Data General

Customer Documentation

Writing a Device Driver for the DG/UX™ System

AViiON™
PRODUCT LINE

Writing a Device Driver for the DG/UX™ System

093-701053-03

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-701053

Printed in the United States of America
Revision 03, May 1990

NOTICE

Writing a Device Driver for the DG/UX™ System

093-701053-03

093-701062-03 (Japan only)

Revision History:	Effective with:
Original Release – April 1989	DG/UX Rel. 4.10
Second Release – June 1989	DG/UX Rel. 4.10
Third Release – March 1990	DG/UX Rel. 4.20
Fourth Release – June 1990	DG/UX Rel. 4.30

DATA GENERAL CORPORATION
4400 Computer Drive
Westboro, MA 01580

Preface

This is a revision of an existing manual. Technical changes from the previous version are marked by vertical revision bars in the outside margin next to the change.

This manual describes how to write your own device driver for a DG/UX™ system running on an AViiON™ machine. Under the AViiON architecture, drivers can be written at two levels: an adapter driver and a device driver for devices connected to an adapter or for units on a controller. This manual addresses both levels of driver.

Who Should Read This Manual?

Users of this document should be generally knowledgeable about operating system design topics such as virtual memory, synchronization, mutual exclusion, locking, and interrupts. They should also be familiar with how multiprocessor hardware can affect these topics. In particular, driver writers should be familiar with the following:

- The AViiON machines, including their I/O architecture and the Motorola 88000 processor. The I/O architecture includes the Small Computer System Interface (SCSI) and the Motorola VMEbus. References for these topics are listed in the "Related Documents" section of this Preface, under the section "Other Documents."

  Readers should also be familiar with general I/O topics such as memory-mapped I/O, interrupt masking, and device masking. Readers should also understand a multiprocessor environment.

- The DG/UX user-level I/O model. This model uses six basic I/O system calls: **open**, **close**, **read**, **write**, **ioctl**, and **select**, which are described in the *Programmer's Reference for the DG/UX™ System (Volume 1)*.

  Readers should also be familiar with the standard UNIX® concept of character special devices, block special devices, and the difference between the two.

- The C programming language, because the interfaces presented in this document are written in C.

We also assume that you have a good understanding of the hardware device or pseudodevice for which you are writing the driver. You must know how your device should behave when it is the target of one of the user-level I/O system calls.

Manual Organization

The manual is organized as follows:

Chapter 1	briefly describes the process of writing a driver and gives an overview of the driver environment.
Chapter 2	describes how to add a driver to the DG/UX system. This chapter also shows you how to configure your device into the system.
Chapter 3	summarizes the functions that a driver must supply to the kernel and also facilities that the kernel supplies to drivers. The chapter also discusses include files and major driver data structures.
Chapter 4	describes in detail the functions, constants, and data structures you must supply for your driver. It also describes the interface for both adapter and device drivers.
Chapter 5	describes how device drivers access their adapter driver's routines via a set of generic adapter manager routines. Using the generic adapter manager routines allows device driver code to work with any and all adapter drivers.
Chapter 6	describes DG/UX routines that relate to process management and timing. It describes routines that handle eventcounters, signals, and clock operations. Also included are descriptions of locking routines.
Chapter 7	describes DG/UX routines that relate to memory and data management. It describes routines for allocating and releasing memory, verifying pointers, and manipulating buffer vectors.
Chapter 8	describes routines used for general driver functions. It describes routines used for error handling, device configuration, driver messaging, and accessing device selection tables.
Appendix A	provides a sample device driver, including its C code, and master and system file entries.

Appendix B	provides a sample adapter driver, including its C code, and master and system file entries.
Appendix C	lists standard peripherals and their default device codes, interrupt levels, and memory-mapped I/O addresses.
Appendix D	provides a short glossary of terms related to writing a device driver.
Documentation Set	provides a complete list of available Data General hardware and software documentation relevant to the DG/UX system.

Related Documents

The following manuals and papers provide information that you may find useful. The first group lists Data General manuals, which can be ordered using the nine-digit ordering number shown in parentheses (see TIPS information in back of manual for ordering instructions). The second group lists manuals and papers available from other organizations. To obtain a document from another organization, contact that organization directly.

Data General Hardware Manuals

AViiON™ 5000 and 6000 Series Systems: Programming System Control and I/O Registers (014-001805)

Describes the system board architecture, including the CPU, memory registers, I/O address decode, and bus arbitration. Discusses how to program the system board registers for addressing, interrupts, I/O and system board control and status.

AViiON™ 300 and 400 Series Stations: Programming System Control and I/O Registers (014-001800)

Describes the workstation architecture and exp;lains how to program the system control logic, monochrome and color graphics controller subsystems, keyboard port, mouse port, serial and parallel ports, LAN interface, and SCSI port.

MC88100 User's Manual, Reduced Instruction Set Computer (RISC) (014-001809)

Describes the Motorola 88100 Central Processing Unit (CPU), including the registers, addressing modes, internal and bus timing, and assembly-language instruction set. This section lists the coduments currently available for the AViiON 400 series stations.

MC88200 User's Manual, Cache/Memory Management Unit (CMMU) (014-001808)

Describes the Motorola 88200 Cache/Memory Management Unit (CMMU), including the CMMU registers, the cache and cache coherency, memory management and user/supervisor space, the Processor bus (Pbus), and the Memory bus (Mbus).

Data General Software Manuals

Installing and Managing the DG/UX™ System (093-701052)

Shows how to install and manage the DG/UX operating system on AViiON hosts that will run as stand-alone, server, or client systems. Aimed at system administrators who are familiar with the UNIX operating system.

Programmer's Reference for the DG/UX™ System (093-701055 and 093-701056)

Alphabetical listing of manual pages for programming commands on the DG/UX system. This two-volume set includes information on system calls, file formats, subroutines, and libraries.

A complete list of the manuals contained in the DG/UX documentation set is provided at the back of this manual, in front of the TIPS information.

Other Organizations' Documents

American National Standard for Information Systems: Small Computer System Interface (SCSI), ANSI X3.131-1986, American National Standards Institute, New York, NY.

The VMEbus Specification, (Revision C.1, Oct. 1985), Motorola Corporation, Phoenix, AZ.

The primary method of synchronization provided by the kernel is eventcounters and sequencers. These were first described in the paper: "Synchronization with Eventcounts and Sequencers," David P. Reed and Rajendra K Kanodia, *Proceedings of the Sixth Symposium on Operating System Principles*, Purdue University, West Lafayette, IN, November 1977. They are also described in: "Synchronization with Eventcounts and Sequencers," David P. Reed and Rajendra K. Kanodia, *Communications of the ACM*, Vol. 22, Number 2, February 1979, pp. 115-123.

Readers, Please Note

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	All DG/UX commands, system calls, pathnames, names of files, directories, and manual pages also use this typeface.
`constant width monospace`	Syntax lines and examples of code use this font.
italic	Represents variables for which you supply values; for example, arguments to routines. In text, italics are also used to emphasize a term that is used for the first time.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (USA only) or contact your local Data General sales representative.

If you have comments on this manual, please use the prepaid Comment Form that appears at the back. We want to know what you like and dislike about this manual.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, and you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS for toll-free telephone support. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

Free telephone assistance is available with your warranty and with most Data General service options. Lines are open from 8:30 a.m. to 8:30 p.m., Eastern Time, Monday through Friday.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

End of Preface

Contents

Chapter 1 — Introduction to Writing a Device Driver

Introduction 1-1
Changes to This Release of the Manual 1-1
Overview of Architectural Issues 1-3
    Memory-Mapped I/O 1-4
    I/O Architecture: Controllers, Adapters, and Devices 1-4
        Adapter, Controller, and Device Layouts on Different Machines 1-5
        Adapter Drivers and Device Drivers 1-6
        The Adapter Manager 1-8
        Do You Have to Write a New Driver? 1-9
    Interrupt Structure 1-9
    The Multiprocessor Environment 1-11
How to Write a Device Driver 1-12
Overview of Device Driver Environment 1-14
    Device Specifications 1-14
    Special Files (Nodes) 1-16
    Block Versus Character Interface 1-18
    Driver Execution 1-18
    Memory Management 1-18
    Interrupts 1-19
    Signals 1-20

Chapter 2 — Adding Your Device Driver to the DG/UX System

Adding a Master File Entry 2-1
    Device Descriptions: The Device Section Entry 2-2
    Parameters: The Keyword Section Entry 2-3
    Master File Aliases: The Alias Section 2-5
Adding a System File Entry 2-5
Rebuilding and Rebooting the System 2-6
Checking the Configuration Process 2-7
    Conf.c 2-7
    Your Special Files 2-9

Chapter 3 — Overview of Driver Facilities and Functions

Include Files 3-1
Overview of User-supplied Device Driver Routines 3-4
Overview of User-supplied Adapter Driver Routines 3-9
    The Device Driver to Adapter Driver Interface 3-9
Overview of Major Data Structures 3-12
    Device Driver Data Structures 3-12
    Data Structures for SCSI Adapter and SCSI Device Drivers 3-14
Other Driver Facilities 3-16

The Driver Daemon and the Generic Daemon 3-16
Error Reporting Facilities 3-17

Chapter 4 — User-Supplied Driver Routines

User-Supplied Device Driver Routines 4-2
Constants and Data Structures 4-2
io_driver_routines_vector_type 4-2
io_device_number_type 4-3
io_device_handle_type 4-3
io_request_info_type 4-4
io_operation_type 4-5
io_operation_record_type 4-5
io_select_intent_type 4-6
io_buffer_vector_type 4-6
io_buffer_descriptor_type 4-7
io_buffer_vector_control_type 4-8
io_channel_flags_type 4-8
Interfaces for Device Driver Routines 4-11
dev_xxx_init: 4-12
dev_xxx_configure: 4-13
dev_xxx_open: 4-16
dev_xxx_close: 4-18
dev_xxx_service_interrupt: 4-20
dev_xxx_read_write: 4-22
dev_xxx_select: 4-25
dev_xxx_ioctl: 4-27
dev_xxx_start_io: 4-29
Kernel I/O Completion Routine Interface 4-31
dev_xxx_open_dump: 4-33
dev_xxx_write_dump: 4-35
dev_xxx_read_dump: 4-37
dev_xxx_close_dump: 4-38
dev_xxx_powerfail: 4-39
dev_xxx_deconfigure: 4-40
dev_xxx_device_to_name: 4-42
dev_xxx_name_to_device: 4-43
dev_xxx_maddmap: 4-45
dev_xxx_mmap: 4-46
dev_xxx_munmap: 4-47
User-Supplied Adapter Driver Routines 4-48
Constants and Data Structures 4-48
dev_scsi_adapter_routines_vector_type 4-48
dev_scsi_interface_routines_vector_type 4-48
dev_scsi_adapter_unit_spec_type 4-49
dev_adapter_request_block_type 4-49
DEV_SCSI_REQUEST_FLAGS 4-51
dev_scsi_adapter_unit_registration_blk_type 4-52
dev_adapter_physical_request_blk_type 4-53
dev_scsi_adapter_unit_options_block_type 4-54
SCSI Adapter Unit Options Block Literals 4-55

Interfaces for Adapter Driver Routines 4-56
dev_xxx_register_requester: 4-58
dev_xxx_set_unit_options: 4-59
dev_xxx_deregister_requester: 4-60
dev_xxx_issue_command: 4-61
dev_xxx_issue_async_command: 4-62
dev_xxx_get_device_info: 4-63
dev_xxx_issue_command_physical_mode: 4-64

Chapter 5 — Managing Your Adapter From Your Device Driver

Constants and Data Structures 5-2
dev_scsi_adapter_configure: 5-3
dev_scsi_adapter_device_to_name: 5-4
dev_scsi_adapter_name_to_device: 5-5
dev_scsi_adapter_open_dump: 5-6
dev_scsi_adapter_register_requester: 5-7
dev_scsi_adapter_set_unit_options: 5-8
dev_scsi_adapter_deregister_requester: 5-9
dev_scsi_adapter_issue_command: 5-10
dev_scsi_adapter_issue_async_command: 5-11
dev_scsi_adapter_get_device_info: 5-12
dev_scsi_adapter_issue_command_physical_mode: 5-13

Chapter 6 — Process Synchronization and Timing

Synchronization Routines 6-2
Constants and Data Structures 6-3
vp_event_type 6-3
vp_add_to_ec_value: 6-4
vp_advance_ec: 6-5
vp_await_ec: 6-6
vp_convert_clock_value_to_ec_value: 6-7
vp_convert_ec_value_to_clock_value: 6-8
vp_get_next_ec_value: 6-9
vp_has_event_occurred: 6-10
vp_increment_ec_value: 6-11
vp_initialize_ec: 6-12
vp_initialize_sequencer: 6-13
vp_read_ec: 6-14
vp_ticket_sequencer: 6-15
vp_are_ec_values_equal: 6-16
Process Signal Management Routines 6-17
Constants and Data Structures 6-17
pm_get_my_pid: 6-18
pm_get_my_pgrp: 6-19
pm_is_interrupted: 6-20
pm_is_terminated: 6-22
pm_send_signal_by_index: 6-23

pm_send_signal_by_process_group: 6-24
pm_send_signal_by_process_id: 6-25
Lock Management Routines 6-26
Constants and Data Structures 6-27
lm_sequenced_lock_type 6-27
lm_unsequenced_lock_type 6-27
misc_spin_lock_type 6-28
lm_initialize_sequenced_lock: 6-29
lm_initialize_unsequenced_lock: 6-30
lm_obtain_sequenced_lock: 6-31
lm_obtain_sequenced_lock_no_wait: 6-32
lm_obtain_unsequenced_lock: 6-33
lm_release_sequenced_lock: 6-34
lm_release_unsequenced_lock: 6-35
misc_obtain_spin_lock: 6-36
misc_release_spin_lock: 6-37
Clock Routines 6-38
Constants and Data Structures 6-39
misc_clock_value_type 6-39
vp_establish_timeout: 6-41
vp_cancel_timeout: 6-42
vp_specify_max_timeouts: 6-43
vp_create_clock_event: 6-44
vp_read_system_clock: 6-45
Interrupt Handling Routines 6-46
Constants and Data Structures 6-46
uc_interrupt_enum_type 6-46
io_mask_interrupt_variety : 6-48
io_unmask_interrupt_variety: 6-49
vp_are_interrupts_disabled: 6-50
vp_disable_interrupts: 6-51
vp_enable_interrupts: 6-52

Chapter 7 — Data and Memory Management Routines

Memory Management Routines 7-2
Constants and Data Structures 7-3
Page Alignment Literals 7-3
vm_get_physical_byte_address: 7-5
vm_get_unwired_memory: 7-6
vm_get_wired_memory: 7-7
vm_map_physical_memory: 7-8
vm_unmap_physical_memory: 7-11
vm_mark_mod_and_ref_and_unwire_memory: 7-13
vm_mark_ref_and_unwire_memory: 7-14
vm_perhaps_get_unwired_memory: 7-15
vm_perhaps_get_wired_memory: 7-16
vm_release_unwired_memory: 7-17
vm_release_wired_memory: 7-18
vm_unwire_memory: 7-19
vm_wire_memory: 7-20

User Data Access Validation Routines 7-21
Constants and Data Structures 7-21
sc_check_access_and_read_string_from_user: 7-22
sc_check_byte_access: 7-24
sc_read_bytes_from_user: 7-25
sc_write_bytes_to_user: 7-26
sc_write_string_to_user: 7-27
Buffer Vector Management Routines 7-28
Constants and Data Structures 7-29
io_add_to_buffer_vector_position: 7-30
io_get_buffer_vector_io_info: 7-31
io_get_buffer_vector_position: 7-33
io_get_buffer_vector_residual: 7-34
io_get_buffer_vector_byte_count: 7-35
io_init_buffer_vector: 7-36
io_init_one_entry_buffer_vector: 7-37
io_read_from_buffer_vector: 7-38
io_reset_buffer_vector_position: 7-39
io_set_buffer_vector_residual: 7-40
io_write_to_buffer_vector: 7-41

Chapter 8 — General Driver Routines

Configuration Routines 8-2
Constants and Data Structures 8-3
fs_dev_request_type 8-3
fs_dev_request_operation_enum_type 8-4
fs_dev_create_request_type 8-4
io_dev_adapt_info_type 8-5
Literals 8-5
uc_device_class_enum_type 8-6
uc_device_code_type 8-6
Integrated Device Code Literals 8-7
uc_reset_enum_type 8-7
fs_submit_dev_request: 8-9
io_add_to_register_list: 8-10
io_allocate_device_number: 8-11
io_deallocate_device_number: 8-13
io_deregister_device_info: 8-14
io_check_device_spec: 8-15
io_forget_device_spec: 8-16
io_do_first_short_board_access: 8-17
io_do_first_long_board_access: 8-18
io_get_device_info: 8-19
io_map_device_number: 8-21
io_parse_device_spec: 8-23
io_perform_reset: 8-25
io_register_device_info: 8-26
Driver Daemon and Generic Daemon Routines 8-28
Constants and Data Structures 8-28
io_queue_message_to_driver_demon: 8-29

io_specify_max_demon_messages: 8-31
io_queue_message_to_generic_demon: 8-32
io_specify_max_generic_demon_messages: 8-34
Error Encoding and Logging Routines 8-35
Constants and Data Structures 8-37
SC_NO_ERRNO 8-37
SC_ENCODE_STATUS: 8-38
io_err_log_error: 8-39
Select Manager Routines 8-41
Constants and Data Structures 8-42
io_select_cancel: 8-43
io_select_init: 8-44
io_select_register: 8-45
io_select_satisfy: 8-46
Miscellaneous Driver Routines 8-47
Constants and Data Structures 8-47
fs_check_self_id: 8-48
io_hex_str_to_int: 8-49
misc_format_line: 8-50
pm_is_super_user: 8-52
sc_panic: 8-53
Nodevice Routine Stubs 8-54
io_nodevice_open: 8-56
io_nodevice_close : 8-57
io_nodevice_read_write : 8-58
io_nodevice_select: 8-59
io_nodevice_ioctl : 8-61
io_nodevice_start_io : 8-63
io_nodevice_configure : 8-64
io_nodevice_deconfigure : 8-65
io_nodevice_name_to_device : 8-66
io_nodevice_device_to_name : 8-67
io_nodevice_open_dump : 8-68
io_nodevice_write_dump : 8-69
io_nodevice_read_dump: 8-70
io_nodevice_close_dump : 8-71
io_nodevice_powerfail : 8-72
io_nodevice_mmap: 8-73
io_nodevice_munmap: 8-74
io_nodevice_maddmap: 8-75
io_nodevice_service_interrupt: 8-76

Appendix A — A Sample SCSI Device Driver

Data Definitions: dev_sd_def.h A-1
Static Global Data: dev_sd_global_data.c A-11
Miscellaneous data: dev_sd_message_data.c A-12
Main Driver C Code: dev_sd_driver.c A-12
System File Entries A-85
Master File Entries A-86

Appendix B — A Sample SCSI Adapter Driver

Data Definitions: dev_cisc_def.h B-1
Static Global Data: dev_cisc_global_data.c B-19
Main Driver C Code: dev_cisc_driver.c B-21
Adapter Management Code: dev_cisc_mgr.c B-32
Driver Utility Code: dev_cisc_util.c B-49
System File Entries B-72
Master File Entries B-73

Appendix C — Standard Peripherals and Their Defaults

AViiON System I/O Defaults C-2
AViiON Station I/O Defaults C-6
SCSI IDs C-7
Device Specifications C-8
Disk and Tape Command Set Compatibility C-10

Appendix D — Glossary

Documentation Set

Index

Tables

Table

3-1 Routine Classes and Their Include Files 3-2

C-1 AViiON System I/O Address and Interrupt Level/Vector Defaults C-2
C-2 AViiON Station I/O Address Defaults C-6
C-3 Default SCSI IDs C-7
C-4 AViiON System Device Specification Parameters C-8
C-5 AViiON Station Device Specification Parameters C-9

Figures

Figure

1-1 Diagram of the AViiON System I/O Architecture 1-5
1-2 Diagram of the AViiON Station I/O Architecture 1-6
1-3 The Adapter/Kernel and Device/Adapter Interfaces 1-8

C-1 AViiON System Memory-Mapped I/O Addresses and Data Width Areas C-4

Chapter 1
Introduction to Writing a Device Driver

This manual contains information you need to integrate a device driver into the DG/UX kernel. It details the rules and interfaces that affect the relationship between the driver and the kernel. It describes when routines in the driver will be called, what assumptions they must take into account, and what actions they must take. It also describes kernel routines that the driver may call.

Introduction

This chapter provides a general overview of the DG/UX operating system environment in which your device driver will reside. This chapter focuses on hardware architectural issues and relevant facets of the kernel environment.

We assume that you have a working knowledge of Data General hardware and software architecture. If you are not familiar with these topics, or if you need more information, please refer to the manuals listed in the Preface, in the section called "Related Documents."

You may write a device driver for either a hardware device or for a software virtual device (called a pseudo-device). Much of this manual applies to both types of devices. However, for pseudo-devices, information about the I/O architecture does not apply.

Changes to This Release of the Manual

The following list summarizes the changes documented in this release of the manual:

In Chapter 2:

Corrections to the compile command line are given in the "Rebuilding and Rebooting the System" section.

In Chapter 3:

A note cautioning users to initialize their data has been removed. Enhancements to the 4.30 linker (**ld**) now make sure that data goes to the correct section of the

program area regardless of whether or not it has been initialized.

The "Driver Daemon" section has been revised and renamed to include the new Generic Daemon that handles I/O completion routines that may pend during processing.

The "Error Reporting Facilities" section has been rewritten to clarify new error reporting facilities.

In Chapter 4:

The **dev_xxx_configure** routine has a clarification regarding verification of the name string parameter.

The **dev_xxx_service_interrupt** routine has a clarification regarding use of the driver and generic daemons to pass information to other processes.

Corrections and clarifications have been made to parameters in the **dev_xxx_ioctl** routine.

Minor changes have been made to the **io_driver_routines_vector_type** **dev_adapter_request_block_type** structure, the DEV_SCSI_REQUEST_FLAGS literals, the **dev_adapter_physical_request_blk_type** structure, and the **sense_bytes** field of the **dev_scsi_adapter_unit_options_block_type** structure. Note also the change to the **version** field of **io_driver_routines_vector_type.**

Dummy interfaces have been added for the **dev_xxx_read_dump**, **dev_xxx_mmap**, **dev_xxx_munmap**, and **dev_xxx_maddmap** operations that will be supported in an upcoming release.

In Chapter 5:

The **dev_scsi_adapter_configure** routine has a parameter change.

A restriction of data transfers to even numbers of bytes with buffers starting on even byte boundaries are noted for the **dev_scsi_adapter_issue_command**, **dev_scsi_adapter_issue_async_command**, and **dev_scsi_adapter_issue_command_physical_mode** routines.

A parameter has been changed in the **dev_scsi_adapter_get_device_info** routine.

In Chapter 6:

Clarifications have been added to the "Clock Routines" section on synchronous versus asynchronous use of the clock.

In Chapter 8:

Minor changes have been made to the **io_add_to_register_list**, **io_check_device_spec**, and **io_parse_device_spec** routines.

Two new routines, **io_forget_device_spec** ("Configuration Routines" section) and **io_err_log_error** ("Error Encoding and Logging Routines" section) have been added.

Two routines for handling the Generic Daemon have been added. These routines, **io_queue_message_to_generic_demon** and **io_specify_max_demon_messages**, are in the "Driver Daemon and Generic Daemon Routines" section.

Nodevice stubs have been added, **io_nodevice_read_dump** and **io_nodevice_maddmap** ("Nodevice Routine Stubs" section). routines.

In Appendixes A and B:

New sample drivers have been provided.

In Appendix C:

A note on mapping logical A24 and A32 address space to a physical address has been added.

The 4.30 driver interface represents the stable base interface for device drivers on the DG/UX system. New interfaces, notably the **dev_xxx_read_dump**, **dev_xxx_mmap**, **dev_xxx_munmap**, and **dev_xxx_maddmap** interfaces, will be added in a upcoming release, but currently defined interfaces are intended to be final.

NOTE: In order to expand discussion on basic kernel programming topics, this manual is scheduled to be restructured into two manuals. The new manuals, *Programming in the DG/UX™ Kernel-Level Environment* and *Writing a Device Driver for the DG/UX™ System*, will be available in the August 1990 time-frame.

Overview of Architectural Issues

This manual applies to drivers for all AViiON series machines (both workstations and systems) that are running the DG/UX operating system. In order to make drivers independent of the architecture of the different AViiON series machines, the DG/UX kernel handles most architecture dependencies itself. However, there are architectural features common to all AViiON machines that are part of your driver's environment. This section discusses these common features as well as certain kernel facilities that help your driver stay architecture-independent.

Throughout this manual we will refer to all AViiON series machines simply as AViiON machines.

Memory-Mapped I/O

On AViiON machines, drivers access their devices via memory-mapped I/O. This means you will read and write to a specific area of memory that is dedicated to your device. With memory-mapped I/O, assembly language programming becomes unnecessary because you can access your device using simple memory reference instructions.

For most devices, you set the device's memory-mapped I/O address by setting jumpers on the device itself. For devices it supplies, Data General pre-assigns and jumpers the memory-mapped I/O addresses according to the manufacturer's default address (that is, the address set at the factory). However, if you add a non-standard device or a second instance of a standard device, you will have to jumper the I/O address on your hardware. More importantly, you will have to choose an address that is not already used by another device. Appendix C shows conventions and restrictions for choosing a memory-mapped I/O address. Appendix C also lists standard devices and their default addresses.

I/O Architecture: Controllers, Adapters, and Devices

For purposes of writing device drivers, the DG/UX kernel defines three major types (or levels) of peripheral devices: controllers, adapters and devices. Your device's peripheral level affects the type of driver you will write. This section discusses these structures and their implications for device drivers.

For the rest of this manual, we will use these terms in specific ways, with specific implications. The following list defines the terms as we will use them.

- The term adapter refers to an I/O device designed to manage an independent secondary bus. An adapter converts signals from the primary system bus to the secondary bus and serves as a conduit between the CPU and devices attached to the secondary bus. An adapter can interrupt the CPU directly. An SCSI adapter supporting an SCSI bus with SCSI devices is an example of an adapter.

- The term controller refers to an I/O device designed to manage several lower level peripherals, all of the same type. It controls them directly not via an independent bus. Like adapter, controllers directly interrupt the CPU. A line controller supporting several asynchronous I/O lines is an example of a controller.

- The term device refers to the lower level peripherals attached to either controllers or adapters. These lower level devices do not interrupt the CPU directly.

For the most part, devices off controllers are simply considered to be sub-units of the controller. On the other hand, devices off adapters (that is off a secondary independent bus) are considered to have a degree of independence from the adapter.

Drivers for the different levels of peripherals are designed to address the different functions at each level.

Adapter, Controller, and Device Layouts on Different Machines

The definitions of adapter, controller and device apply across AViiON machines. However, on a particular machine, the layout of the different peripheral levels will vary with the I/O architecture. For example, Figure 1-1 and Figure 1-2 show adapters, controllers and devices on the AViiON 5000 series systems and the AViiON 300 series stations, respectively. Throughout this manual, we will use the AViiON 5000 series systems and the AViiON 300 series stations to provide concrete examples of adapters, controllers, and devices under different I/O architectures.

NOTE: Throughout this manual we will refer to AViiON 5000 series machines as AViiON systems, and to AViiON 300 series machines as AViiON stations.

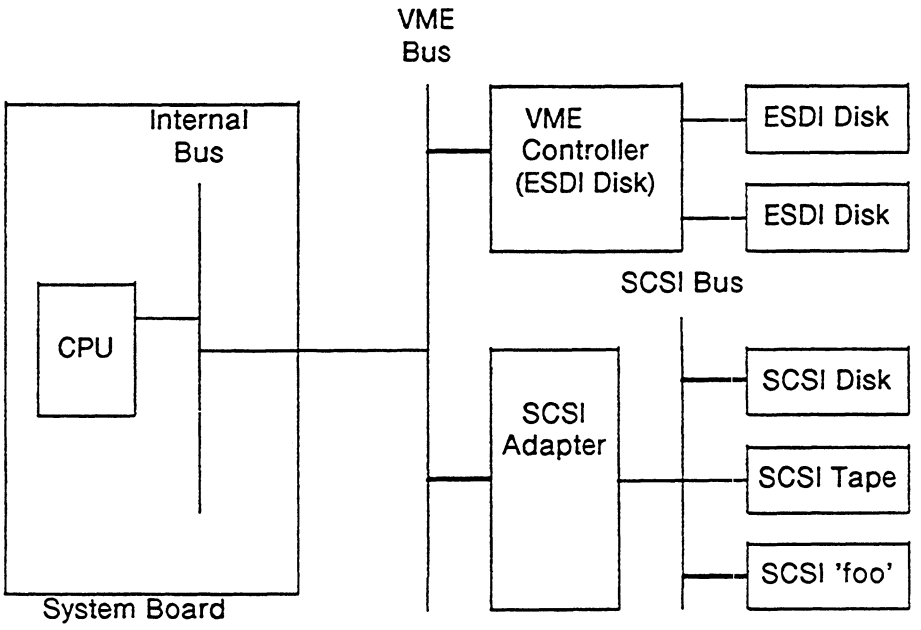

Figure 1-1 Diagram of the AViiON System I/O Architecture

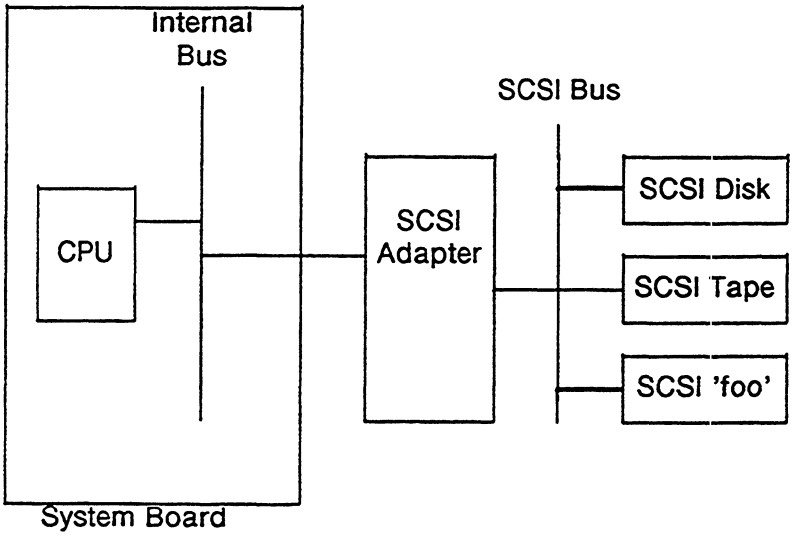

Figure 1-2 Diagram of the AViiON Station I/O Architecture

Both the AViiON system and the AViiON station support SCSI adapters, but their positions in the two architectures are different. In particular, note the layout of the I/O buses and where the SCSI adapters are attached on the two machines. The AViiON station uses an integrated I/O bus with a single SCSI adapter attached to that integrated bus. The AViiON system has an integrated bus on the system board, but it also has a VMEbus as the primary external I/O bus. On the AViiON system, SCSI adapters attach to the VMEbus.

You need to know how your adapter/controller is attached on your target system because architectural differences affect how your driver retrieves interrupts and the interrupt device class to which it is assigned. For example, the I/O bus to which an adapter is attached defines its device class. Thus, on the AViiON station, the SCSI adapter is an integrated device, while on the AViiON system the SCSI adapter is a VME device. We discuss these issues in the "Interrupt Structure" section of this chapter. The next section describes the different kinds of drivers on the DG/UX system.

Adapter Drivers and Device Drivers

The DG/UX system supports two different types of drivers: adapter drivers and device drivers. This section explains the basics of these two types of driver. We use SCSI peripherals as an example of adapter driver/device driver issues.

Portability of driver code is one of the DG/UX kernel's major goals. Therefore, whenever possible, manufacturer-specific operations are separated off from manufacturer-independent operations. This means part of the code will be fully portable and part will be manufacturer-specific. Such separation is particularly possible when an interface is defined by a standard such as is the case with SCSI I/O.

For example, all SCSI devices off an SCSI bus follow the SCSI standard and thus are manufacturer-independent. However, operation of each SCSI adapter is not standard-defined, so different manufacturers' SCSI adapters use different command codes and sequences. From a driver's perspective, all SCSI disks operate the same way regardless of their manufacturer, but one has to go through a manufacturer-specific adapter interface to operate them. In order to maximize the amount of portable code, the DG/UX system puts manufacturer-specific SCSI adapter functions into one driver (an adapter driver). This leaves the manufacturer-independent functions as a fully portable driver. Thus, one SCSI disk driver (the **sd** device driver) works for disks on any supported SCSI adapter.

Separation of manufacturer-specific and manufacturer-independent code is not useful with controllers because their devices are essentially sub-units of the controller. Hence, the controller and all its devices are handled by a single driver. This driver accesses the controller and specifies which unit off the controller it wants to address. Thus, controller drivers are usually specific to a particular manufacturer. For example, the **cied** driver works only for Ciprico ESDI disk controllers.

The adapter drivers and device drivers both consist of a set of externally callable routines. All drivers supply a set of 15 basic I/O routines including configure, open, close, read, write etc. (see Chapter 3). Drivers that supply only these basic routines are called device drivers. In addition to these 15 basic driver routines, adapter drivers have an additional set of adapter routines.

SCSI adapter drivers and SCSI device drivers form a paired system. The kernel passes all user I/O requests to the SCSI device driver (for example, the SCSI tape driver **st** or the SCSI disk driver **sd**). The SCSI device driver in turn issues a request to the SCSI adapter driver, which accesses the physical device and returns the results to the device driver.

Generally, the kernel interfaces to the SCSI device driver, which in turn interfaces to the SCSI adapter driver. The main exception to this rule concerns interrupts. SCSI adapters interrupt the host, but SCSI devices do not. Therefore, it is the SCSI adapter driver that needs to service interrupts. The adapter driver must interface to the kernel in order to receive its interrupts.

NOTE: The kernel may also invoke the adapter driver in response to other system needs. For example, the kernel may invoke the adapter driver's configure routine or its open dump routine. However, because user-level I/O goes to the device driver, most of the adapter driver's basic routines may be left as stubs. For example, adapters will not generally need a read routine. Chapter 3 lists the basic driver routines that the adapter driver must supply (that is, cannot leave as stubs).

Figure 1-3 shows how the different drivers interface to each other and to the hardware.

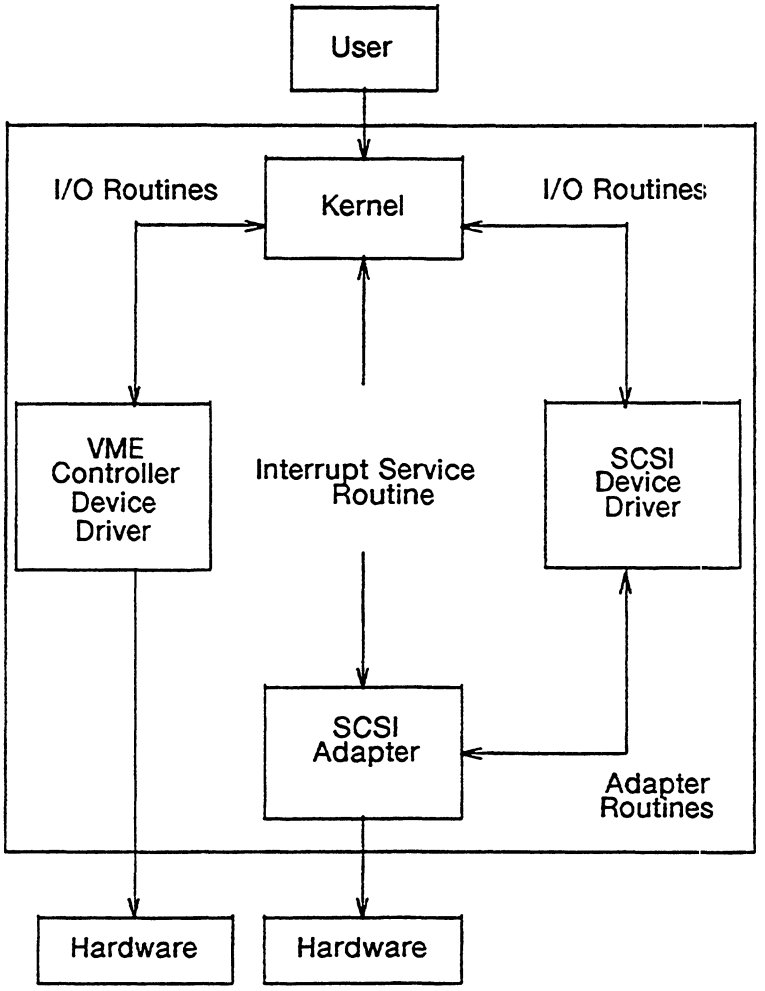

Figure 1-3 The Adapter/Kernel and Device/Adapter Interfaces

Throughout the rest of this manual we will use *device* driver to refer to drivers that provide only the standard set of routines. Thus, both VME controller/unit drivers and SCSI device drivers will be called device drivers. The term *adapter* driver will refer to drivers with the additional adapter routines (which means only those drivers intended for managing an adapter).

The Adapter Manager

The driver interface has one more major component that must be mentioned. We've seen that driver portability is a major focus under the DG/UX kernel. We've also seen that SCSI adapter drivers are specific to a particular manufacturer's adapter whereas SCSI device driver's are not. Therefore, if an SCSI device driver is to work

with all adapter drivers, it must not have calls to a specific adapter driver in its code. To eliminate this problem, the kernel provides a generic "adapter manager" that routes device driver calls to the appropriate adapter driver. Thus, the device driver calls a generic adapter routine (in the kernel's adapter manager), passing a parameter that indicates the adapter driver for which the call is intended. The generic adapter routine decodes the parameter and passes the call on to the actual adapter driver. We discuss the adapter manager routines in Chapter 5.

Do You Have to Write a New Driver?

The DG/UX system provides full System V and BSD functionality with a kernel that supports fully symmetric multiprocessing as well as other enhancements. The DG/UX kernel requires all drivers to conform to a set of standard interfaces. Only drivers that follow the DG/UX interface specifications will run under the DG/UX system.

If you buy an SCSI adapter, SCSI device, or VME controller, the hardware is likely to be compatible (see Appendix C for a list of compatibility specifications). Nevertheless, software drivers written for System V or BSD will not run under the DG/UX kernel. However, this does not necessarily mean that you will have to write a new device driver for your hardware.

As we have seen, SCSI device drivers work for all SCSI adapters, and the DG/UX system supplies SCSI device drivers for a number of standard devices. For example, the DG/UX SCSI disk driver, **sd**, and the SCSI tape driver, **st**, work with disks or tapes that adhere to the basic Common Command Set plus several non-mandatory Common Command Set commands (Appendix C lists these additional commands). If you are adding a new device of the same family as an existing Data General-supplied device driver, chances are you will be able to use the supplied device driver. If you add a new type of SCSI adapter you will probably need to write an SCSI adapter driver for it. On the other hand, you can use the existing Data General-supplied device drivers with your new adapter driver.

In general, you will only need to write an SCSI driver: 1) if you add an SCSI adapter of a type not already supported under the DG/UX system; or 2) if you add an SCSI device of a type not already supported under the DG/UX system. You will need to write a driver for a VME controller if you add a controller of a type not covered by Data General-supplied drivers.

Interrupt Structure

Traditionally, a machine's interrupt structure is a major cause of machine-dependencies in device drivers. In order to make drivers machine-independent, the DG/UX kernel hides most interrupt structure dependencies from the drivers.

Three closely related features allow drivers to remain independent of most interrupt structure dependencies. The first feature has to do with registering interrupt

handlers. If your device generates interrupts, your driver will include an interrupt handler that will service your device's interrupts. When a device of your driver's type is configured into the system, the kernel will call your driver's configuration routine, which will in turn call a routine to register your interrupt handler with that device. Once the handler is registered, whenever that particular device generates an interrupt, the kernel will pass control to your driver's handler.

Yet, for the kernel to pass control to the right interrupt handler, it must be able to identify which interrupt comes from the particular device. Further, in order to be machine-independent, the driver must be able to supply interrupt information using kernel-supplied literals instead of machine-specific values. This leads to the second machine-independent feature — a set of interrupt identifiers that can uniquely identify devices both across machines and regardless of particular configurations within a particular machine.

The kernel's approach to creating a set of such unique identifiers requires that you supply two parameters. One parameter gives the class of interrupts and the other gives a unique device identifier within that class. Thus, to register your interrupt handler, you must supply a device interrupt class, a device code, and a pointer to a device information structure that identifies your interrupt handler.

The interrupt class and device code parameters are defined as follows:

- **Interrupt (or Device) Class**

  The interrupt's class is defined by the bus on which the device is located. If the device is attached to a Data General proprietary bus integrated on the system board or a bus expansion slot off the system board, it belongs to the **integrated class** of device interrupts. If it is attached to any bus other than these, the device class is defined by the particular bus. For example, the AViiON 5000 series systems support an external VME-188 bus. All devices on this VME bus would belong to the **VMEbus class** of device interrupts. The kernel supplies an enumeration type that defines device class literals. Use these literals to specify your device's interrupt class. See Chapter 8 for a discussion of the device class enumeration types.

  Note: A device's class is defined by the external bus to which it is attached and which interrupts the CPU. Thus, if a device is attached to a secondary bus serviced by an adapter, it is the adapter's bus that will define the device's class because it is the adapter that will interrupt the host CPU. For example, SCSI devices (those on an SCSI bus) are serviced by an SCSI adapter. On an AViiON 5000 series system, the SCSI adapter will be attached to a VME bus. Because only the adapter interrupts the CPU, the device class for the SCSI devices and the SCSI adapter is **VMEbus**.

- **Device Identifiers (or Device Codes)**

Unique device identifiers within a device class are called device codes. Device code definitions vary with the device class.

For devices in the integrated class, the kernel supplies a set of literals for all possible types of devices found in the integrated class. For example, you use the literal **UC_DUART_DEVICE_CODE** to identify any integrated duart device. Chapter 8 describes the device code literals for the integrated class of devices.

For other device classes, the device code is generally defined by a unique identifier jumpered on the board. For example, on the AViiON 5000 series machines, controllers on the VME bus are jumpered to a particular vector number. The bus passes this vector number to the CPU and, when the device interrupts, it appears in the Interrupt Acknowledge register as the device identifier. Appendix C describes conventions for the **VMEbus** class device codes.

Note that your driver will be specific to a particular device class (for example, integrated or **VMEbus**). Thus, you will need different drivers for SCSI adapters in the integrated class (as on AViiON stations) and for SCSI adapters in the **VMEbus** class (as on AViiON systems). Drivers dependency on device class results from the fact that device codes are interpreted differently in the different device interrupt classes.

When an interrupt occurs, the kernel will read the interrupt status register (IST) and pass control to the registered interrupt service routine whose device code and device class match the interrupt. Once it recieves cotrol, the interrupt service routine must clear the interrupt if reading the IST did not clear it.

The final way the kernel helps you avoid architecture-specific code is by providing interrupt handling routines that you can use to mask and unmask interrupts. Chapter 6 describes these routines.

The combination of registering interrupt handlers with interrupt identifiers and kernel-supplied mask and unmask routines allow the bulk of your driver to be architecture independent; it should run on any AViiON machine.

