
143  #define RIC_ETB         (unsigned short)0x2001  /* --- ETB : 0 1 */
```

```
144   #define RIC_DISC         (unsigned short)0x200F  /* --- DSC : 0 F */
145   #define RIC_STX_ENQ      (unsigned short)0x202C  /* STX ENQ : 2 C */
146   #define RIC_SOH_ENQ      (unsigned short)0x203C  /* SOH ENQ : 3 C */
147
148   /***************************************************************************
149    *   RIC COMREG VALUES
150    ***************************************************************************/
151   #define COM_RC           0x01            /* Reset Card           .COMREG  */
152   #define COM_IE           0x10            /* Interrupt Enable     .COMREG  */
153   #define COM_IP           0x20            /* Interrupt Pending    .COMREG  */
154
155   /***************************************************************************
156    *   RIC TRANSMIT CHAIN FLAG VALUES
157    ***************************************************************************/
158   #define XMT_DMA_REQ      (unsigned short)0x01
159   #define XMT_FREE_MBUF    (unsigned short)0x02
160   #define XMT_STAT_REQ     (unsigned short)0x04
161   #define XMT_TCWS_PORT    12      /* number TCWs/port      */
162   #define XMT_CHN_ELEM     (2*XMT_TCWS_PORT) /* number xmit chain elements */
163
164   /* Port Control Commands    */
165
166   #define XMIT_SHORT       0x10            /* Transmit Short                */
167   #define XMIT_LONG        0x11            /* Transmit Long                 */
168   #define XMIT_GATHER      0x12            /* Transmit Gather               */
169   #define RCV_BUF_INDC     0x13            /* Receive Buffer Indicate       */
170   #define SET_PARAM        0x21            /* Set Parameters                */
171   #define START_PORT       0x22            /* Start Port                    */
172   #define STOP_PORT        0x23            /* Stop Port                     */
173   #define TERM_PORT        0x24            /* Terminate Port                */
174   #define FLUSH_PORT       0x25            /* Flush Port                    */
175   #define QURY_MDM_INT     0x2a            /* Query Modem Interrupts        */
176   #define STRT_AUTO_RSP    0x2b            /* Start Auto Response           */
177   #define STOP_AUTO_RSP    0x2c            /* Stop Auto Response            */
178   #define CHG_PARAM        0x2d            /* Change Parameters             */
179
180   /* Port Command Modifiers */
181
182   #define ADAP_TX_ACK      0x80            /* Tx ack for Transmit command  */
183   #define ADAP_TRANSP      0x40            /* Transparent mode */
184   #define ADAP_DMA_ACK     0x01            /* DMA ack for Transmit command */
185
186   /* Adapter Constants    */
187
188   #define ADAP_TX_AREA     0x50000         /* Adapter TX buffer area */
189   #define ADAP_BUF_SIZE    4096            /* Size of TX and RX buffers */
190
191   /*----------------------------------------------------------------------*/
192   /*  Adapter Reset:                                                      */
193   /*----------------------------------------------------------------------*/
194
195   # define RESET_TIMEOUT  8       /* eight seconds */
196
197   /* Adapter States: */
198
199   # define UNKNOWN         0       /* adapter is in an unknown state */
200   # define RESET           1       /* adapter has been reset */
201   # define INITIALIZED     2       /* adapter is reset and initialized */
202   # define RESETTING       3       /* adapter is being reset */
203   # define SUSPENDED       0x80    /* adapter is waiting for command blocks */
204
205
206   /*
207    * The following macro will be used exchange the supplied character
208    * with the one which exists in bus memory at the specified address
209    */
210
211   #define BUS_XCHGC(p,v)  (BusXchgC(p,v))
212
213   /*
214    *   BUS accessors
215    */
216   #define PUTSR(p,v)       ((void)BusPutSR(p,v))
217   #define PUTLR(p,v)       ((void)BusPutLR(p,v))
218
```

```
219
220   # define C       1       /* Character type of PIO access */
221   # define S       2       /* Short type of PIO access */
222   # define SR      3       /* Short-reversed type of PIO access */
223   # define L       4       /* Long type of PIO access */
224   # define LR      5       /* Long-reverse type of PIO access */
225
226   # define PIO_GETC( a )          ((int) PioGet( a, C ))
227   # define PIO_GETS( a )          ((int) PioGet( a, S ))
228   # define PIO_GETSR( a )         ((int) PioGet( a, SR ))
229   # define PIO_GETL( a )          ((int) PioGet( a, L ))
230   # define PIO_GETLR( a )         ((int) PioGet( a, LR ))
231
232   # define PIO_PUTC( a, v )       ((int) PioPut( a, v, C ))
233   # define PIO_PUTS( a, v )       ((int) PioPut( a, v, S ))
234   # define PIO_PUTSR( a, v )      ((int) PioPut( a, v, SR ))
235   # define PIO_PUTL( a, v )       ((int) PioPut( a, v, L ))
236   # define PIO_PUTLR( a, v )      ((int) PioPut( a, v, LR ))
237
238   # define PIO_GETSTR( d, s, l )  ((int) PioBusCopy( d, s, l ))
239   # define PIO_PUTSTR( d, s, l )  ((int) PioBusCopy( d, s, l ))
240   # define PIO_XCHGC( a, v )      ((int) PioXchgC( a, v ))
241
242   # define PIO_RETRY_COUNT        3
243
244   /***************************************************************************
245    *   RIC  TASKREG VALUES                                                   *
246    ***************************************************************************/
247   #define TASKREG         0x04            /* Mailbox Register    (TASKREG) */
248   #define TR_ARQ_I        (unsigned char)0x00     /* ARQ Now non-empty (int) */
249   #define TR_TXFL         (unsigned char)0x01     /* Tx Free List non-empty */
250   #define TR_DMA0         (unsigned char)0x80     /* DMA TX Ack, Port 0 */
251   #define TR_DMA1         (unsigned char)0x81     /* DMA TX Ack, Port 1 */
252   #define TR_DMA2         (unsigned char)0x82     /* DMA TX Ack, Port 2 */
253   #define TR_DMA3         (unsigned char)0x83     /* DMA TX Ack, Port 3 */
254   #define TR_WDE          (unsigned char)0xFE     /* Watchdog timer expired */
255   #define TR_NOI          (unsigned char)0xFF     /* No interrupt pending */
256
257   #define GA_CNTNDR_3     0x80            /* GAID, Contender 3      .GAID  */
258   #define GA_CNTNDR_4     0x81            /* GAID, Contender 4      .GAID  */
259   #define GA_CNTNDR_5     0x82            /* GAID, Contender 5      .GAID  */
260
261   /*
262    *   RIC Power On Self test definitions
263    */
264   #define IF_BLK          0x400           /* Page 0 address, Interface Block*/
265   #define ERRLOG_PTR      0x14            /* Offset, Error Log for POST */
266   #define STATOFF         0x7c            /* Offset, primary & secondary stat */
267   #define ROSREADY        0x40            /* ROS Ready Bit, INITREG1 */
268
269   /* RQE Types: */
270
271   # define XMIT_COMPLETE          0x0             /* Transmit complete */
272   # define RECV_COMPLETE_DMA      0x1             /* Receive complete, DMA */
273   # define COMMAND_SUCCESS        0x2             /* Command complete, success */
274   # define SOL_STATUS             0x3             /* Solicited status */
275   # define FATAL_ERROR            0x6             /* Adapter error, fatal */
276   # define XMIT_ERROR             0x8             /* Transmit error */
277   # define RECV_COMPLETE          0x9             /* Recv complete, no DMA */
278   # define COMMAND_FAILURE        0xA             /* Command complete, failure */
279   # define UNSOL_STATUS           0xB             /* Unsolicited status */
280   # define RECOV_ERROR            0xE             /* Adapter error, recoverable */
281   # define DIAGNOSTIC_ERROR       0xF             /* Diagnostic error */
282
283   # define RQE_TYPE(rqe)          (((rqe) >> 4 ) & 0x0F)
284   # define RQE_PORT(rqe)          ((rqe) & 0x0F)
285   # define RQE_COMMAND(rqe)       ((unsigned  char)((rqe) >> 8))
286   # define RQE_SEQUENCE(rqe)      ((unsigned short)((rqe) >> 16))
287   # define RQE_STATUS(rqe)        ((unsigned short)((rqe) >> 16))
288   # define RQE_XESTATUS(rqe)      ((unsigned  char)((rqe) >> 8))
289
290   /*
291    *  These two macros allow the setting of values for the
292    *  CPUPAGE register and ACMDREG value
293    */
```