The Multiprocessor Environment

The DG/UX kernel is designed to operate symmetrically on one or more processors. To do this, the kernel creates an abstraction of a physical processor called a *virtual processor* (VP). By using a VP abstraction, the hardware implementation (actual number of physical processors) can be made transparent to the higher levels of the kernel. The actual physical processor is called a *job processor* (JP).

A given instance of the kernel will have a fixed number of VPs that is usually greater than the number of physical processors but less than the number of processes wanting

to execute. A two-level scheduling scheme is used to balance between processes, VPs, and JPs. The lower level of scheduling multiplexes VPs onto physical processors so that the VPs appear to be active entities that execute code. This short-term scheduling is performed by the dispatcher. A higher level of scheduling multiplexes processes onto VPs so that the processes may execute. This higher scheduling is performed by the medium-term scheduler using the operations defined on VPs.

As you write your driver, keep in mind that it may be operating in a multiprocessor environment. There are two important points about such an environment:

- Do not presume that the driver is the only process running at a given time. Another process might be running on another processor and accessing common memory. This means that synchronization and locking issues are very important. The driver should protect (lock) access to critical data structures because another process might seek access at the same time. Chapter 6 describes the kernel routines that support synchronization and locking operations.

- Driver code may be executed simultaneously on two or more different processors. In particular, when a device interrupts on a multiprocessor system, one of the processors is picked by the system to take the interrupt. The driver has no control over which processor takes the interrupt. The processor chosen for interrupts from a particular device may vary from interrupt to interrupt. In addition, more than one processor can service an interrupt at a time. As a result, it is possible for a controller to have two interrupt requests serviced at the same time.

  Finally, if you disable interrupts on a multiprocessor system, you do so only for the processor that is currently running. The device may still interrupt on another processor. As a result, base-level driver code cannot disable interrupts to protect against collision with the driver's own interrupt service routine. Disabling interrupts on one processor does not guarantee that the interrupt service routine will not access a structure at the same time on another processor.

  You can mask interrupts for the device. This prevents the device from interrupting on all processors. Remember, however, that masking does not guarantee that an interrupt is not already in progress.

How to Write a Device Driver

In this section, we describe the major steps involved in writing a device driver. These steps are as follows:

1) **Write I/O routines for your device.**

   - The kernel requires that you supply a set of I/O routines for your device. You supply these routines in a file called **dev_*xxx*_driver.c**,

where *xxx* is a two- to eight-character device mnemonic identifying your device. The mnemonic may be composed of digits and uppercase and lowercase letters; it is case sensitive. The first character must be a letter. This mnemonic is also used in the master file entry described in Step 2.

The I/O routines you supply will include such routines as **open**, **close**, **read**, and **write**. Throughout this manual, we refer to these routines in the following way: **dev_*xxx*_*name***. For example, the **read** routine for the **xdev** driver would be **dev_xdev_read**. In this book, we use **xxx** as a generic driver name. Thus, our generic driver's **read** routine will be **dev_xxx_read**.

DG/UX I/O is device-independent, which means that all user-supplied I/O routines conform to a specified interface with a standard set of parameters. The kernel calls routines at the appropriate times (for example, **read** when a read operation is requested). The interface specifications for these routines are discussed in Chapter 4.

- The DG/UX kernel contains many routines for system-level operations that you can call from your driver. The operations performed range from managing memory to handling signals to interfacing with the driver daemon. Chapters 6 through 8 describe these routines in detail.

2) **Supply master file information describing your device driver.**

The basic information about all devices is contained in files called master files. You will need to make an entry in a master file to identify your device to the kernel. We discuss master file entries in Chapter 2.

3) **Supply system file information for each controller of your device type.**

The system file supplies information linking master file information to specific instances of devices in a particular configuration. Your driver services all *xxx* type devices, and you may have more than one such device on the system. For example, a system might have two **cied** type disks. One driver services all **cied** disks. The system file will have an entry identifying each disk of the **cied** type. We discuss system file entries in Chapter 2.

4) **Rebuild and reboot the system.**

After the entry has been placed in a master and system file, you can use the standard system-generation procedures. The module or modules implementing the new driver must be separately compiled and included on the link line when the system image is linked. In Chapter 2, we describe how to incorporate your driver in the system build.

Overview of Device Driver Environment

This section describes a number of terms and concepts that are fundamental to writing a driver on a DG/UX system. Also, because a device driver is an integral part of the kernel, the driver must conform to the restrictions that apply to various parts of the kernel. We discuss some of these restrictions in this section.

Device Specifications

All devices (controllers, adapters, and devices/units off controllers and adapters) must also have a unique software descriptor called a *device specification*. You will need to provide device specifications at various times; for example, for system file entries, for various system utilities and at boot time.

The kernel uses device specifications to link specific devices with the appropriate device driver. Specifically, the device specification for devices your driver will service must begin with your driver's device mnemonic.

The kernel passes the device specification string to the driver for interpretation. Therefore, you could write your driver to use a device specification syntax different from that used by drivers supplied with the DG/UX system. However, for consistency and intelligibility, we recommend that you implement your device specification like DG/UX drivers. You can use the **io_parse_device_spec** routine (described in Chapter 8) to parse a device specification according to Data General's conventions.

DG/UX drivers use the following device specifications syntax:

```
device_mnemonic [@device_code] ([parameters])
```

where:

`device mnemonic` is the two- to eight-letter mnemonic used to identify the device driver for the device. The **xxx** code described in Chapter 2 is your device mnemonic. Appendix C lists the device mnemonics for Data General-supplied device and adapter drivers.

`device code` is a device identifier that uniquely identifies a physical device within its interrupt class. For devices with device codes, you enter the device code preceded by an @ (at) sign (for example, @18). Device codes are defined within each specific device class. However, only devices that directly interrupt the host have device codes. Devices that do not have device codes (such as pseudo-devices or SCSI devices off SCSI adapters) must omit the device code field in their device specification.

`parameters` are values that provide additional information to the driver. The parameters for the device specification depend on the type of device and whether the device is a controller, adapter, or device (unit).

Controller/Adapter Parameters

The device specification for an adapter consists of the adapter's name, its device code, and a single parameter identifying which adapter is being addressed. The device specification for a controller consists of the controller's name, its device code, a parameter identifying which controller is being addressed, and a second parameter specifying which device off the controller is addressed (for example, unit #1 off the controller).

For both controllers and adapters, the first parameter indicates which controller or adapter is being addressed. For drivers supplied with the DG/UX system, you can identify which controller/adapter is being addressed in either of two ways:

1) You can specify the controller/adapter by giving its base memory-mapped I/O address. For example, the first **cied** adapter would be **cied(ffffef00)** (the **cied** mnemonic stands for Ciprico ESDI disk). Appendix C lists the base addresses for drivers supplied with the DG/UX system.

2) If the controller/adapter is located at one of the standard base addresses for a device of its type (see Appendix C), you can use the numbers zero (0) or one (1) to indicate the first or second instance of this device. For example, you can use **cied(0)** and **cied(1)** to specify the first and second **cied** controllers. If you omit the first parameter, the driver should assume a value of zero. Drivers supplied with the DG/UX system can deduce the base address from this information.

NOTE: You cannot use this form if you are addressing a controller or adapter whose base address is not a default (as shown in Appendix C).

SCSI Device Parameters

For SCSI devices, the first parameter indicates on which adapter the device is located. You identify the adapter with its device specification as just described. For example, device specification for the SCSI disk off the AViiON station integrated SCSI adapter would be **sd(insc(0),2)**.

The second parameter is the device's SCSI ID. The SCSI ID is a bus identifier jumpered on the device. A device's SCSI ID must be unique on its adapter but not across adapters. (Appendix C lists the default SCSI IDs for standard devices on the DG/UX system.)

The device specification has a third parameter that you can use to specify a unit number if the SCSI device is a controller with multiple units.

NOTE: In device specification, device codes and base addresses are interpreted as hexadecimal numbers. You must *not* precede them with "0x" as is conventional in C language programming.

The following are valid device specifications:

cied()	**cied** disk controller with all parameters assuming their default values. (See Appendix C for a list of default values.)
cied(0,1)	Drive 1 on **cied** disk controller 0.
cied@77(fffff500,0)	Drive 0 on the **cied** disk controller at the non-standard base address 0xfffff500, with the non-standard device code 0x77.
sd(cisc(1),2)	The SCSI disk at SCSI ID 2, reachable through SCSI adapter 1.
st(insc(0),2)	The SCSI tape at SCSI ID 2, reachable through integrated SCSI adapter.
sd(cisc@77(fffff500,0),2)	Disk drive 2 on the **cisc** SCSI adapter at the non-standard base address 0xfffff500, with the non-standard device code 0x77.

Device specifications are described in more detail in *Installing and Managing the DG/UX™ System*.

Special Files (Nodes)

A user accesses a specific unit of a specific device through a *device special file*, also called a *node*. When you open a special file, you get a file descriptor that identifies the specific unit. The user-level code will then use this file descriptor to access the device. Special files are stored in the **/dev** directory.

For example, a disk special file called *node_name* is located in **/dev/dsk**. To open the device and get a file descriptor, the user program issues the following DG/UX system call from a C program:

```
int fd;
fd = open ("/dev/dsk/node_name", O_RDWR)
```

The kernel returns a file descriptor into **fd**. The user will now use **fd** to access the specific device. For example,

```
read (fd, Buffer, 20);
```

causes the kernel to read from the device identified by the special file *node_name*. **fd** points to this file, and 20 is the number of bytes to be read into the memory area

denoted by `Buffer`.

Users can create special files with **rc** scripts and with the **mknod**(1) command. However, at configuration time, the device driver's **dev_xxx_configure** routine also creates special files from device entries given in the system file. The driver usually uses the system file entry (device specification) as the special file's name. Thus, many special files will have a device specification for a name. We describe the interface for the **dev_xxx_configure** routine in Chapter 4.

The file descriptor identifies a special file for a specific device. The special file must therefore describe that device. A special file represents the following information:

- Type of I/O interface — block or character
- The major number
- The minor number
- Access rights

The major number identifies a family of devices all serviced by the same device driver. The kernel uses the major number as an index into a table of vectors containing pointers to each driver's I/O routines. When the kernel recieves a user I/O request, it identifies the correct driver routine to call using the major number and the driver's I/O routines vector. We discuss how to supply a major number in Chapter 2.

The minor number is used to identify a specific unit in a particular device class. The driver **dev_xxx_configure** operation calls the kernel device number manager that allocates minor numbers for each unit and links unit-specific information to the minor number.

When a user opens a device, the kernel sends to the driver **dev_xxx_open** routine the major and minor device numbers for the opened device. The **dev_xxx_open** routine then calls the **io_map_device_number** routine to map the major and minor numbers to unit-specific information.

The driver can use unused bits of the allocated minor number to hold additional information about a unit. For example, tape devices use bits in the minor number to specify density selection on a unit. Note that special-purpose bits must be masked out before any interaction with the device number manager.

See *Installing and Managing the DG/UX™ System* for more information on special files.

Block Versus Character Interface

The DG/UX system supports two major types of I/O interfaces: block special (buffered) and character special (raw). Depending on the device type, drivers can support one or both interfaces.

The block special interface treats the device like a file. The kernel buffers input and output to the device and controls when to do actual reading or writing. Information that is read or written to a block device must pass through the kernel's buffers. An example of a block device is a buffered disk access.

The character interface treats the device as a raw device. The read or written information is transferred directly to and from the user's address space, bypassing the kernel's buffers. The device determines the correct block size and handles all data transfers. An example of a character device is a terminal.

A driver may support either block and character access or character access only. Most driver-supplied routines are the same for both types of access (for example, the same **dev_xxx_open** routine serves both interfaces). The exception is that the **dev_xxx_start_io** routine is used for block special access only.

Driver Execution

All device driver code executes as part of some user or system process running in the kernel. A device driver has access to all of system memory and to all devices. Kernel code is protected from write access so that access errors can be isolated more quickly (note that this protection means a driver cannot use self-modifying code), but no other protection is provided against a driver writing to kernel databases and/or otherwise destroying the kernel internals.

Driver code executes on the kernel stack of the running process. The kernel stack is of fixed size, so driver code must not nest calls too deeply. A system panic results if a process's kernel stack overflows. Panic codes are listed in a file in **/usr/release**; your DG/UX system Release Notice discusses this file.

Because of its special status as part of the kernel, a device driver may not use the standard C libraries or DG/UX system calls (described in Chapters 2 and 3 of the *Programmer's Reference for the DG/UX™ System, Volume 1*).

Memory Management

Two types of kernel memory are visible to a device driver. *Global kernel memory* is addressable by all processes in the system. *Per-process kernel memory* belongs to a particular process and can be addressed by only that process.

You must use care in deciding whether to declare data structures in per-process versus global kernel memory. For example, if you declare an argument to a call in

per-process memory, and then your process completes, the argument will be deleted with your process. Similarly, if you declare a structure in per-process memory, it will not be accessible by interrupt-level code because the interrupt code runs on the currently executing kernel process which has its own per-process memory. Thus, some arguments to device driver calls may be in per-process memory, while other arguments should be restricted to global memory. Note that the user process's kernel stack is in per-process memory.

Logical addresses do not equal physical addresses in the kernel. The addresses may be equal in some situations, but a device driver should not depend upon this. Chapter 7 describes kernel functions that you can use to convert from a logical address to a physical address.

Device driver code and static data reside in wired memory so that they can be accessed from interrupt handlers.

Interrupts

Most device driver code executes with interrupts enabled. The driver should not manipulate the state of the interrupt enable register unless absolutely necessary. If the driver must change the interrupt state, it should use the kernel's interrupt enable/disable routines (described in Chapter 6).

If your device generates hardware interrupts, the driver must supply an interrupt service routine (interrupt handler) to service those interrupts. The interrupt service routine will run with all interrupts disabled on the current processor (interrupts on other processors are not affected).

The interrupt service routine must operate in a severely restricted environment. It is expected to quickly determine what action to take (usually advancing one or more eventcounters) and then dismiss the interrupt. It must not pend or page fault. To avoid page faults, the service routine should not reference unwired memory. It should also avoid calls to routines that might pend or page fault. The kernel routines described in Chapters 6 through 8 indicate whether or not they might pend or page fault.

Interrupts do not nest in the DG/UX system, so each interrupt handler must quickly finish its job and return to base level. Furthermore, interrupts are handled on the kernel stack of the currently running process; no separate interrupt stack is used. Therefore, the interrupt service routine must limit the amount of stack space used by it and any procedure it calls.

For VME devices, reading the Interrupt Acknowledge register acknowledges the interrupt, and on many devices (Release-on-acknowledge devices) this action also clears the interrupt. However, some devices require additional action to clear the interrupt. Consult the documentation for your device to see when and how your device stops asserting interrupts.

Clearing the interrupt frees the device to issue another interrupt. Because another interrupt may be serviced by another processor, it may be handled before the first interrupt service routine has completed.

Signals

Before a device driver waits for an indefinite amount of time for an I/O operation to complete (such as on a read of a user keyboard), it must prepare to receive a signal by calling the appropriate kernel functions. If a signal should occur, the driver must abort the operation and return an appropriate status.

For devices that do not normally require user intervention for an I/O operation to complete (such as a disk), signals do not have to be handled while waiting for the device to respond. The device must, however, be timed out if it fails to respond within a few seconds so that the calling process will not become hung indefinitely if the device should lose power or otherwise fail.

Higher levels of the system are responsible for providing reasonable response to signals. These higher levels may break large user requests into smaller driver-level requests so that signals are not ignored for too long a time. For example, if a user requests that 100 Mbytes be written to the disk, the driver may see only a succession of 256 Kbyte requests. A device driver need not be concerned about the size of a user's request as long as it is making progress on the request and is not depending upon some indefinite external event for continued progress.

End of Chapter

Chapter 2
Adding Your Device Driver to the DG/UX System

This chapter describes the information you will need to perform the following operations:

- Add an entry for your device to a master file.
- Add an entry to the system file for each new hardware device or virtual device attached to your system.
- Rebuild the system and reboot with the new system image.

These operations correspond to steps 2, 3, and 4 of the steps listed in Chapter 1 for adding a driver to your system. This chapter also describes ways in which you can check whether you built your driver into the system properly.

Adding a Master File Entry

Master files are administrative files that contain default information for all supported devices. These files hold information needed for the system configuration. Master files are stored in the **master.d** directory. The main DG/UX master file is **master.d/dgux**. You may want to list this file to clarify the master file entries discussed in this chapter. Master files are discussed in the **master**(4) man page.

You must add an entry for the driver to a master file in **master.d**. The master file has three sections to which you may want to add entries. The sections are as follows:

- *Device section*: holds descriptions of all devices.
- *Keywords section*: defines and sets all configurable parameters.
- *Alias section*: allows you to define aliases for master file device entries.

Within sections, entry lines consist of a number of fields separated by blanks or tabs. Comment lines are preceded by a pound sign (#).

You must add a device description entry for your device in the device section of the master file. You may also want to add a device alias (alias section) and/or

configuration parameters (keyword section), depending upon your implementation needs. We discuss these entries in the next section.

NOTE: All files listed in the **master.d** directory are included in the configuration process. Therefore, do not keep old or backup copies of your master file in **master.d**.

Device Descriptions: The Device Section Entry

For easy management, entries in the device description section of the master file are grouped according to type of device. For example, all types of magnetic tape devices are listed together. (Such grouping is helpful but not necessary.)

Each device description entry contains four fields. The following diagram shows some sample master file device description entries. Lines that start with # are comments.

```
#-----------------------------------------------------
# DISKS
#
#                          Maximum  #
#   Name       Major       of units per     Restriction
#   Prefix     Number(s)   Controller       Flags
#   --------   ---------   -------------    -----------
#
    cied         7         7                n
    sd           6         7                n
#
    xdev        10         4                n
#
#-----------------------------------------------------
```

The **xdev** entry above is a non-standard device we have added to the master file. We'll use this entry as an example to describe the fields in the device section. Information is case sensitive.

```
    xdev       10          4                  n
```

xdev **entry name** — This field identifies a family of devices, specifically, all devices that use the same device driver. The entry name or name prefix is a two- to eight-letter device mnemonic. It is also used as part of the corresponding device driver's name, in the device specification (the device mnemonic field) and in corresponding system file entries. The device mnemonic uses any characters that are valid for C language filenames.

10 **major number** — The kernel uses a device's major number as an index into its I/O routine table. Your major number can be any positive number that is less than 255 and that is not already in use. It is a decimal number. To choose a number, scan all master files for major numbers already allocated. We recommend choosing the smallest possible number, as this will keep the size of the table small.

4 **maximum units per controller** — This a decimal number that specifies the maximum number of units a controller can support.

n **restrictions flag** — This flag signals configuration restrictions for this device. The flags are specified as a string of characters with the following definitions (these options are case sensitive):

Option	Meaning
o	Specifies that the driver will allow only one device of this type to be configured. For example, the system console is defined as being the only device of its type.
r	Indicates that the device is required and will be placed in the system whether or not the system file specifies it. If the device is not specified, default values will be given for device specification values.
s	This option indicates that the device is a STREAMS device.
n	No restrictions apply. Choose this option if you do not use any of the others listed above.
z	This device may be configured either explicitly or implicitly as part of nested declaration of another device. For example, "st(insc()),4)" declares "insc()" implicitly.

Parameters: The Keyword Section Entry

If you want to create a parameter for your driver code that can be set at system configuration time, you will need to add an entry to both the master file and system file. For example, the pseudoterminal driver has a variable giving the number of pseudoterminals to be configured. Most device drivers will not use the keyword section.

The master file entry for a parameter should be placed in the keyword section. This entry has four fields:

- **The variable name.** The variable name is used in the corresponding system file entry.
- **The default value for this variable.** This value is used if you do not add a corresponding system file entry to declare the variable's actual value.
- **The variable's data type.** If you don't specify this field, the kernel uses long integer for the data type.
- **The implied value.** This value is used if you add a system file entry but do not give that entry a value. This field is optional and exists primarily to give configuration flexibility for certain special devices such as the Network Filesystem (ONC™/NFS®).

Some sample keyword section entries are shown below:

```
#
# Variable                        Default                        Implied
# Name                            Value          Type            Value
# --------                        -------        -------         ------
#
cf_sc_nodename[]                  "no_node"      char
cf_sci_daylight_savings_kind       1             uint16_type
physbuf                            256           uint16_type
```

To add a configurable parameter, you must add both a master and a system file entry. The system file entry should be placed in the tunable parameters section (see the **system**(4) man page). For example, to change the number of physical buffers (the **physbuf** master file entry), add the the following system file entry:

```
physbuf        150
```

At configuration time, the **config** program combines the master and system entries to produce the file **conf.c**. As a result of the system file entry shown above, **conf.c** will contain a constant **physbuf** with an updated value of 150. After configuration, you can check **conf.c** to see if your variable has been properly set.

You reference your variable as an external variable by inserting a line similar to the following in your device driver:

```
extern int physbuf;
```

Master File Aliases: The Alias Section

The Alias section of the master file allows you to create aliases for your master file entry name. You use such aliases in the system file entry to help distinguish between different controllers of the same device. For example, the asynchronous controllers can have 8 or 16 lines even though the same device driver and master file device prefix are used. The asynchronous controller's (**syac**) aliases might be as follows:

```
Alias   Entry name
-----   ----------
syac8   syac
syac16  syac
```

In the system file, specific 8-line controllers can be referenced as follows:

```
syac8(1)
```

Adding a System File Entry

To configure the new device into the system, you must modify the system file. The system file lists the physical devices or each instance of a pseudo-device that will be configured into the system. It contains device configuration information, particularly hardware I/O addresses. System file entries are described in the **system**(4) man page.

The system file contains two sections: the device selection section and the tunable parameters section. We have already described how to add an entry to the tunable parameters section to set a parameter defined in the master file (see "Parameters: The Keyword Section"). As described, entries to this section are optional.

You must add entries to the device selection section for each physical device of your driver's device type. Use the device specification for this entry.

A typical set of device entries for our **xdev** device might be as follows:

```
xdev@72()
xdev@73(ffff6000,4)
```

Here, `xdev` is the entry name for the master file device description entry. The number 72 is the device code for the first controller, and 73 is the device code of the second controller of this particular class of device. The empty parentheses () in the first entry indicate that the default parameters, including the default base address, apply for this device. The second instance of the `xdev` device shows a non-standard base address and a second parameter of four (4). The parameter's meaning will be specific to the driver's implementation.

Rebuilding and Rebooting the System

You use the standard system-generation procedure, **sysadm**, to build a new system image. However, before you use **sysadm**, you must complete the following steps:

1) Make your changes to the system file and master file as described in this chapter. We recommend you put your master file entries into your own master file. Create a file with you master file entries and put it in **usr/etc/master.d**. You may give this file any name you want as long as it does not match any existing file names in the **master.d** directory.

2) Compile your driver file **dev_xxx_driver.c** to create the object file **dev_xxx_driver.o**.

   If you compile using the GNU compiler that comes with the DG/UX system, we recommend you use the following compile command line:

   ```
   gcc -DSTANDALONE -DKERNEL -D_PRODUCT_DGUX
                             -fno-omit-frame-pointer
                             -mno-underscores
                             -I/usr/src/uts/aviion dev_xxx_driver.c
   ```

   If you compile using the Green Hills compiler, we recommend you use the following compile command line:

   ```
   ghcc -DSTANDALONE -DKERNEL -D_PRODUCT_DGUX
                           -ga -X58 -X153  -X405
                           -I/usr/src/uts/aviion dev_xxx_driver.c
   ```

   If you want to avoid specifying the three defines (STANDALONE, KERNEL and _PRODUCT_DGUX) during compilation, you can add these to one of your source files.

3) Place your driver object file and any archive files you may need into the directory **/usr/src/uts/aviion/lb**.

4) Create a file called **Libs.***driver_name* that lists all the object files and archive files you want included in the build. Place this file in the directory **/usr/src/uts/aviion/cf**. You can get the format of this file by examining other **Libs.** files.

Once you have completed these steps you are ready to build a new system. *Installing and Managing the DG/UX™ System* describes how to use **sysadm** to build a new kernel. The output of the build is a new system image that you will move to the root directory (*/*).

After the new system image is ready, you can shut down the current system and reboot.

Checking the Configuration Process

To verify that your device is properly configured, check both **conf.c** and the special files for your devices. We describe both of these sources below.

Conf.c

The **conf.c** file contains the system tables generated by the **config** program. You can use these structures to verify your configuration and to determine the location of the I/O routines accessing your device. A partial listing of **conf.c** structures and variables is given below with descriptions on how to use the information to verify proper configuration.

Configurable Variable Section

The configurable variable section lists the variables as defined in the keyword section of the master files and modified in the tunable parameters section of the system file. You can check this section for the proper setting of any parameters you set. A partial listing of this section is given below:

```
/*
/* Configurable Variable Section  */
/* ----------------------------   */
/*                                */
char           cf_sc_machine[]                      =     "AViiON";
char           cf_sc_sysname[]                      =       "dgux";
char           cf_sc_release[]                      =       "4.30";
char           cf_sc_version[]                      =         "00";
uint16_type    cf_sci_daylight_savings_time_kind    =            1;
uint8_type     cf_sfm_max_modules_per_stream        =            9;
uint32_type    cf_sfm_max_data_message_length       =         4096;
uint32_type    cf_sfm_max_control_message_length    =         1024;
uint16_type    cf_ps_max_semaphore_sets             =           10;
uint16_type    cf_ps_max_semaphores_per_set         =           25;
   .
   .
   .
```

I/O Driver Tables

The kernel uses a device's major number as an index into a table of driver routines vectors. The I/O Drivers Table in **conf.c** listed below shows this table of routines vectors. Note the entry for our sample **xdev** device in **uint32e_type cf_io_device_driver_vector** below. Also note the major number index listed to the right. The major number you supplied in the system file entry should now reflect the position of your driver in the routine vector table. Chapter 4 explains how you supply

a routines vector for you driver.

```
/* ----------------- */
/* IO Drivers Table  */
/* ----------------- */

extern uint32e_type cfv_syscon_routines_vector;
extern uint32e_type cfv_cied_routines_vector;
extern uint32e_type cfv_devtty_routines_vector;
extern uint32e_type cfv_mem_routines_vector;
extern uint32e_type cfv_ldm_routines_vector;
extern uint32e_type cfv_st_routines_vector;
extern uint32e_type cfv_syac_routines_vector;
extern uint32e_type cfv_err_routines_vector;
extern uint32e_type cfv_con_routines_vector;
extern uint32e_type cfv_xdev_routines_vector;
extern uint32e_type cfv_pcfv_routines_vector;
extern uint32e_type cfv_ptc_routines_vector;
extern uint32e_type cfv_prf_routines_vector;
extern uint32e_type cfv_meter_routines_vector;
extern uint32e_type cfv_nodevice_routines_vector;

uint32e_type cf_io_device_driver_vector[ 29] =
     [
     &cfv_syscon_routines_vector,              /*   0 */
     &cfv_sd_routines_vector,                  /*   1 */
     &cfv_devtty_routines_vector,              /*   2 */
     &cfv_mem_routines_vector,                 /*   3 */
     &cfv_ldm_routines_vector,                 /*   4 */
     &cfv_nodevice_routines_vector,            /*   5 */
     &cfv_st_routines_vector,                  /*   6 */
     &cfv_syac_routines_vector,                /*   7 */
     &cfv_err_routines_vector,                 /*   8 */
     &cfv_con_routines_vector,                 /*   9 */
     &cfv_xdev_routines_vector,                /*  10 */
     &cfv_nodevice_routines_vector,            /*  11 */
     &cfv_nodevice_routines_vector,            /*  12 */
     &cfv_pcfv_routines_vector,                /*  13 */
     &cfv_ptc_routines_vector,                 /*  14 */
     &cfv_prf_routines_vector,                 /*  16 */
     &cfv_nodevice_routines_vector,            /*  17 */
     &cfv_meter_routines_vector,               /*  18 */
     &cfv_nodevice_routines_vector,            /*  19 */
     &cfv_nodevice_routines_vector,            /*  20 */
     &cfv_nodevice_routines_vector,            /*  21 */
     &cfv_nodevice_routines_vector,            /*  22 */
     &cfv_nodevice_routines_vector,            /*  23 */
     &cfv_nodevice_routines_vector,            /*  24 */
     &cfv_syac_routines_vector,                /*  25 */
```

```
      &cfv_syac_routines_vector,                    /*  26 */
      &cfv_syac_routines_vector,                    /*  27 */
      &cfv_nodevice_routines_vector,                /*  28 */
      };
uint8e_type cf_dev_device_driver_count = 29;
   .
   .
   .
```

Configuration List

The configuration list shows all the devices configured on the system. Check for all your system file entries.

```
/* ------------------ */
/* Configuration List */
/* ------------------ */

char * cf_init_configuration_list [] =
     {
     "syscon()",
     "cied()",
     "devtty()",
     "mem()",
     "ldm()",
     "st(cisc(0,0),*)",
     "syac(0)",
     "err()",
     "con()",
     "xdev@72()",
     "xdev@73(0xffff6000,4)",
     "pts()",
     "ptc()",
     "prf()",
     "hken()",
     "meter()",
     "loop()",
     ""
     };
```

Your Special Files

At reboot time, the system will call the **dev_xxx_configure** routine you supply with your driver (we describe how to write this routine in Chapter 4). Among other things, your **dev_xxx_configure** routine generates the *special files* that point to your device driver (see "Special Files (Nodes)" in Chapter 1).

The special files are stored in **/dev**. You may list the files for your devices to verify their setup. There should be a special file for each unit serviced by your driver. The devices shown here will reflect those you specified in the system file. You determine the special files' names through your **dev_xxx_configure** routine. Listing the special files with **ls -l** will display the major and minor device numbers of each unit, as well as the access permissions. You can also verify that appropriate special files exist for block versus character access for a device.

End of Chapter

Chapter 3
Overview of Driver Facilities and Functions

This chapter describes the functions that device and adapter drivers must supply. It defines the interface between each function and the kernel, and it describes the operations each routine must perform. Where needed, we indicate whether a routine or data structure applies only to an adapter driver. Unless otherwise specified, descriptions apply to both device and adapter drivers.

NOTE: In all references below, use your own device's prefix in place of **xxx**. Your prefix is the one specified in your master file entry.

Include Files

When writing your device driver, you will need to include a number of standard include files and to supply two additional files of your own: **dev_xxx_def.h** and **dev_xxx_global_data.c**. These files are described below.

- Your driver's personal include file: **dev_xxx_def.h**

  You create this include file to hold any constant or data structure definitions you need for your driver.

- You driver's data file: **dev_xxx_global_data.c**

  For consistency between drivers, we recommend that you put your driver's statically allocated global data structures in a file called **dev_xxx_global_data.c**. You can use **dev_xxx_global_data.c** to allocate any global data structures your driver needs, but, specifically, you should use it to allocate a **cfv_xxx_routines_vector** for your driver. Your **cfv_xxx_routines_vector** specifies the locations of your driver functions.

  CAUTION:

  The kernel must find a correctly named routines vector in order to locate your driver routines. Proper allocation of the routines vector is crucial to your driver's operation.

- General driver include files

  All drivers must include the file **i_io.h**. This file contains most of the constants and structures needed by any program adhering to the standard driver interfaces. The file **i_io.h** is found in **/usr/src/uts/aviion/ii**.

  In addition to **i_io.h**, SCSI device and adapter drivers require two additional include files, **dev_scsi_def.h** and **dev_scsi_adapter_def.h**. Both of these files are found in **/usr/src/uts/aviion/dev**.

- Include files for the kernel itself

  All drivers must include three files that contain constants and data structures used by the kernel itself. These files are **c_generics.h**, **os_generics.h**, and **architecture.h**. These files are found in **/usr/src/uts/aviion/ext**.

- Include files for kernel-supplied routines

  If you use a kernel-supplied routine, you will need to include an include file specific to that routine's class. The routine's class is indicated by the first few letters of its name. The include file for a class of routines starts with these same few letters. For example, if you use a virtual memory ("vm") routine, like **vm_wire_memory**, you must include the **i_vm.h** include file. The possible include files are listed in Table 3-1 below.

Table 3-1 Routine Classes and Their Include Files

Routine Class	Acronym	Include File
File system	fs	**i_fs.h**
I/O	io	**i_io.h**
Lock management	lm	**i_lm.h**
Miscellaneous	misc	**i_misc.h**
Process management	pm	**i_pm.h**
System control	sc	**i_sc.h**
Virtual memory	vm	**i_vm.h**
Virtual process	vp	**i_vp.h**
Micro-code	uc	**i_uc.h**

  These files are stored in **/usr/src/uts/aviion/ii**. While this manual discusses some of the constants and data structures used by the various kernel-supplied routines, you may need to list these files to examine particular structures.

  Be sure to define a literal _PRODUCT_DGUX in one of your source files or at compile time if you use any of the **ii** include files.

You compile the file containing your driver routines (in **dev_xxx_driver.c**) and global data (in **dev_xxx_global_data.c**) with your **dev_xxx_def.h** and the appropriate system

include files to produce object files that will be linked into the system image.

Overview of User-supplied Device Driver Routines

All device drivers (SCSI devices and VME controllers) must supply the 15 routines (external interfaces) listed below in the section "Required Routines." These routines constitute the interface between a device driver and the kernel.

The kernel calls these routines as needed, generally when a user addresses an operation to a special file that maps to the driver's major device number. Nevertheless, some routines don't make sense for some drivers. For instance, a mouse driver cannot act on a **write_dump** operation. In such cases, your driver must still supply a routine of the appropriate type and have that routine return an error.

In addition to the 15 basic interface routines, some drivers may need 2 additional routines (internal interfaces). These routines relate to servicing interrupts and handling asynchronous I/O. The routines are used by the driver's own routines but the kernel may be involved in the process of invoking them.

To write a driver, you write your versions of the required routines and combine them into a file named **dev_xxx_driver.c**. This file will be your driver. The internal and external routines that your driver can have are summarized below in the section "Required Routines."

NOTE: In the following routines, there are important differences between device drivers that service interrupts and those that do not. Remember, SCSI device drivers do not service interrupts. Therefore, descriptions relevant to interrupts do not apply to SCSI device drivers.

Required Routines

- The kernel calls the **dev_xxx_init** routine for every driver at system initialization time — before configuration. **dev_xxx_init** allows the device driver to perform any initialization that is necessary before any devices are actually configured into the system. You do not necessarily need to initialize the device itself in **dev_xxx_init**; this routine simply provides you with the opportunity to set up any data structures or other operations that you might want done prior to configuration.

- **dev_xxx_configure** performs operations necessary to make a peripheral of your driver's class accessible to the system. During configuration, the kernel calls the driver's **dev_xxx_configure** routine once for each peripheral listed in the system file. In addition, peripherals not listed in the system file may be configured at some other time in the life of the system. Thus, **dev_xxx_configure** should be able to run at any time in the life of the system.

  Because this is the first time your driver actually interfaces to the device, you

will want to ensure that the device is alive and well. You will set up your special files (**/dev** entries), and assign minor device numbers. You may also need to query the controller to find out how many units it has, because you will want to configure each unit on the controller.

If you are writing an SCSI device driver, you will also need to identify and get a pointer to the adapter driver routines you will be addressing. You will also have to make sure that your adapter has already been configured before you can query your device. To do this you can call the kernel-supplied routine **dev_scsi_adapter_configure** with your device's device specification. The first parameter of this device specification contains the adapter's device specification. **dev_scsi_adapter_configure** will make sure that the adapter is configured and return the adapter's major and minor device number.

For drivers that service interrupts, the **dev_xxx_configure** routine must allocate a *device information structure* for the device and then register the information in this structure. Registering links the device code with the driver's interrupt service routine. This link is made via a *device interrupt table* (DIT). SCSI device drivers must call the adapter manager's **dev_scsi_adapter_register_requester** routine to associate their device with their adapter.

Note that on AViiON machines, devices come up with interrupts enabled by default.

- The kernel calls a device's **dev_xxx_open** and **dev_xxx_close** routines when the user opens and closes the device. The kernel calls **dev_xxx_open** each time the device is opened, even if the device is already open. Similarly, the kernel calls the **dev_xxx_close** function each time the device is closed.

  One of **dev_xxx_open**'s important functions is to return a device handle that the kernel will use later to pass to the driver's other routines. The **dev_xxx_open** routine may also do further checking of the device to ensure that it is ready for operation (for example, that a tape is mounted, online, and write-enabled). Finally, if the configuration routine has not initialized the device, the **dev_xxx_open** routine will initialize the device and make it ready for operation.

- The kernel calls the driver's **dev_xxx_read_write** routine for any user read or write operation that is to be handled synchronously (that is, the user process will be pended until the I/O completes). Character special (raw) I/O is always done synchronously. Thus, for character I/O, each user call to a read/write system call results in the kernel calling **dev_xxx_read_write**.

- **dev_xxx_start_io** is the asynchronous counterpart of the **dev_xxx_read_write**. Whenever the kernel decides not to let the process pend until the I/O completes, it invokes **dev_xxx_start_io** instead of **dev_xxx_read_write**. **dev_xxx_start_io** is used for only block special I/O operations.

For block special I/O, the kernel determines whether the request will be processed synchronously or asynchronously (see **dev_xxx_start_io**). Thus, the kernel may call either **dev_xxx_read_write** or **dev_xxx_start_io**. In fact, the kernel manages block special operations such that it may not call either of these routines in a one-to-one correspondence with the user's **read/write** system call. In other words, sometimes the kernel may have previously buffered the data the user wants.

- The **select** operation is usually used for devices (such as terminals) that must wait for an external event before I/O can proceed. Your driver's **dev_xxx_select** routine implements this operation. The kernel provides **select** facilities that help a driver manage the list of events used to notify processes awaiting a **select** event. We discuss the **select** manager facilities in Chapter 8.

  The kernel calls your **dev_xxx_select** routine whenever the user calls the **select** system call for your driver's device.

- The **ioctl** operation is used to issue control functions to a device. For example, a user might invoke **ioctl** to set forms on a line printer.

  The kernel calls your **dev_xxx_ioctl** routine whenever the user issues the **ioctl** system call for your driver's device. Note that some **ioctl** calls are actually file descriptor operations and do not actually refer to any device. The kernel will handle these calls directly and not call **dev_xxx_ioctl**. For example, the kernel will handle the **FIONCLEX ioctl** command directly and not call **dev_xxx_ioctl**.

  Because control functions are specific to each device driver, you can define your own control parameters. The kernel simply passes the parameters from the user request to the driver's **dev_xxx_ioctl** routine. The kernel does not interpret these parameters.

  If you are writing a disk driver and want to implement a hardware formatter, we recommend you use **ioctl**.

- The kernel calls the **dev_xxx_open_dump**, **dev_xxx_write_dump**, and **dev_xxx_close_dump** routines during system panic. The **dev_xxx_open_dump** function does all initialization required for the dump device to be accessed during system panic. The **dev_xxx_write_dump** routine writes data to the dump device. The **dev_xxx_close_dump** routine is called to terminate the dump operation to the device.

  Note that these routines are necessary only if your device will be a dump destination. For example, a mass spectrometer driver cannot be a dump destination.

- The **dev_xxx_deconfigure** routine does the opposite of the **dev_xxx_configure** operation. It deallocates all resources and performs any cleanup necessary

to completely remove a device from the system. As with **dev_xxx_configure**, **dev_xxx_deconfigure** should be able to work at anytime during the life of the system.

The deconfiguration routine is optional; you may support it if you wish. The benefit of including it is that, in case of an erroneous configuration, the user can deconfigure and then reconfigure and re-use the device. This might occur, for example, if the user accidentally configures a tape at device code 23 when a disk is actually resident at that device code. Deconfiguration will allow the tape to be deconfigured so the disk can be correctly configured without re-booting the entire system.

If you do include deconfiguration, you should try to allow for future enhancements such as repair-under-power. In repair-under-power a single device must be deconfigured so it can be removed from the system and repaired.

- The kernel calls the **dev_xxx_powerfail** routine for every driver when power is restored to the system after a power failure (assuming that battery backup has preserved the process and memory state such that automatic recovery makes sense). The **dev_xxx_powerfail** function should be able to work anytime after **dev_xxx_init** has completed, regardless of whether or not the device is open.

  The DG/UX system has not yet implemented **powerfail**. However, the routine still has a place in the kernel's table of driver routines. The driver only has to provide a stub routine for this interface.

- You must supply **dev_xxx_device_to_name** and **dev_xxx_name_to_device** name translation routines. These routines translate between your devices' names and their numbers (major and minor numbers combined). Kernel and system administration utilities will use these routines to identify devices they want to access. Your **dev_xxx_device_to_name** routine should be able to function anytime after the device's configuration.

NOTE: Your **cfv_xxx_routines_vector** includes place holders for **dev_xxx_mmap**, **dev_xxx_munmap**, **dev_xxx_maddmap**, and **dev_xxx_read_dump** routines. These interfaces will be operational in a later release of the DG/UX kernel. For now, put stubs in the routines vector fields for these routines (see the "Nodevice Routine Stubs" section in Chapter 8).

Optional Routines

In addition to the required routines described above, devices that use interrupts will need an interrupt service routine, and drivers that perform asynchronous I/O may need an I/O completion routine for follow-up processing.

- The **dev_xxx_service_interrupt** routine processes any incoming interrupts for your device.

- You can create an I/O completion routine to complete processing for asynchronous I/O operations. (We'll refer to your I/O completion routine as the *complete_io* routine.) The **dev_xxx_start_io** routine starts the asynchronous request, but the follow-up processing must be handled elsewhere. Frequently, the completion operations are too lengthy to be done in the interrupt service routine. Most drivers handle completion by scheduling a message to the Driver Daemon or Generic Daemon (see Chapter 8). The message specifies a *complete_io* routine that the daemon will execute.

  NOTE: You can name your *complete_io* routine anyway you see fit. We italicize the term *complete_io* to emphasize this point.

  The kernel has its own I/O completion routine (hereafter called the Kernel I/O completion routine) that the driver must invoke as part of its *complete_io* routine. The driver's **dev_xxx_start_io** routine receives the Kernel I/O completion address as a field in the **op_record** parameter. The driver returns control back to the higher levels of the kernel by calling the Kernel I/O completion routine. See the **dev_xxx_start_io** description in Chapter 4 for discussion of "start I/O" and the kernel I/O completion routine.

Overview of User-supplied Adapter Driver Routines

In order to access the physical device, an SCSI device driver invokes the SCSI adapter manager, which in turn invokes the SCSI adapter driver that controls the specified device. Thus, most of the adapter driver routines are used to interface between the SCSI device driver and the SCSI adapter driver. These routines are listed below in the "The Device Driver to Adapter Driver Interface" section.

There are also two cases in which the kernel will invoke an SCSI adapter driver routine. These routines are described in the "The Kernel to Adapter Driver Interface" section.

The Device Driver to Adapter Driver Interface

A device driver will call the following adapter routines as needed. The data structures referred to here are described in more detail in the "Overview of Major Data Structures" section.

- Since SCSI devices do not interrupt the host directly, SCSI device drivers need not register their device information structure with the kernel. Instead, they register themselves with the adapter driver that will handle their interrupts for them. Thus, during configuration, each SCSI device driver calls its adapter's **dev_xxx_register_requester** routine in order to identify itself and establish a link between itself and the adapter driver. **dev_xxx_register_requester** establishes the link by adding an entry with the device's SCSI ID and unit number to its device information structure.

- During deconfiguration, a device driver calls its adapter's **dev_xxx_deregister_requester** routine to close the link between device and adapter drivers.

- SCSI device drivers are specific to a particular class of device. For example, you can have an SCSI tape driver or an SCSI disk driver. Adapter drivers, on the other hand, may handle many different types of devices (such as tapes, disks, and printers). Device drivers use their adapter's **dev_xxx_set_unit_options** routine to set certain adapter I/O parameters to fit their specific device. For example, a tape may need a longer timeout value than a disk.

- When an SCSI device driver wants to send a command to its device, it must process that request via the adapter. If the command is to be processed synchronously, the device driver calls the adapter's **dev_xxx_issue_command** routine. If it is an asynchronous request, the device driver calls the adapter's **dev_xxx_issue_async_command** routine. These routines send an SCSI command to the target device via the adapter. These routines can be invoked as a result of either user-initiated requests or internal driver needs.

The device driver sends information about the request in an *adapter request block* given as a parameter to **dev_xxx_issue_command**. The adapter request block is a generic structure used for all adapters. The adapter driver will return the results of the operation as a return value status. Chapter 4 lists the possible return values for each adapter interface. Sense information describing an error is returned in the request block's sense buffer.

Depending on the adapter's architecture, **dev_xxx_issue_command** or **dev_xxx_issue_async_command** may need to transfer request information to a structure appropriate for the particular adapter. Throughout the rest of this manual we will refer to such structures as the adapter-specific parameter block.

- If the system panics, the kernel enters shutdown mode and no longer provides its usual services. If an SCSI device is a designated dump destination, the adapter driver will have to access the device without normal kernel support. The **dev_xxx_issue_command_physical_mode** routine you supply will be used in these situations. The **dev_xxx_issue_command_physical_mode** provides access to the SCSI device when no interrupts, no locks, no eventcounters and no virtual memory are available.

- Since an SCSI device driver registers itself with its adapter instead of with the kernel, it will need to access its device information via the adapter driver. The device driver calls the adapter's **dev_xxx_get_device_info** routine to get information about its device from the adapter's device information structure. The adapter's **dev_xxx_get_device_info** routine returns the device's unit handle just as the **io_get_info** routine does for the kernel.

The Kernel-to-Adapter Driver Interfaces

From the kernel's perspective an adapter driver is simply another driver. It is theoretically possible for a user to address an I/O request to the adapter as if it were an end device. To handle this possibility, the kernel requires that adapter drivers supply all the routines listed for device drivers.

The majority of these routines would be accessed only because of a user error. Hence, they are really just error-returning program stubs. You can write your own versions of these stubs or you can use the set of DG/UX "nodevice" routines listed in Chapter 8. If you use the DG/UX routines, you will substitute a "nodevice" routine for each routine you do not supply. For example, if you do not supply **dev_xxx_read_write**, you would substitute **io_nodevice_read_write** in **cfv_xxx_routines_vector**.

Of the required device driver routines, an adapter driver must supply four actual routines, and the rest may be stubs. You must supply **dev_xxx_configure**, **dev_xxx_open_dump**, **dev_xxx_device_to_name** and **dev_xxx_name_to_device**. The device driver descriptions of these routines pertain to the adapter driver as well.

Note that since adapter drivers service interrupts, their **dev_xxx_configure** routines must allocate a device information structure for the device and then register the information in this structure. Registering links the device code with the driver's interrupt service routine via the *device interrupt table* (DIT).

Overview of Major Data Structures

Many of the major data structures your driver will need are contained in the include file **i_io.h**. In this section, we describe some of the important types found in **i_io.h**. We show the actual type definitions for these structures in Chapter 4.

Device Driver Data Structures

The major structures a driver might need are as follows:

- *Routines vector*

  The main routines vector is a table of your driver's basic routines (the 17 required and 2 optional described for all drivers). The structure for this routines vector type is defined in **i_io.h**. You allocate your driver's main routines vector in **dev_xxx_global_data.c**. The kernel accesses your driver's routines via its own internal table of routines vectors. It uses the device's major number as an index into this table.

  SCSI adapter drivers have their own routines vector, which includes both the standard and adapter-specific routines. The adapter's routines vector type is defined in **dev_scsi_adapter_def.h**

  Chapter 4 shows the layout of both types of routines vectors.

- *Device number*

  Your driver uses the device number to identify a specific device before the device has been opened. The device number is a combination of the device's major and minor numbers. Your **dev_xxx_configure** routine passes the device number to the kernel's **fs_submit_dev_request** routine to create a special file. The kernel also passes the device number to your **dev_xxx_open** routine to identify the device. After the **dev_xxx_open**, the user will use a file descriptor to identify the device. The kernel identifies the device to the driver's routines by passing the device handle.

  The **dev_xxx_configure** routine gets the device's major number as a parameter from the kernel and gets the device's minor number by calling the kernel's device number **io_allocate_device_number** routine (see Chapter 8). It combines these numbers to form the device number.

- *Device information structure*

  You allocate this structure at configuration time. You create a device information structure for each peripheral of your driver's type listed in the system file. It should be dynamically allocated in global wired memory.

  You can define most of the contents of this structure anyway you see fit.

However, for devices that interrupt the host, the kernel requires that the first field of the device information structure contain a pointer to your interrupt service routine.

- *Device interrupt table*

  The kernel uses this table to match interrupts and interrupt service routines. This table matches each device code with a pointer to the first field in the driver's device information structure. This first field contains the address of the driver's interrupt service routine.

  When you register your device information structure, the **io_register_device_info** routine copies a pointer to your device information structure into the DIT entry for you controller using the device code as an index. The first field of this structure points to your interrupt service routine.

  The kernel declares the device interrupt table.

- *Device handle*

  After the device is open, the kernel passes the device handle to the driver routines. Thus, your **dev_xxx_read_write** routine will get the device handle when it is invoked. The device handle is specific to a single unit of a device. The kernel does not interpret this field.

  You are allowed to define and use the device handle as you want. It is intended to be a pointer to an information table describing the operations occurring on a particular device. Most drivers make the device handle point to a unit-specific area in the device information structure. (We will presume this implementation in the rest of this manual.)

- *Buffer vector*

  Buffer vectors are the DG/UX kernel's interface for data transfer. When users make an I/O request such as a **read**, they specify a buffer and a transfer byte count. The kernel allocates a buffer vector and packages the I/O request information in this structure. Thus, the buffer vector holds a transfer byte count and pointers to memory buffers. The kernel then passes the buffer vector for the request to your driver routine (for example, **dev_xxx_read_write**). You can manipulate the buffer vector using the kernel routines described in Chapter 5.

  Buffer vectors are specifically designed to handle buffers that span non-contiguous memory. Non-contiguous buffers are needed for the **readv** and **writev** operations. For simplicity, the same buffer-vectoring scheme is used for the standard **read** and **write** operations even though they do not need non-contiguous buffers.

- *Request information packet*

  For the **dev_xxx_read_write** routine, the kernel packages information about the user I/O request in a request information packet. This packet contains the device handle, the buffer vector, and a set of I/O flags that specify restrictions on this particular operation (see Chapter 4). In addition, it contains a device offset value, which specifies to the driver where on the device the information transfer is to begin. For example, for a disk driver, the offset might indicate the number of bytes from the start of the disk. The driver can divide this number by 512 to determine the logical sector on the disk.

- *Operation record packet*

  The operation record packet is the same as a request information packet except that it is used with the asynchronous **dev_xxx_start_io** routine and hence has several extra fields. In addition to the same fields found in the request information packet, the operation record packet contains: 1) a **complete_io** routine field, which holds the address of the kernel's I/O completion routine; and 2) a **link** field, which allows the driver to link requests together in an asynchronous request queue.

Data Structures for SCSI Adapter and SCSI Device Drivers

In addition to the structures listed above, SCSI adapter and SCSI device drivers will also use the following.

- *Adapter unit specification structure*

  The adapter driver uses the device specification structure to save the device specification information for each currently active device. The entry for each active device shows the device's SCSI ID and unit number. Unit numbers are used for SCSI devices that are controllers with units.

- *Adapter request blocks* and *Adapter-specific parameter block*

  Device drivers use adapter request blocks to pass information about their current request to the adapter driver. The structure is a generic block of parameters used to issue a request to the **dev_xxx_issue_command** and **dev_xxx_issue_async_command** routines. Depending on the architecture, before issuing the request, the adapter driver may need to transfer request information to a structure appropriate for its particular adapter. Throughout the rest of this manual, we refer to such structures as adapter-specific parameter blocks.

- *Other adapter parameter blocks*

Dev_xxx_register_requester, **dev_xxx_set_unit_options**, and **dev_xxx_issue_command_physical_mode** each have defined blocks through which the device driver passes them information. Chapter 4 shows the layout of these parameter blocks.

Other Driver Facilities

In this section, we describe two driver facilities: the Driver and Generic Daemons and the error reporting facilities. See Chapter 8 for more information on these facilities and a description of the routines you use to interface to them.

The Driver Daemon and the Generic Daemon

Driver Daemons and Generic Daemons are classes of daemon processes that drivers use for handling asynchronous I/O requests. Asynchronous I/O requests generally require the use of an interrupt service routine. In a symmetric multiprocessing environment, the interrupt service routine cannot be allowed to pend or to call any routine that might pend. Thus, the interrupt service routine can perform only very minimal operations. In most cases it will need a way to continue processing the interrupt outside the service routine's restricted environment. Driver Daemons and Generic Daemons provide an appropriate way to handle this continued processing.

The two classes of daemon process have exactly the same interface and method of operation. Each class has a global queue on which requests are placed. Requests consist of a pointer to a routine to execute and an argument to be passed to the routine. A daemon process will remove an entry from the request queue and call the routine with the specified argument. More than one daemon process may be removing requests from the same queue so that multiple requests can be executed in parallel on a multiprocessor system. Each individual request, however, is only executed once and by a single daemon. All the daemon processes that are working off the same queue are in the same class.

By executing the requestor's routine, the daemon can take the place of the requestor in performing the device service operations such as examining the status, retrying errors, and starting previously queued requests. Furthermore, if the driver code determines that an asynchronous request has completed, the requestor's I/O completion routine (see Chapter 4) will be called, again with the daemon actually executing the code.

The two classes of daemon differ in what kinds of operations the routine in the request may perform. Routines in Driver Daemon requests must not perform any operation that might have to wait for the completion of a disk I/O operation. For example, such routines may not page fault, because servicing the page fault may require waiting for a disk I/O to complete. In addition, such routines must not directly or indirectly send signals or perform terminal-related operations. Because of all these restrictions, the Driver Daemons will generally only be used by disk device drivers.

On the other hand, routines in Generic Daemon requests are allowed to wait on disk I/O, send signals, and perform terminal-related operations. The lesser restrictions make the Generic Daemons usable by terminal-handling code and other higher level parts of the system.

NOTE: Disk device drivers must not use the Generic Daemons because a deadlock condition could result.

Error Reporting Facilities

Drivers can choose between two major error-reporting destinations: 1) the user-level calling process; and 2) the system error-logging facility. Drivers do not need to perform any special operation to report statuses back to the user-level process; the kernel passes driver routines' return values back to the user as a completion status after the routine completes. Because users receive return values as statuses, we strongly recommend you encode your driver's unique return values according to standard encoding procedures (see Chapter 8). Users can decode standardly encoded statuses using the **dg_ext_errno** system call.

To send an error to the system error-logging facility, the driver must use the services of the system error daemon, **syslogd**, and the psuedodevice, **err**(7). **Err** receives and stores errors from kernel-level processes. **Syslogd** receives and stores errors from all processes connected to the system, remote or local, user- or kernel-level. **Syslogd** periodically retreives and processes the errors stored in **err**.

How **syslogd** processes errors is determined by its configuration file, **/etc/syslog.conf**. For example, **syslog.conf** may specify that the logged errors are to be printed out to the system console or written to a disk file, and so forth. See **logger**(1), **syslog**(3), **syslog.conf**(5), and **syslogd**(8) for more information on the system error daemon and how to configure error processsing.

The **err** psuedodevice receives and stores errors from drivers on an internal error queue. Your driver can store error messages on this queue using the kernel-supplied routine, **io_err_log_error**. **Io_err_log_error** is described in Chapter 8.

End of Chapter

Chapter 4
User-Supplied Driver Routines

In Chapter 3, we gave you an overview of the routines that your driver should supply to the kernel. In this chapter, we describe what each routine does, give details on parameters and arguments, and tell you about assumptions you should make while writing the routines.

The chapter is divided into the following major sections:

- User-Supplied Device Driver Routines
- User-Supplied Adapter Driver Routines

The device driver interfaces describe the routines you must write to build a device driver, SCSI, or VME. The adapter driver interfaces describe routines you must write to build an SCSI adapter driver.

Each routine specification includes a "Return Values" section that lists specific return values that the kernel can process when the routine returns. When no return value is specified, the routine must not fail (the kernel will not process any returns or exceptions). If the driver routine experiences an exception other than those specified in the "Return Values" section, it can proceed in one of the following three ways:

1) It may return an exception by returning a value other than one of the specified values. The kernel will filter this value back to the user as a standard **errno**. You can either define your own values for this **errno** or use values already defined by the system. Check **/usr/include/sys/errno.h** for a listing of the existing **errnos** and their definitions. In Chapter 8 we describe how to define an error status.

2) It may panic the system. In Chapter 8 we describe the routines used to panic the system. Some driver routines are not allowed to panic. We indicate whether or not a routine can panic in the "Return Values" section of the interface description.

3) It may use the error daemon to log an error. In Chapter 8 we describe the procedures used for error logging. If the routine decides to log the error, it should still return an exception (**errno**) to the user directly.

User-Supplied Device Driver Routines

This section describes the routines and data structures you will need to create a device driver.

Constants and Data Structures

The device driver routines you write will use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, since these specifics may change in later releases of the software. The best way to avoid such dependencies is to use kernel-supplied routines to manipulate these structures. We discuss kernel-supplied routines in Chapter 5 through Chapter 8.

The constants and data structures listed here are given for convenience only and may change. Check the appropriate include file (for example, **i_io.h** for structures beginning with the **io** acronym) for the exact definition of all constants and data structures. Chapter 3 describes the various include files.

io_driver_routines_vector_type

```
typedef struct
    {
    uint16_type         version;
    bit16_type          flags;
    status_type         (*open)();
    void                (*close)();
    status_type         (*read_write)();
    void                (*select)();
    status_type         (*ioctl)();
    status_type         (*start_io)();
    void                (*init)();
    status_type         (*configure)();
    status_type         (*deconfigure)();
    status_type         (*device_to_name)();
    status_type         (*name_to_device)();
    status_type         (*open_dump)();
    status_type         (*write_dump)();
    status_type         (*read_dump)();
    status_type         (*close_dump)();
    status_type         (*powerfail)();
    status_type         (*mmap)();
    status_type         (*munmap)();
    status_type         (*maddmap)();

    }  io_driver_routines_vector_type  ;
```

Description

The kernel must have a pointer to each of your routines that will be externally visible. You provide a vector of pointers to your driver's routines in a routines vector described by **io_driver_routines_vector_type**. You must allocate a variable of this type for your driver in **dev_xxx_global_data.c**.

A **version** field is present to allow the system to change this structure and still be compatible with older, user-written device drivers. The version should be one (1) except when **io_routine_vector_type** is used as part of the SCSI adapter routines vector, **dev_scsi_adapter_routines_vector_type**. In this latter case, the **version** should be IO_DRIVER_ROUTINES_VECTOR_SCSI_ADAPTER_VERSION.

io_device_number_type

```
typedef struct
    {
    io_major_device_number_type          major;
    io_minor_device_number_type          minor;

    }  io_device_number_type  ;
```

Description

A device number is a composite of the device's major and minor device numbers. During configuration, the kernel calls your **dev_xxx_configure** routine with the device's major number. Your **dev_xxx_configure** will get the device's minor number (using the kernel's **io_allocate_device_number** routine) and then create the device number variable for the unit. It then uses this device number to create the special file (node) for the specific units. The kernel also passes the device number to your **dev_xxx_open** routine to identify the special file of the unit to be opened. After the open, a file descriptor will be used to identify the unit to the user, and a device handle will be used to identify the unit to your driver routines. The kernel will not interpret the device number value.

io_device_handle_type

```
typedef opaque32_type     io_device_handle_type  ;
```

Description

A device handle identifies an open device to other calls to the device driver. Your **dev_xxx_open** routine defines and returns the device handle when the device is opened. The device handle becomes invalid when the device is closed.

Many drivers use a pointer to the unit-specific portion of the device information

structure as the device handle. However, what makes up a device handle and its interpretation is up to each individual driver. Higher levels of the kernel that hold device handles will not interpret their contents.

io_request_info_type

The kernel supplies a variable of this type for every I/O request system call made to your driver.

```
typedef struct
    {
    io_operation_type                   op;
    io_channel_flags_type               flags;
    io_device_handle_type               device_handle;
    uint32_type                         device_offset_extender;
    uint32_type                         device_offset;
    io_buffer_vector_type               buffer_vector;
    df_self_id_type                     self_id;

    }  io_request_info_type  ;
```

Description

The request information package described by this type groups several related values that are needed to specify an I/O request. The request information package fields are as follows:

op — The operation indicated by this request. See **io_operation_type** for a list of the operation types. The **op** request is modified by the **flags** field.

flags — An additional set of flags that modify the operation indicated by the **op** field. These **io_channel_flags** are described later in this section.

device_handle — The device handle of the device to which the request is to be directed. The device handle must be a device handle that was returned by the open function of the driver for the device to which this request is to be directed.

device_offset_extender — This field exists for device offsets needing more than 32 bits. This field should be zero if large offsets are not used (for example, non-disk devices). It should be checked in disk drivers. If your disk does not support offsets needing the extender, you should reject requests where this offset is non-zero.

device_offset — The offset on the device where the transfer is to begin. The interpretation of this field is defined by the driver to which the request is directed.

buffer_vector — A buffer vector describing the main memory area that is to be involved in the I/O operation. The addresses may be logical or physical

depending upon the operation specified in the **op** field.

self_id — The home system identification against which read data is to be checked if the **IO_CHECK_SELF_ID** flag is TRUE.

io_operation_type

```
typedef bit16_type                  io_operation_type  ;

#define IO_OPERATION_READ              ((bit16_type)00000001)
#define IO_OPERATION_WRITE             ((bit16_type)00000002)
#define IO_OPERATION_RECALIBRATE       ((bit16_type)00000004)

#define IO_OPERATION_CHECK_SELF_ID     ((bit16_type)00000010)
#define IO_OPERATION_PHYSICAL_BUFFER   ((bit16_type)00000020)
#define IO_OPERATION_USER_BUFFER       ((bit16_type)00000040)
```

This type defines a bit field that describes an I/O operation to be performed. Only one of READ, WRITE, or RECALIBRATE will be on at any one time. The CHECK_SELF_ID flag may be present only on a READ operation. The PHYSICAL_BUFFER flag may be present only on a READ or WRITE and indicates that the buffer address supplied with the operation is a physical memory address rather than a logical memory address. The USER_BUFFER flag may be present only on a READ or WRITE and indicates that the buffer address supplied with the operation is a user memory address rather than a kernel memory address.

io_operation_record_type

```
typedef struct
    {
    misc_queue_links_type               links;
    io_request_info_type                ri;
    io_completion_routine_ptr_type      completion_routine;

    }  io_operation_record_type  ;
```

Description

You use the operation record when starting an asynchronous I/O request using your driver's **dev_xxx_start_io** function. The structure is basically an extension of the **io_request_info_type** that you use for synchronous requests. The extension includes extra information that is needed to service the request in an asynchronous manner. The operation record's fields are as follows:

links — Space that may be used by a device driver to link this operation record into a queue with other operation records. The driver determines the actual use of this space.

ri — The request information structure that specifies the request.

completion_routine — The address of the function that should be called when the operation denoted by this operation record is complete. This function must conform to the I/O completion routine interface described in the "Kernel I/O Completion Routine Interface" section.

io_select_intent_type

```
typedef bit16_type  io_select_intent_type  ;

IO_SELECT_INTENT_READ
IO_SELECT_INTENT_WRITE
IO_SELECT_INTENT_EXCEPTION
IO_SELECT_INTENT_NONE
```

Description

This type describes the select options that may be specified to a device driver's **dev_xxx_select** routine. The READ, WRITE, and EXCEPTION options start a select for the corresponding operation. You can use any combination of these three options in a single **dev_xxx_select** call. IO_SELECT_INTENT_NONE is used as a return value from **io_select_cancel** when no intent has been satisfied.

io_buffer_vector_type

```
typedef struct
        {
        union
        {
        io_buffer_vector_control_type    many;
        io_buffer_descriptor_type        one;
        } u;
        uint16_type                      descriptor_count;
        uint16_type                      current_descriptor;
        uint32_type                      current_offset;
        uint32_type                      total_remaining;

        }  io_buffer_vector_type ;
```

Description

This structure defines a buffer vector, which is a collection of individual buffer descriptors plus an associated state. A buffer vector may be the source or destination of a single read or write operation; the individual buffer descriptors define the locations from which the data is being read or into which the data is being written.

The current position is where the next byte of data will be read from or written to. The current position is initialized to the first byte of the first buffer descriptor. The current position within the buffer vector is maintained by the associated state.

The fields in this structure are as follows:

many — This structure contains a pointer to the array of buffer descriptors and the total of the sizes of all the elements of the array. This field of the union is used only when **descriptor_count** is non-zero. **io_buffer_vector_control_type** is described in this section.

one — This structure contains the single buffer descriptor when the buffer vector consists of a single descriptor. This field of the union is used only when **descriptor_count** is zero. **io_buffer_vector_control_type** is described later in this section.

descriptor_count — The number of entries in the **many** array of the **io_buffer_vector_control_type**. Not all of these entries are presumed valid; the **total_size** field controls the number of entries that are used. This field is used to determine the actual amount of memory allocated to the array. If this field is zero, then there is no memory allocated to the array and a single descriptor is stored in the union field **one**.

current_descriptor — The index of the descriptor that contains the current position. **io_buffer_vector_control_type** is described later in this section.

current_offset — The offset of the current position in the buffer descriptor indexed by **current_descriptor**.

total_remaining — The total number of bytes remaining to be moved to or from this buffer vector since it was initialized.

io_buffer_descriptor_type

```
typedef struct
     {
     pointer_to_any_type  buffer_ptr;
     uint32_type          size;

     }  io_buffer_descriptor_type  ;
```

Description

This structure describes a buffer from which data is to be read or to which data is to be written.

The fields in this structure are as follows:

buffer_ptr — Pointer to the start of the buffer.

size — The size of the buffer, in bytes.

io_buffer_vector_control_type

```
typedef struct
    {
    io_buffer_descriptor_ptr_type     descriptors_ptr;
    uint32_type                       total_size;

    }  io_buffer_vector_control_type  ;
```

Description

This structure is used in the **many** field of **buffer_vector_type**.

The fields in this structure are as follows:

descriptors_ptr — A pointer to an array of buffer descriptors. The array may contain as many as **UINT16_MAX** entries. (See **c_generics.h** for the definition of **UINT16_MAX.**)

total_size — The sum of the size fields in all the elements of the array buffer descriptors.

io_channel_flags_type

```
typedef bit32_type io_channel_flags_type  ;

#define   IO_CHANNEL_NO_FLAGS                ((bit16_type)00000000)
#define   IO_CHANNEL_READ_INTENT             ((bit16_type)00000001)
#define   IO_CHANNEL_WRITE_INTENT            ((bit16_type)00000002)
#define   IO_CHANNEL_EXCLUDE_WRITERS_INTENT  ((bit16_type)00000004)
#define   IO_CHANNEL_APPEND_INTENT           ((bit16_type)00000010)
#define   IO_CHANNEL_SYNC_IO                 ((bit16_type)00000020)
#define   IO_CHANNEL_NO_WAIT                 ((bit16_type)00000040)
#define   IO_CHANNEL_ASYNC_IO                ((bit16_type)00000100)
#define   IO_CHANNEL_NONBLOCK                ((bit16_type)00000200)
#define   IO_CHANNEL_NDELAY                  ((bit16_type)00000400)
#define   IO_CHANNEL_BLOCK_SPECIAL           ((bit16_type)00001000)
#define   IO_CHANNEL_NO_RETRIES              ((bit16_type)00002000)
#define   IO_CHANNEL_NOTIFY_IF_MANDATORY     ((bit16_type)00004000)
#define   IO_CHANNEL_NOTIFY                  ((bit16_type)00010000)
```

Description

When users open a device, they can open with a set of conditions. The channel flags specify the open conditions that the user requested. These conditions are passed to the **dev_xxx_open** routine. See **dev_xxx_open** for descriptions of the conditions.

The open options are as follows:

IO_CHANNEL_NO_FLAGS — None of the conditions described below applies.

IO_CHANNEL_READ_INTENT — The channel is opened with read intent. This flag corresponds to the O_RDONLY or O_RDWR option on the **open** system call.

IO_CHANNEL_WRITE_INTENT — The channel is opened with write intent. This flag corresponds to the O_WRONLY or O_RDWR option on the **open** system call.

IO_CHANNEL_EXCLUDE_WRITERS_INTENT — The channel is opened only if there are currently no writers, and future attempts to open with write intent are disallowed. This flag is used internally by the file system to prevent other processes from writing to a disk it is managing.

IO_CHANNEL_APPEND_INTENT — The channel is opened with append intent. This flag corresponds to the O_APPEND option on the **open** system call.

IO_CHANNEL_SYNC_IO — The channel is opened with the synchronous I/O option. This flag corresponds to the O_SYNC option on the **open** system call.

IO_CHANNEL_NO_WAIT — The channel is opened with the no-wait I/O option. This flag corresponds to the O_NDELAY or to the O_NONBLOCK option on the **open** system call.

IO_CHANNEL_ASYNC_IO — The channel is opened with the asynchronous I/O option. This flag corresponds to setting the FASYNC option with the **fcntl** system call.

IO_CHANNEL_NONBLOCK — The channel is opened with the O_NONBLOCK option.

IO_CHANNEL_NDELAY — The channel is opened with the O_NDELAY option. The driver should not look at this flag.

IO_CHANNEL_BLOCK_SPECIAL — The driver is being opened as a block special device. This flag is used only internally.

IO_CHANNEL_NO_RETRIES — I/O performed via this channel should not be retried if errors occur; all errors are treated as hard errors. This flag may or may not be supported by a given device driver.

IO_CHANNEL_NOTIFY_IF_MANDATORY — The kernel uses this option internally to avoid deadlock on mandatory locks. Drivers should not use this option.

IO_CHANNEL_NOTIFY — The driver is being opened with the O_NOCTTY

open flag set. The kernel uses this option to prevent the controlling terminal from being set.

Interfaces for Device Driver Routines

In this section, we detail the following device driver routines, which you must supply.

- **dev_xxx_init**
- **dev_xxx_configure**
- **dev_xxx_open**
- **dev_xxx_close**
- **dev_xxx_service_interrupt**
- **dev_xxx_read_write**
- **dev_xxx_select**
- **dev_xxx_ioctl**
- **dev_xxx_start_io**
- **dev_xxx_open_dump**
- **dev_xxx_write_dump**
- **dev_xxx_read_dump**
- **dev_xxx_close_dump**
- **dev_xxx_powerfail**
- **dev_xxx_deconfigure**
- **dev_xxx_device_to_name**
- **dev_xxx_name_to_device**
- **dev_xxx_maddmap**
- **dev_xxx_mmap**
- **dev_xxx_munmap**

dev_xxx_init

Syntax

```
void    dev_xxx_init  ()
```

Summary

This routine performs any pre-configuration initialization your driver might need.

Parameters

None.

Description

The kernel calls the **dev_xxx_init** routine as part of system initialization. **dev_xxx_init** gives the driver an opportunity to perform any initialization needed before any of the driver's devices are configured into the system. **dev_xxx_init** is invoked once in the life of the system. No devices controlled by the driver will be configured until after the **dev_xxx_init** routine completes.

The **dev_xxx_init** routine operates in a restricted environment. It may not await or take a page fault.

Return Values

The **dev_xxx_init** routine does not return a status; any errors that it encounters must result in a panic or in some method of flagging the error to **dev_xxx_configure** for further processing.

dev_xxx_configure

Syntax

```
status_type dev_xxx_configure (device_name_ptr, major_number)

char_ptr_type device_name_ptr;                /*READ ONLY*/
io_major_device_number_type   major_number;   /*READ ONLY*/
```

Summary

This routine configures a single device of the class supported by this driver.

Parameters

device_name_ptr — A pointer to the name of the device to be configured. The name is in the form of a null-terminated string. The device name is the name specified in the system file.

major_number — The major device number on which the device is to be configured. This is the major number specified in the master file.

Description

This routine performs operations that make a physical device (of the class supported by the driver) accessible to the system. The **dev_xxx_configure** routine can be called anytime. If your device has system and master file entries, it will be called by system initialization code during system boot. It is called once for each system file entry in the system.

The **dev_xxx_configure** routine receives a **device_name_ptr** variable that points to a device name. The name string is terminated by a null character and has the following form:

device_mnemonic [@*device_code*] ([*parameters*])

Because each **dev_xxx_configure** is called for all system file entries, your **dev_xxx_configure** should verify that the name string for the current call contains its device's name. If the name is not for one of its devices, **dev_xxx_configure** should exit with a return value of **IO_ENXIO_DEVICE_NOT_RECOGNIZED**.

dev_xxx_configure must initialize the device and must make the device accessible to the kernel. Device initialization is unique to the device and to the driver. The **dev_xxx_configure** routine should perform the following functions:

- Allocate a device information structure. The driver uses the device information structure to hold information relating to a specific device (status, permissions, and so on).

  While the driver can define most of this structure's internal specifics, the structure must contain a pointer to the driver's interrupt service routine (if it has one) in the first field.

  In addition, if you want to use the kernel's routines for managing a select list (see Chapter 8), you should allocate a select list header in the device information structure. The select list header type is defined in **i_io.h**. You will also have to initialize this list by calling the kernel's **io_init_select** routine.

- If the device handles interrupts from the host, you must register the device information structure using the **io_register_device_info** routine. Registering the device information links the hardware device code with the interrupt service routine given in the device information structure.

- Define a device handle and device number by calling **io_allocate_device_number**. **io_allocate_device_number** allocates a minor number for the device specified and links the device number and device handle in the kernel's internal tables. Later you can retrieve this information by using kernel routines for accessing device information (see Chapter 8). The kernel will pass the device number to your driver's **dev_xxx_open** routine, but thereafter it will identify a device to all driver routines by passing the device handle.

  If the device has a controller with accessible units, you should establish a device number and device handle for all units that users will access.

- Create device special files. As with device numbers, you should create special files for all the units that users will access. We recommend that you create the special files after registering your device information structure, because it is possible for the register operation to fail. Chapter 8 describes kernel routines that create device special files.

If the device has a controller, the driver usually performs any initialization needed to bring the controller on-line here so that it can initialize the controller's units at open time.

If a failure occurs in any phase of the operation, **dev_xxx_configure** must return the system to the state it was in before the **dev_xxx_configure** routine was called. Data structures must be deallocated and the device interrupt table slot freed.

The **dev_xxx_configure** routine should be written such that it can be called anytime during the life of the system.

Return Values

OK — The device was successfully configured.

IO_ENXIO_DEVICE_NOT_RECOGNIZED — **device_name_ptr** does not specify a device in the class supported by this driver.

IO_ENXIO_DEVICE_NOT_SUPPORTED — **device_name_ptr** specifies a device in the device class supported by this driver, but the particular model is not supported.

IO_EIO_PHYSICAL_UNIT_FAILURE — A request issued to the device controller failed with an error status.

IO_EIO_DEVICE_TIMED_OUT — The controller did not respond to a request within a reasonable length of time.

IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED — A device is already registered at the location specified by **device_name_ptr**.

dev_xxx_open

Syntax

```
status_type   dev_xxx_open (device_number, channel_flags,
                            device_handle_ptr)

io_device_number_type       device_number;     /*READ ONLY*/
io_channel_flags_type       channel_flags;     /*READ ONLY*/
io_device_handle_ptr_type   device_handle_ptr; /*WRITE ONLY*/
```

Summary

This routine prepares a specified device for future I/O operations. It also adds the device to the set of devices on which I/O may be performed by this driver.

Parameters

device_number — The major and minor device numbers of the device being opened.

channel_flags — The set of channel flags specifying whether the device will be open for reads, writes, or both. The channel flags also indicate whether the open is for block or character special operation. See **i_io.h** for a listing of the channel flags.

device_handle_ptr — A pointer to the location where the device handle (that results from the open) is to be placed. If the routine does not return an OK status, this value is undefined. The driver need not check the validity of this pointer.

Description

The **dev_xxx_open** routine prepares the device for future I/O operations. The kernel calls it whenever one of the driver's devices is opened by a user or by the kernel. The kernel will not call **dev_xxx_open** until both **dev_xxx_init** and **dev_xxx_configure** have completed.

The DG/UX system allows multiple opens on a device, and **dev_xxx_open** should manage this feature as appropriate to its device. **dev_xxx_open** controls the number of outstanding opens. For example, **dev_xxx_open** may impose restrictions such as requiring an exclusive open of a particular minor device. To implement this, **dev_xxx_open** might return an error status if the minor device has already been opened but not closed. Multiple **dev_xxx_opens** may be in progress simultaneously on the same or different minor device numbers.

The **dev_xxx_open** routine must also control the type of open requested. The kernel passes **dev_xxx_open** a set of channel flags which specify the intents given in the higher level **open** call (for example, read, write or both). **dev_xxx_open** may reject the open because of conflicts between the current open intent and open intents that have already taken place or because of conflicts with the device's capabilities. For example, a write intent on a read-only device must fail.

dev_xxx_open typically performs other operations to prepare the device and ensure that it is ready for I/O. For example, it may allocate storage for and initialize databases to hold information describing the I/O operation on the specific unit. If the device is a real hardware device, **dev_xxx_open** may query the device to verify that it is online and ready for the type of I/O specified in the open intent. For example, it may check that there is a write ring in the tape if write intent is specified.

dev_xxx_open must establish the device handle that the kernel will use as a parameter in all future driver operations. It can retrieve the device handle supplied by **dev_xxx_configure** by calling **io_map_device_number** with the device number. If **dev_xxx_open** returns an OK, it must return a pointer to a device handle in **device_handle_ptr**. If it returns a status other than OK, the kernel presumes that the open failed and that the device handle will not be used. If the open fails, the kernel will disregard the returned **device_handle_ptr** argument.

Return Values

OK — The **dev_xxx_open** routine was successful in preparing the device for further operations.

IO_ENXIO_UNIT_NOT_READY — The unit is not ready or online.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — The specified device number cannot be mapped to a configured device.

IO_ENXIO_OPEN_INTENT_CONFLICTS The unit is already open and can only be opened exclusively.

IO_ENXIO_NO_WRITE_RING — The tape was opened with write intent, but the tape did not have a write ring (only applicable to tape devices).

IO_ENXIO_TAPE_DENSITY_NOT_SUPPORTED — The requested density is not supported by the tape controller (only applicable to tape devices).

IO_ENXIO_CANNOT_CHANGE_TAPE_DENSITY — The requested density is not compatible with the current density setting of the unit, and the tape is not at the beginning of the tape (BOT) (only applicable to tape devices).

dev_xxx_close

Syntax

```
status_type dev_xxx_close  (device_handle, channel_flags)

io_device_handle_type       device_handle;  /*READ ONLY*/
io_channel_flags_type       channel_flags;  /*READ ONLY*/
```

Summary

This routine removes a specified device from the set of devices on which this driver may perform I/O.

Parameters

device_handle — The device handle of the device to be closed. This handle will be the device handle that was returned by the driver's **dev_xxx_open** routine. The driver does not need to validate this argument.

channel_flags — Flags indicating how the device was opened (read, write, or read/write). The driver does not need to validate this argument.

Description

The **dev_xxx_close** routine performs operations that remove the specified device from the set of devices on which this driver may perform I/O. It is invoked in one-to-one correspondence to successful **dev_xxx_opens** and always with the same intents supplied to the open and with the device handle returned by the open. However, if a device is opened multiple times, it will not necessarily be closed in the same or reverse order of the opens.

Typically, **dev_xxx_close** performs any necessary exit operations such as flushing any buffers that may be present and releasing previously allocated storage. Some devices will also have special exit requirements. For example, a tape close would probably rewind the tape. Most drivers also use **dev_xxx_close** in coordination with **dev_xxx_open** to manage the number of outstanding opens.

Return Values

OK — The close was successful.

IO_STATUS_ERROR_ON_EARLIER_REQUEST — An error occurred on the last asynchronous request made to the device. Since the last request was asynchronous, this is the first opportunity to notify the interested process.

dev_xxx_service_interrupt

Syntax

```
void  dev_xxx_service_interrupt  (device_info_ptr)

dev_xxx_device_info_ptr_type      device_info_ptr;/*READ ONLY*/
```

Summary

This routine handles interrupts from devices under the control of this driver. It is called by the system interrupt handler.

Parameters

device_info_ptr — A pointer to the device information structure for the interrupting device. A pointer to the **dev_xxx_service_interrupt** routine is the first field in this structure. The driver may assume that this pointer is valid.

Description

The **dev_xxx_service_interrupt** routine performs any steps needed to service the device at interrupt level. It operates in a restricted environment: interrupts are disabled; no page faults may be taken; and the process must not wait. Because of these restrictions, **dev_xxx_service_interrupt** should defer as much device service as possible to a base-level process. The driver designer should determine the proper balance between executing code at interrupt level and at base level.

Because **dev_xxx_service_interrupt** must avoid calling any routine that may pend, it must forgo virtually all the kernel-supplied utilities. To signal or send information back to other processes, the driver should use the Driver or Generic Daemon. You send a message to the appropriate daemon by queuing a message with a completion routine to the daemon's queue. (Chapter 8 describes the **io_queue_message_to_driver_demon** and **io_queue_message_to_generic_demon** routines that you use to queue messages.) The daemon will dequeue the message and execute the completion routine in the daemon's context rather than the service routine's limited context.

Typically, **dev_xxx_service_interrupt** might do any of the following: read the device's status registers; advance an eventcounter for synchronous events; send a message to the Driver or Generic Daemon for an asynchronous event; or do a select satisfy for a select operation. If the interrupt is not cleared automatically by reading the status register, **dev_xxx_service_interrupt** must clear the interrupt before exiting.

The pointer to the device information structure allows **dev_xxx_service_interrupt** to access the device database associated with the I/O request.

NOTE: If the device does not generate hardware interrupts, you do not need to create this routine.

Return Values

None.

dev_xxx_read_write

Syntax

```
status_type  dev_xxx_read_write (request_info_ptr)

io_request_info_ptr_type request_info_ptr;   /*READ ONLY*/
```

Summary

This routine performs a synchronous read or write of the specified device.

Parameters

request_info_ptr — A pointer to a request information structure. The structure may be assumed to have a valid operation code, device handle, and memory address as specified in the buffer vector. The driver must validate the device offset and transfer count as being appropriate for the device.

The transfer count specifies the maximum number of bytes that should be transferred. The driver determines how much data to actually transfer before returning; in the case of an error, the amount may be less than the transfer count. However, under no circumstances may the amount of data transferred exceed the specified maximum.

The offset specified is a file pointer maintained by the kernel and indicates where the read/write operation should begin. For example, on a disk, the offset might specify where, after the start of the sector, the desired data is located. The driver may ignore this parameter if it is not applicable to its device — for example, if the device is character special.

The request information structure may exist in the caller's per-process address space. Therefore it can be accessed only when the requesting process is running and not from the interrupt level."

Description

The **dev_xxx_read_write** routine performs a synchronous read or write of the specified device, transferring data between the device and the specified buffer. It is invoked whenever a user or kernel read or write is performed on a character special device supported by this driver. This routine is usually used for character special I/O, but it may also be used for block special I/O.

Multiple reads/writes of the same or different minor devices may be in progress simultaneously. Therefore the driver should take steps to serialize requests as needed. This usually means using locks on important data structures.

The **dev_xxx_read_write** routine should also handle any special transfer constraints for its device. For example, a disk device might allow only those transfer counts that are multiples of 512 bytes. **dev_xxx_read_write** should be prepared to handle a self-identification check using the **self-id** field in the request information structure. The kernel may set a flag requesting that the driver validate that this **self-id** matches the device's **self-id** as stored in data blocks read from that device. You can use the kernel's **fs_check_self_id** routine to retrieve the **self-id** stored in the data (see Chapter 8).

We say that the **dev_xxx_read_write** routine is synchronous because when it completes, any data that is going to be transferred to or from the buffer must already be transferred. Because it must wait for the I/O to complete, **dev_xxx_read_write** will need to set up await mechanisms such as a timeout, a signal, or an I/O completion event. Chapter 6 describes kernel routines that the driver may use to implement these await mechanisms.

The kernel checks access to the buffer before **dev_xxx_read_write** is called so that the driver is ensured write access to the buffer for read operations and read access to the buffer for write operations. The buffer may exist in the caller's per-process address space and therefore may be accessed only when the requesting process is running.

The specified buffer is not necessarily wired in memory, and many devices must have wired buffers. For such devices, the driver must explicitly wire the buffer and unwire it before returning.

After the read/write, the driver should update the buffer vector pointers to reflect the actual data transferred (which may be less than the transfer count in cases of error). All references and updates to data contained in the **io_buffer_vector** structure must be done through kernel routines. Chapter 7 describes the kernel routines used to manipulate buffer vectors.

Return Values

The **dev_xxx_read_write** routine must return a status indicating the success or failure of the transfer. The definition of success or failure is determined by the driver and need not be related to the number of bytes actually transferred.

OK — The **dev_xxx_read_write** routine was successful.

IO_EINVAL_ILLEGAL_REQUEST_SIZE — The requested count is not valid for the device type.

IO_EINVAL_ILLEGAL_BUFFER_ADDRESS — The buffer was not aligned as required by the device.

IO_EIO_DEVICE_TIMED_OUT — The device controller did not respond to a request in a reasonable length of time.

IO_EIO_HARD_IO_ERROR — An unrecoverable I/O error occurred, resulting from a media failure.

IO_EIO_PHYSICAL_UNIT_FAILURE — An uncorrectable error occurred that presumably affects I/O operations to the entire physical unit.

IO_ENXIO_ILLEGAL_DEVICE_ADDRESS — The location specification for reading/writing is invalid for the device.

IO_EINTR_INTERRUPTED_BY_SIGNAL — A signal was received while waiting for the I/O to complete.

dev_xxx_select

Syntax

```
void  dev_xxx_select (device_handle, select_mode,
                      ec_ptr, intent_ptr)

io_device_handle_type       device_handle;  /*READ ONLY*/
boolean_type                select_mode;    /*READ ONLY*/
vp_ec_ptr_type              ec_ptr;         /*READ ONLY*/
io_select_intent_ptr_type   intent_ptr;     /*READ/WRITE*/
```

Summary

This routine supplies information about whether the specified device is ready to perform an I/O operation.

Parameters

device_handle — The device handle of the device that is the target of the select operation. The device handle argument need not be validated by the driver.

select_mode — If **select_mode** is TRUE, this is the start of a select operation. If **select_mode** is FALSE, this is the end of a select operation.

ec_ptr — **ec_ptr** specifies the eventcounter to be advanced by the driver when the particular type of select is satisfied. The driver does not need to validate this pointer.

intent_ptr — The **intent_ptr** parameter points to an intent variable consisting of a set of intent flags. The kernel calls the driver with the intent flags showing whether the device is being selected for read, write, or exceptional conditions or any combination of these. When the driver returns, it sets the intent flags to show which input conditions are currently TRUE. The driver does not need to validate this argument. The possible intent flags are described in the "Constants and Data Structures" section of this chapter.