```
294
295  #define SET_CPUPAGE( p, s, v ) {\
296          if ( p->cpu_page != v )\
297          {\
298              p->cpu_page = v;\
299              PIO_PUTC( (unsigned long)(p->io_base) | s + CPUPAGE, v);\
300          }}
301
302  #define SET_ACMDREG( p, s, v ) {\
303          if ( p->adap_cmd_reg != v )\
304          {\
305              p->adap_cmd_reg = v;\
306              PIO_PUTS( (unsigned long)(p->p_adap_cmd_reg) | s, v );\
307          }}
308
309  /*
310   *  M_INPAGE determines if the data portion of an mbuf resides within
311   *  one page
312   */
313
314  # define M_INPAGE(m)     ((((int)MTOD((m), uchar *)                  \
315                                   &  (PAGESIZE - 1)) + PAGESIZE) >       \
316                                       ((int)MTOD((m), uchar *) + (m)->m_len))
317
318
319  # define SWAPSHORT(x)    ((((x) & 0xFF) << 8) | ((x) >> 8))
320  # define SWAPLONG(x)     ((((x) & 0xFF)<<24) | (((x) & 0xFF00)<<8) | \
321                            (((x) & 0xFF0000)>>8) | (((x) & 0xFF000000)>>24))
```

E.2.2 ricstruct.h

```
 1  #include <sys/intr.h>
 2  #include <sys/types.h>
 3  #include <sys/lockl.h>
 4  #include "ricfixup.h"
 5  #include <sys/mbuf.h>
 6  #include <sys/mpqp.h>
 7
 8
 9  /***********************************************************************
10   *      Define Device Structure                                        *
11   ***********************************************************************/
12  typedef struct   RICDDS
13  {
14      struct DDS_HDW
15      {
16          unsigned int    slot_num;       /* slot number of adapter */
17
18          unsigned int    bus_intr_lvl;   /* interrupt level */
19
20          unsigned short  intr_priority;  /* interrupt priority */
21
22          unsigned short  dma_lvl;        /* this is the bus arbitration level */
23                                          /* for this adapter */
24
25          unsigned int    bus_io_addr;    /* base of Bus I/O area for this */
26                                          /* adapter */
27
28          unsigned int    bus_mem_addr;   /* base of Bus Memory "Shared" */
29                                          /* addressability for this adapter */
30
31          unsigned int    tcw_bus_mem_addr; /* base of Bus Memory DMA */
32                                            /* addressability for this adapter */
33
34      } dds_hdw;
35
36      struct DDS_DVC
37      {
38          unsigned char   port_num;       /* Port Number for this port */
39
40          unsigned char   port_state;     /* Port State */
41
42          unsigned short  rdto;           /* Receive Data Transfer Offset */
43
```

```
44          int             net_id;         /* Network ID */
45
46      } dds_dvc;
47
48      struct DDS_RAS
49      {
50          t_cio_stats cio_stats;          /* number of receives for port  */
51          t_err_threshold err_thresh;     /* number of transmits for port */
52      } dds_ras;
53
54      struct DDS_VPD
55      {
56          unsigned short  card_id;        /* Card ID...POS0 & POS1 */
57          unsigned short  ver_num;        /* Version Number */
58          char            devname[16];  /* logical device name */
59          char            adpt_name[16];        /* logical adpater name */
60      } dds_vpd;
61
62      struct DDS_WRK
63      {
64          unsigned short  cmd_seq_num;    /* sequence number of command      */
65          unsigned short  cur_chan_num;   /* current channel number          */
66          unsigned char   num_starts;     /* number of starts issued on port */
67                                          /* incremented by successful ioctl */
68                                          /* with CIO_START operator.        */
69                                          /* decremented by successful ioctl */
70                                          /* with CIO_HALT operator.         */
71
72          unsigned char   xmt_ld_flg;     /* flag indicating that the transmit */
73                                          /* chain has been loaded...          */
74
75          struct mbreq  mbreq;            /* mbuf requirements */
76
77          t_chan_info     *p_chan_info[MAX_CHAN];       /* open/select info          */
78
79          t_xmt_chain     *p_xmt_chn;    /* pointer to transmit chain for port */
80
81          t_reg_list      *p_reg_list;   /* pointer to region manager list     */
82
83          unsigned char   modem_intr_mask;
84
85          unsigned char   phys_link;
86
87          unsigned char   field_select;
88
89          unsigned char   dial_proto;
90
91          unsigned char   dial_flags;
92
93          unsigned char   data_proto;
94
95          unsigned char   data_flags;
96
97          unsigned char   modem_flags;
98
99          unsigned char   poll_addr;
100
101         unsigned char   select_addr;
102
103         unsigned char   baud_rate;
104
105         unsigned char   modem_status;
106
107         unsigned short  rcv_timeout;
108
109         unsigned char   cmd_avail_flag;
110
111         struct trb      *ndelay_timer;
112
113         unsigned int    ndelay_timer_pop;
114
115         int             halt_sleep_event;
116
117         int             sleep_on_halt;
118
```