Description

The **dev_xxx_select** routine is called in response to user-level **select** system calls. It operates as follows:

- If the user is selecting a device (**select_mode** argument is TRUE), and the device is ready for at least one of the incoming intents, the driver should set the intent flags to match the device's current state and return. It should set the flags to FALSE for all intents that are

not currently ready. It should set the intent flag to TRUE if that flag was TRUE on input and the intent is currently ready.

If none of the conditions the caller was interested in are TRUE, the driver should add the eventcounter pointed to by **ec_ptr** to a list of events maintained by the driver. Later, when one of the specified intents becomes TRUE, the driver must advance this eventcounter. Usually, drivers have the **dev_xxx_service_interrupt** routine complete select processing via a message to the Driver or Generic Daemon.

- If the user is unselecting the device (the **select_mode** argument is FALSE), the previously saved **ec_ptr** is discarded and any intents that have become TRUE are reported.

Multiple selects of the same or different minor devices may be in progress simultaneously. The **dev_xxx_select** routine must be able to store multiple eventcounter names for each of the read, write, and exception selects and advance them all when the intent becomes TRUE. Kernel routines for managing select lists (adding and removing entries and satisfying selects) are described in Chapter 8. The select list structure should have been allocated and initialized earlier, usually in the **dev_xxx_configure** routine.

Return Values

None.

Remarks

For many devices, such as disks and tapes, **dev_xxx_select** will always return TRUE because the I/O operations are so quick. **dev_xxx_select** is more meaningful on character devices that depend upon external intervention. For example, a terminal might select FALSE for writing when a user's terminal output is being held with Ctrl-S. Similarly, a terminal would select FALSE for reading when the driver is waiting for the user to type something.

dev_xxx_ioctl

Syntax

```
status_type  dev_xxx_ioctl  (device_handle, command,
                             parameter, return_value_ptr)

io_device_handle_type    device_handle;    /*READ ONLY*/
bit32e_type              command;          /*READ ONLY*/
bit32e_type              parameter;        /*READ/WRITE*/
int32e_ptr_type          return_value_ptr;/*WRITE ONLY*/
```

Summary

This routine performs a control operation on the specified device.

Parameters

device_handle — The device handle of the device that is the target of the I/O control operation. The device handle need not be validated by the driver.

command — A command to the device. Because commands are specific to the driver, they must be validated by the driver.

parameter — An argument to the command. The interpretation of the parameter is specific to the driver and the command. The parameter may be used to transfer information between the caller and the device, in either direction. In particular, it may be a pointer to a buffer supplied by the caller. Because the interpretation is specific to the driver, the driver must validate this argument.

return_value_ptr — A pointer to a return value that this routine can define and pass the user. This additional return value increases the flexibility of your **ioctl** operation by providing the user with variations on the generic return value specified in the "Return Values" section.

Description

The **dev_xxx_ioctl** routine performs a control operation on the specified device based on the values of **command** and **parameter**. It is invoked in response to a user or kernel **ioctl** call for one of the driver's devices. However, not all user **ioctl** calls go to **dev_xxx_ioctl**. Some **ioctl** calls are actually file descriptor operations. These are intercepted and handled by the kernel. The **FIONCLEX** operation, for example, would not reach **dev_xxx_ioctl**. Multiple **ioctl** operations on the same or different minor devices may be in progress simultaneously.

The kernel calls **dev_xxx_ioctl** with the **command** and **parameter** arguments given in the higher level **ioctl** call. The kernel will not interpret these arguments. Thus, you can define your driver's **command** and **parameter** arguments as you wish.

Because **ioctl** operations are so specific to each driver, the kernel validates only the device handle argument. The driver must validate the **command** and **parameter** arguments. It should also validate any buffer pointers for proper access. Chapter 7 describes kernel routines you can use to validate pointers.

Return Values

The **dev_xxx_ioctl** routine should return a status indicating the success or failure of the control operation. The definition of success or failure is determined by the driver.

OK — The **dev_xxx_ioctl** routine was successful. No errors should be indicated to the caller.

IO_EINVAL_COMMAND_NOT_SUPPORTED_BY_DEVICE — The **command** was not supported by the driver.

IO_EFAULT_BAD_ADDRESS_IN_IOCTL — The **parameter** argument specified an address that is not a valid part of the caller's address space.

dev_xxx_start_io

Syntax

```
status_type  dev_xxx_start_io (op_record_ptr)

io_operation_record_ptr_type  op_record_ptr;  /*READ ONLY*/
```

Summary

This routine starts an asynchronous I/O operation on the specified device.

Parameters

op_record_ptr — A pointer to the operation record for the asynchronous request. The operation record contains the following fields:

- The device handle that is the target of the operation.
- The operation to be performed; for example, read, write, or both.
- The offset from which the operation is to commence. The offset is a file pointer maintained by the kernel that indicates where the read/write operation should begin. For example, on a disk the offset might indicate where to start reading after the start of the sector. The interpretation of the offset is driver-dependent.
- The address of the kernel's I/O completion routine that is to be called by the driver's *complete_io* routine when the operation completes. The Kernel I/O completion routine follows the interface shown in the "Kernel I/O Completion Routine Interface" section below. Note that you pass the **op_record** and a completion status back as parameters to the Kernel I/O Completion routine.
- An **io_buffer_vector** structure that holds the transfer size and the address of the memory buffers.

The operation record must reside in global kernel memory, so it may be accessed by any process — not just the requestor. The driver need not validate most of the fields of the operation record. The exceptions are the device offset and transfer size fields. The driver may need to check these fields to ensure that they are meaningful for the device.

Description

The **dev_xxx_start_io** routine is invoked only on block special devices. Multiple **dev_xxx_start_io** routines on the same or different minor devices may be in progress simultaneously.

When a user initiates a read or write operation on a block special device, the kernel will invoke **dev_xxx_start_io** to process the request asynchronously. **dev_xxx_start_io** should start the operation and then exit, leaving the completion to be handled by another routine. If **dev_xxx_start_io** cannot initiate the operation (for example, if the device is busy), it should queue the request (usually with the Driver or Generic Daemon) to be handled later and exit. This routine should not pend.

The **dev_xxx_start_io** routine is asynchronous in that when it returns, the data transfer is not necessarily complete. The driver must therefore decide how to finish processing once the operation is complete. The driver is relatively free to handle completion as necessary for its own device.

The only thing the driver must do for completion is to call the kernel completion routine supplied in the operation record. The kernel waits until its I/O completion routine is called before expecting new data in the buffer, modifying data in a buffer that was written, or modifying the operation record that was passed in as an argument. Until the kernel's I/O completion routine is called, the kernel does not consider the operation complete.

The driver decides when to call the kernel's I/O completion routine. Typically, when the operation completes, the **dev_xxx_service_interrupt** routine queues a message to the Driver or Generic Daemon. The message contains a pointer to a routine for the daemon to execute. This routine might be either the kernel's I/O completion routine or a driver-supplied *complete_io* routine that in turn calls the kernel's I/O completion routine. In Chapter 8, we describe the kernel routines for interacting with the Driver or Generic Daemon.

The upcoming section called "Kernel I/O Completion Routine Interface" describes the interface used by the kernel's I/O completion routine. We do not give an interface description for a driver-supplied *complete_io* routine. Such a routine is completely optional. If you want to implement such a routine, you do not need to follow any kernel-specified interface.

The following implementation notes are relevant regardless of how completion is implemented:

- The driver must transfer the exact amount of data specified in the request unless there is an I/O error (in which case less data is acceptable). Under no circumstances may the amount of data transferred exceed the amount specified.

- It is possible that the kernel's I/O completion routine may be called before **dev_xxx_start_io** finishes. Therefore, the Driver or Generic Daemon may actually call the kernel's I/O completion routine before **dev_xxx_start_io** returns. Thus, the kernel's I/O completion routine must not be called by the process that has the **dev_xxx_start_io** in progress.

- The buffer to receive the data is wired in memory. The driver may perform logical-to-physical address translations without having to explicitly wire the buffer. Further, the buffer must reside in global kernel memory, so any process may access the buffer or perform the logical-to-physical translation.

Return Values

OK — This routine always returns OK. Errors that occur on the request are reported via the completion routine.

Kernel I/O Completion Routine Interface

Syntax

```
void kernel_complete_io (op_record_ptr, status)

io_operation_record_ptr_type  op_record_ptr;  /*READ ONLY*/
status_type                   status;         /*READ ONLY*/
```

Summary

The kernel supplies a routine that adheres to this interface to perform work necessary when an asynchronous I/O operation completes.

Parameters

op_record_ptr — A pointer to the operation record for the request that has completed. The operation record contains fields that indicate the device handle that is the target of the operation, the operation to be performed, the offset on the device from which the operation is to commence, the address of the routine that is to be called when the operation completes, an I/O buffer vector structure that contains the size of the transfer, and the address of the main memory buffer.

status — The completion status of the request.

Description

The kernel's I/O completion routine performs the cleanup work necessary when an asynchronous I/O completes. The driver calls it to indicate that the operation is complete.

The **status** argument indicates the result of the asynchronous I/O operation.

Return Values

The kernel's I/O completion routine must always succeed, therefore it does not have a return value.

dev_xxx_open_dump

Syntax

```
status_type  dev_xxx_open_dump (device_name)

char_ptr_type    device_name;    /*READ ONLY*/
```

Summary

This routine prepares one of the driver's devices for use as the destination of a system dump.

Parameters

device_name — A pointer to the null-terminated string identifying the device to be opened as a dump device.

Description

A master file entry specifies the default device for a system dump, but during the dump procedure the user is allowed to specify an alternative dump destination. If your device is selected as the dump destination, the system will call your **dev_xxx_open_dump** routine if the system panics.

The **dev_xxx_open_dump** routine initializes the device as a dump destination. To do this, it must reinitialize the device's controller (and/or units). Because the system is in an undefined state as a result of the panic, the standard kernel facilities will not be available for the initialization procedure. In particular, this means that **dev_xxx_open_dump** must run in physical memory — dynamically allocated memory cannot be accessed. **dev_xxx_open_dump** should statically allocate its data structures in **dev_xxx_global_data.c**.

The **dev_xxx_open_dump** routine should also use busy waits when interacting with the controller, because the standard interrupt mechanism will not be available.

Because the dump procedure is a single-threaded process, kernel locking mechanisms are not required and should not be used.

The **dev_xxx_open_dump** routine receives a device name specified by **device_name_ptr**. It should verify that the device specified is of the driver's type. The device name is of the following form:

device_mnemonic [@*device_code*] ([*parameters*])

Finally, as with any open routine, **dev_xxx_open_dump** should perform any

operations necessary to ensure that the device is on-line and ready for a write operation. For example, if the device is a tape, the tape should be on-line and write-enabled.

NOTE: This routine must not panic, because it is invoked as part of the panic sequence.

Return Values

OK — The open completed successfully.

IO_STATUS_DUMP_NOT_SUPPORTED — The device identified by the **device_name** string is not supported as a dump device by this driver. Either the device mnemonic does not match the mnemonic associated with your driver; the device name is in an unrecognizable format; or the device specified by **device_name_ptr** is supported by the driver but the device type is not a valid dump destination device.

IO_EIO_HARD_IO_ERROR — A request to the dump destination device has resulted in an error condition.

dev_xxx_write_dump

Syntax

```
status_type     dev_xxx_write_dump (buffer_ptr, buffer_size)

pointer_to_any_type     buffer_ptr;     /*READ ONLY*/
uint32_type             buffer_size;    /*READ ONLY*/
```

Summary

This routine writes system dump data to the dump destination device previously opened by **dev_xxx_open_dump**.

Parameters

buffer_ptr — A pointer to the buffer containing the data to be written.

buffer_size — The size of the buffer, in bytes.

Description

During a dump, the system's panic code calls **dev_xxx_write_dump** to write a single physical record of size **buffer_size**. The panic code will call the **dev_xxx_write_dump** routine as many times as necessary to transfer all of the dump data. You will not need to wire buffer memory or verify parameters for this routine.

If the dump destination must use multiple volumes to hold the entire system dump, the **dev_xxx_write_dump** routine should close the completed volume, request that the operator mount a new volume, and open the new volume.

NOTE: Because the normal kernel facilities are not available, this routine should busy-wait for the write operations to complete. The normal system interrupt handler is not available.

Also, this routine must not panic because it is invoked as part of the panic sequence.

Return Values

OK — The write operation completed normally.

IO_STATUS_TAKE_CHECKPOINT — The write operation completed normally but was written as the first record on a volume. The system dump code should checkpoint its current state.

IO_EIO_HARD_IO_ERROR — An unrecoverable I/O error occurred. The system dump code should restore its state from the last checkpoint and begin writing again from there. This error does not occur on the first record of the volume.

dev_xxx_read_dump

Syntax

```
status_type   dev_xxx_read_dump  (buffer_ptr, buffer_size)

pointer_to_any_type   buffer_ptr;     /* WRITE ONLY  */
uint32_type           buffer_size;    /* READ  ONLY  */
```

Summary

This routine handles reading a system dump.

Parameters

buffer_ptr — A pointer to the buffer to which data is to be read.

buffer_size — The size, in bytes, of the buffer.

Description

The DG/UX system does not support this operation at this time. You should use the appropriate **io_nodevice** routine stub for this routine.

Return Value

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_READ_DUMP — An attempt was made to read dump information from a non-existent device.

dev_xxx_close_dump

Syntax

```
status_type     dev_xxx_close_dump ()
```

Summary

The routine closes the dump device previously opened by **dev_xxx_open_dump**.

Parameters

None.

Description

The **dev_xxx_close_dump** routine is called by the system dump code when all of the data has been written to the dump destination. **dev_xxx_close_dump** should perform all the standard exit operations (for example, write End-of-file or rewind the tape). In particular, it should close the completed volume and inform the operator that the dump has completed.

Return Values

OK — The volume was successfully closed.

IO_EIO_HARD_IO_ERROR — An unrecoverable error occurred in closing the volume. The operator is prompted to mount another volume, and the system dump utility should resume operation at its last checkpoint

dev_xxx_powerfail

Syntax

```
status_type     dev_xxx_powerfail  ()
```

Summary

This routine restarts all devices managed by this driver when power has been restored after a power failure.

Parameters

None.

Description

The DG/UX system does not support this operation at this time. You should use the appropriate **io_nodevice** stub for this routine.

Return Values

OK — Return this value in all cases.

dev_xxx_deconfigure

Syntax

```
status_type     dev_xxx_deconfigure (device_name_ptr)

char_ptr_type   device_name_ptr;        /*READ ONLY*/
```

Summary

This routine deconfigures the specified device if it is in the class supported by this driver.

Parameters

device_name_ptr — A pointer to the null-terminated string specifying the device to be deconfigured.

Description

The **dev_xxx_deconfigure** routine does the opposite of the configure routine (see **dev_xxx_configure**). It releases all system resources obtained to configure the device. After **dev_xxx_deconfigure** has completed, the system should be in the state it was in before the device configure routine was executed. **dev_xxx_deconfigure** performs the following functions:

- Deallocates a device information structure
- Frees the minor number
- Deregisters device information
- Releases all memory

NOTE: Device special files created by the configure operation do not have to be removed.

dev_xxx_deconfigure receives a pointer to a device specification of the following form:

device_mnemonic [@*device_code*] ([*parameters*])

The pointer is terminated by a null character.

Return Values

OK — The device was successfully deconfigured.

IO_ENXIO_DEVICE_NOT_RECOGNIZED — **device_name_ptr** does not specify a device in the class supported by this driver. This error is returned when the device mnemonic does not match the mnemonic associated with the driver.

IO_EBUSY_DEVICE_HAS_OPEN_UNITS — The specified device currently has one or more units that are open.

dev_xxx_device_to_name

Syntax

```
status_type dev_xxx_device_to_name (device_number,
                                    name_ptr,size)

io_device_number_type   device_number;  /*READ ONLY*/
char_ptr_type           name_ptr;       /*WRITE ONLY*/
uint32_type             size;           /*READ ONLY*/
```

Summary

This routine returns the device name associated with the specified device number. The name is returned as a null-terminated string.

Parameters

device_number — The device number for the device whose name is desired. The device number consists of a major and minor device number.

name_ptr — A pointer to where the device name is to be written. The name will be in the form of a null-terminated string.

size — The maximum number of bytes, including the terminating null, that is to be written to **name_ptr**.

Description

The **dev_xxx_device_to_name** routine is called by various file system utilities to translate a device number into a device name consisting of a device code and unit number. It returns the name in a string of the following form:

device_mnemonic [*@device_code*] ([*parameters*])

To simplify its operation, **dev_xxx_device_to_name** may call the kernel's **io_map_device_number** to retrieve the device code and unit number for the given device number.

Return Values

OK — The translation was performed successfully.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — The specified device number is not configured.

dev_xxx_name_to_device

Syntax

```
status_type dev_xxx_name_to_device (device_name_ptr,number_ptr)

char_ptr_type                 device_name_ptr;   /*READ ONLY*/
io_device_number_ptr_type     number_ptr;        /*WRITE ONLY*/
```

Summary

This routine returns the device number for the specified device name.

Parameters

device_name_ptr — A pointer to the device name that is to be translated. The name is in the form of a null-terminated string.

number_ptr — A pointer to where the corresponding device number is to be written. The device number consists of a major and minor device number.

Description

This routine is called by various file system utilities to translate a device name into the major and minor device numbers that are required to access the device. The device name specified by **device_name_ptr** is of the following form:

device_mnemonic [@*device_code*] ([*parameters*])

To simplify its processing, **dev_xxx_name_to_device** can call the kernel's **io_get_device_info**, which returns a pointer to the device information structure that will contain the device's device number.

The driver should verify that the device mnemonic given in the name matches its own mnemonic (**xxx**).

Return Values

OK — The device name was successfully translated.

IO_ENXIO_DEVICE_NOT_RECOGNIZED — The specified device is not supported by this driver. This error is returned when the device mnemonic does not match the mnemonic associated with the driver.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — The specified device is supported by this driver but is not currently configured in the system.

dev_xxx_maddmap

Syntax

```
status_type     dev_xxx_maddmap     ()
```

Summary

This routine increments reference counts to memory mapped sections.

Parameters

None.

Description

The DG/UX system does not support this operation at this time. You should use the appropriate **io_nodevice** routine stub for this routine.

Return Value

IO_EINVAL_MMAP_NOT_SUPPORTED — The **maddmap** operation is not supported for this device.

Exceptions

None.

dev_xxx_mmap

Syntax

```
status_type     dev_xxx_mmap     ()
```

Summary

This routine handles the **mmap** system call.

Parameter

None.

Description

The DG/UX system does not support this operation at this time. You should use the appropriate **io_nodevice** routine stub for this routine.

Return Value

IO_EINVAL_MMAP_NOT_SUPPORTED — The **mmap** operation is not supported for this device.

Exceptions

None.

dev_xxx_munmap

Syntax

```
status_type     dev_xxx_munmap     ()
```

Summary

This routine handles the **munmap** system call.

Parameters

None.

Description

The DG/UX system does not support this operation at this time. You should use the appropriate **io_nodevice** routine stub for this routine.

Return Value

IO_EINVAL_MUNMAP_NOT_SUPPORTED — The **munmap** operation is not supported for this device.

Exceptions

None.

User-Supplied Adapter Driver Routines

This section describes the interfaces for routines you must write to build an SCSI adapter driver. Chapter 5 describes the SCSI adapter routines that come with the DG/UX system. You only need to write adapter routines if your application requires changes to these routines or if you are going to add a different type of adapter board to your system. As with user-supplied device driver routines, replace **xxx** with the mnemonic for your SCSI adapter.

Constants and Data Structures

The following constants and data structures are used by SCSI device and adapter drivers. They are found in the **dev_scsi_adapter_def.h**.

dev_scsi_adapter_routines_vector_type

```
typedef struct
    {
    io_driver_routines_vector_type          driver_routines;
    dev_scsi_interface_routines_vector_type scsi_routines;

       }  dev_scsi_adapter_routines_vector_type  ;
```

Description

This structure describes the adapter driver routines vector. Note that it contains both the standard device driver routines vector and the additional adapter-driver-specific routines vector.

dev_scsi_interface_routines_vector_type

```
typedef struct
    {
    uint16_type         version;
    bit16_type          flags;
    status_type         (*register_requester)();
    status_type         (*set_unit_options)();
    void                (*deregister_requester)();
    status_type         (*issue_command)();
    status_type         (*issue_async_command)();
    status_type         (*get_device_info)();
    status_type         (*issue_command_physical_mode)();

       }  dev_scsi_interface_routines_vector_type  ;
```

Description

This type describes the adapter-driver-specific routines vector. This vector contains a pointer to each adapter driver routine that can be called by the adapter manager. The version field is present to allow changes to this structure while retaining compatibility with older user written device drivers. The structure as currently defined is version 1. The flags field is currently unused.

dev_scsi_adapter_unit_spec_type

```
typedef struct
    {
    uint8_type          scsi_id;
    uint8_type          unit;

        }    dev_scsi_adapter_unit_spec_type  ;
```

Description

This structure is used to specify a particular instance of an SCSI device on an SCSI adapter.

The fields in this structure are as follows:

scsi_id — The SCSI ID to which the device responds on the SCSI bus.

unit — The unit number of the device.

dev_adapter_request_block_type

```
typedef struct
  {
  misc_queue_links_type              links;
  uint16_type                        type;
  bit16e_type                        request_flags;
  uint32_type                        reserved;
  io_device_handle_type              adapter_handle;
  dev_scsi_cmd_blk_type              scsi_cmd_blk;
  dev_scsi_adapter_unit_spec_type    unit_spec;
  io_buffer_vector_type              buffer_vector;
  dev_scsi_request_sense_buffer_type sense_buffer;
  boolean_type                       sync_io;
  io_operation_record_ptr_type       op_record_ptr;
  io_completion_routine_ptr_type     complete_io_routine;
  misc_clock_value_type              request_start_time;
  misc_clock_value_type              total_request_busy_time;
```

```
}     dev_adapter_request_block_type;
```

Description

This structure is a generic parameter block that SCSI device drivers use to specify a request to the supporting adapter driver.

The fields in this structure are as follows:

links — The queue manager uses this field to maintain the adapter request block on the various queues on which it may be queued during processing.

type — This field defines the type of the adapter request block and hence provides for multiple adapter request blocks. This provision has been made to allow for new types of adapter request blocks that may be needed as new types of adapters are added. This field must be defined. Check the DEV_SCSI_ARB_TYPE definitions in **dev_scsi_adapter_def.h** for a current listing of supported types.

request_flags — Flags field used to qualify the request. See the DEV_SCSI_REQUEST_FLAGS definitions below for more information.

reserved — This field is reserved for future use by Data General and must always be set to zero.

adapter_handle — The adapter driver uses this handle to map each instance of a device to the data structures used to control it. This handle is used only by the adapter driver.

scsi_cmd_blk — The command block which specifies the request to be made to the device. The SCSI command block is not interpreted by the adapter manager.

unit_spec — The device's SCSI ID and unit number.

buffer_vector — A buffer vector describing the main memory area that is to be involved in the I/O operation. Note that the SCSI interface manager assumes the buffer vector contains only a single buffer descriptor.

sync_io — This is a flag field. When set, this field indicates that the request is to be performed synchronously. If it is clear (false), the operation is performed asynchronously.

op_record_ptr — When the operation is to be performed asynchronously, this field contains a pointer to the original operation record that specified the request.

complete_io_routine — When operation is to be performed asynchronously, this field contains the address of the caller's I/O completion routine. The I/O completion routine will be called when the operation completes.

request_start_time — This field is used to save the starting time of a request for device usage accounting. Upon request completion, the start time is subtracted from the current time to get the total time required to process the request. This field is not used by the SCSI interface manager; it used is only by the SCSI device drivers.

total_request_busy_time — This field specifies the total amount of time that the target physical device spent processing the request. Since the SCSI device drivers have no precise knowledge of when the target device actually starts or completes a request, this information must be obtained from the supporting SCSI interface manager.

DEV_SCSI_REQUEST_FLAGS

The following constants define bit positions of the request flags used in the **request_flags** field of an adapter request block described by **dev_adapter_request_block_type**.

```
#define DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER     0x000001
```

This literal indicates that the data buffer address specified in the request block is a kernel address.

```
#define DEV_SCSI_REQUEST_FLAGS_PHYSICAL_BUFFER  0x000002
```

This literal indicates that the data buffer address specified in the request block is a physical address.

```
#define DEV_SCSI_REQUEST_FLAGS_SB_DONE 0x000004
```

This literal is used by the DG/UX sector-buffering mechanism. When set, it indicates that the current request is done. Sector buffering is performed by the DG/UX SCSI disk driver to allow access to a disk that does not have standard 512-byte sectors.

```
#define DEV_SCSI_REQUEST_FLAGS_SB 0x000008
```

This literal is used by the DG/UX sector-buffering mechanism. When set, it indicates that sector buffering is enabled for device access.

```
#define DEV_SCSI_REQUEST_FLAGS_SB_READ 0x0000010
```

This literal is used by the DG/UX sector-buffering mechanism. When set, it indicates that a sector buffered read is taking place.

```
#define DEV_SCSI_REQUEST_FLAGS_DATA_XFER_IN 0x0000020
```

This literal, if set, indicates that the data transfer direction is from the adapter to the host (that is, a read). If clear, the data transfer direction is from the host to the adapter (that is, a write). This flag is valid only if the number of bytes being

transferred is greater than zero.

dev_scsi_adapter_unit_registration_blk_type

```
typedef struct
    {
    io_device_number_type            adapter_device_number;
    dev_scsi_adapter_unit_spec_type  unit_spec;
    io_device_handle_type            adapter_handle;
    io_device_handle_type            driver_handle;
    uint16_type                      max_concurrent_requests;
    uint32_type                      max_request_size;
    bit8_type                        device_type;

        }    dev_scsi_adapter_unit_registration_blk_type  ;
```

Description

This structure is used by an SCSI device driver to register a physical unit with the SCSI adapter manager. Registration establishes a direct link between the device driver and the adapter manager service routines. Registering a unit consists of the device driver and the adapter driver exchanging information. This structure is a simple packaging of several variables needed for the registration process.

The fields in this structure are as follows:

adapter_device_number — The major and minor device numbers of the SCSI adapter with which the unit is being registered.

unit_spec — The SCSI ID and unit number of the device being registered.

adapter_handle — The handle that the SCSI adapter manager returns to the device driver requesting the registration. It is passed as an argument to the adapter driver routines to identify the physical unit that is the target of a request.

driver_handle — This is a unit handle that points to a driver-defined block of information specific to the unit being addressed. The device driver may pass this handle to the adapter manager when it registers the unit. The adapter manager saves this handle and returns it to the driver when the driver calls the adapter's **dev_xxx_get_device_info** routine. The adapter manager does not interpret this field.

max_concurrent_requests — The maximum number of concurrently executing requests on the unit that the driver will allow.

max_request_size — The adapter driver uses this field to return the maximum number of bytes transferable between the host and device in a single operation.

device_type — If a device is already registered with the specified SCSI ID and unit number, the device type of the registered device is returned in this field. Otherwise, the device type specified by **device_type** is recorded for the device.

dev_adapter_physical_request_blk_type

```
typedef struct
    {
    uint16_type                          type;
    uint16_type                          reserved;
    uint32_type                          reserved1;
    dev_scsi_cmd_blk_type                scsi_cmd_blk;
    io_buffer_vector_type                buffer_vector;
    dev_scsi_request_sense_buffer_type   sense_buffer;
    dev_scsi_mode_buffer_type            mode_select_buffer;
    uint16_type                          volume;
    dev_scsi_adapter_unit_spec_type      unit_spec;
    io_device_number_type                adapter_device_number;
    boolean_type                         is_open;
    boolean_type                         first_block_on_medium;

    }    dev_adapter_physical_request_blk_type  ;
```

Description

This structure defines the information block that is used by SCSI device drivers to specify an SCSI bus request when the system is in shutdown mode and a system dump is in progress.

The fields in this structure are as follows:

type — This field defines the type of the adapter request block and hence provides for multiple adapter request blocks. This provision has been made to allow for new types of adapter request blocks that may be needed as new types of adapters are added. This field must be defined. Check the DEV_SCSI_ARB_TYPE definitions in **dev_scsi_adapter_def.h** for a current listing of supported types.

reserved — This field is reserved for future use by Data General and must always be set to zero.

reserved1 — This field is reserved for future use by Data General and must always be set to zero.

scsi_cmd_blk — The SCSI command block, which specifies the request to be made to the device. The SCSI command block is not interpreted by the adapter manager.

buffer_vector — A buffer vector describing the main memory area that is to be involved in the I/O operation. Note that the SCSI interface manager assumes the buffer vector contains only a single buffer descriptor.

sense_buffer — Buffer to which sense information is returned if a request results in a Check Condition status.

mode_select_buffer — Buffer to which the device's current operating mode information is saved.

volume — Specifies the current volume number.

unit_spec — The device's SCSI ID and unit number.

adapter_device_number — The major and minor device number of the target adapter.

is_open — This is a flag field. When set, it indicates that the tape has been successfully opened as a system dump target.

first_block_on_medium — If an error occurs, this flag is used to determine whether to prompt for a new tape or to flag the shutdown manager to restart from the last checkpoint.

dev_scsi_adapter_unit_options_block_type

```
typedef struct
    [
    misc_clock_value_ptr_type disconnect_timeout_ptr;
    misc_clock_value_ptr_type bus_request_timeout_ptr;
    uint8_type                max_disconn_reconn_per_command;
    uint8_type                adapter_retries;
    uint8_type                sense_bytes;
    boolean_type              synchronous_data_transfers;
    boolean_type              perform_request_sorting;

        }    dev_scsi_adapter_unit_options_block_type  ;
```

Description

This structure is used by an SCSI device driver to specify various unit options to the **dev_xxx_set_unit_options** interface of the supporting SCSI adapter driver.

The fields in this structure are as follows:

disconnect_timeout_ptr — A pointer to a "misc_clock" value. This value determines how long the adapter driver will wait after a disconnect has occurred before assuming that an error has taken place and that the reselect will not be

occurring. If the timeout interval expires, a timeout error will be reported back to the caller. A **disconnect_timeout_ptr** of DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR disables disconnect timeouts for the device.

bus_request_timeout_ptr — A pointer to a "misc_clock" value. This value determines how long the adapter driver will wait after a bus request has been made before assuming an error has taken place and the request is aborted. A **bus_request_timeout_ptr** of DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR disables bus request timeouts for the device.

max_disconn_reconn_per_command — The maximum number of times that the SCSI target device can be expected to disconnect and reconnect during the execution of a single command. This value is used by some SCSI adapter drivers to calculate the maximum amount of time that a single request to the SCSI adapter should take. The time value is used as a backup timeout mechanism for SCSI adapters that manage disconnect timeouts and bus request timeouts internally. A value of zero for this field inhibits the device from disconnecting while a command is executing.

adapter_retries — The number of times the SCSI adapter driver will reissue a request if the request results in a hard I/O error.

sense_bytes — The number of sense bytes that will be returned from the device if a command to the device results in a Check Condition status.

synchronous_data_transfers — If non-zero, this flag indicates that data transfer to/from the device should be done in SCSI synchronous mode. If this option is selected on a device that does not support synchronous transfers, data will be transferred in asynchronous mode with no error reported to the driver.

perform_request_sorting — If non-zero, this flag indicates that the adapter driver should perform request sorting and ordering to provide more efficient access to the specified device.

SCSI Adapter Unit Options Block Literals

These definitions specify various constants that apply to Set Unit Options SCSI adapter routine.

```
#define DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR
                    ((misc_clock_value_ptr_type)DEFAULT_NULL_LINK)
```

This timeout pointer is specified in a unit options block to disable timeouts for a unit.

```
#define DEV_SCSI_ADAPTER_MIN_TIMEOUT_VALUE
```

This literal defines the minimum timeout value supported by an SCSI adapter driver. This value is in units of 1 millisecond.

```
#define DEV_SCSI_ADAPTER_MAX_TIMEOUT_VALUE
```

This literal defines the maximum timeout value supported by an SCSI adapter driver. This value is in units of milliseconds. Currently the maximum timeout interval supported is 30 minutes.

```
#define DEV_SCSI_ADAPTER_MIN_ADAPTER_RETRIES
```

This literal defines the minimum number of adapter retries that may be specified in a Set Unit Options block.

```
#define DEV_SCSI_ADAPTER_MAX_ADAPTER_RETRIES
```

This literal defines the maximum number of adapter retries that may be specified in a Set Unit Options block.

```
#define DEV_SCSI_ADAPTER_MIN_SENSE_BYTES
```

This literal defines the minimum number of sense bytes that may be specified in a Set Unit Options block.

```
#define DEV_SCSI_ADAPTER_MAX_SENSE_BYTES
```

This literal defines the maximum number of sense bytes that may be specified in a Set Unit Options block.

```
#define DEV_SCSI_ADAPTER_MIN_DISCON_RECON
```

This literal defines the minimum number disconnect/reconnects per command that may be specified in a set unit options block.

```
#define DEV_SCSI_ADAPTER_MAX_DISCON_RECON
```

This literal defines the maximum number disconnect/reconnects per command that may may be specified in a Set Unit Options block.

Interfaces for Adapter Driver Routines

DG/UX device driver routines use adapter drivers to interface to the SCSI bus. To write an adapter driver you must supply all the routines listed below. These routines must conform to the interface, as described in the rest of this chapter. As with device driver routines, substitute your own device mnemonic for the **xxx** shown in the routine names in this section.

You supply the entry points to your adapter routines in your routines vector which is defined in **dev_xxx_global_data.c**. SCSI device drivers obtain the vector to their adapter driver's routines during device configuration.

The adapter driver routines described in this chapter are listed below:

- **dev_xxx_register_requester**
- **dev_xxx_set_unit_options**
- **dev_xxx_deregister_requester**
- **dev_xxx_issue_command**
- **dev_xxx_issue_async_command**
- **dev_xxx_get_device_info**
- **dev_xxx_issue_command_physical_mode**

dev_xxx_register_requester

Syntax

```
status_type         dev_xxx_register_requester  (rb_ptr)

dev_scsi_adapter_unit_registration_blk_ptr_type rb_ptr;
                                                    /*READ/WRITE*/
```

Summary

This routine associates the specified device with an SCSI adapter, thereby establishing a link between the device driver and the adapter service routines.

Parameters

rb_ptr — A pointer to an SCSI adapter registration block.

Description

This routine adds an entry to the unit table associated with the specified SCSI ID and unit number. The unit table entry consists of a device type specifier and an opaque unit handle, which is meaningful only to the device driver. The unit table entry provides a bridge between the device driver and the adapter driver routines.

If the unit table entry specified by the SCSI ID and unit number is already occupied, an error is returned. Also, the device type of the device occupying the entry is returned so that the caller can distinguish between **IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED** and **IO_ENXIO_DEVICE_DOES_NOT_EXIST**.

Return Values

OK — The specified device was successfully registered with the adapter.

IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED — A device is already registered at the location specified by **rb_ptr**.

dev_xxx_set_unit_options

Syntax

```
status_type dev_xxx_set_unit_options (adapter_handle,
                                      unit_options_block_ptr)

io_device_handle_type  adapter_handle;  /*READ ONLY*/
dev_scsi_adapter_unit_options_block_ptr_type
                       unit_options_block_ptr;/*READ ONLY*/
```

Summary

Set the unit options of a registered device.

Parameters

adapter_handle — The device handle of the physical unit which is the target of the set unit options operation. This handle must be the device handle that was returned by the register-requester routine of the adapter manager.

unit_options_block_ptr — Pointer to a unit options block that specifies the options to be selected for the unit.

Description

This routine is called to set the various unit options that describe how the SCSI adapter driver manages a request that has been issued over the SCSI bus to a physical unit. See the definition of the **dev_scsi_adapter_unit_options_block** in the file **dev_scsi_adapter_def.h** for a complete description of the unit options supported.

Return Values

OK — The requested options were selected successfully.

DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE — An illegal option value was detected in the callers Set Unit Options Block.

IO_EIO_PHYSICAL_UNIT_FAILURE — The Set Unit Options command issued to the adapter resulted in an error.

dev_xxx_deregister_requester

Syntax

```
void  dev_xxx_deregister_requester  (adapter_handle)

io_device_handle_type    adapter_handle;/*READ ONLY*/
```

Summary

This routine terminates the link between the SCSI adapter manager and the device referenced by **adapter_handle**.

Parameters

adapter_handle — The device handle of the physical unit that is to be deregistered. This handle must be the device handle that was returned by the register-requester routine of the adapter manager.

Description

See Summary.

Return Values

None.

dev_xxx_issue_command

Syntax

```
status_type dev_xxx_issue_command  (arb_ptr)

dev_adapter_request_block_ptr_type  arb_ptr;  /*READ/WRITE*/
```

Summary

Issue an SCSI command synchronously through the adapter to a target device.

Parameters

arb_ptr — A pointer to a generic adapter request block that holds all information which describes the request.

Description

This routine transfers request information from the generic adapter request block to an adapter-specific parameter block and calls the adapter driver to execute the request.

If the request completes with a Check Condition status, sense information from the device is automatically returned in the adapter request block.

Return Values

OK — A synchronous request completed successfully.

Other return values that you deem necessary — The receiving device drivers should be prepared to handle these return values.

dev_xxx_issue_async_command

Syntax

```
status_type dev_xxx_issue_async_command  (arb_ptr)

dev_adapter_request_block_ptr_type arb_ptr;/*READ/WRITE*/
```

Summary

Issue an SCSI command asynchronously through the adapter to a target device.

Parameters

arb_ptr — A pointer to a generic adapter request block which holds all information that decribes the request.

Description

The adapter request block is added to the asynchronous request queue and an attempt is made to obtain the specified controller's command list request lock. If the lock is obtained, **dev_xxx_start_async_request** is called to start the request. Control is returned to the caller as soon as the request has been issued through the adapter to the physical unit. The Driver or Generic Daemon handles request completion and starts the next request in the queue if there is one.

If the command list request lock cannot be obtained, the request is left on the request queue and the function returns immediately. The enqueued request is started when the currently executing request and all requests ahead in the queue have been executed.

Return Values

OK — The request was successfully started. This status does not indicate that the request has completed successfully.

dev_xxx_get_device_info

Syntax

```
status_type dev_xxx_get_device_info (adapter_device_number,
                                     unit_spec,
                                     device_type,
                                     driver_handle_ptr)

io_device_number_type  adapter_device_number;/*READ ONLY*/
dev_scsi_adapter_unit_spec_type unit_spec;/*READ ONLY*/
bit8_type              device_type;       /*READ ONLY*/
bit32e_ptr_type        driver_handle_ptr; /*WRITE ONLY*/
```

Summary

This routine retrieves device information associated with a specified registered device.

Parameters

adapter_device_number — The major and minor device number of the SCSI adapter used to access the target unit.

unit_spec — The SCSI ID and unit number of the target device.

device_type — Device type of device expected to be registered for unit number and SCSI ID.

driver_handle_ptr — A pointer to where the device information is to be returned.

Description

Return the opaque driver handle that was registered with the device. This routine takes the place of **io_get_device_info** for SCSI devices that don't have DIT entries.

Return Values

OK — The opaque driver handle was successfully retrieved and returned.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — A device of the specified type is not registered at the SCSI ID and unit number slot.

dev_xxx_issue_command_physical_mode

Syntax

```
status_type dev_xxx_issue_command_physical_mode
                                                (request_blk_ptr)

dev_adapter_physical_request_blk_ptr_type request_blk_ptr;
                                                /*READ/WRITE*/
```

Summary

Issue a physical I/O request through the SCSI adapter to a target device.

Parameters

request_blk_ptr — A pointer to a request block that holds information that specifies the request. Note that this is a special version of the adapter request block and is not the same as the request block used during normal system operation.

Description

This routine is called to issue a synchronous I/O request over the SCSI bus without the use of the normal operating system facilities. Synchronization is done without the use of event counters or interrupts. All buffer addresses are assumed to be physical. The system is assumed to be running a single thread of control, so no lock management is required.

Return Values

OK — A synchronous request completed successfully or an asynchronous request was started.

DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION — The command completed with a Check Condition status, and sense information is available in the caller's sense buffer.

IO_EIO_HARD_IO_ERROR — The command completed with a check condition status, and the subsequent request sense command failed.

Other return values that you deem necessary. The receiving device drivers should be prepared to handle these return values.

End of Chapter

Chapter 5
Managing Your Adapter From Your Device Driver

In hardware, an adapter controls the devices attached to it. In software, however, the adapter driver routines are invoked by the device drivers. However, in order to keep the device driver code from being fixed to a particular adapter driver, device drivers interface to their adapter drivers through an *adapter manager* which multiplexes device driver calls to the correct adapter driver. The adapter manager consists of a standard set of adapter driver routines with the generic mnemonic **scsi_adapter**. The device driver identifies the target adapter by passing its adapter's device name or device number as parameters to the adapter manager function.

This chapter describes adapter manager routines that SCSI device drivers use to interface to their SCSI adapter drivers. It includes the following commands:

- **dev_scsi_adapter_configure**
- **dev_scsi_adapter_device_to_name**
- **dev_scsi_adapter_name_to_device**
- **dev_scsi_adapter_open_dump**
- **dev_scsi_adapter_register_requester**
- **dev_scsi_adapter_set_unit_options**
- **dev_scsi_adapter_deregister_requester**
- **dev_scsi_adapter_issue_command**
- **dev_scsi_adapter_issue_async_command**
- **dev_scsi_adapter_get_device_info**
- **dev_scsi_adapter_issue_command_physical_mode**

Constants and Data Structures

In general, the adapter manager routines use the same data structures described for user-supplied adapter drivers. See Chapter 4 for a discussion of these data structures.

dev_scsi_adapter_configure

Syntax

```
status_type dev_scsi_adapter_configure  (name_ptr)

char_ptr_type                  name_ptr;   /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by **name_ptr** and calls its configuration routine.

Parameters

name_ptr — Pointer to the SCSI adapter name as specified in the DG/UX system file.

Description

This routine invokes the proper adapter manager configuration routine.

Return Values

The return value will be whatever is returned by the adapter manager configure routine.

dev_scsi_adapter_device_to_name

Syntax

```
status_type dev_scsi_adapter_device_to_name  (device_number,
                                              name_ptr,
                                                size)

io_device_number_type   device_number; /*READ ONLY*/
char_ptr_type           name_ptr;      /*READ ONLY*/
uint32_type             size;          /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its device-to-name routine.

Parameters

device_number — The device number of the SCSI adapter for which the character string name is wanted.

name_ptr — A pointer to where the null-terminated character string name is to be written.

size — The maximum number of bytes, including the terminating null, that is to be written to **name_ptr**.

Description

This routine invokes the proper adapter manager device-to-name routine.

Return Values

The return value will be whatever is returned by the adapter manager device-to-name routine.

dev_scsi_adapter_name_to_device

Syntax

```
status_type dev_scsi_adapter_name_to_device (name_ptr,
                                             device_number_ptr)

char_ptr_type              name_ptr;          /*READ ONLY*/
io_device_number_ptr_type  device_number_ptr; /*WRITE ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by **name_ptr** and calls its name-to-device routine.

Parameters

name_ptr — Pointer to the SCSI adapter name as specified in the DG/UX system file.

device_number_ptr — Pointer to where the SCSI adapter major and minor number is to be returned.

Description

This routine invokes the proper adapter manager name-to-device routine.

Return Values

The return value will be whatever is returned by the adapter manager name-to-device routine.

dev_scsi_adapter_open_dump

Syntax

```
status_type  dev_scsi_adapter_open_dump  (name_ptr,
                                          major_device_ptr)

char_ptr_type                    name_ptr; /*READ ONLY*/
io_major_device_number_ptr_type major_device_ptr;
                                          /*WRITE ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by **name_ptr** and calls its open-dump routine.

Parameters

name_ptr — A pointer to the null-terminated character string identifying the adapter to be opened to allow access to a dump destination device.

major_device_ptr — A pointer to where the major device number of the driver that successfully opens the dump device is to be written.

Description

This routine invokes the proper adapter manager open-dump routine.

Return Values

The return value will be whatever is returned by the adapter manager open-dump routine.

dev_scsi_adapter_register_requester

Syntax

```
status_type dev_scsi_adapter_register_requester
                                                (major_number,
                                                rb_ptr)

io_major_device_number_type     major_number;/*READ ONLY*/
dev_scsi_adapter_unit_registration_blk_ptr_type  rb_ptr;
                                                 /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its register-requester routine.

Parameters

major_number — The major device number of the SCSI adapter device that the device driver is registering with.

rb_ptr — A pointer to an SCSI adapter registration block.

Description

This routine invokes the proper adapter manager register-requester routine.

Return Values

The return value will be whatever is returned by the adapter manager register-requester routine.

dev_scsi_adapter_set_unit_options

Syntax

```
status_type dev_scsi_adapter_set_unit_options (major_number,
                        apter_handle,unit_options_block_ptr)

io_major_device_number_type     major_number;     /*READ ONLY*/
io_device_handle_type           adapter_handle;   /*READ ONLY*/
dev_scsi_adapter_unit_options_block_ptr_type
                                unit_options_block_ptr;
                                                  /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its set-unit-options routine.

Parameters

major_number — The major device number of the SCSI adapter device used to reference the unit that is the target of the set-unit-options operation.

adapter_handle — The device handle of the physical unit the is the target of the set-unit-options operation. This handle must be the device handle that was returned by the register-requester routine of the adapter manager.

unit_options_block_ptr — Pointer to a unit options block that specifies the options to be selected for the unit.

Description

This routine invokes the proper adapter manager set-unit-options routine.

Return Values

The return value will be whatever is returned by the adapter manager set-unit-options routine.

dev_scsi_adapter_deregister_requester

Syntax

```
void dev_scsi_adapter_deregister_requester (major_number,
                                            adapter_handle)

io_major_device_number_type  major_number;    /*READ ONLY*/
io_device_handle_type        adapter_handle;  /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its deregister-requester routine.

Parameters

major_number — The major device number of the SCSI adapter device that the unit is registered with.

adapter_handle — The device handle of the physical unit that is to be deregistered. This handle must be the device handle that was returned by the register-requester routine of the adapter manager.

Description

This routine invokes the proper adapter manager deregister-requester routine.

Return Values

The return value will be whatever is returned by the adapter driver's deregister-requester routine.

dev_scsi_adapter_issue_command

Syntax

```
status_type  dev_scsi_adapter_issue_command
                                              (major_number,
                                               arb_ptr)

io_major_device_number_type  major_number;  /*READ ONLY*/
dev_adapter_request_block_ptr_type arb_ptr  /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its issue-command routine. The issue-command routine is the entry point used to perform synchronous I/O through the SCSI interface.

Parameters

major_number — The major device number of the SCSI adapter device used to reference the target device.

arb_ptr — A pointer to a generic adapter request block that holds all information that describes the request.

Description

This routine invokes the proper adapter manager issue-command routine.

Because of certain hardware restrictions, you may transfer only an even number of bytes when using this routine. In addition, the starting buffer address must be aligned on an even-byte boundary. Thus, the buffer may start on byte zero (0) or two (2) of a word, but not on bytes one (1) or three (3).

Return Values

The return value will be whatever is returned by the adapter manager issue-command routine.

dev_scsi_adapter_issue_async_command

Syntax

```
status_type  dev_scsi_adapter_issue_async_command
                                                (major_number,
                                                 arb_ptr)

io_major_device_number_type        major_number;/*READ ONLY*/
dev_adapter_request_block_ptr_type arb_ptr;     /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its issue-async-command routine. The issue-async-command routine is the entry point used to perform asynchronous I/O through the SCSI interface.

Parameters

major_number — The major device number of the SCSI adapter device used to reference the target device.

arb_ptr — A pointer to a generic adapter request block that holds all information that describes the request.

Description

This routine invokes the proper adapter manager issue-async-command routine.

Because of certain hardware restrictions, you may transfer only an even number of bytes when using this routine. In addition, the starting buffer address must be aligned on an even-byte boundary. Thus, the buffer may start on byte zero (0) or two (2) of a word, but not on bytes one (1) or three (3).

Return Values

The return value will be whatever is returned by the adapter manager issue-command routine.

dev_scsi_adapter_get_device_info

Syntax

```
status_type dev_scsi_adapter_get_device_info (name_ptr,
                                              unit_spec,
                                              device_type,
                                              driver_handle_ptr)

char_ptr_type                    name_ptr;       /*READ ONLY*/
dev_scsi_adapter_unit_spec_type  unit_spec;      /*READ ONLY*/
bit8_type                        device_type;    /*READ ONLY*/
io_device_handle_ptr_type    driver_handle_ptr;/*WRITE ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device name and calls its get-device-info routine.

Parameters

name_ptr — A pointer to the target SCSI adapter's name, as specified in the DG/UX system file.

unit_spec — The SCSI ID and unit number of the target device.

device_type — Device type of device expected to be registered for unit number and SCSI ID.

driver_handle_ptr — A pointer to where the device information is to be returned.

Description

This routine invokes the proper adapter manager get-device-info routine.

Return Values

The return value will be whatever is returned by the adapter manager get-device-info routine.

dev_scsi_adapter_issue_command_physical_mode

Syntax

```
status_type  dev_scsi_adapter_issue_command_physical_mode
                                                (major_number,
                                                request_blk_ptr)

io_major_device_number_type     ajor_number;   /*READ ONLY*/
dev_adapter_physical_request_blk_ptr_type request_blk_ptr;
                                                /*READ ONLY*/
```

Summary

This routine locates the SCSI adapter manager specified by the device number and calls its issue-command-physical-mode routine.

Parameters

major_number — The major device number of the SCSI adapter device used to reference the target of the request.

request_blk_ptr — A pointer to a request block that holds information which specifies the request.

Description

This routine invokes the proper adapter manager issue-command-physical-mode routine.

Because of certain hardware restrictions, you may transfer only an even number of bytes when using this routine. In addition, the starting buffer address must be aligned on an even-byte boundary. Thus, the buffer may start on byte zero (0) or two (2) of a word, but not on bytes one (1) or three (3).

Return Values

The return value will be whatever is returned by the adapter manager issue-command-physical-mode routine.

End of Chapter

Chapter 6
Process Synchronization and Timing

This chapter describes all DG/UX kernel routines used in process management and timing. Included are routines that handle eventcounters, signals, and clock operations. Also included in this chapter are routines for implementing locks on critical sections of data.

This chapter is divided into five major sections, as follows:

- **Synchronization Routines** — Routines used to synchronize processes using eventcounters.
- **Process Signal Management Routines** — Routines used to process signals.
- **Lock Management Routines** — Routines used to protect critical sections of data.
- **Clock Routines** — Routines used to manage the system clock.
- **Interrupt Handling Routines** — Routines used in handling interrupts.

Each section introduces the major features of the routines that follow. Following each introduction is a "Constants and Data Structures" section, which lists some of the constants and data structures used by the routines. For a full list of constants and data structures, see the include files listed in Chapter 3.

Synchronization Routines

The routines in this section are used to manipulate eventcounters. Eventcounters are used as synchronization primitives. The main synchronization operations performed are await and advance. For more information on await and advance, sequencers, and eventcounters, see the *Communications of the ACM* papers listed in the preface, in the section called "Other Documents."

Await allows a process to wait for any of several events to be satisfied. Here an event refers to an eventcounter and an eventcounter value. The event is said to be satisfied when the value of the eventcounter is greater than or equal to the awaited value. If the await call is made, and one or more of the specified events is already satisfied, the process continues execution following the call to await. If none of the specified events is satisfied, the process enters the awaiting state where it does not compete for CPU resources.

The advance operation increments the value of the specified eventcounter and then checks to see whether the new value of the incremented eventcounter causes any events to be satisfied. If the process associated with a satisfied event is still in the awaiting state, it is scheduled to run.

Sequencers are provided to extend the functionality of eventcounters. Sequencer routines allow a caller to allocate unique eventcounter values for use in constructing events.

When one of the events occurs, a process awaiting multiple events is returned an index into the event list submitted to **vp_await_ec**. The index identifies the event in the list that caused the await to be satisfied. However, the event specified by the index is not necessarily the only event that has occurred in the list. A process may determine which events have occurred by calling the routine **vp_has_event_occurred** for each entry in the event list.

A process doing a **vp_await_ec** that can pend indefinitely (such as waiting for terminal input) should not hold any locks. Doing so will inadvertently tie up a virtual processor (VP) the entire time the process is waiting.

If you use routines from this section, you must allocate the space used by the event and eventcounter instances (see the "Constants and Data Structures" section below). Eventcounters are normally allocated from global memory. Event types are allocated dynamically, as needed.

The following routines are described in this section:

- **vp_add_to_ec_value**
- **vp_advance_ec**
- **vp_await_ec**

- **vp_convert_clock_value_to_ec_value**
- **vp_convert_ec_value_to_clock_value**
- **vp_get_next_ec_value**
- **vp_has_event_occurred**
- **vp_increment_ec_value**
- **vp_initialize_ec**
- **vp_initialize_sequencer**
- **vp_read_ec**
- **vp_ticket_sequencer**
- **vp_are_ec_values_equal**

Routines beginning with **vp** require the **i_vp.h** include file.

Constants and Data Structures

This section discusses some of the data structures used by synchronization routines. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_vp.h** for structures beginning with the **vp** acronym). Chapter 3 lists the various include files.

vp_event_type

```
typedef struct
   [
   vp_ec_ptr_type      name;
   vp_ec_value_type    value;
   }
   vp_event_type ;
```

This structure defines an event, which is an eventcounter name and an eventcounter value. The event is said to occur or to be satisfied when the value of the eventcounter pointed to by the `name` field is greater than or equal to the `value` field.

vp_add_to_ec_value

Syntax

```
void   vp_add_to_ec_value  (ec_value_ptr, addend)

vp_ec_value_ptr_type  ec_value_ptr;   /*READ/WRITE*/
uint32_type           addend;         /*READ ONLY*/
```

Summary

This routine adds the given value to the specified eventcounter value.

Parameters

ec_value_ptr — A pointer to the eventcounter value to be added to.

addend — The value to be added to the eventcounter value.

Description

The specified 32-bit integer is added to the specified eventcounter value.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

vp_advance_ec

Syntax

```
void    vp_advance_ec  (ec_name)

vp_ec_ptr_type     ec_name;            /*READ ONLY*/
```

Summary

This routine performs an advance (by one) on the specified eventcounter. Any processes awaiting on the new value of the eventcounter will be notified.

Parameters

ec_name — A pointer to the eventcounter to be advanced.

Description

The eventcounter is indivisibly incremented, and any processes awaiting on the new value are notified. If a higher priority process becomes eligible to run as a result of the notification, it may be rescheduled. Thus, your process may be pre-empted if you call this routine.

Return Values

None.

Exceptions

None.

vp_await_ec

Syntax

```
void vp_await_ec (event_list, list_size, list_index_ptr)

vp_event_type   event_list[];      /*READ ONLY*/
int32_type      list_size;         /*READ ONLY*/
int32_ptr_type  list_index_ptr;    /*WRITE ONLY*/
```

Summary

This routine performs the await operation on one or more events. The calling process will be suspended until at least one of the specified events is satisfied.

Parameters

event_list — An array of events for which the process wishes to await.

list_size — The number of elements in **event_list**.

list_index_ptr — A pointer to the array index (zero based) of an event that is satisfied when the call returns.

Description

This routine causes the calling process to be suspended until any one of the supplied events has been satisfied. If any of the events is satisfied at the time the call is made, the process is not suspended. When the call returns, the **list_index_ptr** is set to the index of an event that is satisfied, but if more than one event is satisfied, no statement is made about which event will be indicated by **list_index_ptr**.

Return Values

None.

Exceptions

None.

vp_convert_clock_value_to_ec_value

Syntax

```
void vp_convert_clock_value_to_ec_value (clock_value_ptr,
                                         ec_value_ptr)

misc_clock_value_ptr_type  clock_value_ptr; /*READ ONLY*/
vp_ec_value_ptr_type       ec_value_ptr;    /*WRITE ONLY*/
```

Summary

This routine converts a clock value into an eventcounter value.

Parameters

clock_value_ptr — A pointer to a clock value.

ec_value_ptr — A pointer to the location where the corresponding eventcounter value is to be written.

Description

This routine converts a clock value into an eventcounter value. Converting from clock value to eventcounter value requires converting the 64-bit clock value to a 32-bit eventcounter value.

The number of bits to take from the high and low word of the clock value are defined in **i_vp.h** as **VP_CLOCK_TO_EC_HIGH_BITS** and **VP_CLOCK_TO_EC_LOW_BITS**.

Return Values

None.

Exceptions

None.

vp_convert_ec_value_to_clock_value

Syntax

```
void vp_convert_ec_value_to_clock_value (ec_value_ptr,
                                         clock_value_ptr)

vp_ec_value_ptr_type       ec_value_ptr;    /*READ ONLY*/
misc_clock_value_ptr_type  clock_value_ptr; /*WRITE ONLY*/
```

Summary

This routine converts an eventcounter value into a clock value.

Parameters

ec_value_ptr — A pointer to an eventcounter value.

clock_value_ptr — A pointer to the location where the corresponding clock value is to be written.

Description

This routine converts an eventcounter value into a clock value. Conversion from eventcounter value to clock value requires converting a 32-bit eventcounter value to a 64-bit clock value.

The number of bits to assign to the high and low word of the clock value are defined in **i_vp.h** as **VP_CLOCK_TO_EC_HIGH_BITS** and **VP_CLOCK_TO_EC_LOW_BITS**.

Return Values

None.

Exceptions

None.

vp_get_next_ec_value

Syntax

```
void vp_get_next_ec_value (ec_name, ec_value_ptr)

vp_ec_ptr_type  ec_name;              /*READ ONLY*/
vp_ec_value_ptr_type  ec_value_ptr;   /*WRITE ONLY*/
```

Summary

This routine indivisibly reads the specified eventcounter and returns its value plus one.

Parameters

ec_name — A pointer to the eventcounter to be read.

ec_value_ptr — A pointer to the location where the eventcounter value (plus one) is to be written.

Description

The eventcounter is read indivisibly with respect to other processors and with respect to the executing processor's interrupt level. The value is then incremented by one, which is equal to the value that will be reached the next time the eventcounter is advanced.

Return Values

None.

Exceptions

None.

vp_has_event_occurred

Syntax

```
boolean_type  vp_has_event_occurred (event_ptr)

vp_event_ptr_type   event_ptr;          /*READ ONLY*/
```

Summary

This routine determines whether the given event has occurred.

Parameters

event_ptr — A pointer to the subject event.

Return Values

TRUE — The event has been satisfied.

FALSE — The event has not yet occurred.

Exceptions

None.

vp_increment_ec_value

Syntax

```
void     vp_increment_ec_value  (ec_value_ptr)

vp_ec_value_ptr_type ec_value_ptr;     /*READ WRITE*/
```

Summary

This routine increments the specified eventcounter value.

Parameters

ec_value_ptr — A pointer to the eventcounter value to be incremented.

Description

This routine simply takes the eventcounter value passed in and increments it.

Return Values

None.

Exceptions

None.

vp_initialize_ec

Syntax

```
void     vp_initialize_ec  (ec_name)

vp_ec_ptr_type     ec_name;            /*READ ONLY*/
```

Summary

This routine initializes an eventcounter.

Parameters

ec_name — A pointer to the eventcounter to be initialized.

Description

The eventcounter value is set to zero.

Return Values

None.

Exceptions

None.

vp_initialize_sequencer

Syntax

```
void            vp_initialize_sequencer  (seq_name)

vp_ec_ptr_type     seq_name;     /*READ ONLY*/
```

Summary

This routine initializes a sequencer.

Parameters

seq_name — A pointer to the sequencer to be initialized.

Description

The sequencer value is set to zero.

Return Values

None.

Exceptions

None.

vp_read_ec

Syntax

```
void              vp_read_ec  (ec_name, ec_value_ptr)

vp_ec_ptr_type         ec_name;          /*READ ONLY*/
vp_ec_value_ptr_type   ec_value_ptr;     /*WRITE ONLY*/
```

Summary

This routine indivisibly reads the specified eventcounter and returns the value in the variable pointed to by **ec_value_ptr**.

Parameters

ec_name — A pointer to the eventcounter to be read.

ec_value_ptr — A pointer to the location in which the eventcounter value is to be written.

Description

The eventcounter is read indivisibly with respect to other processors and with respect to the executing processor's interrupt level.

Return Values

None.

Exceptions

None.

vp_ticket_sequencer

Syntax

```
void     vp_ticket_sequencer  (seq_name, seq_value_ptr)

vp_ec_ptr_type        seq_name;        /*READ ONLY*/
vp_ec_value_ptr_type  seq_value_ptr;   /*WRITE ONLY*/
```

Summary

This routine indivisibly increments the value of the specified sequencer and returns the new value (that is, the value after the increment).

Parameters

seq_name — A pointer to the sequencer to be ticketed.

seq_value_ptr — A pointer to the location in which the new value of the sequencer is to be written.

Description

The sequencer value is incremented and then read as an indivisible operation.

Return Values

None.

Exceptions

None.

vp_are_ec_values_equal

Syntax

```
boolean_type  vp_are_ec_values_equal (value1_ptr, value2_ptr)

vp_ec_value_ptr_type  value1_ptr;    /*READ ONLY*/
vp_ec_value_ptr_type  value2_ptr;    /*READ ONLY*/
```

Summary

This routine compares two eventcounter values for equality.

Parameters

value1_ptr — A pointer to an eventcounter value.

value2_ptr — A pointer to an eventcounter value.

Description

This routine compares two eventcounter values and returns TRUE if they are equal.

Return Values

TRUE — The eventcounter values are equal.

FALSE — The eventcounter values are not equal.

Exceptions

None.

Process Signal Management Routines

The routines in this section are used by a process to send and receive signals. The routines **pm_is_interrupted** and **pm_is_terminated** notify the caller when process signals are received. **pm_is_interrupted** reports all signals to the caller, and **pm_is_terminated** reports only signals that will cause process termination.

The routines provided for signal delivery allow signals to be selectively sent based on the process index, process ID, or process group of the target process.

The routines described in this section are as follows:

- **pm_get_my_pid**
- **pm_get_my_pgrp**
- **pm_is_interrupted**
- **pm_is_terminated**
- **pm_send_signal_by_index**
- **pm_send_signal_by_process_group**
- **pm_send_signal_by_process_id**

Routines beginning with **pm** require the **i_pm.h** include file.

Constants and Data Structures

No special constants or data structures are required by these routines.

pm_get_my_pid

Syntax

```
pm_process_id_type  pm_get_my_pid ()
```

Summary

Returns the process id of the "calling" process.

Parameters

None.

Description

See Summary.

Return Values

The current pid.

pm_get_my_pgrp

Syntax

```
pm_process_id_type  pm_get_my_pgrp ()
```

Summary

Returns the process group of the "calling" process.

Parameters

None.

Description

See Summary.

Return Values

The process group

pm_is_interrupted

Syntax

```
boolean_type pm_is_interrupted (event_ptr)

vp_event_ptr_type  event_ptr;          /*WRITE ONLY*/
```

Summary

This routine handles signals during a system call.

Parameters

event_ptr — The address of a process interrupt event.

Description

This routine handles signal processing. It should be used whenever a system call will pend the calling process until some external event occurs (that is, pend for an arbitrary amount of time). Processing includes the following:

- Interrupting the system call.
- Terminating the process (with or without a core dump).
- Stopping the process for an arbitrary amount of time.

Only the last of these actions is contained entirely within the **pm_is_interrupted** routine. The first two actions are performed in cooperation with the caller.

Typically, you will use the following code fragment:

```
if (pm_is_interrupted(&events[PROCESS_INTERRUPT]))
{
Arrange to return EINTR to the user.  Exit with error EINTR.
}
vp_await_ec(events, N, &index);
Act on the event that was satisfied.
If only the PROCESS_INTERRUPT was satisfied, loop back to
pm_is_interrupted()
```

In the code shown above, the relevant events are those in the `events[]` array in the first line. In addition, the event returned by **pm_is_interrupted** is also important. If the calling process is interrupted, the system call will

return an error and will set errno to EINTR. Otherwise, the system call pends until the calling process is interrupted or one of the relevant events has happened.

Return Values

TRUE — A signal is presented to be handled.

FALSE — No signal is present.

[*event*] — **event_ptr** is set to an event that will occur when it is appropriate to check for signals again.

Exceptions

None.

pm_is_terminated

Syntax

```
boolean_type pm_is_terminated (event_ptr)

vp_event_ptr_type  event_ptr;          /*WRITE ONLY*/
```

Summary

This routine checks for termination signals during a system call.

Parameters

event_ptr — The address of a process interrupt event.

Description

This routine determines whether the calling process has any signals that will cause process termination.

Return Values

TRUE — A signal is presented to be handled.

FALSE — No signal is present. **event_ptr** is set to an event that will occur when it is appropriate to re-check for termination signals.

Exceptions

None.

Remarks

This call is designed for use within the kernel when a potentially long but nevertheless finite operation is started. For example, a spacing operation on a tape drive or an I/O request to an NFS server is essentially indefinite. In both of these cases, the operation is guaranteed to eventually finish, perhaps due to a timeout; but the end user may like the option of terminating the operation mid-stream by sending the process a signal.

pm_send_signal_by_index

Syntax

```
void pm_send_signal_by_index (index, signal, signal_source)

sc_process_index_type          index;          /*READ ONLY*/
pm_signal_type                 signal;         /*READ ONLY*/
pm_signal_source_enum_type     signal_source;  /*READ ONLY*/
```

Summary

If the subject process exists, this routine sends the process a signal. If the subject process does not exist, this routine has no effect.

Parameters

index — The subject process' index. The index is a unique identifier assigned to each process. It is maintained in per process data and is contained in the variable **sc_my_process_index**.

signal — The signal to send.

signal_source — The reason the signal is being sent.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

pm_send_signal_by_process_group

Syntax

```
status_type pm_send_signal_by_process_group (process_group,
                                             signal_number,
                                             signal_source)

pm_process_id_type            process_group;  /*READ ONLY*/
pm_signal_type                signal_number;  /*READ ONLY*/
pm_signal_source_enum_type    signal_source;  /*READ ONLY*/
```

Summary

This routine sends a signal to a process group.

Parameters

process_group — The process group ID of the target process.

signal_number — The signal being sent.

signal_source — The reason the signal is being sent.

Description

Send the signal **signal_number** to the processes whose process group ID is **process_group**. The signal is sent only to processes that are not system processes and to which the calling process has permission to send a signal.

Return Values

The following values may be returned:

PM_ESRCH_NO_SUCH_PROCESS_GROUP — No process corresponding to those specified by **process_group** can be found.

PM_ESRCH_NO_PERMISSION — The calling process does not have permission to signal the processes identified by **process_group**.

pm_send_signal_by_process_id

Syntax

```
status_type pm_send_signal_by_process_id (process_id,
                                          signal_number,
                                          signal_source)

pm_process_id_type            process_id;     /*READ ONLY*/
pm_signal_type                signal_number;  /*READ ONLY*/
pm_signal_source_enum_type    signal_source;  /*READ ONLY*/
```

Summary

This routine sends a signal to a process identified by **process_id**.

Parameters

process_id — The process ID of the target.

signal_number — The signal being sent.

signal_source — The reason the signal is being sent.

Description

Send the signal **signal_number** to the process identified by **process_id**. If **signal_number** is **PM_SIGNAL_SIGKILL**, **pm_send_signal_by_process_id** assumes that **process_id** does not identify a system process.

Return Values

The following values may be returned:

PM_ESRCH_NO_SUCH_PROCESS_ID — No process corresponding to that specified by **process_id** can be found.

PM_EPERM_NO_KILL_ACCESS — The sending process does not have permission to signal the receiving process.

Lock Management Routines

The kernel lock facilities are used to protect critical sections of code. These facilities synchronize code paths and data structures. Because the DG/UX system runs in a multiprocessor environment, you may not use interrupt disable to protect critical sections. The kernel provides three types of locks: sequenced locks, unsequenced locks, and spin locks.

Unsequenced locks provide no ordering of requesters. They require less space than sequenced locks, and the obtain and release operations are faster than for sequenced locks. Unsequenced locks, however, do not perform well under tight contention, because they can cause a cascade of rescheduling. Each time an unsequenced lock is released, all processes waiting for the lock are awakened. One process will get the lock and all others will go back to sleep. If *n* processes are contending for the lock, the first time the lock is released (*n-1*) processes will be rescheduled; the next time the lock is released (*n-2*) will be rescheduled, and so forth. A total of $n(n+1)/2$ reschedulings will occur for every *n* contentions.

Sequenced locks grant access on a first-come-first-serve basis. They avoid the scheduling overhead by ordering contending processes based on when they first tried to obtain the lock. When the lock is released, only the next process in line is awakened.

Spin locks are simple locks that cause the caller to loop if the lock cannot be obtained immediately. They should be used only in a very restricted environment. All code and data you reference while holding a spin lock must be wired. This is because a page fault could cause the lock to be held for a long period of time. This situation could deadlock the system depending on what other processes try to get the lock. Also, any process holding a spin lock must not lose the processor on which it is running. Finally, the caller must ensure that interrupts are disabled while a spin lock is held.

The user of these routines is responsible for allocating the space used by the lock instances. A lock may be created by declaring an instance of type **lm_sequenced_lock_type**, **lm_unsequenced_lock_type**, or **misc_spin_lock_type**.

The routines described in this section are as follows:

- **lm_initialize_sequenced_lock**
- **lm_initialize_unsequenced_lock**
- **lm_obtain_sequenced_lock**
- **lm_obtain_sequenced_lock_no_wait**
- **lm_obtain_unsequenced_lock**

- **lm_release_sequenced_lock**
- **lm_release_unsequenced_lock**
- **misc_obtain_spin_lock**
- **misc_release_spin_lock**

Routines beginning with **lm** and **misc** require the **i_lm.h** and **i_misc.h** include files, respectively.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_lm.h** for structures beginning with the **lm** acronym). Chapter 3 lists the various include files.

lm_sequenced_lock_type

```
typedef struct
    {
    lm_resource_counter_type    rc;

    }  lm_sequenced_lock_type  ;
```

Description

This type is a sequenced lock. A sequenced lock may be created by simply declaring an instance of this type. The user of the lock is responsible for allocating the space occupied by the lock instance and reclaiming that space when the lock is destroyed.

A sequenced lock is simply a resource counter that has an initial value of one.

lm_unsequenced_lock_type

```
typedef struct
    {
```

```
    vp_unsequenced_lock_type      lock;

    }  lm_unsequenced_lock_type  ;
```

Description

This type is an unsequenced lock. An unsequenced lock may be created by simply declaring an instance of this type. The user of the lock is responsible for allocating the space occupied by the lock instance and reclaiming that space when the lock is destroyed.

misc_spin_lock_type

```
typedef bit32e_type       misc_spin_lock_type ;
```

Description

This type defines a spin lock. The spin lock actually uses only the low bit of the 32. The lock is considered held when the low order bit is 1, and is considered not held otherwise.

lm_initialize_sequenced_lock

Syntax

```
void lm_initialize_sequenced_lock (lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr;       /*WRITE ONLY*/
```

Summary

This routine initializes a sequenced lock.

Parameters

lock_ptr — A pointer to the lock to be initialized.

Description

This routine initializes a sequenced lock. None of the obtain or release operations should be performed on a lock until it has been initialized by this routine.

Return Values

None.

Exceptions

None.

lm_initialize_unsequenced_lock

Syntax

```
void lm_initialize_unsequenced_lock (lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr;    /*WRITE ONLY*/
```

Summary

This routine initializes an unsequenced lock.

Parameters

lock_ptr — A pointer to the lock to be initialized.

Description

This routine initializes an unsequenced lock. None of the obtain or release operations should be performed on a lock until it has been initialized by this routine.

Return Values

None.

Exceptions

None.

lm_obtain_sequenced_lock

Syntax

```
void  lm_obtain_sequenced_lock(lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr;       /*WRITE ONLY*/
```

Summary

This routine obtains the specified lock.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Return Values

None.

Exceptions

None.

lm_obtain_sequenced_lock_no_wait

Syntax

```
void   lm_obtain_sequenced_lock_no_wait (lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr; /*read/write*/
```

Summary

This routine obtains the specified lock. The calling process is not pended if the lock is not immediately available. A boolean is returned, which indicates whether the lock was obtained.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Description

See Summary.

Return Values

TRUE — The lock was obtained.

FALSE — The lock was not obtained.

Exceptions

None.

lm_obtain_unsequenced_lock

Syntax

```
void lm_obtain_unsequenced_lock(lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr;     /*WRITE ONLY*/
```

Summary

This routine obtains the specified lock.

NOTE: The calling process will be pended if the lock is not immediately available.

Parameters

lock_ptr — A pointer to the lock to be obtained.

Return Values

None.

Exceptions

None.

lm_release_sequenced_lock

Syntax

```
void  lm_release_sequenced_lock(lock_ptr)

lm_sequenced_lock_ptr_type  lock_ptr;        /*WRITE ONLY*/
```

Summary

This routine releases the specified lock. If other processes are waiting for the lock to become available, the next one in sequence will be awakened.

Parameters

lock_ptr — A pointer to the lock that is to be released.

Return Values

None.

Exceptions

None.

lm_release_unsequenced_lock

Syntax

```
void  lm_release_unsequenced_lock(lock_ptr)

lm_unsequenced_lock_ptr_type  lock_ptr;       /*WRITE ONLY*/
```

Summary

This routine releases the specified lock. If other processes are waiting for the lock to become available, all waiting processes will be awakened and one will be given the lock.

Parameters

lock_ptr — A pointer to the lock that is to be released.

Return Values

None.

Exceptions

None.

misc_obtain_spin_lock

Syntax

```
void    misc_obtain_spin_lock     (lock_ptr)

misc_spin_lock_ptr_type lock_ptr; /*READ/WRITE*/
```

Summary

This routine obtains a spin lock. If the lock is not immediately available, the process will loop until it becomes available.

Parameters

lock_ptr — A pointer to the spin lock that is to be obtained. **misc_obtain_spin_lock** assumes that **lock_ptr** is a word pointer to a word-aligned structure.

Description

An attempt is made to obtain the lock. If the lock is already held, the code loops until the lock is obtained. Spin locks are the only locks that can be obtained at interrupt level.

Return Values

None.

Exceptions

None.

misc_release_spin_lock

Syntax

```
void    misc_release_spin_lock    (lock_ptr)

misc_spin_lock_ptr_type lock_ptr; /*READ/WRITE*/
```

Summary

This routine releases a spin lock.

Parameters

lock_ptr — A pointer to the spin lock that is to be released.

Semantics

The high-order bit of the lock word is cleared.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

Clock Routines

The system clock is a 64-bit logical counter that increments at a fixed rate in real time. The counter is given value zero at system boot time. System clock values are continuous and monotonically increasing. Continuous means that the value of the system clock is not changed even if the external time-of-day is changed. Therefore, you can use the system clock to do interval timing without having to worry about its value changing during the interval.

External time-of-day is computed as an offset relative to the system clock. The offset is set initially when the system is booted. The current time-of-day is obtained by reading the system clock and adding the offset. If the time-of-day is changed while the system is running, only the offset is changed. Because time-of-day is calculated from an offset, interval calculations (such as process run time) based on the system clock will remain valid even if time-of-day is changed. However, if the time-of-day is changed while the system is running, externally visible time-stamps set by the system (such as the time-last-modified on a file) may be anomalous.

The DG/UX system provides timeout services for doing asynchronous processing. Routines are provided to both establish and cancel timeouts. Timeouts are identified by a timeout ID that is returned by the establish call. You must supply this ID in order to cancel the timeout. You must cancel a timeout when it is no longer needed, regardless of whether or not it has started.

If you are doing synchronous processing, you will connect to the clock via a clock event. The **vp_create_clock_event** will allow you to establish a clock event. See the "Synchronization Routines" section of this chapter for other routines you will need to service the clock event you establish.

The routines described in this section are as follows:

- **vp_establish_timeout**
- **vp_cancel_timeout**
- **vp_specify_max_timeouts**
- **vp_create_clock_event**
- **vp_read_system_clock**

Routines beginning with **vp** require the **i_vp.h** include file.

Constants and Data Structures

This section describes the format of system clock values and the general clock value constants that may be needed by other subsystems. These constants are allocated in global memory, and the data types are defined in **i_misc.h**. Pointers to the constants are passed to the clock management routines to specify time values. Generally useful values are defined in this section; if a subsystem has a need for a special clock value, it may define the value itself.

Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_misc.h** for structures beginning with the **misc** acronym). Chapter 3 lists the various include files.

misc_clock_value_type

```
typedef struct
    {
    uint32e_type            high;
    uint32e_type            low;
    }
    misc_clock_value_type
```

Description

This type describes a value that the system clock can have. The clock value is treated as a 64-bit signed integer with time values contained in the bottom 63 bits. (The bits are numbered such that bit 63 is the most-significant bit, and bit 0 is the least significant bit with bit 32 representing one second.) Actual resolution of timing may vary but will be accurate to at least 10 milliseconds.

You may use the following defined constants in your driver. They are defined in **i_misc.h**.

misc_five_minutes
misc_one_hundred_seconds
misc_one_minute
misc_ten_seconds
misc_five_seconds
misc_three_seconds
misc_two_seconds
misc_one_second
misc_one_half_second

misc_two_hundred_fifty_milliseconds
misc_two_hundred_milliseconds
misc_ten_milliseconds

vp_establish_timeout

Syntax

```
opaque32_type vp_establish_timeout (time_ptr, routine_ptr,
                                    argument)
misc_clock_value_ptr_type    time_ptr;       /*READ ONLY*/
vp_timeout_routine_ptr_type  routine_ptr;    /*READ ONLY*/
bit32e_type                  argument;       /*READ ONLY*/
```

Summary

This routine establishes a timeout. The timeout will occur **time_ptr** time from the current time, and then the specified routine will be called with the specified argument.

Parameters

time_ptr — A pointer to a clock value indicating the amount of real time that is to elapse before the timeout occurs. Use the clock constants in the "Constants and Data Structures" section for increment values.

routine_ptr — A pointer to a routine that is to be called by the I/O daemon when the timeout occurs.

argument — A 32-bit value that is to be passed to the timeout routine as an argument.

Return Values

timeout_id — The return value is an opaque 32-bit identifier for the timeout. This value may be used only as an argument to **vp_cancel_timeout**.

Exceptions

None.

vp_cancel_timeout

Syntax

```
void    vp_cancel_timeout   (timeout_id)

opaque32_type  timeout_id;  /*READ ONLY*/
```

Summary

This routine cancels a previously established timeout.

Parameters

timeout_id — The **timeout_id** of the timeout to be cancelled. This value was returned by the **vp_establish_timeout** routine.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

vp_specify_max_timeouts

Syntax

```
void          vp_specify_max_timeouts  (count)

uint32_type   count;      /*READ ONLY*/
```

Summary

This routine reserves space for the specified number of timeouts. A device driver should call it to reserve space for the maximum number of timeouts it will ever have in effect simultaneously.

Parameters

count — The number of timeouts for which to reserve space.

Description

Space is reserved for the specified number of timeouts. The space must be reserved before any timeouts are established. This routine will presumably be called several times, once by each driver in the system, as part of its initialization.

The amount of space reserved for timeouts cannot be reduced. Therefore you should try not to ask for more space than you will need during the life of the system.

Return Values

None.

Exceptions

None.

vp_create_clock_event

Syntax

```
void    vp_create_clock_event  (event_ptr, increment_ptr)

vp_event_ptr_type          event_ptr;      /*WRITE ONLY*/
misc_clock_value_ptr_type  increment_ptr;  /*READ ONLY*/
```

Summary

This routine sets up a clock event for a specified (**increment_ptr**) time in the future.

Parameters

event_ptr — A pointer to the event that is to be set up.

increment_ptr — A pointer to a clock value that is to be added to the current system time. Use the clock constants in the "Constants and Data Structures" section for increment values.

Description

event_ptr is set to an event. The value of the eventcounter is set to make the event occur at current time plus the increment. See the "Synchronization Routines" section of this chapter for other routines used in servicing the event. For example, you may want to use **vp_await_ec** to await the occurrence of this event. You do this by specifying the event in **vp_await_ec**'s event list.

Return Values

None.

Exceptions

None.

vp_read_system_clock

Syntax

```
void    vp_read_system_clock  (current_time_ptr)

misc_clock_value_ptr_type  current_time_ptr;   /*READ ONLY*/
```

Summary

The current value of the system clock is returned.

Parameters

current_time_ptr — A pointer to where the current value of the system clock is to be written.

Return Values

None.

Exceptions

None.

Interrupt Handling Routines

This section describes routines you use to handle interrupts, including the masking, enabling, disabling, and servicing interrupts.

In order to make efficient use of the multiprocessor environment, the masking routines perform stacking of mask requests. This means that the kernel maintains a count for each mask bit, and the actual hardware mask is changed only on transitions of the count between zero and one.

The routines described in this section are as follows:

- **io_mask_interrupt_variety**
- **io_unmask_interrupt_variety**
- **vp_are_interrupts_disabled**
- **vp_disable_interrupts**
- **vp_enable_interrupts**

Routines beginning with **vp** and **io** require the **i_vp.h** and **i_io.h** include files, respectively.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, because these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_uc.h** for structures beginning with the **uc** acronym). Chapter 3 lists the various include files.

uc_interrupt_enum_type

```
typedef enum
    {
    Uc_System_Alarm_Clock_Interrupt =    0,
    Uc_Keyboard_Interrupt =              1,
    Uc_Parallel_Port_Interrupt =         2,
    Uc_Ethernet_Interrupt =              3,
```

```
Uc_SCSI_Interrupt =                    4,
Uc_Duart_Interrupt=                    5,
Uc_Graphics_Device_Interrupt =         6,
Uc_Level_1_VME_Interrupt =             7,
Uc_Level_2_VME_Interrupt =             8,
Uc_Level_3_VME_Interrupt =             9,
Uc_Level_4_VME_Interrupt =            10,
Uc_Level_5_VME_Interrupt =            11,
Uc_Level_6_VME_Interrupt =            12,
Uc_Level_7_VME_Interrupt =            13,
Uc_Dma_Terminal_Count_Interrupt =     14,
Uc_System_Console_Interrupt =         15,
Uc_Interrupt_Enum_Last =              16,

}     uc_interrupt_enum_type  ;
```

Description

This type is used to describe the type (or variety) of interrupt to be masked or unmasked. Note that this type does not define all interrupts, but only those for which non-standard drivers may be written.

io_mask_interrupt_variety

Syntax

```
void  io_mask_interrupt_variety (interrupt_variety)

uc_interrupt_enum_type interrupt_variety; /*READ ONLY*/
```

Summary

This routine masks a variety of interrupt specified in **interrupt_variety**.

Parameters

interrupt_variety — The type of interrupt to be masked. Any device that uses the interrupt variety to interrupt the system is effectively masked.

Description

This routine masks interrupts for a device with the interrupt type given in **interrupt_variety**. Any devices that use the interrupt variety to interrupt the system are effectively masked. If there are multiple processors, the interrupt is disabled for all processors. It also nests mask and unmask requests.

The routine uses a mask depth associated with the specified device to nest interrupts. This routine increments the mask depth, and if the new value is one, the hardware is updated to reflect a change in the mask.

You may call this routine from base level or from interrupt level. It remembers and correctly restores the state of the interrupt enable flag.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

IO_PANIC_ILLEGAL_MASK_INTERRUPT — Either the mask depth associated with the specified device has become larger than it should, or the interrupt variety is illegal. The former must be due to incorrect pairing of the mask and unmask functions by the caller.

io_unmask_interrupt_variety

Syntax

```
void io_unmask_interrupt_variety (interrupt_variety)

uc_interrupt_enum_type  interrupt_variety; /*READ ONLY*/
```

Summary

This routine unmasks a variety of interrupt specified in **interrupt_variety**.

Parameters

interrupt_variety — The type of interrupt to be unmasked. Any device that uses this interrupt variety is effectively unmasked.

Description

This routine unmasks interrupts for a device with the interrupt type given in **interrupt_variety**. Any devices that use the interrupt variety to interrupt the system are effectively unmasked. If there are multiple processors, the interrupt is enabled for all processors. The routine nests mask and unmask requests.

The routine uses a mask depth associated with the specified device to nest interrupts. This routine decrements the mask depth, and if the new value is 0, the hardware is updated to reflect a change in the mask.

You may call this routine from base level or from interrupt level. It remembers and correctly restores the state of the interrupt enable flag.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

IO_PANIC_ILLEGAL_UNMASK_INTERRUPT — Either the device's mask depth is equal to zero, or the interrupt variety is illegal. The former must be due to incorrect pairing of the mask and unmask function calls.

vp_are_interrupts_disabled

Syntax

```
bool    vp_are_interrupts_disabled  ()
```

Summary

This routine returns TRUE if interrupts are disabled in the calling processor.

Parameters

None.

Return Values

TRUE — Interrupts are disabled in the calling processor.

FALSE — Interrupts are enabled in the calling processor.

Exceptions

None.

vp_disable_interrupts

Syntax

```
void    vp_disable_interrupts  ()
```

Summary

This routine disables interrupts in the calling processor.

Parameters

None.

Description

Interrupts are disabled and the interrupt disable depth count is decremented. An interrupt disable depth count is maintained so that calls to this routine and **vp_enable_interrupts** will nest properly.

Return Values

None.

Exceptions

None.

vp_enable_interrupts

Syntax

```
void    vp_enable_interrupts  ()
```

Summary

This routine counters a previous call to disable interrupts in the calling processor. Interrupts are enabled if the disable depth is returned to zero.

Parameters

None.

Description

Multiple disable interrupt calls are tracked by the disable count depth. This routine counteracts one disable call by decrementing the disable depth count. The interrupt disable depth count is decremented. If this decrement restores the count to its initial value, interrupts are enabled.

Return Values

None.

Exceptions

None.

End of Chapter

Chapter 7
Data and Memory Management Routines

This chapter describes the kernel memory and data management routines that your driver can call. Included are routines for verifying pointers to data buffers, manipulating buffer vectors, and allocating and releasing memory.

The chapter is divided into three major sections:

- **Memory Management Routines** — Routines for allocating and releasing wired and unwired memory in the global kernel address space.
- **User Data Access Validation Routines** — Routines used for accessing user address space. These routines are used to validate user-supplied pointers and copy data to or from user memory.
- **Buffer Vector Management Routines** — Routines for managing user data buffers for the **readv**, **read**, **write**, and **writev** system calls.

Each section introduces the major features of the routines that follow. Following each introduction is a "Constants and Data Structures" section that lists some of the constants and data structures used by the routines. For a full listing of constants and data structures, see the include files listed in Chapter 3.