```
119         int             buff_ctr;       /* rx buffer counter */
120
121         union
122         {
123                 t_x21_data      x21_data;
124                 t_auto_data     auto_data;
125         } t_dial;
126
127     } dds_wrk;
128
129 } t_ric_dds;
130
131 #define NUM_RIC_TCWS  64                /* number of TCWs per RIC adapter */
132
133 /*----------------------------------------------------------------------*/
134 /*  Adapter Queue definitions:                                          */
135 /*----------------------------------------------------------------------*/
136
137 typedef struct {
138         unsigned char   length;         /* length of queue */
139         unsigned char   end;            /* last element of queue */
140         unsigned char   out;            /* first item to remove */
141         unsigned char   in;             /* last item inserted */
142         unsigned char   q_elem[1];    /* queue elements */
143 } BYTE_Q;
144
145 typedef struct {
146         unsigned char   length;         /* length of queue */
147         unsigned char   end;            /* last element of queue */
148         unsigned char   out;            /* first item to remove */
149         unsigned char   in;             /* last item inserted */
150         unsigned long   q_elem[1];    /* queue elements */
151 } LONG_Q;
152
153 /*
154  *  Old RIC Byte and Word queue structure types:
155  */
156
157 typedef struct
158 {
159         unsigned char   length;         /* length of queue */
160
161         unsigned char   end;            /* last element of queue */
162
163         unsigned char   out;            /* first item to remove */
164
165         unsigned char   in;             /* last item inserted */
166
167         unsigned char   bqueue[1];    /* byte queue elements */
168
169 }t_byte_queue;
170
171 typedef struct
172 {
173         unsigned char   length;         /* length of queue */
174
175         unsigned char   end;            /* last element of queue */
176
177         unsigned char   out;            /* first item to remove */
178
179         unsigned char   in;             /* last item inserted */
180
181         unsigned int    wqueue[1];    /* word queue elements */
182
183 }t_word_queue;
184
185
186 /************************************************************************
187  *      Receive Queue Chain Data Structure                              *
188  ************************************************************************/
189
190 typedef struct  RCVMAP
191 {
192     struct mbuf         *p_rcv_mbuf;    /* pointer to associated mbuf */
193
```

```
194     char                *p_host_buf;    /* host memory mbuf extension ptr */
195
196     char                *p_bus_buf;     /* bus address for above */
197
198 } t_rcv_map;
199
200 typedef struct  RCVCHAIN
201 {
202
203     int                 length;         /* Number of elements in chain */
204
205     int                 head;           /* index of oldest element sent */
206                                         /* to the adapter */
207
208     int                 tail;           /* index of latest element sent */
209                                         /* to the adapter */
210
211                                         /* receive chain which contains */
212                                         /* mbuf pointers and TCW mapping */
213                                         /* information */
214     t_rcv_map           rcv_map_chn[NUM_RIC_TCWS/4];
215
216 } t_rcv_chain;
217
218
219 /*
220  *  RIC Adapter Command Block
221  */
222
223 typedef struct ADCMDB
224 {
225         unsigned char   cmd_typ;        /* diagnostic command */
226
227         unsigned char   port_nmbr;      /* port number for command      */
228
229         unsigned short  seq_num;        /* command sequence number      */
230
231         unsigned short  rsrvd_1;        /* filler                       */
232
233         unsigned char   lngth;          /* byte length for data         */
234
235         unsigned char   cntrl;          /* control information          */
236
237         char            *p_edrr;        /* pointer to response region   */
238
239         unsigned int    rsrvd_2;        /* filler                       */
240
241         union
242         {
243             struct
244             {
245                 char            data[48]; /* data area associated with          */
246                                           /* this command.              */
247             }d_ovl;
248            struct
249            {
250                 unsigned int    tst_addr;
251                 unsigned short  tst_length;
252                 unsigned char   cntl;
253                 unsigned char   fyller[41];
254             }c_ovl;
255         }u_data_area;
256 }t_adap_cmd;
257
258 /*
259  *  RIC Transmit Gather Adapter Command Block Overlay
260  */
261
262 typedef struct TX_GTHR_CMD
263 {
264         unsigned char   cmd_type;
265
266         unsigned char   port_num;
267
268         unsigned short  seq_num;
```

```
269
270         unsigned short  bongo_1;
271
272         unsigned char   num_blocks;
273
274         unsigned char   control;
275
276         unsigned long   bongo_2;
277
278         unsigned long   bongo_3;
279
280         unsigned char   *p_gthr_blk[8];
281
282         unsigned short  gthr_len[8];
283
284 } t_tx_gather;
285
286 /*
287  *  RIC Offlevel Intr Structure
288  */
289 typedef struct OFFL_INTR
290 {
291         struct intr     offl_intr;      /* system wide bit */
292
293         struct t_acb            *p_acb_intr;    /* pointer to acb for i_sched */
294
295 }t_offl_intr;
296
297 /*
298  *  RIC Adapter Control Block
299  */
300 typedef struct ACB
301 {
302         struct intr     slih_intr;      /* interrupt handler structure  */
303
304         t_offl_intr     offl;           /* offlevel interrupt structure */
305
306         t_offl_intr     dmaoffl;        /* dma offlevel intr structure  */
307
308         int             rasw_sleep;     /* reload adapter software sleep cell */
309
310         char            *p_dma_tst_buf; /* pointer to test for bus master dma */
311
312         unsigned int    arb_lvl;        /* MicroChannel Arbitration Level */
313
314         unsigned int    int_lvl;        /* interrupt level this adapter  */
315                                         /* responds to                   */
316
317         unsigned int    int_pri;        /* interrupt priority */
318
319         unsigned int    offl_lvl;       /* offlevel level for interrupts */
320                                         /* from this adapter             */
321
322         unsigned int    offl_pri;       /* offlevel priority             */
323
324         int             txfl_event_lst; /* transmit free list event list */
325
326         unsigned char   pos0;           /* POS Register 0 Value */
327
328         unsigned char   pos1;           /* POS Register 1 Value */
329
330         unsigned char   pos2;           /* POS Register 2 Value */
331
332         unsigned char   pos3;           /* POS Register 3 Value */
333
334         unsigned char   pos4;           /* POS Register 4 Value */
335
336         unsigned char   pos5;           /* POS Register 5 Value */
337
338         unsigned char   pos6;           /* POS Register 6 Value */
339
340         unsigned char   pos7;           /* POS Register 7 Value */
341
342         unsigned char   slot_num;       /* slot number adapter is in    */
343
```