Memory Management Routines

Memory management routines are provided for allocating and releasing wired and unwired memory in the global kernel address space.

There are two types of allocation routines for wired and unwired memory: **standard** and **perhaps**. The **standard** and **perhaps** allocation routines are essentially the same. The difference is that **vm_get_wired/unwired_memory** will panic if memory cannot be allocated, whereas **vm_perhaps_get_wired/unwired_memory** will return an error indication. Memory allocation fails because internal allocation limits have been exceeded; because these limits may be lower for the **perhaps** versions, the **perhaps** versions may fail when the **standard** versions would not.

Memory must always be released as the same type (wired or unwired) and size as obtained. If unwired memory is obtained and then wired, it must be unwired before being released and must be released as unwired memory.

The wire and unwire operations affect both the memory itself and any page tables needed to reference the memory. Hence, one is guaranteed that the physical frames assigned to wired memory will not change and that no page faults will occur on wired memory. The wire and unwire operations nest so that multiple calls on the same memory do not interfere.

Calls to wire memory may be nested. After memory has been wired, subsequent wire requests result in the incrementing of a wired count for the memory. Unwire calls decrement the wired count. Memory becomes unwired when the count is decremented to zero.

The routines described in this section are as follows:

- **vm_get_physical_byte_address**
- **vm_get_unwired_memory**
- **vm_get_wired_memory**
- **vm_map_physical_memory**
- **vm_unmap_physical_memory**
- **vm_mark_mod_and_ref_and_unwire_memory**
- **vm_mark_ref_and_unwire_memory**
- **vm_perhaps_get_unwired_memory**
- **vm_perhaps_get_wired_memory**

- **vm_release_unwired_memory**
- **vm_release_wired_memory**
- **vm_unwire_memory**
- **vm_wire_memory**

Routines beginning with **vm** require the **i_vm.h** include file.

Constants and Data Structures

This section discusses the literals used to specify data alignment in calls to **vm_get_wired_memory** and **vm_get_unwired_memory**.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_vm.h** for structures beginning with the **vm** acronym). Chapter 3 lists the various include files.

Page Alignment Literals

After memory has been aligned, you cannot ask for a quantity smaller than the specified alignment. You cannot ask for page alignment for less than one page, except in the case of VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS (shown below).

`VM_PAGE_ALIGNED`

This constant will request page alignment. Don't use this alignment when VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS is sufficient, because it will result in wasted space.

`VM_DOUBLE_WORD_ALIGNED`

This constant will request double word alignment (64-bit).

`VM_WORD_ALIGNED`

This constant will request word alignment (32-bit).

`VM_BYTE_ALIGNED`

This constant will request byte alignment.

`VM_DEFAULT_ALIGNMENT`

This constant represents the most efficient alignment for use when allocating strings and structures. The default is double word alignment (64-bit) because, in most cases, the system deals with double word alignment most efficiently. Use this constant whenever possible.

`VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS`

This constant represents an alignment that is guaranteed not to cross a page boundary and will be default-aligned. Allocations that use this alignment are restricted to one page or less.

`VM_INVALID_MEMORY_PTR`

This constant will be returned by **vm_perhaps_get_wired_memory** and **vm_perhaps_get_unwired_memory** when the memory allocation fails.

vm_get_physical_byte_address

Syntax

```
void  vm_get_physical_byte_address   (logical_address,
                                      is_user_address,
                                      physical_address_ptr)

byte_address_type    logical_address;       /*READ ONLY*/
boolean_type         is_user_address;       /*READ ONLY*/
byte_address_type *  physical_address_ptr; /*WRITE ONLY*/
```

Summary

This function returns the physical address that corresponds to the given logical byte address.

Parameters

logical_address — The logical address for which a physical address is needed.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

physical_address_ptr — A pointer to the physical address corresponding to the given logical address, filled by this routine. If the logical address was invalid, this address will be **VM_INVALID_PHYSICAL_ADDRESS_PTR**.

Description

See Summary.

Return Values

None.

Exceptions

None.

vm_get_unwired_memory

Syntax

```
pointer_to_any_type vm_get_unwired_memory  (bytes, alignment)

uint32_type         bytes;              /*READ ONLY*/
uint32_type         alignment;          /*READ ONLY*/
```

Summary

This routine allocates unwired memory from available address space.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

This routine allocates unwired memory on the alignment specified by the user. The amount of memory allocated is specified by the **bytes** parameter. If the allocation fails, the system panics.

Return Values

memory_ptr — The routine returns a byte pointer to the allocated space.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

VM_PANIC_GET_UNWIRED_MEMORY — The requested memory could not be allocated.

vm_get_wired_memory

Syntax

```
pointer_to_any_type  vm_get_wired_memory  (bytes, alignment)

uint32_type          bytes;           /*READ ONLY*/
uint32_type          alignment;       /*READ ONLY*/
```

Summary

This routine allocates wired memory from available address space.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

This routine allocates wired memory on the alignment specified by the user. The amount of memory allocated is specified by the **bytes** parameter. If the allocation fails, the system panics.

Return Values

memory_ptr — The routine returns a byte pointer to the allocated space.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

VM_PANIC_GET_WIRED_MEMORY — The requested memory could not be allocated.

vm_map_physical_memory

Syntax

```
status_type       vm_map_physical_memory   ( logical_addr,
                                             physical_addr,
                                             num_bytes,
                                             access_mode,
                                             sharing,
                                             control_flags )

byte_address_type     logical_addr;        /*READ ONLY*/
byte_address_type     physical_addr;       /*READ ONLY*/
uint32_type           num_bytes;           /*READ ONLY*/
bit32_type            access_mode;         /*READ ONLY*/
int32_type            sharing;             /*READ ONLY*/
bit32_type            control_flags;       /*READ ONLY*/
```

Summary

This routine supports the **mmap**(2) system call.

Parameters

logical_addr — The first logical address in a contiguous block of **num_bytes** to be mapped to **physical_addr**. This address must be on a page boundary.

physical_addr — The first physical address in a contiguous block of **num_bytes** to which the **logical_addr** will be mapped. This address also must be on a page boundary.

num_bytes — The total number of bytes to be mapped, which must be an integral multiple of the logical page size.

access_mode — This is the bitwise OR of all appropriate access modes. It may contain any of **PROT_READ**, **PROT_WRITE**, and/or **PROT_EXEC**, from **sys/mman.h**. No checking will be performed as to the appropriateness of the specified modes, and it is assumed that privilege violations will not occur, or be enforced by higher level routines.

sharing — Under the current implementation, this must be **MAP_SHARED**. **MAP_PRIVATE** is not yet supported though it is specified as an option.

control_flags — This is a bit-field used to send special control information. One bit is used for cache control. This bit-field may include information for inhibiting caching which should be passed on to the page table entries of the mapped physical memory. A second bit is used to specify whether the passed logical address is to be in the user or kernel address space. The remaining bits

are reserved for later expansion.

The following list describes the bit positions of the currently supported flags:

Bit 0 — This flag specifies whether user or kernel address space is to be used. Use **VM_MMAP_IS_KERNEL_ADDRESS_SPACE_MASK** to specify kernel addresses; otherwise, user addresses are assumed by default. (Bit 0 is the lowest order bit.)

Bit 1 — This flag specifies whether to write through or inhibit caching. Use **VM_MMAP_WRITE_THROUGH_MASK** to specify write-through caching on architectures that support it; otherwise, by default, no caching (cache inhibiting) is assumed.

Bits 2-31 — All other bits are unused and reserved for expansion.

Description

This routine will support the **mmap**(2) system call. It provides the kernel data structure modifications to **map** an area of a process's address space to real physical memory. Basically, this is done by searching for a contiguous region of a process's data area and setting pointers to the appropriate physical memory frame.

It may be called by either user or kernel processes; that is, the passed **logical_addr** may be either a kernel or user address. A bit in the **control_flags** parameter controls this distinction. The **logical_addr** passed, offset by **num_bytes**, must be part of the current address space before this call is made. Kernel processes should have already allocated unwired memory, while users should **valloc**() the appropriate range before making the mapping call.

The range addresses to be mapped must have referred to an existing region of a process's data area, or else an error will result. Any existing data that was addressed in this range will be discarded. No other explicit cleanup is needed for an address space that has been mapped. If the process that called **mmap**(2) exits, its mapped region will be implicitly unmapped by the exit path. Likewise, if the process calls any version of **exec**(2), the mapped region will also be implicitly unmapped before the new program begins to execute. Finally, in the case of a **fork**(2), the new child will implicitly inherit the parent's mapped address space.

Return Values

OK — All frames were mapped successfully.

VM_EINVAL_MMAP_UNSUPPORTED — This error will be returned if the

parameter **sharing** is set to **MAP_PRIVATE**; currently, only **MAP_SHARING** is supported. The function will abort before any modifications are made.

VM_EINVAL_MMAP_BYTES_NOT_MULTIPLE — This error code will be returned if the parameter **num_bytes** is not an integral multiple (greater than zero) of the size of a physical page. The function will abort before any modifications are made.

VM_EINVAL_MMAP_BAD_ADDR_BOUNDARY — This error code will be returned if either of the address parameters, **physical_addr** or **logical_addr**, is not aligned on a page boundary. The function will abort before any modifications are made.

VM_EINVAL_MMAP_SPACE_UNALLOCATED — This error will occur when the **logical_addr** is not already a part of the current process's address space. The function will abort before any modifications are made.

VM_EINVAL_MMAP_ADDRESS_NOT_DATA — This error will be returned when the **logical_addr** is not a part of the calling process's data area. Only regions of a process's address space of the data variety will be accepted for mapping. The function will abort before any modifications are made.

VM_EINVAL_MMAP_BAD_REGION — This error will be returned in several circumstances when the range of addresses to be mapped is inappropriate. The following are specific examples of this: when **logical_addr** offset by the **num_bytes** is greater than the maximum address (4G); or when the region of addresses to be mapped, **logical_addr** to **logical_addr+num_bytes**, does not fit in the size of the process's current data area; or when **logical_addr** cannot be located in any address area. The function will abort before any modifications are made.

VM_EINVAL_MMAP_ALREADY_MAPPED — This error will be returned when any data contained within the passed range of addresses (**logical_addr** through **logical_addr+num_bytes**) is already mapped. A region of a process's data area can be remapped, but only if an explicit **munmap**(2) is done before the remap attempt. The function will abort, and this error will return before any modifications are made.

Abort Conditions

None.

vm_unmap_physical_memory

Syntax

```
status_type  vm_unmap_physical_memory   (logical_addr,
                                         num_bytes,
                                         control_flags )

byte_address_type  logical_addr;         /*READ ONLY*/
uint32_type        num_bytes;            /*READ ONLY*/
bit32_type         control_flags;        /*READ ONLY*/
```

Summary

This routine will support the **munmap**(2) system call.

Parameters

logical_addr — The first logical address in a contiguous block of **num_bytes** to be unmapped. This address must be on a page boundary.

num_bytes — The total number of bytes to be unmapped. This does not necessarily have to be identical to the number that were mapped originally; however, it must be an integral multiple of the logical page size.

control_flags — This is a bit-field used to send special control information. A bit is necessary to specify whether the passed logical address is to be in the user or kernel address space. The remaining bits may be used later, and are reserved for expandability. The following list describes the bit positions of the currently supported flags:

Bit 0 — This flag specifies whether to use user or kernel address space. Use **VM_MUNMAP_IS_KERNEL_ADDRESS_SPACE_MASK** to specify kernel addresses; otherwise, user addresses are assumed by default. Bit 0 is the lowest order bit.

Bits 1-31 — All other bits are unused and reserved for expansion.

Description

This routine will support the **munmap**(2) system call. It will **unmap** an area of a process's address space to which was previously mapped by **mmap**(2). Only previously mapped areas can be unmapped. Upon unmapping, the appropriate address space will be reset to be non-resident but will still be a valid area of the data area.

It may be called by either user or kernel processes and still work properly; that is, the passed **logical_addr** may be either a kernel or user address. A bit in the parameter **control_flags** will be used to make this distinction.

Return Values

OK — All frames were unmapped successfully.

VM_EINVAL_MUNMAP_BYTES_NOT_MULTIPLE — This error code will be returned if the parameter **num_bytes** is not an integral multiple of the physical page size. The function will abort before any modifications are made.

VM_EINVAL_MUNMAP_BAD_ADDR_BOUNDARY — This error code will be returned if the address parameter, **logical_addr**, is not aligned on a page boundary. The function will abort before any modifications are made.

VM_EINVAL_MUNMAP_BAD_REGION — This error code will be returned when several situations occur: if the range of addresses to unmap (**logical_addr** through **logical_addr+num_bytes**) is not entirely valid; if the starting address cannot be found in the process's data area; or if the address range overflows beyond the 4G upper limit.

VM_EINVAL_MUNMAP_DATA_NOT_MAPPED — This error will occur if an attempt is made to unmap any portion of an address space that has not been previously mapped. Only previously mapped regions may be unmapped.

Abort Conditions

None.

vm_mark_mod_and_ref_and_unwire_memory

Syntax

```
void  vm_mark_mod_and_ref_and_unwire_memory (start_address,
                                             is_user_address,
                                             bytes_to_unwire)


pointer_to_any_type  start_address;   /* READ ONLY */
boolean_type         is_user_address; /*READ ONLY*/
uint32_type          bytes_to_unwire; /* READ ONLY */
```

Summary

This routine marks the frames indicated as having been referenced and modified, and then unwires the frames.

Parameters

start_address — The byte address indicating the start of the memory to be unwired. The value in **start_address** is rounded down to a page boundary.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

1bytes_to_unwire — The number of bytes to be unwired.

Description

This routine marks frames as having been referenced and modified and then unwires them. It starts at **start_address** (rounded down to a page boundary), goes for **bytes_to_unwire** number of bytes and rounds up to a page boundary.

Memory needs to be marked as modified if it has been wired and then used as an I/O buffer. I/O uses direct memory access, which does not cause the frame to be marked as modified automatically. Therefore, this routine will set the modified bit explicitly.

Return Values

None.

vm_mark_ref_and_unwire_memory

Syntax

```
void  vm_mark_ref_and_unwire_memory  (start_address,
                                      is_user_address,
                                      bytes_to_unwire)

pointer_to_any_type  start_address;    /*READ ONLY*/
boolean_type         is_user_address;  /*READ ONLY*/
uint32_type          bytes_to_unwire;  /*READ ONLY*/
```

Summary

This routine marks the indicated frames as having been referenced and then unwires them.

Parameters

start_address — The byte address indicating the start of the memory to be unwired.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_unwire — The number of bytes to be unwired.

Description

This routine marks the frames as having been referenced and then unwires them. It starts at **start_address** (rounded down to a page boundary), goes for **bytes_to_unwire** number of bytes and rounds up to a page boundary.

Memory needs to be marked as referenced if it has been wired and then used as an I/O buffer. I/O uses direct memory access, which does not cause the frame to be marked as referenced automatically. Therefore, this routine will set the referenced bit explicitly.

Return Values

None.

vm_perhaps_get_unwired_memory

Syntax

```
pointer_to_any_type vm_perhaps_get_unwired_memory (bytes,
                                                   alignment)

uint32_type         bytes;          /*READ ONLY*/
uint32_type         alignment;      /*READ ONLY*/
```

Summary

This routine allocates unwired memory.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

Memory is allocated from unwired memory on the alignment specified by the user. The amount of memory allocated is specified by the **bytes** parameter.

Return Values

memory_ptr — The memory was allocated successfully.

VM_INVALID_MEMORY_PTR — The memory could not be allocated.

Exceptions

None.

vm_perhaps_get_wired_memory

Syntax

```
pointer_to_any_type  vm_perhaps_get_wired_memory  (bytes,
                                                   alignment)

uint32_type          bytes;          /*READ ONLY*/
uint32_type          alignment;      /*READ ONLY*/
```

Summary

This routine allocates wired memory.

Parameters

bytes — The number of bytes to be allocated; **bytes** must be a positive value.

alignment — The byte alignment of the allocated space. Constants for the alignment parameter are defined in **i_vm.h**.

Description

Memory is allocated from wired memory on the alignment specified by the user. The amount of memory to be allocated is specified by the **bytes** parameter.

Return Values

memory_ptr — The memory was allocated successfully.

VM_INVALID_MEMORY_PTR — The memory could not be allocated.

Exceptions

None.

vm_release_unwired_memory

Syntax

```
void  vm_release_unwired_memory   (memory_ptr, bytes)

pointer_to_any_type  memory_ptr;        /*READ ONLY*/
uint32_type          bytes;             /*READ ONLY*/
```

Summary

This routine releases unwired memory that was previously obtained via a **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call.

Parameters

memory_ptr — A byte pointer to the start of the memory to be released. **memory_ptr** must be the same pointer that was returned by the **vm_get_unwired_memory** or **vm_perhaps_get_uwired_memory** call when memory was originally requested.

bytes — The number of bytes to be released. **bytes** must be the same number of bytes as given to the **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call when memory was originally requested.

Description

This routine releases the given number of bytes of unwired memory, starting at the given byte address. This memory must have been obtained via a **vm_get_unwired_memory** or **vm_perhaps_get_unwired_memory** call.

Return Values

None.

Exceptions

None.

vm_release_wired_memory

Syntax

```
void  vm_release_wired_memory   (memory_ptr, bytes)

pointer_to_any_type  memory_ptr;        /*READ ONLY*/
uint32_type          bytes;             /*READ ONLY*/
```

Summary

This routine releases wired memory that was previously obtained via a **vm_get_wired_memory** or **vm_perhaps_get_wired_memory** call.

Parameters

memory_ptr — A byte pointer to the start of the memory that is to be released. **memory_ptr** must contain the same pointer that was returned by the **vm_get_wired_memory** or the **vm_perhaps_get_wired_memory** call when memory was originally requested.

bytes — The number of bytes to be released. **bytes** must be the same number of bytes as requested in the **vm_get_wired_memory** or **vm_perhaps_get_wired_memory** call when memory was originally requested.

Description

This routine releases the given number of bytes of wired memory, starting at the given byte address. This memory must have been obtained via a **vm_get_wired_memory** or **vm_perhaps_get_wired_memory** call.

Return Values

None.

Exceptions

None.

vm_unwire_memory

Syntax

```
void  vm_unwire_memory (start_address,
                        is_user_address,
                        bytes_to_unwire)

pointer_to_any_type  start_address;    /*READ ONLY*/
boolean_type         is_user_address;  /*READ ONLY*/
uint32_type          bytes_to_unwire;  /*READ ONLY*/
```

Summary

This routine unwires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_unwire**.

Parameters

start_address — The byte address indicating the start of the memory to be unwired.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_unwire — The number of bytes to be unwired.

Description

This routine unwires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_unwire**. Unwiring is done only on blocks of a complete page. Therefore, if **start_address** is not the start of a page, **vm_unwire_memory** starts at the next lowest page boundary. Similarly, if **bytes_to_unwire** does not end on a page boundary, unwiring continues into the next higher page boundary.

Return Values

None.

vm_wire_memory

Syntax

```
status_type vm_wire_memory (start_address,is_user_address,
                            bytes_to_wire)

pointer_to_any_type  start_address;   /*READ ONLY*/
boolean_type         is_user_address; /*READ ONLY*/
uint32_type          bytes_to_wire;   /*READ ONLY*/
```

Summary

This routine wires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_wire.**

Parameters

start_address — The byte address indicating the start of the memory to be wired.

is_user_address — Indicates whether the logical address specified is a user or kernel address. If **is_user_address** is TRUE, the address is a user address. If FALSE, it is a kernel address.

bytes_to_wire — The number of bytes to be wired.

Description

This routine wires the memory indicated by **start_address** for the number of bytes indicated by **bytes_to_wire.** Wiring is done only on blocks of a complete page. Therefore, if **start_address** is not the start of a page, **vm_wire_memory** starts at the next lowest page boundary. Similarly, if **bytes_to_wire** does not end on a page boundary, wiring continues into the next higher page boundary.

Return Values

OK — The memory was successfully wired.

[*other error statuses*] — A hard I/O error occurred that prevented a page from being brought in. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 8.

User Data Access Validation Routines

Routines in this section are used to validate user-supplied memory addresses or to transfer data between user memory and kernel memory.

For most I/O operations, the kernel will validate user-specified buffers before performing the operation. However, the kernel cannot validate user-specified buffers for an **ioctl** operation because **ioctl** packets may contain buffer pointers embedded in the packet. Therefore, the driver must validate user buffers itself for **ioctl** operations.

The routines in this section perform read, write, and execute access checking. These checks verify that the buffer memory has permissions appropriate for the requested operation (for example, read permission is granted for a write operation).

The routines described in this section are as follows:

- **sc_check_access_and_read_string_from_user**
- **sc_check_byte_access**
- **sc_read_bytes_from_user**
- **sc_write_bytes_to_user**
- **sc_write_string_to_user**

Routines beginning with **sc** require the **i_sc.h** include file.

Constants and Data Structures

No special constants or data structures are required for these routines.

sc_check_access_and_read_string_from_user

Syntax

```
status_type  sc_check_access_and_read_string_from_user
                          (buffer_ptr_ptr, dest_ptr, count_ptr)

pointer_to_any_ptr_type  buffer_ptr_ptr;  /*READ ONLY*/
pointer_to_any_type      dest_ptr;        /*WRITE ONLY*/
uint32_ptr_type          count_ptr;       /*READ/WRITE*/
```

Summary

This routine checks the user address space starting at **buffer_ptr_ptr** for **count** bytes, or through the terminating null, to verify that read access is available for the entire string. The string is also copied into the destination buffer.

Parameters

buffer_ptr_ptr — A pointer to the byte pointer that marks the start of the string for which access is to be checked.

dest_ptr — A pointer to the kernel buffer into which the string is to be copied.

count_ptr — On input, a pointer to the maximum size, in bytes, of the string, including the terminating null. On output, the size of the string copied into the kernel buffer, including the terminating null.

Return Values

OK — Read access is available for the entire string. The string has been copied to the destination with a terminating null.

SC_EFAULT_STRING_TOO_LONG — Read access is available for the maximum size of the string, but there is no terminating null in that length. The contents of the destination are undefined.

SC_EFAULT_NO_READ_ACCESS — Read access is available for less than the maximum size of the string, and no terminating null was found in the area to which read access was available. The contents of the destination are undefined.

[*other error statuses*] — The bytes could not be read because of an error. The specific list of possible errors is too long to give here. You can decode any

status returned here using the error status decoding methods described in Chapter 8.

Exceptions

None.

sc_check_byte_access

Syntax

```
status_type sc_check_byte_access (buffer_ptr_ptr,
                                  count,access)

pointer_to_any_ptr_type  buffer_ptr_ptr;  /*READ/WRITE*/
uint32_type              count;           /*READ ONLY*/
sc_access_mode_type      access;          /*READ ONLY*/
```

Summary

This routine checks the user address space starting at **buffer_ptr_ptr** for **count** bytes to verify that **access** access is available for the entire area.

Parameters

buffer_ptr_ptr — A pointer to the byte pointer that marks the start of the area for which access is to be checked.

count — The size, in bytes, of the area to be checked.

access — The access modes to be checked.

Return Values

OK — The requested access is available for the entire area.

SC_EFAULT_NO_ACCESS — One or more bytes of the specified area do not have the required access.

Exceptions

None.

sc_read_bytes_from_user

Syntax

```
status_type  sc_read_bytes_from_user (source_ptr,
                                      dest_ptr, count)

pointer_to_any_type  source_ptr;   /*READ ONLY*/
pointer_to_any_type  dest_ptr;     /*READ ONLY*/
uint32_type          count;        /*READ ONLY*/
```

Summary

This routine moves the specified number of bytes from the user's address space to the kernel address space.

Parameters

source_ptr — A pointer to the location in the user's address space from which the data is to be moved.

dest_ptr — A pointer to the location in the kernel address space to which the data is to be moved.

count — The number of bytes to be moved.

Description

The specified number of bytes are moved from the source to the destination. Access should be checked before reading.

Return Values

OK — The bytes were successfully read from the user address space into kernel address space.

[*other error statuses*] — The bytes could not be read because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 8.

Exceptions

None.

sc_write_bytes_to_user

Syntax

```
status_type  sc_write_bytes_to_user (source_ptr,
                                     dest_ptr, count)

pointer_to_any_type  source_ptr;  /*READ ONLY*/
pointer_to_any_type  dest_ptr;    /*READ ONLY*/
uint32_type          count;       /*READ ONLY*/
```

Summary

This routine moves the specified number of bytes from the kernel address space to the user's address space.

Parameters

source_ptr — A pointer to the location in the kernel address space from which the data is to be moved.

dest_ptr — A pointer to the location in the user address space to which the data is to be moved.

count — The number of bytes to be moved.

Description

The specified number of bytes are moved from the source to the destination. This routine assumes that access has already been checked.

Return Values

OK — The bytes were successfully written to the user's address space.

[*other error statuses* — The bytes could not be written because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 8.

Exceptions

None.

sc_write_string_to_user

Syntax

```
status_type  sc_write_string_to_user (source_ptr, dest_ptr)

pointer_to_any_type source_ptr;    /*READ ONLY*/
pointer_to_any_type dest_ptr;      /*READ ONLY*/
```

Summary

This routine moves bytes from the kernel address space to the user's address space up to and including the first null byte in the source string.

Parameters

source_ptr — A pointer to the location in the kernel address space from which the data is to be moved.

dest_ptr — A pointer to the location in the user address space to which the data is to be moved.

Description

Bytes are moved from the source to the destination until a null byte is found in the source. The null is transferred to the destination. This routine assumes that access has already been checked.

Return Values

OK — The bytes were successfully written to the user's address space.

[*other error statuses* — The bytes could not be written because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 8.

Exceptions

None.

Buffer Vector Management Routines

This section describes routines you can use to manage buffer vectors. Buffer vectors are data structures used to package user data buffers specified by the **read, readv, write,** and **writev** system calls. The "v" system calls (**readv** and **writev**) require such non-contiguous buffer space, and buffer vectors allow a buffer area to be spread across non-contiguous memory.

While the **read** and **write** system calls do not use non-contiguous buffers, they still use the buffer vector interface. These system calls will have a buffer vector array with only one entry (see **io_init_one_entry_buffer_vector**).

A buffer vector consists of a collection of individual buffer descriptors with associated state variables. Each buffer descriptor consists of a buffer pointer and a buffer size. A buffer vector may be either the source of a read or the destination of a write operation; the individual buffer descriptors define the locations from which the data is being read or into which the data is being written.

The current position within the buffer vector is maintained by the associated state variable. The current position defines where the next byte of data will be read from or written to. The current position is initialized to the first byte of the first buffer descriptor.

The routines described in this section are as follows:

- **io_add_to_buffer_vector_position**
- **io_get_buffer_vector_io_info**
- **io_get_buffer_vector_position**
- **io_get_buffer_vector_residual**
- **io_get_buffer_vector_byte_count**
- **io_init_buffer_vector**
- **io_init_one_entry_buffer_vector**
- **io_read_from_buffer_vector**
- **io_reset_buffer_vector_position**
- **io_set_buffer_vector_residual**
- **io_write_to_buffer_vector**

Routines beginning with **io** require the **i_io.h** include file.

Constants and Data Structures

See Chapter 4 for a description of **io_buffer_vector_type** and other data structures used with buffer vectors.

io_add_to_buffer_vector_position

Syntax

```
void io_add_to_buffer_vector_position
                                    (buffer_vector_ptr, count)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
int32_type                 count;             /*READ ONLY*/
```

Summary

This routine adds the given count to the current position associated with the given buffer vector.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose current position is to be changed.

count — The number of bytes to be added to the current buffer position.

Description

This routine adds the given count to the current position associated with the given buffer vector. The amount added may be positive or negative. If the new value of the current position would be less than zero or greater than the byte count associated with the buffer vector, the result is undefined. Note that changing the current position changes the residual count by implication, so that the relationship between the current position plus residual count and the overall byte count remains true.

Return Values

None.

Exceptions

None.

io_get_buffer_vector_io_info

Syntax

```
void io_get_buffer_vector_io_info (buffer_vector_ptr,
                                   buffer_ptr_ptr, count_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr;/*READ ONLY*/
pointer_to_any_ptr_type    buffer_ptr_ptr;   /*WRITE ONLY*/
uint32_ptr_type            count_ptr;        /*WRITE ONLY*/
```

Summary

This routine takes the current buffer descriptor and returns the buffer pointer and the number of contiguous bytes left in the buffer from that pointer.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose I/O information is to be returned.

buffer_ptr_ptr — A pointer to where the buffer pointer at the current position is to be returned.

count_ptr — A pointer to where the number of contiguous bytes starting at the current position is to be returned. This returned value will always be greater than zero.

Description

This routine returns the actual buffer pointer and contiguous byte count associated with that position so that direct access I/O operations can be performed on the buffer.

Drivers can use this routine to produce the same effect as **io_read_bytes_from_buffer_vector** or **io_write_bytes_to_buffer_vector**, but with the transfer going directly between the device and the buffer vector instead of through an intermediate memory buffer. To do this, the driver successively gets the I/O information for the current position, performs direct access I/O, and updates the current position with **io_add_to_buffer_vector_position**.

NOTE: This routine must not be called when the buffer vector residual is zero, as the returned count is defined to always be strictly greater than zero.

Return Values

None.

Exceptions

None.

io_get_buffer_vector_position

Syntax

```
uint32_type io_get_buffer_vector_position
                                        (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
```

Summary

This routine gets the current position of the specified buffer vector.

Parameters

buffer_vector_ptr — A pointer to the buffer vector from which the current position is to be retrieved.

Return Values

[*position*] — The current position associated with the given buffer vector.

Exceptions

None.

io_get_buffer_vector_residual

Syntax

```
uint32_type io_get_buffer_vector_residual (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr;  /*READ ONLY*/
```

Summary

This routine gets the number of bytes remaining in the specified buffer vector.

Parameters

buffer_vector_ptr — A pointer to the buffer vector from which the residual byte count is to be retrieved.

Description

This routine gets the number of bytes remaining in the specified buffer vector. This residual count is always equal to the byte count of the buffer vector minus the current position. The **buffer_vector_ptr** is assumed to be valid.

Return Values

count — The residual bytes associated with the given buffer vector.

Exceptions

None.

io_get_buffer_vector_byte_count

Syntax

```
uint32_type io_get_buffer_vector_byte_count
                                        (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
```

Summary

This routine gets the byte count for the specified buffer vector. This count is the number of bytes of data that this vector can hold.

Parameters

buffer_vector_ptr — A pointer to the buffer vector from which the byte count is to be retrieved.

Description

This routine gets the byte count for the specified buffer vector. This count is the number of bytes of data that this vector can hold. The **buffer_vector_ptr** is assumed to be valid.

Return Values

count — The byte count associated with the given buffer vector.

Exceptions

None.

io_init_buffer_vector

Syntax

```
void  io_init_buffer_vector (buffer_vector_ptr, total_size,
                             buffer_descriptors, count)

io_buffer_vector_ptr_type  buffer_vector_ptr;/*READ/WRITE*/
uint32_type                total_size;       /*READ ONLY*/
io_buffer_descriptor_ptr_type
                           buffer_descriptors;/*READ ONLY*/
uint16_type                count;            /*READ ONLY*/
```

Summary

This routine is used to initialize a buffer vector.

Parameters

buffer_vector_ptr — The buffer vector to be initialized.

total_size — The sum of sizes from the buffer descriptors.

buffer_descriptors — Pointer to the array of buffer descriptors to be associated with the buffer vector.

count — The number of entries in the **buffer_descriptors** array.

Description

This routine is used to initialize a buffer vector. The **buffer_vector_ptr** is assumed to be valid.

Return Values

None.

Exceptions

None.

io_init_one_entry_buffer_vector

Syntax

```
void io_init_one_entry_buffer_vector (buffer_vector_ptr,
                                      buffer_ptr, size)

io_buffer_vector_ptr_type buffer_vector_ptr;/*READ/WRITE*/
pointer_to_any_type       buffer_ptr;       /*READ ONLY*/
uint32_type               size;             /*READ ONLY*/
```

Summary

This routine is used to initialize a buffer vector that will have only one entry in the **buffer_descriptors** array.

Parameters

buffer_vector_ptr — The buffer vector to initialize.

buffer_ptr — A pointer to the buffer that is to be the sole entry in the **buffer_descriptors** array.

size — The size, in bytes, of the sole entry in the **buffer_descriptor** array.

Description

This routine is called if a buffer vector structure is being created with a single buffer descriptor entry. Using this routine to initialize a single entry buffer vector allows optimizations to be performed in buffer vector management.

Return Values

None.

Exceptions

None.

io_read_from_buffer_vector

Syntax

```
status_type io_read_from_buffer_vector (buffer_vector_ptr,
                                        buffer_ptr, count_ptr)

io_buffer_vector_ptr_type buffer_vector_ptr;  /*READ/WRITE*/
pointer_to_any_type       buffer_ptr;         /*READ/WRITE*/
uint32_ptr_type           count_ptr;          /*READ/WRITE*/
```

Summary

This routine is used to read data from the buffer vector into the specified buffer.

Parameters

buffer_vector_ptr — Pointer to the buffer vector from which data is to be read.

buffer_ptr — Pointer to where data from the buffer vector is to be placed.

count_ptr — On entry, the number of bytes to move. On exit, the actual number of bytes moved.

Description

Data is moved into the specified buffer starting at the current position of the specified buffer vector until all the data in the buffer vector has been exhausted or until **count_ptr** bytes have been moved. **count_ptr** is set to the actual number of bytes moved.

Return Values

OK — The bytes were successfully written to the buffer area.

[*other error statuses*] — The bytes could not be read because of an error. The specific list of possible errors is too long to give here. You can decode any status returned here using the error status decoding methods described in Chapter 8.

Exceptions

None.

io_reset_buffer_vector_position

Syntax

```
void io_reset_buffer_vector_position (buffer_vector_ptr)

io_buffer_vector_ptr_type  buffer_vector_ptr; /*READ ONLY*/
```

Summary

This routine resets the current position of the buffer vector to zero.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose position is to be reset to zero.

Return Values

None.

Exceptions

None.

io_set_buffer_vector_residual

Syntax

```
void io_set_buffer_vector_residual (buffer_vector_ptr, count)

io_buffer_vector_ptr_type  buffer_vector_ptr;/*READ ONLY*/
uint32_type                count;            /*READ ONLY*/
```

Summary

This routine sets the number of bytes remaining in the specified buffer vector. The current position is unchanged.

Parameters

buffer_vector_ptr — A pointer to the buffer vector whose residual byte count is to be set.

count — The value to which to set the residual.

Description

Because the residual is always equal to the total size minus the current position, and the current position is unchanged by this routine, this routine changes the total size by implication.

Return Values

None.

Exceptions

None.

io_write_to_buffer_vector

Syntax

```
status_type io_write_to_buffer_vector (buffer_ptr,
                                       buffer_vector_ptr, count_ptr)

pointer_to_any_type         buffer_ptr;       /*READ ONLY*/
io_buffer_vector_ptr_type   buffer_vector_ptr;/*READ/WRITE*/
uint32_ptr_type             count_ptr;          /*READ/WRITE*/
```

Summary

This routine is used to write data from the specified buffer into the buffer vector.

Parameters

buffer_ptr — Pointer to the buffer from which data is to be read.

buffer_vector_ptr — Pointer to the buffer vector to which data is to be written.

count_ptr — On entry, the number of bytes to be moved. On exit, the actual number of bytes moved.

Description

Data is moved into the buffer vector. The transfer starts at the beginning of the specified buffer and goes until the end of the buffer vector has been reached or until **count_ptr** bytes have been moved. **count_ptr** is set to the actual number of bytes moved.

Return Values

OK — The bytes were successfully written to the buffer area.

[*other error statuses*] — An error terminated the write operation. The list of possible errors is too long to give here. You can decode any status returned here using the status decoding methods described in Chapter 8.

Exceptions

None.

End of Chapter

Chapter 8
General Driver Routines

This chapter describes the DG/UX kernel routines used for a variety of driver operations, including error handling, set up/configuration, sending messages to the driver daemon, and accessing device selection tables.

The chapter is divided into the following sections:

- **Configuration Routines** — Routines used when you configure or deconfigure a device.
- **Driver Daemon and Generic Daemon Routines** — Routines that help you service asynchronous I/O requests. You must process such requests through the kernel facilities. The Driver Daemon and Generic Daemon are the kernel processes used to handle asynchronous I/O requests for all drivers.
- **Error Encoding and Logging Routines** — Routines used to create system-compatible statuses for your device.
- **Select Manager Routines** — Routines used in conjunction with your **dev_xxx_select** routine. The select manager facilities help administer multiple outstanding I/O requests for a single device.
- **Miscellaneous Driver Routines** — Other general routines used in driver operations.
- **Nodevice Routine Stubs** — Routine stubs used to handle erroneous I/O calls.

Each section introduces the major features of the routines that follow. Following each introduction is a "Constants and Data Structures" section which lists some of the constants and data structures used by the DG/UX routines. For a full list of constants and data structures, see the include files listed in Chapter 3.

Configuration Routines

This section describes routines your that driver's **dev_xxx_configure** and **dev_xxx_deconfigure** routines can use to configure/deconfigure a device and its units.

The system build process creates a list of devices to be configured from the entries in the system file. At boot time, the system initialization code scans this list and invokes the driver's **dev_xxx_configure** routine for each device of the driver's type in that list. The initialization code passes **dev_xxx_configure** the device code for the device in the standard format shown below:

device_mnemonic [*@device_code*] ([*parameters*])

You can use **io_parse_device_spec** to separate the different fields in this format.

The routines described in this section are as follows:

- **fs_submit_dev_request**
- **io_add_to_register_list**
- **io_allocate_device_number**
- **io_deallocate_device_number**
- **io_deregister_device_info**
- **io_check_device_spec**
- **io_forget_device_spec**
- **io_do_first_short_board_access**
- **io_do_first_long_board_access**
- **io_get_device_info**
- **io_map_device_number**
- **io_parse_device_spec**
- **io_perform_reset**
- **io_register_device_info**

Routines beginning with **fs** and **io** require the **i_fs.h** and **i_io.h** include files, respectively.

Constants and Data Structures

The routines in this section use the following constants and data structures. Try to avoid dependencies on the specifics of these structures, such as size or location of fields, since these specifics may change in later releases of the software.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_fs.h** for structures beginning with the **fs** acronym). Chapter 3 lists the various include files.

fs_dev_request_type

```
typedef struct
        {
fs_dev_request_operation_enum_type  operation;
        char_type  dirname[33];
        char_type  filename[33];
        union {
              fs_dev_create_request_type  create;
              }
              op;
        }
        fs_dev_request_type;
```

Description

This structure contains the information required to change a node in **/dev**.

The fields in this structure are as follows:

operation — The type of operation requested; for example, delete, or create.

dirname — The directory in which the node should reside. This name will be appended to **/dev/**. For example, set **dirname** to "rdsk" to create a node in **/dev/rdsk**. If you don't want the node in a directory under **/dev**, set **dirname[0]** to **FS_NULL_CHAR**.

filename — The filename of the node.

op — The information necessary for the operation requested.

fs_dev_request_operation_enum_type

```
typedef enum
        {
        Fs_Dev_Request_Operation_Create,
        Fs_Dev_Request_Operation_Delete
        }
             fs_dev_request_operation_enum_type  ;
```

Description

This enum type contains the valid operations supported by the /dev manager.

The fields in this structure are as follows:

Fs_Dev_Request_Operation_Create — Request to create a node in **/dev**. See **fs_dev_create_request_type**.

Fs_Dev_Request_Operation_Delete — Request to delete a node from /dev.

fs_dev_create_request_type

```
typedef struct
        {
        io_device_number_type     device;
        df_file_mode_type         mode_bits;
        }
             fs_dev_create_request_type  ;
```

Description

This structure contains the information required by the adapter driver to create a node in **/dev**.

The fields in this structure are as follows:

device — The device number of the node.

mode_bits — The initial mode bits of the node. This includes the file type information.

io_dev_adapt_info_type

```
typedef struct
    {
    char_ptr_type  name;
    char_ptr_type  device_code;
    char_ptr_type  params[IO_DEV_ADAPT_MAX_PARAMS];

}    io_dev_adapt_info_type  ;
```

This structure provides a method to pass data back from the **i_io_parse_dev_adapt_spec** routine.

The fields in this structure are as follows:

name — A pointer to the null terminated string of a device or adapter name.

device_code — A pointer to the null terminated string of a device code.

params — An array of pointers to null terminated strings for each of the parameters.

Literals

The following IO_DEV_ADAPT values define constants relating to the construction of both device and adapter specification strings.

```
#define IO_DEV_ADAPT_MAX_PARAMS((int16_type)3)
```

This liter defines the maximum number of parameters that may be specified in either a device or adapter specification string. This controls the size of the IO_DEV_ADAPT_INFO structures **params** element size.

```
#define IO_DEV_ADAPT_MAX_SPEC_SIZE((int32_type)256)
```

This literal specifies the maximum string length of a device or adapter specification, including the terminating null character.

```
#define IO_DEV_ADAPT_DEVICE_CODE_DELIMITER((char_type)'@')
```

This literal specifies the character that must prefix the sequence of characters of the device code of either a device or adapter specification.

```
#define IO_DEV_ADAPT_START_PARAMS_DELIMITER((char_type)'(')
```

This literal specifies the character that must prefix the sequence of characters of the parameters of either a device or adapter specification.

```
#define IO_DEV_ADAPT_END_PARAMS_DELIMITER((char_type)')')
```

This literal specifies the character that must suffix the sequence of characters of the parameters of either a device or adapter specification.

```
#define IO_DEV_ADAPT_PARAMS_DELIMITER((char_type)',')
```

This literal specifies the character that must separate parameter components of either a device or adapter specification.

uc_device_class_enum_type

```
typedef enum
    {
    Uc_Integrated_Device_Class =    0,
    Uc_Vmebus_Device_Class =        1,
    Uc_Invalid_Device_Class =       2,

    }    uc_device_class_enum_type  ;
```

Description

This type describes the classes of devices supported by the DG/UX kernel. A device is uniquely identified by its interrupt class and device code.

As new classes of device are supported this type definition will change. Check the **i_uc.h** (in **/usr/src/uts/aviion/ii**) include file for the latest supported classes.

uc_device_code_type

```
typedef  uint32_type              uc_device_code_type  ;
```

Description

This type is used to describe a device code, which, along with its associated device class, is used to identify an I/O device.

Device codes must be unique within a class, but the same value device code can be found in multiple classes. Thus, device codes are fit to the device class to which they apply.

The device codes for integrated devices are pre-defined and will be the same across all architectures. Note that there is no association between the pre-defined integrated device codes and physical hardware. The kernel will map the pre-assigned device code to the device interrupt on a given machine.

VME188 class devices do not have pre-assigned device codes because the VME interrupt vector mechanism allows devices to be set up to use any valid VME vector. In the VME188 class, the device code is the value of the VME vector. It is up to drivers to register device information specifying the appropriate VME vector to the kernel. When a VME class interrupt occurs, the kernel will return the VME vector of the interrupting device.

Integrated Device Code Literals

This section defines the values for the integrated device type preassigned device codes. The values below apply for all machine architectures. During driver initialization, a device's driver links its device code with an interrupt handler by registering the device. Use the following literals as the device codes for integrated class devices:

```
UC_SYSTEM_ERROR_DEVICE_CODE
UC_SYSTEM_TIMER_DEVICE_CODE
UC_KEYBOARD_DEVICE_CODE
UC_DUART_DEVICE_CODE
UC_PARALLEL_PORT_DEVICE_CODE
UC_ETHERNET_DEVICE_CODE
UC_SCSI_DEVICE_CODE
UC_DMA_TERMINAL_COUNT_DEVICE_CODE
UC_GRAPHICS_CARD_DEVICE_CODE
UC_CROSS_INTERRUPT_DEVICE_CODE
UC_PER_JP_TIMER_DEVICE_CODE
UC_DUART_TIMER_DEVICE_CODE
UC_SIGHP_DEVICE_CODE
UC_LOCATION_MONITOR_DEVICE_CODE
UC_POWER_FAIL_DEVICE_CODE
```

uc_reset_enum_type

```
typedef enum
    {
    Uc_Reset_Scsi =      0,
    Uc_Reset_Ethernet =  1,
    Uc_Reset_Async =     2,
    Uc_Reset_Keyboard =  3,
    Uc_Reset_Vme =       4,

    }    uc_reset_enum_type  ;
```

Description

This enumeration describes the various reset types available. These resets are all for

integrated class devices. Resets of non-integrated class devices are not supported. An enumeration is provided for any reset supported by any architecture.

The possible members of this type are as follows:

Uc_Reset_Scsi — Reset the SCSI integrated device.

Uc_Reset_Ethernet — Reset the Ethernet integrated device.

Uc_Reset_Async — Reset the Asynchronous integrated ports, such as DUARTS.

Uc_Reset_Vme — Reset the VME bus.

fs_submit_dev_request

Syntax

```
void   fs_submit_dev_request (dev_request_ptr)

fs_dev_request_ptr_type  dev_request_ptr;  /*READ ONLY*/
```

Summary

This routine is used to submit a request to create or delete a **/dev** entry. If the root is not mounted, then the request will not be performed until the root is mounted.

Parameters

dev_request_ptr — A pointer to the necessary information to manipulate a **/dev** entry. For the create operation, this information includes the file's major and minor device numbers, mode bits, type (block or character), containing directory (for example, "." or "rdsk") and the filename of the new file (for example, "tty05"). For the delete operation, only the filename and containing directory fields are required.

Description

A request to manipulate a **/dev** entry is accepted. The request will be processed immediately if the root has been mounted. Otherwise, the request is added to a queue for later processing.

Return Value

None.

io_add_to_register_list

Syntax

```
void  io_add_to_register_list  (device_number)

io_device_number_type   device_number;     /*READ ONLY*/
```

Summary

This routine adds the specified device to the list of disks that may be implicitly registered as part of system initialization. This routine is optional and is used only with disks.

Parameters

device_number — Device number of the disk to be registered.

Description

Implicitly registered disks are known to the file system without being specifically mounted. The specified device is added to a linked list of device numbers.

Return Value

None.

Exceptions

None.

io_allocate_device_number

Syntax

```
status_type io_allocate_device_number (major, handle,
                                       unit, minor_ptr)

io_major_device_number_type     major;     /*READ ONLY*/
bit32e_type                     handle;    /*READ ONLY*/
uint16_type                     unit;      /*READ ONLY*/
io_minor_device_number_ptr_type minor_ptr; /*WRITE ONLY*/
```

Summary

This routine assigns the device a minor device number. The major device number identifies the family of devices to which the device belongs.

Parameters

major — The device's major device number.

handle — The device handle which identifies the device to its driver.

unit — The unit number that identifies the device to its controller.

minor_ptr — A pointer to the location where the allocated minor device number is returned.

Description

The file system maintains a minor device number table for each family of devices as identified by a major device number. If no units of the specified type have been previously configured, the minor device number table address will be null. In this case, a minor device number table is allocated and its address is entered into **io_device_number_map**.

This routine searches the minor device number table for the first unused slot. The offset of the first unused slot in the table is assigned as the minor device number of the unit. The given device handle and unit number are stored in the minor number table entry to provide a mapping from minor number to device handle and unit number.

If no slots in the minor device number table are available, a new table, twice as large as the existing table, is allocated. The existing table is copied into the new table and deallocated. This procedure is repeated each time the minor device number table becomes full, until the table grows to contain MAX_MINOR_NUMBER_TABLE_ENTRIES entries. At this point, the error

IO_ENXIO_ALL_MINOR_NUMBERS_IN_USE is returned on subsequent minor number allocation requests.

Return Value

OK — No errors were discovered, so all returned arguments are valid.

IO_ENXIO_ALL_MINOR_NUMBERS_IN_USE — The minor device number table for this major device number contains no unused slots and has grown to the maximum size.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_MAJOR_NUMBER_EXCEEDS_MAX — The major device number argument exceeds the maximum specified by **cf_io_device_driver_count (see conf.c).**

io_deallocate_device_number

Syntax

```
void io_deallocate_device_number (device_number)

io_device_number_type  device_number;   /*READ ONLY*/
```

Summary

This routine terminates the association between the device and its minor device number.

Parameters

device_number — Contains the major and minor device numbers of the device being deconfigured.

Description

The minor device number table is found using the major number. The table is indexed by the given minor number, and the device handle field of the table entry is set to null. A null entry in the device handle field of a minor device table entry indicates that the entry is inactive and may be reused.

Return Value

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_MAJOR_NUMBER_EXCEEDS_MAX — The major device number argument exceeds the maximum specified by **cf_io_device_driver_count** in **conf.c**.

IO_PANIC_DEVICE_IS_NOT_CONFIGURED — An active entry in the minor device number table does not exist at the offset specified by the minor device number argument.

io_deregister_device_info

Syntax

```
void   io_deregister_device_info  (dev_code, dev_class)

io_device_code_type               dev_code;  /*READ ONLY*/
uc_device_class_enum_type         dev_class; /*READ ONLY*/
```

Summary

This routine deregisters the device by removing its current interrupt handler and device information structure from the DIT.

Parameters

dev_code — device code for which the current interrupt handler is to be disassociated.

dev_class — device class for which the current interrupt handler is to be disassociated.

Description

This routine reverses the effect of **io_register_device_info**. It deregisters the device by removing its current interrupt handler and device information structure from the DIT. After this call completes, future interrupts on the specified device code will be directed to the system supplied "nodevice" interrupt handler. If you make this call on a device code that does not currently have an interrupt handler, a panic will occur.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine may invoke the **sc_panic** routine with the following error code:

IO_PANIC_ILLEGAL_DEREGISTER_DEVICE_INFO — An attempt was made to deregister a device on a device code that did not have information registered.

io_check_device_spec

Syntax

```
status_type  io_check_device_spec (device_address,
                                   device_code)

opaque_ptr_type       device_address;      /*READ ONLY*/
io_device_code_type   device_code;         /*READ ONLY*/
```

Summary

This routine checks that the address and device code specified for the device are not already in use.

Parameters

device_address — The address of the primary registers for the device.

device_code — The device code for the device.

Description

This routine checks that the address and device code specified for the device are not already in use. Such address and device code validation will not prevent overlap of registers or RAM areas. It does help avoid the most common user errors in device specification. Only the first address for a device is checked.

Return Values

OK — The address and device code are not already in use.

IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED — The address or device code are already in use.

Exceptions

None.

io_forget_device_spec

Syntax

```
status_type  io_forget_device_spec     (device_address,
                                        device_code)

opaque_ptr_type        device_address;   /* READ ONLY  */
io_device_code_type    device_code;      /* READ ONLY  */
```

Summary

Release (that is, forget) a device specification that was claimed as the result of a previous call to the **io_check_device_spec** routine.

Parameters

device_address — The address of the primary registers for the device.

device_code — The device code for the device.

Description

When a device is deconfigured, the **device_address** claimed for the device must be freed by calling this routine. If you do not free the device address, calls to **io_check_device_spec** using this device address will fail.

Return Values

OK — The address/device code pair is freed.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — The address/device code to be freed were not found.

Exceptions

None.

io_do_first_short_board_access

Syntax

```
status_type io_do_first_short_board_access (register_ptr,
                                            register_contents_ptr,
                                            write_to_register)

bit16e_ptr_type    register_ptr;          /*READ/WRITE*/
bit16e_ptr_type    register_contents_ptr; /*READ/WRITE*/
boolean_type       write_to_register;     /*READ ONLY*/
```

Summary

This routine tests for the existence of the board at a particular memory-mapped I/O address. Use this routine for boards with short (16-bit) registers.

Parameters

register_ptr — A pointer to the register on the board to be accessed.

register_contents_ptr — A pointer a one-word read/write buffer. For a write operation, the contents of this buffer will be written to the register. For a read operation, the data read from the register will be stored in this buffer.

write_to_register — A boolean indicating whether the operation is read or write. When it is TRUE, the routine writes to the register. When it is FALSE, the routine reads from the register.

Description

Do the first access to a board register such that if a board is not present, the system will not hang or panic. The board should not be accessed again if IO_ENXIO_DEVICE_DOES_NOT_EXIST is returned. This routine assumes that the register is a short register.

Return Values

OK — The register was accessed successfully.

IO_ENXIO_DEVICE_DOES_NOT_EXIST — The board is not accessible.

Exceptions

None.

io_do_first_long_board_access

Syntax

```
status_type io_do_first_long_board_access (register_ptr,
                                           register_contents_ptr,
                                           write_to_register)

bit32e_ptr_type    register_ptr;           /*READ/WRITE*/
bit32e_ptr_type    register_contents_ptr;  /*READ/WRITE*/
boolean_type       write_to_register;      /*READ ONLY*/
```

Summary

This routine tests for the existence of the board at a particular memory-mapped I/O address. Use this routine for boards with long (32-bit) registers.

Parameters

register_ptr — A pointer to the register on the board to be accessed.

register_contents_ptr — A pointer to the contents of the given register. On input, if **write_to_register** is TRUE, this value will be written to the register. On output, when write_to_register is FALSE, this value will be the value read from the register.

write_to_register — A boolean indicating, when TRUE, to write to the register. Otherwise a read will be done.

Description

Do the first access to a long board register such that if a board is not present, the system will not hang or panic. The board should not be accessed again if IO_ENXIO_DEVICE_DOES_NOT_EXIST is returned. This routine assumes that the register is a long register.

Return Values

OK — The register was accessed successfully.

IO_ENXIO_DEVICE_DOES_NOT_EXIST — The board is not accessible.

Exceptions

None.

io_get_device_info

Syntax

```
status_type io_get_device_info (dev_code, dev_class,
                                interrupt_handler,
                                dit_entry_ptr)

io_device_code_type              dev_code;          /*READ ONLY*/
uc_device_class_enum_type        dev_class;         /*READ ONLY*/
io_service_interrupt_routine_ptr_type
                                 interrupt_handler;/*READ ONLY*/
word_address_ptr_type            dit_entry_ptr;     /*WRITE ONLY*/
```

Summary

This routine retrieves the device information pointer associated with the device specified by the device code and device class.

Parameters

dev_code — The device code of the device for which the device information pointer is to be retrieved.

dev_class — The device class of the device for which class the device information pointer is to be retrieved.

interrupt_handler — The service interrupt routine pointer stored at the beginning of the device information structure. This argument is used to ensure that the device information pointer returned by this routine really does belong to the requestor.

dit_entry_ptr — A pointer to where the device information pointer is to be returned.

Description

The device information pointer registered with the specified device is retrieved. If the specified device code has no device information registered to it, or if the service interrupt routine pointer in the device information structure does not match the service interrupt routine pointer supplied as an argument to this call, then an error status is returned and the returned device information pointer is undefined.

Return Value

OK — The device information pointer was successfully returned.

IO_ENXIO_DEVICE_CODE_OUT_OF_RANGE — The supplied device code is not supported on this system.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — No device information pointer was found for the device code or the device code does not belong to the requestor.

Exceptions

None.

io_map_device_number

Syntax

```
status_type  io_map_device_number (device_number,
                                   handle_ptr, unit_ptr)

io_device_number_type  device_number;     /*READ ONLY*/
bit32e_ptr_type        handle_ptr;        /*WRITE ONLY*/
uint16_ptr_type        unit_ptr;          /*WRITE ONLY*/
```

Summary

This routine translates major and minor device numbers to device handle and unit number.

Parameters

device_number — Contains the major and minor device numbers of the device.

handle_ptr — Pointer to the location where the device handle is returned.

unit_ptr — Pointer to the location where the unit number is returned.

Description

The system's **io_device_number_map** table is indexed by the given major device number to obtain the location of the minor device number table for this family of devices. The minor device number table is then indexed by the given minor number, and the device handle and unit are extracted and returned.

This routine is typically called by a driver's **dev_xxx_open** routine to map the major and minor device numbers to a specific device.

Return Value

OK — No errors occurred.

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — An active entry in the minor device number table does not exist at the offset specified by the minor device number argument.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_MAJOR_NUMBER_EXCEEDS_MAX — The major device number argument exceeds the maximum specified by **cf_io_device_driver_count**.

io_parse_device_spec

Syntax

```
boolean_type  io_parse_device_spec     (spec_ptr,
                                        dev_adapt_info_ptr,
                                        spec_size_ptr)
char_ptr_type                spec_ptr;          /*READ-WRITE*/
io_dev_adapt_info_ptr_type   dev_adapt_info_ptr;/*WRITE ONLY*/
int32_ptr_type               spec_size_ptr;     /*WRITE ONLY*/
```

Summary

Parse the device or adapter specification string for the positions of all specification components.

Parameters

spec_ptr — Pointer to a null terminated device or adapter specification string.

dev_adapt_info_ptr — Pointer to a structure where the pointers to the parsed string are to be returned.

spec_size_ptr — Pointer to the location where the length of the parsed device/adapter specification is returned. This location remains unchanged on error.

Description

This routine parses a device or adapter specification string (null terminated) into components. The components parsed for are: the device/adapter name, device code, and up to IO_DEV_ADAPT_MAX_PARAMS parameters. The parse leaves the original string intact. If a given component was not present, its pointer will point to a null character. Upon successful parsing, the length, in bytes, of the parsed specification will be returned in **spec_size_ptr**.

At a minimum the device/adapter specification must consist of a sequence of characters followed by an open and a close parenthesis. If a device code is present it must be prefixed with an IO_DEV_ADAPT_DEVICE_CODE_DELIMITER (at-sign, @), consist of two characters, and occupy the space immediately in front of the open parenthesis. Any number of parameters up to IO_DEV_ADAPT_MAX_PARAMS may be present, but they must be separated by commas. For more detailed information about the device and adapter specification, refer to Chapter 1. If the parsing fails, then all information within the **dev_adapt_info** structure must be assumed to be invalid.

Return Values

TRUE — The specification was successfully parsed.

FALSE — The parsing failed and the state of the **spec_ptr** string and the **dev_adapt_info_ptr** structure elements are unknown.

io_perform_reset

Syntax

```
void io_perform_reset  (reset_variety)

uc_reset_enum_type      reset_type;   /*READ ONLY*/
```

Summary

This routine performs the specified type of reset.

Parameters

reset_variety — An enumeration specifying which type of reset is to be done.

Description

This routine performs the specified type of reset. It uses the clock and await mechanisms, so it should not be used in an environment where this is not possible (for example, during shutdown or after reset).

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_BAD_RESET_TYPE — The parameter passed is not recognizable.

io_register_device_info

Syntax

```
status_type  io_register_device_info (dev_code,dev_class,
                                      info_ptr)

io_device_code_type        dev_code;  /*READ ONLY*/
uc_device_class_enum_type  dev_class; /*READ ONLY*/
word_address_type          info_ptr;  /*READ ONLY*/
```

Summary

This routine associates a pointer given in **info_ptr** with the device specified by the device code and device class. This process establishes an interrupt handler for the given device code.

Parameters

dev_code — The device code of the device with which a device information structure is to be associated.

dev_class — The device class of the device with which a device information structure is to be associated.

info_ptr — A pointer to the device information structure to be associated with the specified device code. The device information structure must contain a pointer to an interrupt handler as the first field. This interrupt handler becomes the handler for interrupts from the specified device code.

Description

This routine creates an entry in the appropriate device class device interrupt table (DIT) for the device code. If the slot in the DIT is already occupied or if the device code is larger than the maximum device code supported on this system, then an error is returned and the association between the device code and device information structure is NOT established.

Return Value

OK — The **device_info** was successfully registered.

IO_ENXIO_DEVICE_CODE_OUT_OF_RANGE — The supplied device code is not supported on this system. The **device_info** is not registered.

IO_ENXIO_DEVICE_CODE_ALREADY_ASSIGNED — An attempt was made to configure a device on a device code that is already assigned.

Exceptions

None.

Driver Daemon and Generic Daemon Routines

This section describes the routines you can call to interact with either the Driver Daemon or the Generic Daemon. Both daemons are processes that are permanently bound to a kernel virtual processor (VP) and are responsible for helping to service asynchronous I/O requests for all devices.

The routines described in this section are as follows:

- **io_queue_message_to_driver_demon**
- **io_specify_max_demon_messages**
- **io_queue_message_to_generic_demon**
- **io_specify_max_generic_demon_messages**

Routines beginning with **io** require the **i_io.h** include files.

Constants and Data Structures

No special constants or data structures are required by the routines in this section.

io_queue_message_to_driver_demon

Syntax

```
vp_ec_ptr_type  io_queue_message_to_driver_demon
                    (completion_routine_ptr, data, do_advance)

io_completion_routine_ptr_type completion_routine_ptr;
                                              /*READ ONLY*/
bit32e_type                    data;          /*READ ONLY*/
boolean_type                   do_advance;    /*READ ONLY*/
```

Summary

This routine queues a message to the Driver Daemon.

Parameters

completion_routine_ptr — A pointer to the value to go in the **completion_routine** field of the message. When the Driver Daemon dequeues this message, it will call the routine pointed to by the **completion_routine_ptr** field.

data — The value to go in the **data** field of the message. The Driver Daemon will use this value as a parameter when it calls the routine pointed to by **completion_routine_ptr**.

do_advance — A boolean indicating whether to advance the Driver Daemon eventcounter. See Description below.

Description

This routine queues a message to the I/O Driver Daemon. A free message is allocated from the Driver Daemon free list, filled in with the arguments given, and queued to the I/O Driver Daemon queue.

If **do_advance** is TRUE and the queue is empty, the null eventcounter pointer and the daemon eventcounter will be advanced by one.

If **do_advance** is FALSE, the daemon eventcounter is not advanced under any circumstances. Rather, if the message queued is the only message in the queue, the address of the daemon eventcounter is returned. Otherwise, the null eventcounter pointer is returned.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_DEMON_FREE_LIST_EMPTY — A free message could not be allocated from the Driver Daemon free list when needed. A device driver has used more messages than the number of messages it requested to be allocated for the daemon. See **io_specify_max_demon_messages**.

Remarks

The **do_advance** boolean is needed to handle timeouts. When a driver's timeout routine queues a message to the daemon, the eventcounter must not be advanced because the await table lock is already held by the await table routine that found the timeout entry. Instead, the eventcounter address is passed all the way back to the await table code, which will perform the advance when the await table is unlocked.

io_specify_max_demon_messages

Syntax

```
void    io_specify_max_demon_messages  (count)

uint32_type  count;        /*READ ONLY*/
```

Summary

This routine defines the maximum number of messages that the calling driver can have in the daemon's queue simultaneously.

Parameters

count — The maximum number of messages. The **count** parameter must be a positive integer; it is not possible to reduce the maximum number of messages.

Description

This routine allocates space for the specified number of messages and adds them to the daemon's free queue. It must be called by each device driver before that driver sends a message to the daemon. A given driver may make this call more than once if the maximum number of messages grows. The maximum number of messages may not be reduced.

In general, the maximum number of messages a driver will need depends on the number of devices it must service and on the way the driver handles and clears interrupts from those devices.

Return Values

None.

Exceptions

None.

io_queue_message_to_generic_demon

Syntax

```
vp_ec_ptr_type  io_queue_message_to_generic_demon
                                        (completion_routine_ptr,
                                         data, do_advance)

io_completion_routine_ptr_type
                completion_routine_ptr;/*READ ONLY*/
bit32e_type     data;                  /*READ ONLY*/
boolean_type    do_advance;            /*READ ONLY*/
```

Summary

This routine queues a message to the Generic Daemon.

Parameters

completion_routine_ptr — The value to go in the **completion_routine** field of the message.

data — The value to go in the data field of the message.

do_advance — A boolean indicating whether to advance the generic daemon eventcounter. See "Description" below.

Description

This routine queues a message to the Generic Daemon. A free message is allocated from the Generic Daemon free list, is filled in with the arguments given, and queued to the Generic Daemon queue.

If the **do_advance** boolean is TRUE, the return value will be the null eventcounter pointer and the Generic Daemon eventcounter will be advanced if the message queue is the only message in the queue.

If the **do_advance** boolean is FALSE, the Generic Daemon eventcounter is not advanced under any circumstances. Rather, if the message queued is the only message in the queue, the address of the demon eventcounter is returned. Otherwise, the null eventcounter pointer is returned.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_GENERIC_DEMON_FREE_LIST_EMPTY — A free message could not be allocated from the Generic Daemon free list when needed. The device driver has used more messages than the number of messages it requested to be allocated for the daemon.

io_specify_max_generic_demon_messages

Syntax

```
void   io_specify_max_generic_demon_messages  (count)

uint32_type  count; /*READ ONLY*/
```

Summary

This routine informs the Generic Daemon of the maximum number of messages that the calling driver will have in the daemon's queue.

Parameters

count — The maximum number of messages. This value must be a positive integer. Once **count** has been set you can add to but not reduce the maximum number of messages.

Description

This routine allocates space for the specified number of messages and adds them to the Generic Daemon's free queue. It must be called by each device driver before that driver sends a message to the Generic Daemon. A given driver may make this call more than once if the maximum number of messages grows. However, the maximum number of messages may not be reduced.

In general, the maximum number of messages a driver will need depends on the number of devices it must service and on the way the driver handles and clears interrupts from those devices.

Return Values

None.

Exceptions

None.

Error Encoding and Logging Routines

This section describes a macro you can use to create system-compatible error numbers for your device's errors and a routine you can use to log errors to the system error facility.

You use the **io_err_log_error** routine to queue your driver's error messages on the psuedodevice **err**(7) until they can be retrieved by the system error daemon and written to system error log. You pass your message to **io_err_log_error** in the form of a **printf** string with a format parameter and accompanying variables.

The error encoding macro helps you integrate system compatible **errnos** into your status codes. Compatible **errnos** can be passed all the way back to the user level. In normal processing, once a status is sent to the user-level, the **errno** is extracted from the status and returned to the user.

To create a status containing an **errno**, use the following convention:

SS_EEEE_DDDDDD

Here, SS is a two- to four-letter subsystem name; your device driver will use "DEV." EEEE is the full name of the **errno** to be returned to the user; use standard **errnos** found in **errno.h**. DDDDDD is a description of the state that caused the status to be returned. An example of a status code is as follows:

IO_EIO_DEVICE_TIMED_OUT

If you do not want to use the status to return an **errno** to the user, pass SC_NO_ERRNO to this macro. Higher levels of code will deal with the status before it gets back to the user.

Whenever possible, use I/O statuses already defined in **dev_status_codes.h**. For statuses that you will handle within your driver, use DEV as the subsystem, SC_NO_ERRNO as the **errno**, and a higher sequence number than the last used in **dev_status_codes.h**. The **DEV_ENCODE** macro in **dev_status_codes.h** will set up the status for you correctly. **dev_status_codes.h** is in **aviion/dev**.

Note that the convention through the rest of the kernel is to use STATUS instead of the EEEE **errno** when no errno is used. For example, IO_STATUS_REQUEST_STILL_IN_PROGRESS will not return a status to the user. An example of how to create a new status for your device is as follows:

```
#define DEV_STATUS_FOO_DEVICE_IN_BAR_STATE DEV_ENCODE(SC_NO_ERRNO,0107)
```

The routines described in this section are as follows:

- **SC_ENCODE_STATUS**

- **io_err_log_error**

Routines beginning with **sc** require the **i_sc.h** include file and those beginning with **io** require **i_io.h**.

Constants and Data Structures

This section defines the "no error" constant.

NOTE: Because constants and data structures are subject to change, you must verify exact variable definitions in the appropriate include file (for example, check **i_sc.h** for structures beginning with the **sc** acronym). Chapter 3 lists the various include files.

SC_NO_ERRNO

```
#define SC_NO_ERRNO                    0
```

Use this value to indicate that the status does not contain an **errno** value.

SC_ENCODE_STATUS

Syntax

```
#define    SC_ENCODE_STATUS (subsystem_id, errno, sequence)

(status_type)((subsystem_id << 18) + (errno << 9) + sequence))
```

Summary

This macro constructs a status value from the subsystem ID for a subsystem, the **errno** to be inserted into the status, and a sequence number to distinguish multiple statuses with the same subsystem ID.

Parameters

subsystem_id — The subsystem ID for the subsystem.

errno — The **errno** that is to be inserted into the status. The value of errno must be less than or equal to 511.

sequence — A sequence number to distinguish multiple statuses with the same subsystem ID. The sequence number must have a value between 1 and 511.

Description

The status is constructed so the sequence number occupies bits 0-8, the errno occupies bits 9-17, and the subsystem ID occupies bits 18-26. Bits 27-31 are unused and set to 0. The **errno** parameter specifies the **errno** value that will be returned to the user. To get the subsystem ID and sequence numbers, the user should call the **dg_ext_errno** system call.

Return Value

status — The newly encoded status.

io_err_log_error

Syntax

```
boolean_type  io_err_log_error  (priority, format,
                                 value_00,value_01,value_02,
                                 value_03,value_04,value_05,
                                 value_06,value_07,value_08,
                                 value_09,value_10,value_11,
                                 value_12,value_13,value_14,
                                 value_15,value_16,value_17)

uint32e_type  priority;       /* READ ONLY  */
char_ptr_type format;         /* READ ONLY  */
bit32e_type   value_00;       /* READ ONLY  */
bit32e_type   value_01;       /* READ ONLY  */
bit32e_type   value_02;       /* READ ONLY  */
bit32e_type   value_03;       /* READ ONLY  */
bit32e_type   value_04;       /* READ ONLY  */
bit32e_type   value_05;       /* READ ONLY  */
bit32e_type   value_06;       /* READ ONLY  */
bit32e_type   value_07;       /* READ ONLY  */
bit32e_type   value_08;       /* READ ONLY  */
bit32e_type   value_09;       /* READ ONLY  */
bit32e_type   value_10;       /* READ ONLY  */
bit32e_type   value_11;       /* READ ONLY  */
bit32e_type   value_12;       /* READ ONLY  */
bit32e_type   value_13;       /* READ ONLY  */
bit32e_type   value_14;       /* READ ONLY  */
bit32e_type   value_15;       /* READ ONLY  */
bit32e_type   value_16;       /* READ ONLY  */
bit32e_type   value_17;       /* READ ONLY  */
```

Summary

If an error queue element is available on the free queue, the indicated message is formatted and copied into it, and the element is placed on the ready queue.

Parameters

priority — The priority of this error message. See **syslog.h** for priority definitions.

format — A **printf** format string that specifies the format to be used for the message.

value_00-17 — The parameters to be substituted into the **printf** string format.

Description

If the error daemon **syslogd** has not opened the **err** pseudodevice, the message is formatted and printed on the console. If an empty record is available, the priority number and the message are formatted into it. Long messages are truncated. The formatted record is placed on the ready queue, and the event counter is advanced. If there are no available records, the message is ignored.

Return Values

TRUE — If an error queue element was available.

FALSE — If no error queue element was available.

Select Manager Routines

The "select" operations allow multiple users to wait for I/O from a device without directly suspending. The kernel's select routines (select manager) help your driver manage select operations by maintaining lists of outstanding select operations for each device.

The select manager has the following routines and features:

- For each device, the select manager keeps a list of the processes interested in I/O events on that device. During initialization you allocate a data structure of type **io_select_list_type** for each physical device. This structure will hold the list of processes (select list) interested in I/O events on that device.

- Initialize each select list by calling **io_select_init** before the list is used in any other select manager call.

- When a user makes a select request, control will be passed to the driver's **dev_xxx_select**. If the select operation cannot be completed immediately, then the driver should place the request on the select list for the device by calling **io_select_register**. This routine will put an entry in the select list for the device with the intent (type of I/O) of the select and a pointer to the process's select eventcounter.

- When an I/O event occurs on the device (for example, the driver receives data, learns the device is ready for writing, or discovers an exceptional condition on a device), the driver should call **io_select_satisfy** with the pertinent information. This will advance the eventcounters of the processes that are interested in that particular sort of event. Note that **io_select_satisfy** leaves the process's entry on the select list.

- After **io_select_satisfy** finishes, control will return to the driver's **dev_xxx_select** as a result of the event. If appropriate, **dev_xxx_select** can then remove the calling process from the select list by calling **io_select_cancel**.

CAUTION:

It is essential that you note the following items:

It is important that only one thread of control access a given select list at a time. The kernel select manager routines do not lock the select list structures; therefore, the device driver should lock this structure so that multiple threads cannot access the select lists.

Io_select_satisfy will frequently be called from an interrupt service routine. If you call it from a service routine, be sure to mask out the device's interrupts before you call other routines with its select list. If

you don't, the interrupt level may encounter a halfway processed list.

The following routines are described in this section:

- **io_select_cancel**
- **io_select_init**
- **io_select_register**
- **io_select_satisfy**

Routines beginning with **io** require the **i_io.h** include file.

Constants and Data Structures

See Chapter 4 for a definition of **io_select_intent_type**.

io_select_cancel

Syntax

```
io_select_intent_type io_select_cancel (select_list_ptr,
                                        ec_ptr)

io_select_list_ptr_type  select_list_ptr;    /*WRITE ONLY*/
vp_ec_ptr_type           ec_ptr;             /*READ ONLY*/
```

Summary

This routine removes the process identified by **ec_ptr** from the select list.

Parameters

select_list_ptr — A pointer to a select list.

ec_ptr — A pointer to a process's select eventcounter.

Return Values

The type of select intent satisfied (or none).

io_select_init

Syntax

```
void     io_select_init (select_list_ptr)

io_select_list_ptr_type  select_list_ptr;   /*WRITE ONLY*/
```

Summary

This routine initializes the given select list.

Parameters

select_list_ptr — A pointer to a select list.

Return Values

None.

io_select_register

Syntax

```
void   io_select_register (select_list_ptr, intent, ec_ptr)

io_select_list_ptr_type  select_list_ptr;    /*READ/WRITE/*
io_select_intent_type    intent;             /*READ ONLY*/
vp_ec_ptr_type           ec_ptr;             /*READ ONLY*/
```

Summary

This routine registers a select with the given intent and eventcounter on the given select list.

Parameters

select_list_ptr — A pointer to a select list.

intent — The intent of the select.

ec_ptr — A pointer to the select eventcounter of the selecting process.

Description

See the "Constants and Data Structures" section for a list of defines for **intent**.

Return Values

None.

io_select_satisfy

Syntax

```
void   io_select_satisfy (select_list_ptr, intent)

io_select_list_ptr_type  select_list_ptr;  /*WRITE ONLY*/
io_select_intent_type    intent;           /*READ ONLY*/
```

Summary

This routine searches the given select list for processes interested in the given I/O event. The select eventcounters for those processes are advanced.

Parameters

select_list_ptr — A pointer to a select list.

intent — The type of select to satisfy.

Return Values

None.

Miscellaneous Driver Routines

The following routines described in this section are:

- **fs_check_self_id**
- **io_hex_str_to_int**
- **misc_format_line**
- **pm_is_super_user**
- **sc_panic**

Routines beginning with **fs**, **misc**, **pm**, **sc**, and **io** require the **i_fs.h**, **i_misc.h**, **i_pm.h**, **i_sc.h**, and **i_io.h** include files, respectively.

Constants and Data Structures

There are no special constants or data structures required for these routines.

fs_check_self_id

Syntax

```
boolean_type      fs_check_self_id  (blocks_ptr, self_id_ptr,
                                     count_ptr)

pointer_to_any_type   blocks_ptr;      /*READ ONLY*/
df_self_id_ptr_type   self_id_ptr;     /*READ ONLY*/
uint32_ptr_type       count_ptr;       /*READ/WRITE*/
```

Summary

This routine checks the self-ID for the given set of blocks.

Parameters

blocks_ptr — Pointer to the beginning of the first block to be checked.

self_id_ptr — Pointer to the self-ID that the first block is expected to have.

count_ptr — On input, the number of bytes to be checked. On output, the number of bytes that checked out OK. On both input and output, **count_ptr** must be a multiple of the block size, though this is not checked.

Description

A self-ID is an identifying number used to identify different non-data disk blocks used in disk administration (for example, header blocks). For each block, this routine checks its self-ID against the prototype self-ID. If any block fails, FALSE is returned along with the number of bytes that passed the check. If all blocks pass, then TRUE is returned along with the number of bytes that were checked.

Return Value

TRUE — All blocks were successfully checked.

FALSE — At least one block failed a self-ID check.

io_hex_str_to_int

Syntax

```
boolean_type      io_hex_str_to_int   (str_ptr, int_value_ptr)
char_ptr_type     str_ptr;            /*READ  ONLY*/
uint32_ptr_type   int_value_ptr;      /*WRITE ONLY*/
```

Summary

Return the integer value of the null terminated hexadecimal string at **str_ptr**.

Parameters

str_ptr — A pointer to the beginning of the string to convert.

int_value_ptr — Pointer to location where the integer value is to be returned.

Description

Scan a string **str_ptr** consisting only of the characters '0' - '9', 'a' - 'f', and 'A' - 'F', and terminated with a null character, returning its unsigned 32-bit value at **int_value_ptr**. If any other characters are encountered or the value exceeds what can be expressed in a 32-bit unsigned value then **int_value_ptr** is unchanged and an error is returned.

Return Values

FALSE — Successful conversion occurred.

TRUE — The string conversion failed.

misc_format_line

Syntax

```
uint32_type misc_format_line
(result_buf, rb_size, format, value_00, value_01, value_02
                              value_03, value_04, value_05
                              value_06, value_07, value_08
                              value_09, value_10, value_11
                              value_12, value_13, value_14
                              value_15, value_16, value_17
                              value_18, value_19)

char_ptr_type  result_buf;    /*WRITE ONLY*/
uint32_type    rb_size;       /*READ ONLY*/
char_ptr_type  format;        /*READ ONLY*/
bit32e_type    value_00;      /*READ ONLY*/
bit32e_type    value_01;      /*READ ONLY*/
bit32e_type    value_02;      /*READ ONLY*/
bit32e_type    value_03;      /*READ ONLY*/
bit32e_type    value_04;      /*READ ONLY*/
bit32e_type    value_05;      /*READ ONLY*/
bit32e_type    value_06;      /*READ ONLY*/
bit32e_type    value_07;      /*READ ONLY*/
bit32e_type    value_08;      /*READ ONLY*/
bit32e_type    value_09;      /*READ ONLY*/
bit32e_type    value_10;      /*READ ONLY*/
bit32e_type    value_11;      /*READ ONLY*/
bit32e_type    value_12;      /*READ ONLY*/
bit32e_type    value_13;      /*READ ONLY*/
bit32e_type    value_14;      /*READ ONLY*/
bit32e_type    value_15;      /*READ ONLY*/
bit32e_type    value_16;      /*READ ONLY*/
bit32e_type    value_17;      /*READ ONLY*/
bit32e_type    value_18;      /*READ ONLY*/
bit32e_type    value_19;      /*READ ONLY*/
```

Summary

This routine provides limited **sprintf**(3) functionality; it formats output and performs value substitutions. It formats a line, creating a string by substituting values according to field descriptors. The field descriptors in the input format are a subset of the field descriptors that the standard library routine **printf** provides. Currently provided are **%c, %s, %d, %o, %x, %u** and the ability to specify field length and zero padding.

Parameters

result_buf — Resulting formatted output placed here.

rb_size — Size of the buffer.

format — The format string. This format string is the same as those used in **printf**.

value_00...value_19 — Place holder for 0 to 19 format substitution values. Use **value_00** for the first value, **value_01** for the second value, etc.

Return Values

None.

Exceptions

None.

pm_is_super_user

Syntax

```
boolean_type pm_is_super_user ()
```

Summary

This routine determines whether the calling process has superuser permission. If so, it notifies the kernel that the process has used superuser permission so it can be recorded for accounting information.

Parameters

None.

Return Value

TRUE — Caller is superuser.

FALSE — Caller is not superuser.

Exceptions

None.

Abort Conditions

None.

sc_panic

Syntax

```
void  sc_panic     (panic_code)

sc_panic_code_type  panic_code;    /*READ ONLY*/
```

Summary

This routine is the panic routine that you call when serious errors or inconsistencies are detected. A panic message is written to the system console, and the emergency shutdown sequence is entered.

Parameters

panic_code — A value identifying the cause of the panic. This value will be written to the system console along with the panic message. Non-standard devices should use a panic code between 0 and 511 decimal to avoid collision with existing system panic codes.

Description

The panic lock is obtained to ensure that only one processor enters the panic and emergency shutdown code. If any other processors are running, they are stopped. The routine **sc_write_line** is called to write the panic message to the system console and the emergency shutdown routine is entered.

Return Values

None.

Exceptions

None.

Abort Conditions

None.

Nodevice Routine Stubs

You can call the routines listed below anytime your driver does not process the I/O operation indicated in the routine's name. For example, if your device cannot be used as a dump device, you can use the **io_nodevice_open_dump** routine instead of supplying your own open dump stub. The routines in this section generally return at least an error and, in some cases, a panic. Before you use one of these routines, make sure its error return is acceptable and appropriate for your device.

The routines supplied support both block and character operations so they can serve as stubs for both types of requests. The following routines are described in this section:

- **io_nodevice_open**
- **io_nodevice_close**
- **io_nodevice_read_write**
- **io_nodevice_select**
- **io_nodevice_ioctl**
- **io_nodevice_start_io**
- **io_nodevice_configure**
- **io_nodevice_deconfigure**
- **io_nodevice_name_to_device**
- **io_nodevice_device_to_name**
- **io_nodevice_open_dump**
- **io_nodevice_write_dump**
- **io_nodevice_read_dump**
- **io_nodevice_close_dump**
- **io_nodevice_powerfail**
- **io_nodevice_mmap**
- **io_nodevice_munmap**
- **io_nodevice_maddmap**

- **io_nodevice_service_interrupt**

io_nodevice_open

Syntax

```
status_type io_nodevice_open (device_number, channel_flags,
                              device_handle_ptr)

io_device_number_type       device_number;     /*READ ONLY*/
io_channel_flags_type       channel_flags;     /*READ ONLY*/
io_device_handle_ptr_type   device_handle_ptr; /*WRITE ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to open a non-existent device.

Parameters

device_number — The major and minor device number from the special file that is being opened.

channel_flags — A set of flags specifying whether the I/O to the device will be reads, writes, or both.

device_handle_ptr — A pointer to the location where the device handle is to be returned. Because **io_nodevice_open** always fails, no device handle is ever returned.

Description

This routine returns a status indicating that the device does not exist.

Return Value

IO_ENXIO_DEVICE_DOES_NOT_EXIST — This value is always returned.

Exceptions

None.

io_nodevice_close

Syntax

```
status_type io_nodevice_close (device_handle, channel_flags)

io_device_handle_type  device_handle;  /*READ ONLY*/
io_channel_flags_type  channel_flags;  /*READ ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

device_handle — The device handle for the device that is being closed.

channel_flags — The flags with which the device was opened.

Description

Panic is invoked.

Return Values

None.

Exceptions

None.

Abort Conditions

This routine always panics with the following panic code:

IO_PANIC_NODEVICE_CLOSE — An attempt was made to close a major device number for which no driver exists.

io_nodevice_read_write

Syntax

```
status_type     io_nodevice_read_write (request_info_ptr)

io_request_info_ptr_type  request_info_ptr;  /*READ ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

request_info_ptr — A pointer to a packet containing the information necessary to specify a read or write request.

Description

Panic is invoked.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always called with the following panic code:

IO_PANIC_NODEVICE_READ_WRITE — An attempt was made to do a read or write operation on a major device number for which no driver exists.

io_nodevice_select

Syntax

```
void  io_nodevice_select (device_handle, select,
                          ec_ptr, intent_ptr)

io_device_handle_type     device_handle;  /*READ ONLY*/
boolean_type              select;         /*READ ONLY*/
vp_ec_ptr_type            ec_ptr;         /*READ ONLY*/
io_select_intent_ptr_type intent_ptr;     /*READ WRITE*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

device_handle — The device handle of the device that is the target of select. This handle must be a device handle that was returned by the open routine of this driver.

select — If TRUE, this is the start of a select operation; conditions that are not immediately TRUE should be recorded so that the eventcounter can be advanced when they become TRUE. If FALSE, this is the end of a select operation; any previously remembered conditions should be forgotten.

ec_ptr — Specifies the eventcounter to be advanced by the driver when the select is satisfied if it is not immediately satisfied.

intent_ptr — On input, **intent_ptr** specifies whether a select is to be instituted for a combination of read, write, or exceptional conditions.

Description

Panic is invoked.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always invoked with the following panic code:

IO_PANIC_NODEVICE_SELECT — An attempt was made to do a select operation on a device for which no driver exists.

io_nodevice_ioctl

Syntax

```
status_type io_nodevice_ioctl  (device_handle, command,
                                parameter, return_value_ptr)

io_device_handle_type     device_handle;  /*READ ONLY*/
bit32e_type               command;        /*READ ONLY*/
bit32e_type               parameter;      /*READ/WRITE*/
int32e_ptr_type           parameter;      /*WRITE ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

device_handle — The device handle of the device that is the target of the I/O control operation.

command — A command to the device. The interpretation of the command is specific to the driver.

parameter — An argument to the command. The interpretation of the parameter is specific to the driver and the command. The parameter may be used to transfer information in either direction between the caller and the device. In particular, it may be a pointer to a buffer supplied by the caller.

return_value_ptr — A pointer to the value to be returned to the user.

Description

This routine causes a system panic.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always invoked with the following panic code:

IO_PANIC_NODEVICE_IOCTL — An attempt was made to do an ioctl operation on a device for which no driver exists.

io_nodevice_start_io

Syntax

```
status_type  io_nodevice_start_io (op_record_ptr)

io_operation_record_ptr_type  op_record_ptr;  /*READ ONLY*/
```

Summary

This routine is a stub for handling erroneous I/O operations. Either the driver does not support this operation or the device does not exist. Calling this routine will cause a system fatal error.

Parameters

op_record_ptr — A pointer to the operation record for the asynchronous request. The operation record contains fields indicating the minor device that is the target of the operation, the operation to be performed, the offset on the device from which the operation is to commence, the size of the transfer, the address of the main memory buffer, and the address of the routine that is to be called when the operation completes.

Description

This routine causes a system panic.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic is always invoked with the following panic code:

IO_PANIC_NODEVICE_START_IO — An attempt was made to do an start_io operation on a device for which no driver exists.

io_nodevice_configure

Syntax

```
status_type io_nodevice_configure (device_name_ptr,
                                   major_number)

char_ptr_type                 device_name_ptr;/*READ ONLY*/
io_major_device_number_type   major_number;   /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to configure a non-existent device.

Parameters

device_name_ptr — A pointer to the character string name of the device to be configured.

major_number — The major device number on which the device is to be configured.

Description

This routine always returns an error.

Return Value

IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED — This status is always returned.

Exceptions

None.