```
344         unsigned char   adapter_state;  /* 0 - uninitialized            */
345                                         /* 1 - initialization begun     */
346                                         /* 2 - initialization complete  */
347                                         /* 3 - reset requested          */
348                                         /* 0x80 - Suspended: Or Mask    */
349
350         unsigned char   diag_flag;      /* 0 - no diagnostic mode       */
351                                         /* 1 - diagnostic open requested*/
352                                         /* 2 - opened for diagnostics   */
353                                         /* Note: value of 1 set in      */
354                                         /* mpqmpx on request for open   */
355                                         /* and value of two set in      */
356                                         /* mpqopen upon successful open */
357                                         /* for diagnostics.             */
358
359         unsigned char   asw_load_flag;  /* 0 - Adapter software is not  */
360                                         /*     loaded                   */
361                                         /* 1 - Load completed           */
362                                         /* 0xff - locked                */
363
364         unsigned char   cur_intr_val;   /* current interrupt value       */
365
366         unsigned char   n_cfg_ports;    /* number of ports configured    */
367                                         /* on this adapter               */
368
369         unsigned char   n_open_ports;   /* number of ports opened on     */
370                                         /* this adapter                  */
371
372         unsigned char   ds_base_page;   /* CPUPAGE value for data struct-*/
373                                         /* ures on adapter               */
374
375         unsigned char   cpu_page;       /* BUSIO Addr - 5               */
376                                         /* cpu_page is a copy of the    */
377                                         /* last value written in shared */
378                                         /* memory of the adapter.       */
379
380         unsigned char   num_starts;     /* aggregate number of starts   */
381                                         /* on this adapter.  incremented */
382                                         /* on successful start, decremented */
383                                         /* on successful halt.  used to  */
384                                         /* determine when to allocate and */
385                                         /* deallocate receive mbuf/tcw(s)*/
386                                         /* ## JULY 90 - This is now a flag */
387                                         /* ## 1 = Yes & 0 = No for a start */
388                                         /* ## on the adapter. */
389
390         unsigned char   rcv_buf_ind_snt;/* receive buffer indicates sent */
391                                         /* to the adapter...this flag    */
392                                         /* should be set when the first  */
393                                         /* successful start port takes   */
394                                         /* place and reset when adapter  */
395                                         /* software is reloaded          */
396
397         unsigned char   adap_cmd_que_in;/* index to the next place to   */
398                                         /* receive a command number in  */
399                                         /* the adapter command queue    */
400
401         lock_t          cmd_queue_lock; /* lock to access cmd queue */
402
403         unsigned short  adap_cmd_reg;   /* adap_cmd_reg is a copy of the */
404                                         /* last value written in shared  */
405                                         /* memory of the adapter.        */
406
407         int             dma_channel_id; /* DMA Channel ID returned from  */
408                                         /* d_init call                   */
409
410         unsigned long   io_base;        /* base io address              */
411
412         unsigned long   mem_base;       /* base memory address          */
413
414         unsigned long   dma_base;       /* base address of bus memory   */
415                                         /* for this adapter, set in     */
416                                         /* mpqconfig                    */
417
418         unsigned long   io_segreg_val;  /* Segment register value for   */
```

```
419                                               /* io space indicator          */
420
421         t_ric_dds      *p_port_dds[NUM_PORTS];  /* an array of pointers to */
422                                                   /* device data structures  */
423                                                   /* for all the ports for   */
424                                                   /* this current adapter    */
425
426         t_byte_queue    *p_txfree_q;     /* pointer to the transmit free */
427                                          /* buffer queue data structure  */
428
429         t_byte_queue    *p_adap_cmd_que; /* pointer to the adapter com-  */
430                                          /* mand queue data structure    */
431
432         t_word_queue    *p_adap_rsp_que; /* pointer to the adapter re-   */
433                                          /* sponse queue data structure  */
434
435         unsigned short  *p_adap_cmd_reg; /* pointer the the adapter command  */
436                                           /* register                        */
437
438         unsigned short  *p_num_cmd;      /* pointer to number of commands */
439
440         unsigned short  *p_num_rcv_buf; /* pointer to number of receive  */
441                                          /* buffers                       */
442
443         unsigned short  *p_rcv_buf_siz; /* pointer to receive buffer size */
444
445         unsigned short  *p_rcv_buf_para_num; /* pointer to receive buffer */
446                                               /* paragraph number (addr)   */
447
448         unsigned short  *p_num_xmit_buf; /* pointer to number of xmit    */
449                                          /* buffers                      */
450
451         unsigned short  *p_xmit_buf_siz; /* pointer to xmit buffer size */
452
453         unsigned short  *p_xmit_buf_para_num; /* pointer to xmit buffer */
454                                               /* paragraph number (addr)   */
455
456                                          /* Per port pointer to extended */
457         unsigned char   *p_edrr[NUM_PORTS];
458                                          /* diagnostic response region */
459
460         unsigned char   *p_adap_trc_data; /* pointer to adapter trace data */
461
462                                          /* pointer to port trace data   */
463         unsigned char   *p_port_trc_data[NUM_PORTS];
464
465         t_adap_cmd      *p_cmd_blk;      /* pointer to an array of adapter */
466                                          /* command block data structures */
467
468                                          /* local TX free buffer queue */
469         t_byte_queue    *p_lcl_txfree_buf_q;
470
471         t_rcv_chain     *p_rcv_chain;    /* pointer to receive mbuf */
472                                          /* managment chain         */
473
474         unsigned int    c_rcv;           /* receive count             */
475
476         unsigned long   c_intr_rcvd;     /* interrupt counter         */
477
478         unsigned char   dma_sched;       /* flag for dma sched pending */
479
480         unsigned char   arq_sched;       /* flag for arq sched pending */
481
482         struct trb      *sleep_timer;    /* timer structure for sleep */
483
484         unsigned int    sleep_timer_pop;        /* timer pop flag */
485
486 }t_acb;
487
488 typedef struct
489 {
490         int     length;          /* length of transfer */
491         char    *usr_buf;        /* address of user buffer */
492         unsigned long mem_off;   /* offset in adapter memory where */
493                                  /* transfer will begin  */
```

```
494   }t_rw_cmd;
495
```

E.2.3 ricsmisc.h

```
 1   /*
 2      Region management functions.  These services create a control
 3      structure for managing a region of memory.  Additionally, they
 4      provides lookup/retrieval and return services which enable a
 5      user to request any contiguous size of that memory region.
 6   */
 7
 8   typedef struct REG_LIST
 9   {
10           unsigned char          *_p_region;     /* Region Base Address */
11           unsigned long          _rsize;         /* Region size, bytes */
12           unsigned               _l2rsize;       /* Log2, Region Size */
13           unsigned               _n_region;      /* Number of regions */
14           unsigned char          _free [1];      /* Usage map, n_region long */
15   } t_reg_list;
16
17   /*
18      Region management function prototypes.  Aid parameter checking of callers.
19   */
20
21   extern t_reg_list       *reg_init( unsigned char *, unsigned, unsigned );
22   #define reg_release( p_reg )   xmfree( p_reg, pinned_heap )
23
24   extern unsigned char    *reg_alloc( t_reg_list *, unsigned );
25   extern int               reg_free( t_reg_list *, unsigned, unsigned char * );
26   extern void              reg_clear( t_reg_list * );
27   extern int               reg_avail( t_reg_list * );
28
29   #define REG_FREE        0xFF
30   #define REG_USED        0x00
31
```

E.3 Device Driver Makefile

```
 1 #-----------------------------------------------------------------------
 2 #       Makefile for ric device driver
 3 #-----------------------------------------------------------------------
 4 KDEFS = -D_AIX -D_KERNEL
 5
 6 LIBRARIES = -ericconfig -bimport:/lib/my.exp \
 7                 -bimport:/lib/syscalls.exp -lsys -lcsys
 8
 9 ricdd_obj = \
10         ricdd.o
11
12 ricdd:  ricdd.c ric.h ricstruct.h
13         cc $(KDEFS) -c -O ricdd.c
14         cc -o ricdd $(ricdd_obj) $(LIBRARIES)
15
16
17
```

```
18
19
```

Appendix F. Device Driver Miscellaneous

F.1 The busresolve system call

busresolve Device Configuration Subroutine

Purpose

Allocates bus resources to Micro Channel adapters.

Syntax

```
#include <cf.h>

int busresolve(logname, flags, conf_list, not_res_list, busname)
char *logname;
char *conf_list;
char *not_res_list;
int flags;
char *busname;
```

Parameters

logname Specifies the device logical name.

flags Specifies the boot phase or 0.

conf_list Pointer to an array of characters of at least 512 characters.

not_res_list Pointer to an array of characters of at least 512 characters.

busname Specifies the logical name of the bus.

Description

The "busresolve" device configuration subroutine allocates bus resources for devices having predefined bus resource attributes. It queries the Customized Attribute and Predefined Attribute object classes to get a list of current bus resource attribute settings and a list of possible settings for each attribute. It adjusts the values for attributes of devices in the Defined state as necessary to resolve all conflicts. It does this by modifying the values in the Customized Attribute object class. It will never modify attributes of devices that are already in the Available state. "busresolve" will ignore devices in the Defined state if their "change status" indicates that they are Missing.

When "logname" is set to the logical name of a device, "busresolve" will adjust that device's bus resource attributes if necessary to resolve any conflicts with already Available devices. A device's configuration method should invoke "busresolve" to ensure that its bus resources are allocated properly when configuring the device at run time. The configuration method need not do this when run as part of the boot process as this will already have been done for the device by the bus device's configuration method.

© Copyright IBM Corp. 1991

If "logname" is set to a Null string, "busresolve" will allocate bus resources for all devices that are not already in the Available state. This is how "busresolve" is invoked by the bus device's configuration method during the boot process.

The "flags" parameter is to be set to 1 for boot phase 1, 2 for boot phase 2, and 0 when "busresolve" is invoked during run time. "busresolve" can only be invoked to resolve a specific device's bus resources at run time, i.e. "flags" must be 0 when "logname" specifies a device logical name.

If the return code is E_OK, all attributes were resolved successfully. If the return code is E_BUSRESOURCE, "busreolve" was not able to resolve all conflicts. In this case, "conf_list" contains a list of the logical names of the devices for which it successfully resolved attributes and "not_res_list" contains a list of the logical names of the devices for which it could not successfully resolve all attributes. Devices whose names appear in the "not_res_list" must not be configured into the Available state. A configure method that is invoked at run time for a device having bus resources should fail and return E_BUSRESOURCE if "busresolve" does not return E_OK. Both the "conf_list" and "not_res_list" strings must be at least 512 characters or there may not be enough space to hold the device names.

File

/lib/libcfg.a

Return Values

E_OK All bus resources were resolved and allocated successfully.

E_ARGS Invalid parameters to busresolve, i.e. "logname" specifies a device logical name but "flags" is not set to 0 for run time.

E_MALLOC malloc of necessary memory storage failed.

E_NOCuDv No customized device data for the bus device whose logical name is specified by "busname".

E_ODMGET An ODM error occurred while retrieving data from the configuration database.

E_PARENTSTATE The bus device whose name is specified by "busname" is not in the Available state.

E_BUSRESOURCE A bus resource for the device specified by "logname", or any device with bus resources if "logname" is Null, could not be resolved.

Index

A

address space
- addressing model 2-3
- bus memory address space 2-2
- IOCC address space 2-2
- I/O address space 2-1
- real memory address space 2-2

addressing modes 2-8
- bus memory mode 2-8
- IOCC control mode 2-9
- I/O devices mode 2-8
- RT compatibility mode 2-9
- system address mode 2-8

AIX
- avoiding deadlock 3-20
- device classes A-1
- device subclasses A-1
- device types A-1
- interrupt environment 3-1
- process environment 3-1
- process preemption 3-5

AIX Commands
- bosboot 9-19
- ckprereq 8-4
- crash 9-1, 9-4, 9-5, 9-6
- errlogger 9-43
- errupdate 9-43
- installp 8-1, 8-2, 8-5
- inurest 8-4
- mkdev 6-18, 6-21, 6-23, 6-27, 6-32, 6-35, 7-17
- mknod 1-1, 3-7
- nice 3-4
- odmadd 7-21
- odmcreate 7-20
- sysdumpdev 9-1
- updatep 8-2

AIX kernel services
- bawrite 3-12
- bdwrite 3-12
- bflush 3-12
- binval 3-12
- blkflush 3-12
- bread 3-12
- breada 3-12
- brelse 3-12
- bwrite 3-12
- cfgnadd 3-15, 3-16
- cfgndel 3-15, 3-16
- clrbuf 3-12
- clrjmpx 3-19, 3-21
- copyin 3-14
- copyinstr 3-14
- copyout 3-14

AIX kernel services *(continued)*
- creatp 3-19, 3-21
- devdump 3-16
- devstrat 3-16
- devswadd 3-7, 3-15, 3-16
- devswdel 3-7, 3-15, 3-16
- devswqry 3-7, 3-15, 3-16
- dmp_add 9-3
- dmp_del 9-3
- d_align 3-11
- d_clear 3-11
- d_complete 3-11
- d_init 3-11
- d_mask 3-11
- d_master 3-11
- d_move 3-11
- d_roundup 3-11
- d_slave 3-11
- d_unmask 3-11
- errsave 9-35, 9-41, 9-42, 9-43, 9-44
- e_post 3-21
- e_sleep 3-21, 4-31, 4-36
- e_sleepl 3-21
- e_wait 3-21
- e_wakeup 3-21, 4-31, 4-36
- fp_access 3-18
- fp_close 3-18, 4-26, 4-27
- fp_fstat 3-18
- fp_getdevno 3-18
- fp_getf 3-18
- fp_hold 3-18
- fp_ioctl 3-18
- fp_lseek 3-18
- fp_open 3-17, 3-18, 4-20
- fp_opendev 3-17, 3-18, 4-20, 4-21
- fp_poll 3-18
- fp_read 3-18
- fp_readv 3-18
- fp_rwuio 3-18, 4-31, 4-36
- fp_select 3-18
- fp_writev 3-18
- fubyte 3-14
- fuword 3-14
- getblk 3-12
- getc 3-12
- getcb 3-12
- getcbp 3-12
- getcf 3-12
- getcx 3-13
- geteblk 3-12
- geterror 3-12
- getexcept 3-16
- getpid 3-20, 3-21
- getuerror 3-16

© Copyright IBM Corp. 1991

AIX kernel services *(continued)*
initp 3-19, 3-21
init_heap 3-13
iodone 3-12, 5-4
iostadd 3-16, 3-17
iostdel 3-16, 3-17
iowait 3-12
i_clear 3-10
i_disable 3-3, 3-10
i_enable 3-3, 3-10
i_init 3-10
i_mask 3-10
i_reset 3-10
i_sched 3-3, 3-10
i_sched kernel service 2-13
i_umask 3-3
i_unmask 3-10
kmod_entrypt 3-15, 3-17
kmod_load 3-15, 3-17
kmod_unload 3-15, 3-17
lockl 3-6, 3-20, 3-21
longjmpx 3-19, 3-21
nodev 3-8
nulldev 3-8
pdsignal 3-21
pidsig 3-21
pin 3-13
pincf 3-13
pincode 3-13, 10-1, 10-2
pinu 3-13
pioassist 3-16
pio_assist 3-17
prochadd 3-15, 3-17
prochdel 3-15, 3-17
purblk 3-12
putc 3-13
putcb 3-13
putcbp 3-13
putcf 3-13
putcfl 3-13
putcx 3-13
selnotify 3-16, 3-17, 4-48
setjmpx 3-19, 3-21
setpinit 3-19, 3-21
setuerror 3-16, 3-17
sig_chk 3-19, 3-21
sleep 3-21
subyte 3-14
suword 3-14
sysconfig 3-19
uexadd 3-15, 3-17
uexblock 3-16, 3-17
uexclear 3-16, 3-17
uexdel 3-15, 3-17
uiomove 3-14, 4-3, 4-31, 4-36
unlockl 3-20, 3-21
unpin 3-13
unpincode 3-13

AIX kernel services *(continued)*
unpinu 3-13
uphysio 3-12, 4-31
ureadc 3-14, 4-31
uwritec 3-14, 4-36
waitcfree 3-13
wakeup 3-21
xmalloc 3-13
xmattach 3-14
xmdetach 3-14
xmemdma 3-14, 3-15
xmemin 3-14, 3-15
xmemout 3-14, 3-15
xmfree 3-13
AIX system calls
attrval 6-31
busresolve 2-11, 6-25, F-1
close 4-16
creat 4-14
create 4-21
genmajor 6-25, 6-29
genminor 6-22, 6-25
genseq 6-20
getattr 4-10
getminor 6-25, 6-29
ioctl 4-43, 4-44
loadext 6-25, 6-27, 6-29
mknod 6-25
odmadd 6-13, 6-34
odm_add_obj 6-34
odm_initialize 6-19, 6-22, 6-24, 6-29, 6-31
odm_lock 6-19, 6-22, 6-24, 6-29, 6-31
odm_run_method 6-32
odm_terminate 6-23
open 4-14, 4-21
poll 4-48
read 4-31
readx 4-31
reldevno 6-22
relmajor 6-22
select 4-48
setleds 6-24
setprio 3-4
sysconfig 3-15, 4-2, 4-4, 4-9, 6-26, 6-27
write 4-36
writex 4-36
attrval routine B-21
attrval subroutine 6-31

B

Bibliography xix
block I/O
iodone 3-12
iowait 3-12
uphysio 3-12
block I/O kernel services 3-12

bosboot command 9-19
breakpoints 9-29, 9-30
buffer cache 3-12
  bdwrite 3-12
  bflush 3-12
  binval 3-12
  blkflush 3-12
  bread 3-12
  breada 3-12
  brelse 3-12
  bwrite 3-12
  clrbuf 3-12
  getblk 3-12
  geteblk 3-12
  geterror 3-12
  purblk 3-12
BUID 2-7
bus memory address space 2-2
busresolve system call 2-11, 6-25, F-1
byte reversal on Micro Channel bus 2-19

C

cfgdev command 6-5
cfgmgr command A-3
cfgnadd 3-15
cfgndel 3-15
change method 6-4, 6-14
character I/O 3-12
character I/O kernel services
  getc 3-12
  getcb 3-12
  getcbp 3-12
  getcf 3-12
  getcx 3-13
  pincf 3-13
  putc 3-13
  putcb 3-13
  putcbp 3-13
  putcf 3-13
  putcfl 3-13
  putcx 3-13
  waitcfree 3-13
chdev command A-3
chgdev command 6-5
ckprereq command 8-4
compiling device drivers 10-3
component dump table 9-2
component dump table routine entry point 3-10
configuration manager 6-7, 6-10
configuration method 6-17
configuration rules (Config_Rules) B-17
configure method 6-4, 6-14
crash command 9-1, 9-4, 9-5, 9-6
crash subcommands 9-7
  buf 9-7
  buffer 9-7
  callout 9-8
  cm 9-8
crash subcommands *(continued)*
  ds 9-9
  du 9-9
  dump 9-9
  file 9-10
  fs 9-10
  inode 9-10
  kfp 9-11
  knlist 9-11
  mbuf 9-11
  nm 9-12
  od 9-12
  pcb 9-12
  proc 9-13
  socket 9-14
  stack 9-14
  stat 9-15
  trace 9-15
  ts 9-15
  tty 9-16
  user 9-16
  var 9-17
  vfs 9-17
  vnode 9-18
  xmalloc 9-18
creat subroutine call 4-14
create subroutine 4-21
cross memory kernel services
  xmattach 3-14
  xmdetach 3-14
  xmemdma 3-14, 3-15
  xmemin 3-14, 3-15
  xmemout 3-14, 3-15
customized attribute (CuAt) B-13
customized dependency (CuDep) B-15
customized device driver (CuDvDr) B-15
customized devices (CuDv) B-10
customized VPD (CuVPD) B-16

D

ddclose 3-8, 5-3
ddclose device driver entry point 4-24, 4-26
ddconfig 3-8, 5-2
ddconfig device driver entry point 4-1, 4-2, 4-3, 4-12
dddump 3-9, 5-6
dddump device driver entry point 4-52
ddioctl 3-9, 5-6
ddioctl device driver entry point 4-42, 4-43
ddmpx 3-9
ddmpx device driver entry point 4-12
ddopen 3-8, 5-3
ddopen device driver entry point 4-19
ddopen entry point 4-15
ddread 3-9
ddread device driver entry point 4-29
ddrevoke 3-9

ddselect 3-9
ddselect device driver entry point 4-46, 4-47
ddstrategy 3-9, 5-3
ddwrite 3-9
ddwrite device driver entry point 4-34, 4-35
defdev command 6-5
define method 6-4, 6-14
device configuration library routines B-21
  attrval B-21
  genmajor B-21
  genminor B-21
  genseq B-21
  getattr B-21
  getminor B-21
  loadext B-21
  putattr B-21
  reldevno B-21
  relmajor B-21
  relseq B-21
device dependent structure (dds) 4-9
device driver
  bottom half 1-6
  compiling 10-3
  linking 10-3
  multiplexed ddclose routine 4-26
  non-multiplexed ddclose routine 4-26
  packaging 8-1
  setting breakpoints 9-29
  top half 1-6, 1-7
device driver entry points
  ddclose 4-16, 4-24, 4-26, 5-3
  ddconfig 4-1, 4-2, 4-3, 4-12, 5-2
  dddump 4-52, 5-6
  ddioctl 4-42, 5-6
  ddmpx 4-12, 4-16
  ddopen 4-15, 4-19, 5-3
  ddread 4-29
  ddselect 4-46, 4-47
  ddstrategy 5-3
  ddwrite 4-34, 4-35
device driver management services
  kmod_entrypt 3-15
  kmod_load 3-15
  kmod_unload 3-15
  sysconfig 3-19
device handler role 1-4
device head role 1-4
device methods 6-14
  change 4-9, 6-14, 6-30
  change method 6-4
  configure 4-9, 6-14, 6-23
  configure method 6-4
  define 6-14, 6-17
  define method 6-4
  start 6-15, 6-33
  start method 6-5
  stop 6-14, 6-33
  stop method 6-5
device methods *(continued)*
  unconfigure 4-10, 6-14, 6-27
  unconfigure method 6-4
  undefine 6-14, 6-21
  undefine method 6-4
Device Switch Table 3-7
devswadd 3-15
devswdel 3-15
devswqry 3-15
dma (direct memory access)
  assignment of dma channels 2-10
  bawrite 3-12
  dma master 2-9
  dma slave 2-9
  d_align 3-11
  d_clear 3-11
  d_complete 3-11
  d_complete kernel service 2-10
  d_init 3-11
  d_mask 3-11
  d_master 3-11
  d_master kernel service 2-10
  d_move 3-11
  d_roundup 3-11
  d_slave 3-11
  d_slave kernel service 2-10
  d_unmask 3-11
dmp_add kernel service 9-3
dmp_del kernel service 9-3

E

error logging 9-35
errsave command 9-35, 9-41, 9-42, 9-43
errupdate command 9-43
e_sleep kernel service 4-31, 4-36
e_wakeup kernel service 4-31, 4-36

F

fp_close kernel service 4-26, 4-27
fp_open kernel service 4-20
fp_opendev kernel service 4-20, 4-21
fp_rwuio kernel service 4-31, 4-36

G

genmajor subroutine 6-25, 6-29, B-21
genminor subroutine 6-22, 6-25, B-21
genseq subroutine 6-20, B-21
getattr subroutine 4-10, B-21
getexcept 3-16
getminor subroutine 6-25, 6-29, B-21
getuerror 3-16
gnode 4-14, 4-16, 4-26

H

hft subsystem 1-5

I

installp command 8-1, 8-2, 8-5
installp/updatep files D-1
  al File D-4
  al_Level File D-9
  arp File D-7
  config File D-10
  copyright File D-4
  Filename.err File D-12
  Filename.evt File D-13
  Filename.trc File D-12
  instal Script File D-2
  inventory File D-14
  liblpp File D-2
  lpp.acf File D-13
  lpp.cleanup File D-5
  lpp.deinst File D-14
  lpp.doc File D-12
  lpp.instr File D-15
  lpp_name File D-1
  Option.config File D-10
  Option.prereq File D-10
  prereq File D-10
  productid File D-14
  rename File D-14
  service_num File D-7
  size File D-5
  special File D-6
  update Script File D-7
interrupt
  i_disable kernel service 3-3
  i_enable kernel service 3-3
  i_init 3-10
  i_mask 3-10
  i_reset 3-10
  i_sched 3-10
  i_sched kernel service 2-13, 3-3
  i_umask kernel service 3-3
  i_unmask 3-10
  off-level 2-13
  priority assignment 2-12
  processing 2-11
interrupt handling routine entry point 3-9
inurest command 8-4
IOCC 2-3
IOCC address space 2-2
ioctl subroutine call 4-43, 4-44
iodone 5-4
iostadd 3-16
iostdel 3-16
I/O address space 2-1
I/O macros 2-14
  BUSIO_ATT(bid,io_addr) 2-16
  BUSIO_DET(io_addr) 2-16
I/O macros *(continued)*
  BUSIO_GETC 2-15
  BUSIO_PUTC 2-15
  BUSMEM_ATT 2-15, 2-16
  BUSMEM_DET 2-15, 2-16
  IOCC_ATT 2-15
  IOCC_ATT(bid,iocc_addr) 2-16
  IOCC_DET 2-15
  IOCC_DET(iocc_addr) 2-16
  POSREG 2-15

K

kernel debugger 9-1, 9-18
kernel debugger commands
  alter 9-21
  back 9-21
  break 9-21
  breaks 9-22
  clear 9-22
  display 9-22
  drivers 9-22
  find 9-22
  float 9-23
  go 9-23
  help 9-23
  loop 9-23
  map 9-23
  next 9-24
  origin 9-24
  proc 9-24
  quit 9-24
  screen 9-25
  set 9-25
  sregs 9-25
  st 9-26
  stack 9-26
  stc 9-26
  step 9-26
  sth 9-26
  swap 9-26
  trace 9-26
  trb 9-27
  tty 9-27
  user 9-27
  vars 9-27
  vmm 9-27
  xlate 9-27
kernel dump routine 4-52
kernel processes 10-3
  compiling a kernel processs 10-3
  linking a kernel processs 10-4
  loading a kernel processs 10-4
  starting a kernel processs 10-4
  writing a kernel processs 10-3
kernel program/device driver management services
  cfgnadd 3-16
  cfgndel 3-16
  devdump 3-16

kernel program/device driver management services (continued)
devstrat 3-16
devswadd 3-16
devswdel 3-16
devswqry 3-16
getexcept 3-16
getuerror 3-16
iostadd 3-17
iostdel 3-17
kmod_entrypt 3-17
kmod_load 3-17
kmod_unload 3-17
pio_assist 3-17
prochadd 3-17
prochdel 3-17
selnotify 3-17
setuerror 3-17
uexadd 3-17
uexblock 3-17
uexclear 3-17
uexdel 3-17
kernel system dump 9-1
kernel_lock 3-20
kernel_lock. 3-5

L

linking device drivers 10-3
loadext subroutine 6-25, 6-27, 6-29, B-21
lockl kernel service 3-20
logical file system kernel services 3-17
fp_access 3-18
fp_close 3-18
fp_fstat 3-18
fp_getdevno 3-18
fp_getf 3-18
fp_hold 3-18
fp_ioctl 3-18
fp_lseek 3-18
fp_open 3-17, 3-18
fp_opendev 3-17, 3-18
fp_poll 3-18
fp_read 3-18
fp_readv 3-18
fp_rwuio 3-18
fp_select 3-18
fp_writev 3-18

M

macros
attach/detach 2-16
BUSIO_GETC 2-17
BUSIO_GETL 2-17
BUSIO_GETS 2-17
BUSIO_GETSTR 2-17
BUSIO_PUTC 2-17
BUSIO_PUTS 2-17
macros (continued)
BUSIO_PUTSTR 2-17
BUS_GETC 2-17
BUS_GETL 2-17
BUS_GETS 2-17
BUS_PUTCL 2-17
BUS_PUTL 2-17
BUS_PUTS 2-17
BUS_PUTSTR 2-17
data transfer 2-17
makefile 8-6, D-16
master dump table 9-2, 9-3
memory access services 3-13
copyin 3-14
copyinstr 3-14
copyout 3-14
fubyte 3-14
fuword 3-14
subyte 3-14
suword 3-14
uiomove 3-14
ureadc 3-14
uwritec 3-14
memory access to/from user space 3-13
memory management kernel services 3-13
init_heap 3-13
pin 3-13
pincode 3-13
pinu 3-13
unpin 3-13
unpincode 3-13
unpinu 3-13
xmalloc 3-13
xmfree 3-13
Micro Channel
adapter identification by software 2-13
byte reversal 2-19
displaying registers on 9-30
overview 2-1
querying POS registers 2-18
setting POS registers 2-18
unique identification of adapter 2-14
mkdev command 6-18, 6-21, 6-23, 6-27, 6-32, 6-35, 7-17, A-3
mknod subroutine 6-25

N

nodev 3-8
nulldev 3-8

O

object classes 6-5
object classes in odm
configuration rules (Config_Rules) B-17
Config_Rules 6-6, 6-7
CuAt 6-6
CuDep 6-6

object classes in odm *(continued)*
CuDv 6-6
CuDvDr 6-6
customized attribute (CuAt) B-13
customized dependency (CuDep) B-15
customized device driver (CuDvDr) B-15
customized devices (CuDv) B-10
customized VPD (CuVPD) B-16
CuVPD 6-6
PdAt 6-5, 6-10
PdCn 6-6
PdDv 6-5
predefined attributes (PdAt) B-6
predefined connection (PdCn) B-9
predefined devices (PdDv) B-1
ODM 7-1
ODM commands B-19
odmadd B-19
odmchange B-19
odmcreate B-19
odmdelete B-19
odmdrop B-19
odmget B-19
odmshow B-19
ODM object classes B-1
ODM subroutines B-20
odm_add_obj B-20
odm_change_obj B-20
odm_close_class B-20
odm_create_class B-20
odm_err_msg B-20
odm_free_list B-20
odm_get_by_id B-20
odm_get_first B-20
odm_get_list B-20
odm_get_next B-20
odm_get_obj B-20
odm_initialize B-20
odm_lock B-20
odm_mount_class B-20
odm_open_class B-20
odm_rm_by_id B-20
odm_rm_class B-20
odm_rm_obj B-20
odm_run_method B-20
odm_set_path B-20
odm_set_perms B-20
odm_terminate B-20
odm_unlock B-20
odm (Object Data Manager) 6-5
odmadd command 6-34, 7-21, B-19
odmadd system call 6-13
odmchange command B-19
odmcreate command 7-20, B-19
odmdelete command B-19
odmdrop command B-19
odmget command B-19
odmshow command B-19
odm_add_obj command 6-34
odm_add_obj subroutine B-20
odm_change_obj subroutine B-20
odm_close_class subroutine B-20
odm_create_class subroutine B-20
odm_err_msg subroutine B-20
odm_free_list subroutine B-20
odm_get_by_id subroutine B-20
odm_get_first subroutine B-20
odm_get_list subroutine B-20
odm_get_next subroutine B-20
odm_get_obj subroutine B-20
odm_initialize subroutine 6-19, 6-22, 6-24, 6-29, 6-31, B-20
odm_lock subroutine 6-19, 6-22, 6-24, 6-29, 6-31, B-20
odm_mount_class subroutine B-20
odm_open_class subroutine B-20
odm_rm_by_id subroutine B-20
odm_rm_class subroutine B-20
odm_rm_obj subroutine B-20
odm_run_method command 6-32
odm_run_method subroutine B-20
odm_set_path subroutine B-20
odm_set_perms subroutine B-20
odm_terminate subroutine 6-23, B-20
odm_unlock subroutine B-20
open subroutine 4-21
open subroutine call 4-14

P

pincode kernel service 10-1, 10-2
pioassist 3-16
poll subroutine 4-48
predefined attributes (PdAt) B-6
predefined connection (PdCn) B-9
predefined devices (PdDv) B-1
process management kernel services 3-19
clrjmpx 3-19, 3-21
creatp 3-19, 3-21
e_post 3-21
e_sleep 3-21
e_sleepl 3-21
e_wait 3-21
e_wakeup 3-21
getpid 3-20, 3-21
initp 3-19, 3-21
lockl 3-21
longjmpx 3-19, 3-21
pdsignal 3-21
pidsig 3-21
setjmpx 3-19, 3-21
setpinit 3-19, 3-21
sig_chk 3-19, 3-21
sleep 3-21
unlockl 3-21
wakeup 3-21

prochadd 3-15
prochdel 3-15
programmed I/O 2-9
publications xix
putattr routine B-21

R

raw I/O 5-1, 5-6
raymond chiang
read subroutine call 4-31
readx subroutine call 4-31
real memory address space 2-2
reldevno subroutine 6-22, B-21
relmajor subroutine 6-22, B-21
relseq routine B-21
rmdev command A-3
run time configuration commands A-3

S

select subroutine 4-48
selnotify kernel service 3-16, 4-48
setleds subroutine 6-24
setuerror 3-16
smetco world headquarters
SMIT 7-1, 7-21
SMIT database 7-5
SMIT (System Management Interface Tool)
  database 7-5
  database creation 7-20
  Dialog header Object Class (sm_cmd_hdr) C-5
  dialog screens 7-1, 7-2, 7-4
  Dialog/selector command option Object Class (sm_cmd_opt) C-7
  extentions debugging 7-20
  Menu Object Class (sm_menu_opt) C-1
  Menu Object Class (sm_menu_opt) used for aliases C-2
  menu screens 7-1, 7-2, 7-3
  object classes C-1
  Selector header Object Class (sm_name_hdr) C-3
  selector screens 7-1, 7-2, 7-3
  task additions 7-21
special file 1-1
start I/O routine entry point 3-9
start method 6-5, 6-15
stop method 6-5, 6-14
stpfdev command 6-5
strategy routine 5-1
sttdev command 6-5
sysconfig subroutine 3-15, 4-2, 4-4, 4-9, 6-26, 6-27
sysdumpdev command 9-1
system dump formatting 9-5

U

ucfgdev command 6-5
udefdev command 6-5
uexadd 3-15
uexblock 3-16
uexclear 3-16
uexdel 3-15
uio structure 4-31, 4-36
uiomove 4-3
uiomove kernel service 4-31, 4-36
unconfigure method 6-4, 6-14
undefine method 6-4, 6-14
unlockl kernel service 3-20
updatep command 8-2
uphysio kernel service 4-31
ureadc kernel service 4-31
uwritec kernel service 4-36

W

write subroutine 4-36
writex subroutine 4-36
writing a change method 6-30
writing a configure method 6-23
writing a define method 6-17
writing a start method 6-33
writing a stop method 6-33
writing a unconfigure method 6-27
writing a undefine method 6-21

READER'S COMMENT FORM

Title: **Writing a Device Driver for AIX Version 3**
GG24-3629-00

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

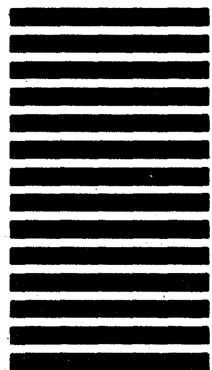
Your comments:

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp is necessary if mailed in the U.S.A.. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and Tape | Please Do Not Staple | Fold and Tape



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:
IBM Corporation, International Technical Support Center
11400 Burnet Road
Internal Zip 2201
Austin, Texas 78758
U.S.A.

Fold | Fold

Return address:

Your Name ______________________

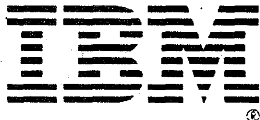
Company Name ______________________ *Department* __________

Street Address ______________________

City ______________________

State ______________________ *Zip Code* __________

IBM Branch Office serving you ______________________



PRINTED IN THE U.S.A.

GG24-3629-00

Writing a Device Driver for AIX Version 3

GG24-3629-00

PRINTED IN THE U.S.A.