Abort Conditions

None.

io_nodevice_deconfigure

Syntax

```
status_type     io_nodevice_deconfigure (device_name_ptr)

char_ptr_type   device_name_ptr;  /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to deconfigure a non-existent device.

Parameters

device_name_ptr — A pointer to the null-terminated string specifying the device to be deconfigured.

Description

This routine always returns an error.

Return Value

IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED — This status is always returned.

Exceptions

None.

io_nodevice_name_to_device

Syntax

```
status_type io_nodevice_name_to_device (device_name_ptr,
                                        number_ptr)

char_ptr_type               device_name_ptr;/*READ ONLY*/
io_device_number_ptr_type   number_ptr;     /*WRITE ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to do name-to-device conversion on a non-existent device.

Parameters

device_name_ptr — A pointer to the null-terminated device name that is to be translated.

number_ptr — A pointer to where the corresponding device number is to be written.

Description

This routine always returns an error.

Return Value

IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED — This status is always returned.

Exceptions

None.

io_nodevice_device_to_name

Syntax

```
status_type io_nodevice_device_to_name  (device_number,
                                         name_ptr, size)

io_device_number_type   device_number;  /*READ ONLY*/
char_ptr_type           name_ptr;       /*WRITE ONLY*/
uint32_type             size;           /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to do device-to-name conversion on a non-existent device.

Parameters

device_number — The device number to be translated into a device name character string.

name_ptr — A pointer to where the null-terminated character string name is to be written.

size — The maximum number of bytes, including the terminating null, that is to be written to **name_ptr**.

Description

This routine always returns an error.

Return Value

IO_ENXIO_DEVICE_IS_NOT_CONFIGURED — This status is always returned.

Exceptions

None.

io_nodevice_open_dump

Syntax

```
status_type     io_nodevice_open_dump (device_name)

char_ptr_type   device_name;  /*READ ONLY*/
```

Summary

This routine is a stub routine that returns an error if an attempt is made to dump to a non-existent device or to a device that does not support dumps.

Parameters

device_name — The character string name of the device to which the dump is being written.

Description

This routine always returns an error.

Return Value

IO_STATUS_DUMP_NOT_SUPPORTED — This status indicates that the device does not support dumps. This status is always returned.

Exceptions

None.

Abort Conditions

None. This routine must not panic because it is invoked as part of the panic sequence.

io_nodevice_write_dump

Syntax

```
status_type io_nodevice_write_dump (buffer_ptr, buffer_size)

pointer_to_any_type  buffer_ptr;   /*READ ONLY*/
uint32_type          buffer_size;  /*READ ONLY*/
```

Summary

This routine is a stub for handling devices that do not exist. It is a system fatal error to call this routine.

Parameters

buffer_ptr — A pointer to the buffer of data to be written to the system dump.

buffer_size — The size, in bytes, of the buffer.

Description

This routine should never be called because **io_nodevice_open_dump** always fails.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_WRITE_DUMP — An attempt was made to write dump information to a non-existent device.

io_nodevice_read_dump

Syntax

```
status_type io_nodevice_read_dump  (buffer_ptr, buffer_size)

pointer_to_any_type   buffer_ptr;     /* WRITE ONLY  */
uint32_type           buffer_size;    /* READ  ONLY  */
```

Summary

This function is a stub for handling devices that do not exist. It is a system fatal error to call this function.

Parameters

buffer_ptr — A pointer to the buffer to which data is to be read.

buffer_size — The size, in bytes, of the buffer.

Description

This function should never be called because **io_nodevice_open_dump** always fails.

Return Value

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error codes:

IO_PANIC_NODEVICE_READ_DUMP — An attempt was made to read dump information from a non-existent device.

io_nodevice_close_dump

Syntax

```
status_type io_nodevice_close_dump ()
```

Summary

This routine is a stub for handling devices that do not exist. It is a system fatal error to call this routine.

Parameters

None.

Description

This routine should never be called because **io_nodevice_open_dump** always fails.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_CLOSE_DUMP — An attempt was made to close a non-existent dump device.

io_nodevice_powerfail

Syntax

```
status_type     io_nodevice_powerfail  ()
```

Summary

This routine is a stub routine that simply returns OK, because there is nothing to do in order to perform powerfail restart on nodevice.

Parameters

None.

Description

The status OK is returned.

Return Value

OK — This value is always returned.

Exceptions

None.

Abort Conditions

None.

io_nodevice_mmap

Syntax

```
status_type     io_nodevice_mmap     ()
```

Summary

This routine is a stub for handling the **mmap** system call. The **errno** EINVAL is returned.

Parameter

None.

Description

This routine always returns an error.

Return Value

IO_EINVAL_MMAP_NOT_SUPPORTED — The **mmap** operation is not supported for this device.

Exceptions

None.

io_nodevice_munmap

Syntax

```
status_type     io_nodevice_munmap      ()
```

Summary

This routine is a stub for handling the **munmap** system call. The **errno** EINVAL is returned.

Parameters

None.

Description

This routine always returns an error.

Return Value

IO_EINVAL_MUNMAP_NOT_SUPPORTED — The **munmap** operation is not supported for this device.

Exceptions

None.

io_nodevice_maddmap

Syntax

```
status_type    io_nodevice_maddmap    ()
```

Summary

This function is a stub for incrementing reference counts to memory mapped sections. The **errno** EINVAL is returned.

Parameters

None.

Description

EINVAL is returned.

Return Value

IO_EINVAL_MMAP_NOT_SUPPORTED — The **maddmap** operation is not supported for this device.

Exceptions

None.

io_nodevice_service_interrupt

Syntax

```
void  io_nodevice_service_interrupt  (device_code, device_class)

io_device_code_type       device_code;  /*READ ONLY*/
uc_device_class_enum_type device_class;  /*READ ONLY*/
```

Summary

This routine handles unexpected interrupts from devices that are not configured into the kernel.

Parameters

device_code — The device code of the interrupting device.

device_class — The device class of the interrupting device.

Description

This routine runs at interrupt level. It handles interrupts from devices that are not configured and, therefore, which should not be generating interrupts. This routine does not obey the standard interface for service interrupt routines. Because it must service interrupts from all devices, it uses a device code as the argument instead of a device information structure pointer.

Return Values

None.

Exceptions

None.

Abort Conditions

Panic may be invoked with the following error code:

IO_PANIC_NODEVICE_INTERRUPT_OVERRUN — Too many unexpected interrupts were received in too short a time. This panic probably indicates the existence of a hardware problem that is generating spurious interrupts.

End of Chapter

Appendix A
A Sample SCSI Device Driver

This appendix gives sample code for a disk driver. We include the type definitions, global data definitions, driver supplied I/O routines, an example system file and an example master file. For this example, **xxx** is replaced by **sd**.

NOTE: The code provided here is only a sample. It is not guaranteed to be either complete or operational.

Data Definitions: dev_sd_def.h

```
/*<-------------------------------------------------------*/
/*              dev_sd_def.h                             */
/*>-------------------------------------------------------*/

/*.Contents[=-------------------------------------------------------------
/*
/* DEV_SD_STANDARD_SECTOR_SIZE -- subsystem
/* DEV_SD_DEFAULT_SCSI_ID -- subsystem
/* DEV_SD_DEVICE_TYPES_SUPPORTED -- subsystem
/* DEV_SD_DEFAULT_UNIT_NUMBER -- subsystem
/* DEV_SD_MAX_BUFFERED_CONCURRENT_UNIT_REQUESTS -- subsystem
/* DEV_SD_MAX_CONCURRENT_UNIT_REQUESTS -- subsystem
/* DEV_SD_MAX_INQUIRY_RETRIES -- subsystem
/* DEV_SD_NULL_UNIT_INFO_PTR -- subsystem
/* DEV_SD_NULL_BUFFER_PTR -- subsystem
/* DEV_SD_NULL_ADAPTER_HANDLE -- subsystem
/* DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_OFFSET -- subsystem
/* DEV_SD_FLOPPY_INQUIRY_CONFIG_5_25 -- subsystem
/* DEV_SD_FLOPPY_INQUIRY_CONFIG_MIXED -- subsystem
/* DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_SIZE -- subsystem
/* DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR")subsystem
/* DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR_MIXED")subsystem
/* DEV_SD_FLOPPY_5_25INCH_1200KB_TRANSFER_RATE -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_1200KB_SECTORS_PER_TRACK -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_1200KB_NUMBER_OF_CYLINDERS  -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_1200KB_STEP_PULSES_PER_CYLINDER")subsystem
/* DEV_SD_FLOPPY_DEFLAULT_WRITE_PRECOMPENSATION_VALUE")subsystem
/* DEV_SD_FLOPPY_MEDIUM_TYPE_96_135TPI_7958BPR -- subsystem
/* DEV_SD_FLOPPY_720KB_TRANSFER_RATE -- subsystem
/* DEV_SD_FLOPPY_720KB_SECTORS_PER_TRACK -- subsystem
/* DEV_SD_FLOPPY_720KB_NUMBER_OF_CYLINDERS -- subsystem
/* DEV_SD_FLOPPY_720KB_STEP_PULSES_PER_CYLINDER -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_360KB_TRANSFER_RATE -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_360KB_SECTORS_PER_TRACK -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_360KB_NUMBER_OF_CYLINDERS -- subsystem
/* DEV_SD_FLOPPY_5_25INCH_360KB_STEP_PULSES_PER_CYLINDER")subsystem
/* DEV_SD_FLOPPY_MEDIUM_TYPE_3_50INCH_135TPI_15916BPR")subsystem
/* DEV_SD_FLOPPY_3_50INCH_1440KB_TRANSFER_RATE -- subsystem
/* DEV_SD_FLOPPY_3_50INCH_1440KB_SECTORS_PER_TRACK -- subsystem
/* DEV_SD_FLOPPY_3_50INCH_1440KB_NUMBER_OF_CYLINDERS")subsystem
/* DEV_SD_FLOPPY_3_50INCH_1440KB_STEP_PULSES_PER_CYLINDER")subsystem
/* DEV_SD_FLOPPY_MODE_SELECT_TEST_SECTOR -- subsystem
```

```
/* dev_sd_unit_info_type -- subsystem
/* dev_sd_unit_info_ptr_type -- subsystem
/* dev_sd_worm_optimem_mode_buffer_type -- subsystem
/* dev_sd_worm_optimem_mode_buffer_ptr_type -- subsystem
/* DEV_SD_DISK_TYPE_RIGID -- subsystem
/* DEV_SD_DISK_TYPE_FLOPPY -- subsystem
/* DEV_SD_DISK_TYPE_ERASABLE_OPTICAL -- subsystem
/* DEV_SD_DISK_TYPE_WORM -- subsystem
/* DEV_SD_DISK_TYPE_READ_ONLY -- subsystem
/*
/*.Description
/*
/*   This module contains definitions that support the SD (SCSI disk)
/*   class disk device driver, which is in module dev_sd_driver.c.
/*
/*   All of the definitions herein describe host-side only data
/*   structures that are used by the driver to keep track of the
/*   state of the disk and outstanding requests, and are subject
/*   to change as the driver changes.
/*
/*
/*   SCSI disk literals.
/*

#define DEV_SD_STANDARD_SECTOR_SIZE   ((uint16_type)0x00000200)
    /*
     * This value is the number of bytes per sector on a SCSI disk. It
     * is used to convert the byte count in an I/O request into a sector
     * count to give to the disk controller.
     */

#define DEV_SD_DEFAULT_SCSI_ID   ((uint16_type)0x00000000)
    /*
     * The default SCSI id that is used to identify a disk if a SCSI id
     * is not present in a device specification.
     *
     */

#define DEV_SD_DEFAULT_UNIT_NUMBER   ((uint16_type)0x00000000)
    /*
     * The default unit number that is used to identify a disk if a unit
     * number is not present in a device specification.
     */

#define DEV_SD_MAX_BUFFERED_CONCURRENT_UNIT_REQUESTS   0x00000001
    /*
     * The maximum number of requests that can be executing
     * concurrently on a single SCSI disk unit when disk request
     * buffering is being performed.
     */

#define DEV_SD_MAX_CONCURRENT_UNIT_REQUESTS   0x00000020
    /*
     * The maximum number of requests that can be executing
     * concurrently on a single SCSI disk unit. Although SCSI
     * disks can only process on request at a time, the SCSI interface
     * to the disk can queue up multiple requests and optimize the
     * way that the requests are actually issued to the disk.
     */

#define DEV_SD_MAX_INQUIRY_RETRIES   0x00000002
    /*
     * The maximum number of times that the SCSI Inquiry command is
     * retried during the disk configure operation before deciding
     * that a device is not present at the LUN.
     */

#define DEV_SD_DEVICE_TYPES_SUPPORTED   ((bit8e_type)
    /*
     * The set of SCSI devices that are supported by the SD driver.
     */

#define DEV_SD_NULL_UNIT_INFO_PTR          ((dev_sd_unit_info_ptr_type)DEFAULT_NULL_LINK)
    /*
     * A null SCSI disk unit information table pointer. It is used to
```

```
     * identify unallocated unit information table entries.
     */

#define DEV_SD_NULL_BUFFER_PTR   ((pointer_to_any_type)DEFAULT_NULL_LINK)
    /*
     * A null buffer pointer used in the adapter request block
     * buffer vector when no data buffer is required.
     */

#define DEV_SD_NULL_ADAPTER_HANDLE
     ((io_device_handle_type)DEFAULT_NULL_LINK)
    /*
     * A null adapter handle, used as a place holder in the unit
     * information structure.
     */

/*.Literal_Section[=------------------------------------------------
/*
/*   TEAC FC-1 Configuration Literals.
/*
/*.Description
/*
/*   These constants are used to interpret inquiry data returned
/*   from the TEAC FC-1 SCSI to floppy (SA450) adapter card.
/*   Inquiry data describes the floppy unit configuration
/*   present on an adapter card. The product information bytes
/*   of the inquiry buffer indicate which combination of the
/*   F, G, and H jumpers are set on the card. If the product
/*   information bytes contain "GF" then the G and F jumpers
/*   are installed indicating that LUNs 0-3, if present,  will
/*   be 5.25 inch floppy units (supporting 1.2 Mb, .720 Mb, .360 Mb
/*   formatted capacities). If the product information bytes contain
/*   "HF" then the H and F jumpers are installed indicating a mixed
/*   configuration. LUNs 0 and 1, if present, will be 3.5 inch
/*   floppy units (supporting 1.44 Mb and .760 Mb formatted capacities).
/*   LUNs 2 and 3, if present, will be 5.25 inch floppy units.
/*

#define DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_OFFSET   0x0014
    /*
     * Offset in the vendor unique byte array to the product information
     * data (jumper settings).
     */

#define DEV_SD_FLOPPY_INQUIRY_CONFIG_5_25    "GF"
    /*
     * Jumper setting which indicates that LUNs 0-3, if present, will
     * be 5.25 inch floppy units.
     */

#define DEV_SD_FLOPPY_INQUIRY_CONFIG_MIXED    "HF"
    /*
     * Jumper setting which indicates that LUNs 0-1, if present, will
     * be 3.5 inch floppy units and LUNs 2-3, if present will be
     * 5.25 inch floppy units.
     */

#define DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_SIZE   ((uint32_type)2)
    /*
     * Size in bytes of the config information used in the product
     * information part of the inquiry buffer.
     */

/*.Literal_Section[=------------------------------------------------
/*
/*   SCSI 1.2 Mbyte (formatted) 5.25 inch floppy disk literals.
/*
/*.Description
/*
/*   These constants are used to perform mode sense/select operations
/*   on a 1.2 Mbyte 5.25 inch floppy. To set the floppy disk controller
/*   to the correct mode to access a floppy of this type, a mode
/*   sense is first done to get a template for the mode select
/*   operation. Mode values that are not fixed for all 5.25 inch
/*   floppies are defined here and used to set the controller
```

```
/*    to the mode that matches the medium type.
/*

#define DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR      ((uint8_type)0x00084)
    /*
     * Medium type code used in a mode sense/select header
     * to specify a 1.2 Mbyte 5.25 inch floppy.
     */
#define DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR_MIXED     ((uint8_type)0x00088)
    /*
     * Medium type code used in a mode sense/select header
     * to specify a 1.2 Mbyte 5.25 inch floppy when the sa450 adapter
     * card is jumpered for a mixed configuration of 3.5 and 5.25 inch
     * units. The sa450 card is jumpered for a 3.5 inch configuration
     * when the unit types are mixed. As a result, the card will not
     * allow DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR to be
     * selected as the medium type. The work around for this is to
     * set the medium type to 3.5 inch 1.44 Mb, forcing the sa450 card
     * to set the  disk parameters according to the flexible disk drive
     * geometry paramters specified with the mode select command.
     */

#define DEV_SD_FLOPPY_5_25INCH_1200KB_TRANSFER_RATE         ((uint16_type)0x0001f4)
    /*
     * Transfer rate value used in a mode select command
     * for a 1200 Kbyte 5.25 inch floppy.
     */

#define DEV_SD_FLOPPY_5_25INCH_1200KB_SECTORS_PER_TRACK      ((uint8_type)0x0000f)
    /*
     * The number of sectors per track on a 1.2 Mbyte 5.25 inch
     * floppy.
     */

#define DEV_SD_FLOPPY_5_25INCH_1200KB_NUMBER_OF_CYLINDERS      ((uint16_type)0x00050)
    /*
     * Number of cylinders on a 5.25 inch 1.2 Mbyte floppy.
     */

#define DEV_SD_FLOPPY_5_25INCH_1200KB_STEP_PULSES_PER_CYLINDER      ((uint8_type)0x00001)
    /*
     * Step pulses per cylinder value used in a mode select command
     * of a 5.25 inch 1.2 Mbyte floppy.
     */

#define DEV_SD_FLOPPY_DEFLAULT_WRITE_PRECOMPENSATION_VALUE      ((uint8_type)0x00002)
    /*
     * Write precompensation value used for all floppy
     * medium types.
     */


/*.Literal_Section[-------------------------------------------------------
/*
/*    SCSI 720 Kbyte (formatted) floppy disk literals.
/*
/*.Description
/*
/*    These constants are used to perform mode sense/select operations
/*    on a 720 Kbyte inch floppy. To set the floppy disk controller
/*    to the correct mode to access a floppy of this type, a mode
/*    sense is first done to get a template for the mode select
/*    operation. Mode values that are not fixed for all 720 Kbyte
/*    floppies are defined here and used to set the controller
/*    to the mode that matches the medium type. Note that identical
/*    mode select values are used for 3.50 and 5.25 inch 720 Kbyte
/*    floppies.
/*

#define DEV_SD_FLOPPY_MEDIUM_TYPE_96_135TPI_7958BPR    ((uint8_type)0x00080)
    /*
     * Medium type code used in a mode sense/select header
     * to specify a 720K 5.25 inch, 720K 3.50 inch,  or 360K 5.25
     * inch floppy.
```

```
     */

#define DEV_SD_FLOPPY_720KB_TRANSFER_RATE   ((uint16_type)0x000fa)
    /*
     * Transfer rate value used in a mode select command
     * for a 720 Kbyte floppy.
     */

#define DEV_SD_FLOPPY_720KB_SECTORS_PER_TRACK   ((uint8_type)0x00009)
    /*
     * Number of sectors per track on a 720 Kbyte floppy.
     */

#define DEV_SD_FLOPPY_720KB_NUMBER_OF_CYLINDERS   ((uint16_type)0x00050)
    /*
     * Number of cylinders on a 5.25 inch 720 Kbyte floppy.
     */

#define DEV_SD_FLOPPY_720KB_STEP_PULSES_PER_CYLINDER     ((uint8_type)0x00001)
    /*
     * Step pulses per cylinder value used in a mode select command
     * of a 720 Kbyte floppy.
     */

/*.Literal_Section[=-------------------------------------------------
/*
/*   SCSI 360 Kbyte (formatted) 5.25 inch floppy disk literals.
/*
/*.Description
/*
/*   These constants are used to perform mode sense/select operations
/*   on a 360 Kbyte 5.25 inch floppy. To set the floppy disk controller
/*   to the correct mode to access a floppy of this type, a mode
/*   sense is first done to get a template for the mode select
/*   operation. Mode values that are not fixed for all 5.25 inch
/*   floppies are defined here and used to set the controller
/*   to the mode that matches the medium type.
/*


#define DEV_SD_FLOPPY_5_25INCH_360KB_TRANSFER_RATE   ((uint16_type)0x000fa)
    /*
     * Transfer rate value used in a mode select command
     * for a 360 Kbyte floppy.
     */

#define DEV_SD_FLOPPY_5_25INCH_360KB_SECTORS_PER_TRACK   ((uint8_type)0x0009)
    /*
     * Number of sectors per track on a 5.25 inch 360 Kbyte floppy.
     */

#define DEV_SD_FLOPPY_5_25INCH_360KB_NUMBER_OF_CYLINDERS     ((uint16_type)0x00028)
    /*
     * Number of cylinders on a 5.25 inch 360 Kbyte floppy.
     */

#define DEV_SD_FLOPPY_5_25INCH_360KB_STEP_PULSES_PER_CYLINDER     ((uint8_type)0x00002)
    /*
     * Step pulses per cylinder value used in a mode select command
     * of a 5.25 inch 360 Kbyte floppy.
     */

/*.Literal_Section[=-------------------------------------------------
/*
/*   SCSI 1.44 Mbyte (formatted) 3.5 inch floppy disk literals.
/*
/*.Description
/*
/*   These constants are used to perform mode sense/select operations
/*   on a 1.44 Mbyte 3.5 inch floppy. To set the floppy disk controller
/*   to the correct mode to access a floppy of this type, a mode
/*   sense is first done to get a template for the mode select
/*   operation. Mode values that are not fixed for all 3.5 inch
/*   floppies are defined here and used to set the controller
/*   to the mode that matches the medium type.
```

```
/*
#define DEV_SD_FLOPPY_MEDIUM_TYPE_3_50INCH_135TPI_15916BPR     ((uint8_type)0x00088)
    /*
     * Medium type code used in a mode sense/select header
     * to specify a 1.44 Mbyte 3.50 inch floppy.
     */

#define DEV_SD_FLOPPY_3_50INCH_1440KB_TRANSFER_RATE        ((uint16_type)0x0001f4)
    /*
     * Transfer rate value used in a mode select command
     * for a 1440 Kbyte 3.50 inch floppy.
     */

#define DEV_SD_FLOPPY_3_50INCH_1440KB_SECTORS_PER_TRACK
       ((uint8_type)0x00012)
    /*
     * Number of sectors per track on a 3.5 inch 1.44 Mbyte floppy.
     */

#define DEV_SD_FLOPPY_3_50INCH_1440KB_NUMBER_OF_CYLINDERS     ((uint16_type)0x00050)
    /*
     * Number of cylinders on a 3.50 inch 1.44 Mbyte floppy.
     */

#define DEV_SD_FLOPPY_3_50INCH_1440KB_STEP_PULSES_PER_CYLINDER     ((uint8_type)0x00001)
    /*
     * Step pulses per cylinder value used in a mode select command
     * of a 3.50 inch 1.4 Mbyte floppy.
     */

/*.Literal_Section[=-------------------------------------------------
/*
/*   Floppy Mode Select Test Literals.
/*
/*.Description
/*
/*   These constants are used to determine empirically whether
/*   the operation mode selected on a floppy unit actually
/*   matches the medium type currently inserted in the device.
/*   A trial and error method is used to make the determination.
/*   A mode is selected and an access to the device is attempted to
/*   see if the medium can be successfully accessed. If the access
/*   fails another mode is tried.
/*
/*   This driver supports only double sided floppies with 96 tpi
/*   capacity. 48 tpi formatted floppies are made from 96 tpi
/*   floppies by skipping the odd numbered tracks. The following
/*   floppy densities are supported by the TEAC floppy units:
/*
/* Double Density - 7958 bits/rad (720 Kb formatted at 96 tpi
/*   or 360 Kb formatted at 48 tpi).
/* High Capacity - 13262 bits/rad (1.2 Mb formatted).
/* High Density - 15916 bits/rad (1.44 Mb formatted).
/*
/*   The medium type specified when a floppy disk is formatted
/*   determines the recording method used to layout the disk sectors.
/*   In general, floppies supporting higher densities may be formatted
/*   to a lower density format. However, low density floppies are
/*   not physically capable of supporting high density formats.
/*
/*   A seek and read operation is done to determine if the selected
/*   mode matches the formatted medium type inserted. Based on the
/*   operation mode selected, the controller expects the specified
/*   sector being read to exist on a particular track. If the sector
/*   isn't where it should be, an error is returned and we know we
/*   selected the wrong mode. Take as an example a 720 Kb formatted
/*   floppy inserted with a mode select issued for a 1.2 Mb formatted
/*   floppy. The 720 Kb floppy supports 9 (decimal) sectors per track
/*   times 2 sides -> 18 sectors per track.  The 1.2 MB floppy supports
/*   15 (decimal) sectors per track times 2 sides -> 30 sectors per
/*   track. If we try to read sector 20 the controller thinks the
/*   floppy is 1.2 Mb and expects to find sector 20 in track 0 so the
/*   operation will fail.
/*
```

```
/*   Testing between 720 Kb 96 tpi formatted floppies and 360 Kb 48
/*   tpi formatted floppies should be done only on even numbered tracks.
/*   48 tpi floppies are produced by skipping odd numbered tracks. If
/*   a 48 tpi floppy is formatted from a previously used 96 tpi floppy,
/*   residual data will exist on the odd tracks and make the floppy
/*   appear to the controller to be a 96 tpi floppy if an odd numbered
/*   track is examined.
/*
/*   The sector number used to test for the correct mode was determined
/*   by the following:
/*
/*   It must be small enough to be common to all floppy types.
/*   It must be big enough so that no overlap exists between
/*          medium types (e.g. sector 34 will exist on track 1 of both
/*   1.2 Mb and 760 Kb floppies.
/*   It must be on an even numbered track for all medium types
/*   supported.
/*
/*

#define DEV_SD_FLOPPY_MODE_SELECT_TEST_SECTOR   0x00dc
    /*
     * Sector number to seek to and read to determine if the operating
     * parameters (mode select) of the controller match the currently
     * inserted medium. See above for description of how this sector
     * number was selected.
     */

/*.Section[=-------------------------------------------------------------
/*
/*   SCSI Disk Driver Data Structures.
/*
/*.type */

typedef struct
    {
    io_interleave_lock_type          request_lock;
    misc_queue_header_type           arb_free_queue;
    dev_scsi_adapter_unit_spec_type  unit_spec;
    uint32_type                      max_request_size;
    uint16_type                      open_count;
    uint16_type                      writer_count;
    uint16_type                      exclude_writers_count;
    misc_queue_header_type           async_request_queue;
    io_device_number_type            device_number;
    io_device_number_type            adapter_device_number;
    io_device_handle_type            adapter_handle;
    dev_sd_request_sense_buffer_type sense_buffer;
    dev_scsi_inquiry_buffer_type     inquiry_buffer;
    lm_unsequenced_lock_type         unit_lock;
    misc_counter_type                read_block_count;
    misc_counter_type                write_block_count;
    misc_counter_type                read_request_count;
    misc_counter_type                write_request_count;
    misc_clock_value_type            total_busy_time;
    misc_clock_value_type            total_response_time;
    uint16_type                      sector_size;
    uint32_type                      sector_count;
    boolean_type                     inhibit_error_logging;
    uint16_type                      disk_type;

        /*<------------------*/
    }    dev_sd_unit_info_type  ;
        /*>------------------*/

/*.Description[=---------------------------------------------------------
/*
/*   This data structure contains all of the per-unit information for
/*   a physical disk drive that is under the jurisdiction of the SD
/*   driver.  For the purposes of the SD driver, "unit" refers to a
/*   complete physical disk drive and has exactly the same meaning
/*   as it does in the CPU-to-controller interface.
/*
/*.Members
/*
```

```
/* request_lock -- This lock controls access to the unit
/*   for the purposes of doing I/O to the unit.  The lock must
/*   be obtained before performing any I/O operations.  This lock
/*   is a special interleave lock that allows synchronous and
/*   asynchronous requests to be queued and processed in roughly
/*   the order that the requests are made.
/*
/* arb_free_queue -- Queue of free adapter request blocks.
/*   Adapter request blocks are used to isuue request through
/*   the SCSI interface to a target device.
/*
/* unit_spec -- The SCSI id and unit number of the device
/*   controlled by this data structure.
/*
/* max_request_size -- The largest single request size that the
/*   SCSI interface will support.
/*
/* open_count -- The number of opens that are currently
/*   outstanding on this unit.
/*
/* writer_count -- The number of opens for writing that
/*   are currently outstanding on this unit.
/*
/* exclude_writers_count -- The number of times the unit is
/*   currently open with EXCLUDE_WRITERS intent.  When this field is
/*   non-zero, the unit may not be opened with WRITE intent.  The
/*   field is incremented in 'open' and decremented in 'close'
/*   depending on the open intent.
/*
/* async_request_queue -- The queue header for the queue of
/*   asynchronous requests that could not be immediately serviced by
/*   the unit.  When the unit becomes free, requests are removed
/*   from this queue and assigned to the unit.
/*
/* device_number -- The major and minor device numbers of the
/*   disk unit.
/*
/* adapter_device_number -- The major and minor device numbers
/*   of the SCSI adpater which supports the disk unit.
/*
/* adapter_handle -- An opaque handle returned by the
/*   unit registration function which is used by the SCSI
/*   interface to identify the device controlled by this data
/*   structure.
/*
/* sense_buffer -- Buffer used to store sense data returned
/*   from the unit.
/*
/* inquiry_buffer -- Inquiry data saved from device
/*   configuration.
/*
/* unit_lock -- Lock used to insure serialized base level
/*      access to some members of this data structure. This lock
/*      is used by both the synchronous and asynchronous execution
/*      paths and should be held for only very brief access times
/*      (i.e. the I/O demon should not be kept waiting).
/*
/* read_block_count -- The number of 512 byte blocks read
/*      from this unit since the system was booted.
/*
/* write_block_count -- The number of 512 byte blocks written
/*      to this unit since the system was booted.
/*
/* read_request_count -- The number of read requests
/*      handled by this unit since the system was booted.
/*
/* write_request_count -- The number of write requests
/*      handled by this unit since the system was booted.
/*
/* total_busy_time -- The total amount of time this unit has
/*      been busy since the system was booted.
/*
/* total_response_time -- The total amount of time
/*      that requests spent waiting in the driver for I/O to complete. This
/*      time includes driver overhead as well as device processing time.
```

```
/*
/* sector_size -- Size in bytes of each sector on the
/*   disk.
/*
/* sector_count -- The total number of host accessible
/*      sectors on the disk.
/*
/* inhibit_error_logging -- Flag, when set indicates
/*   that error logging for the device controlled by this
/*   data structure should be disabled. This option is used
/*   during mode selection of some device types. A mode is
/*   selected and an I/O operation is attempted to determine
/*   if the mode was correct. An I/O error occurs if the mode
/*   is not correct and another mode is tried. In this case,
/*   we don't want to log the error.
/*


/*.type */
        /*<----------------------*/
typedef   dev_sd_unit_info_type *          dev_sd_unit_info_ptr_type ;
        /*>----------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*      A pointer to a SCSI disk unit info structure.
/*


/*.type */

typedef struct
    {
    bit8e_type          reserved1;
    bit8e_type          reserved2;
    skip_type           reserved3 : 7;
    field_type          enable_blank_check : 1;
    bit8e_type          blk_desc_len;
    field_type          enable_physical_read : 1;
    field_type          delay_error_reporting : 1;
    field_type          enable_sector_relocation : 1;
    field_type          disable_seek_immediate : 1;
    skip_type           reserved4 : 4;
    field_type          disable_retry_times : 1;
    field_type          error_detection_level : 1;
    skip_type           reserved5 : 1;
    field_type          disable_error_detection_and_correction : 1;
    field_type          parity_enable : 1;
    field_type          enable_diagnostics : 1;
    skip_type           reserved6 : 1;
    skip_type           reserved7 : 1;

        /*<---------------------------------*/
    }    dev_sd_worm_optimem_mode_buffer_type  ;
        /*>---------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure defines a Mode Sense/Select buffer used by the
/*   Optimem 1000 Write Once Read Multiple (WORM) device to report/select
/*   device mode parameters. This format for Mode Sense/Select is
/*   vendor unique.
/*
/*.Members
/*
/* reserved1 -- Not used, must be zero.
/*
/* reserved2 -- Not used, must be zero.
/*
/* reserved3 -- Not used, must be zero.
/*
/* enable_blank_check -- Flag, when set, indicates
/*   that the driver should report an error if a write is
/*   attempted at a block that has already been written to.
/*
```

```
/* blk_desc_len -- The length of the block descriptor in
/*   bytes. The Optimem does not use a block descriptor so this
/*   value is always zero.
/*
/* enable_physical_read -- Flag, when set, allows the disk
/*   controller to read disks whose address fields have been written
/*   in physical format. When zero, normal logical address translation
/*   done.
/*
/* delay_error_reporting -- Flag, when set, inhibits the drive
/*   from reporting read errors until until the read operation has
/*   completed. When zero, the driver reports errors normally.
/*
/* enable_sector_relocation -- Flag, when set, causes the
/*   controller to relocate defective sectors when using the Write
/*   And Verify command. Normal six byte write commands are not
/*   affected by this bit. This bit takes precedence over the Enable
/*   Sector Relocation jumper on the controller board.
/*
/* disable_seek_immediate -- Flag, when set, causes Seek commands
/*   to be of the non-immediate type, no sure what this means.
/*
/* reserved4 -- Not used, must be zero.
/*
/* disable_retry_times -- Flag, when set, causes controller retry
/*   operations to be disabled. When zero, the controller automatically
/*   performs up to five retries before reporting an error.
/*
/* error_detection_level -- Flag, when set, causes the controller
/*   to report a Check Condition status with a Sense Key Medium Error
/*   whenever an error cannot be corrected by the front end processor
/*   of the controller.
/*
/* reserved5 -- Not used, must be zero.
/*
/* disable_error_detection_and_correction -- Flag, when set,
/*   causes the controller to perform a read or write operation without
/*   error correction. Note that sectors written with DEDAC are of a
/*   different format than in the ordinary mode of operation.
/*
/* parity_enable -- Flag, when set, enables controller parity
/*   checking of incoming data. When clear, parity checking is disabled.
/*   This field takes precedence over the Parity Check jumper on the
/*   controller board.
/*
/* enable_diagnostics -- Flag, when set, enables the use of
/*   group 7 diagnostics commands.
/*
/* reserved6 -- Not used, must be zero.
/*
/* reserved7 -- Not used, must be zero.
/*

/*.type */

typedef   dev_sd_worm_optimem_mode_buffer_type *

                        /*<-------------------------------------*/
                         dev_sd_worm_optimem_mode_buffer_ptr_type ;
                        /*>-------------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to an Optimem Write Once Read Many mode select buffer.
/*

/*.Literal_Section[=---------------------------------------------------------
/*
/*   SCSI disk type literals.
/*


#define DEV_SD_DISK_TYPE_RIGID   ((uint16_type)0x00000)
#define DEV_SD_DISK_TYPE_FLOPPY   ((uint16_type)0x00001)
#define DEV_SD_DISK_TYPE_ERASABLE_OPTICAL   ((uint16_type)0x00002)
```

```
#define DEV_SD_DISK_TYPE_WORM   ((uint16_type)0x00003)
#define DEV_SD_DISK_TYPE_READ_ONLY   ((uint16_type)0x00004)
```

Static Global Data: dev_sd_global_data.c

```
/*<-------------------------------------------*/
/*              dev_sd_global_data.c           */
/*>-------------------------------------------*/

/*.Contents[=-------------------------------------------------------------
/*
/* cfv_sd_routines_vector -- exported
/* dev_sd_open_lock -- subsystem
/*
/*.Description
/*
/*   This module contains global data for the SCSI disk driver (SD).
/*
                                  /*<-------------------*/
WIRED
io_driver_routines_vector_type    cfv_sd_routines_vector  =
                                  /*>-------------------*/
        {
        1,                        /* Version 1 of this structure */
        0,                        /* Flags -- currently unused */
        dev_sd_open,
        dev_sd_close,
        dev_sd_read_write,
        dev_sd_select,
        dev_sd_ioctl,
        dev_sd_start_io,
        dev_sd_init,
        dev_sd_configure,
        dev_sd_deconfigure,
        dev_sd_device_to_name,
        dev_sd_name_to_device,
        io_nodevice_open_dump,
        io_nodevice_write_dump,
        io_nodevice_read_dump,
        io_nodevice_close_dump,
        io_nodevice_powerfail,
        io_nodevice_mmap,
        io_nodevice_munmap,
        io_nodevice_maddmap
        };

/*.Description[=-----------------------------------------------------------
/*
/*   This variable contains pointers to each of the externally
/*   referencable functions provided by this device driver.  The
/*   driver is accessed by higher levels of the kernel by
/*   indirecting through these pointers.
/*

/*.variable */
                                        /*<--------------*/
UNWIRED
lm_unsequenced_lock_type                 dev_sd_open_lock  = {0};
                                        /*>--------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This lock protects all operations that involve configuring/
/*   deconfiguring, opening/closing, and mapping device numbers
/*   for devices under the juristiction of this driver.
/*
```

Miscellaneous data: dev_sd_message_data.c

```
/*<-------------------------------------------------*/
/*              dev_sd_message_data.c                */
/*>-------------------------------------------------*/

/*.Contents[=-------------------------------------------------------------
/*
/* dev_sd_hard_error_message -- subsystem
/* dev_sd_soft_error_message -- subsystem
/*
/*.Description
/*
/*   This module contains the text of messages from the sd()
/*   device driver that may be logged to the error logger.
/*   They are collected here in a single module so they may be easily
/*   changed for languages other than English and so their allocation
/*   in memory can be specially managed.
/*
/*.variable */

/*<---------------------*/
WIRED
char     dev_sd_hard_error_message    []  =
/*>---------------------*/

"Disk device at SCSI ID %d, unit %d encountered a hard error at
block %d with status = %o0;

/*.Description[=---------------------------------------------------------
/*
/*   The message that is output when the sd() device encounters
/*   a hard media error.
/*


/*.variable */

/*<-----------------------*/
WIRED
char     dev_sd_soft_error_message    []  =
/*>-----------------------*/

            "Disk device at SCSI ID %d, unit %d encountered a soft error at block %d0;

/*.Description[=---------------------------------------------------------
/*
/*   The message that is output when the sd() device encounters
/*   a soft media error.
```

Main Driver C Code: dev_sd_driver.c

```
/*<-------------------------------------------------*/
/*                    dev_sd_driver.c                */
/*>-------------------------------------------------*/

/*.Contents[=-------------------------------------------------------------
/*
/* dev_sd_open -- subsystem
/* dev_sd_close -- subsystem
/* dev_sd_read_write -- subsystem
/* dev_sd_select -- subsystem
/* dev_sd_ioctl -- subsystem
/* dev_sd_start_io -- subsystem
/* dev_sd_init -- subsystem
/* dev_sd_configure -- subsystem
/* dev_sd_deconfigure -- subsystem
```

```
/* dev_sd_name_to_device -- subsystem
/* dev_sd_device_to_name -- subsystem
/*
/* dev_sd_parse_device_name -- internal
/* dev_sd_complete_io -- internal
/* dev_sd_start_async_request -- internal
/* dev_sd_start_sync_request -- internal
/* dev_sd_evaluate_sense_info -- internal
/* dev_sd_log_error -- internal
/* dev_sd_init_rigid_disk_unit -- internal
/* dev_sd_init_floppy_disk_unit -- internal
/* dev_sd_init_worm_disk_unit -- internal
/* dev_sd_init_optical_disk_unit -- internal
/* dev_sd_sense_unit_mode -- internal
/* dev_sd_select_unit_mode -- internal
/* dev_sd_test_mode_select -- internal
/* dev_sd_control_medium_removal -- internal
/* dev_sd_read_disk_capacity -- internal
/* dev_scsi_get_bytes_requested -- internal
/* dev_sd_determine_disk_type -- internal
/* dev_sd_complete_async_sb_io -- internal
/*
/*.Description
/*
/*   This module is the main part of the device driver for the
/*   set of disks which communicate with the host system
/*   through a SCSI (Small Computer System Interface) I/O bus
/*   interface. Definitions that support this module are in
/*   dev_sd_def.h and dev_scsi_def.h.
/*
/*   The SCSI disk driver supports all SCSI disks which implement
/*   the ANSI SCSI-1 Specification (X3.131-1986) mandatory
/*   commands and the optional Inquiry command. The Inquiry
/*   command is required for self-configuration to work.
/*
/*   This device driver obeys the Standard Driver Interface
/*   for the DG/UX kernel.
/*

/*.function */

               /*<--------*/
UNWIRED
status_type     dev_sd_open  (device_number, channel_flags, device_handle_ptr)
               /*>--------*/

io_device_number_type               device_number;     /* READ ONLY  */
io_channel_flags_type               channel_flags;     /* READ ONLY  */
io_device_handle_ptr_type           device_handle_ptr; /* WRITE ONLY */

/*.Summary[=-------------------------------------------------------------
/*
/*   This function prepares a SCSI disk device (sd) for further I/O
/*   operations and places the specified device number in the set for
/*   which further operations will be valid.
/*
/*.Parameters
/*
/* device_number -- The major and minor device numbers from the
/*      special file that is being opened.
/*
/* channel_flags -- The set of channel flags specifying the
/*   type of access requested on the device.
/*
/* device_handle_ptr -- A pointer to the location where the
/*   device handle is to be returned.
/*
/*.Functional_Description
/*
/*   The device number is mapped to a physical disk unit and the
/*   unit is tested for on-line and ready. If the unit has never
/*   been opened or formatted by a DG/UX system, a mode selection
/*   is done to put the unit in the correct operating mode. The
/*   selected modes are saved on the disk so they become the default
/*   modes for the life of the disk.
```

```
/*
/*   Once the unit is initialized (either because we did it
/*   or because it was already done), we check for open intent
/*   conflicts. If there are no conflicts, the open count of the
/*   unit is incremented and the device handle is returned.
/*
/*.Return_Value
/*
/* OK -- The unit was successfully opened. Any necessary
/*   initializations were successfully performed.
/*
/* DEV_ENXIO_DEVICE_IS_NOT_CONFIGURED -- The speicfied device
/*   number could not be mapped to a physical device.
/*
/* IO_ENXIO_UNIT_NOT_READY -- The disk unit is not ready and
/*   on-line.
/*
/* IO_EIO_PHYSICAL_UNIT_FAILURE -- The disk is not responding
/*   to any commands. This indicates that either the device is not
/*   really present at the specified SCSI id or the device is broken.
/*
/* IO_ENXIO_DEVICE_NOT_SUPPORTED -- A disk has been encountered
/*   with characteristics such that it cannot be supported by this
/*   driver.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*

{

uint16_type                 i;
status_type                 status;
dev_sd_unit_info_ptr_type                           uip;
dev_adapter_request_block_ptr_type                  arb_ptr;
uint16_type                 unit;
dev_sd_read_capacity_buffer_type                    capacity_buffer;
dev_scsi_test_unit_ready_cmd_blk_ptr_type           scsi_cmd_blk_ptr;
uint16_type                 disk_type;

/*.Implementation[=-------------------------------------------------
/*
/*   Obtain the SCSI disk open lock. The open lock insures that
/*   simultaneous opens of the same device do not occur. The lock
/*   also insures that the device will not be deconfigured
/*   during the open operation.
/*
/*   First we check that the given device number corresponds to
/*   a configured device, and in the process get a pointer to the
/*   unit information structure for use throughout the rest of
/*   the function.
/*
/*.End]=-----------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_sd_open_lock);
status = io_map_device_number(device_number, (bit32e_ptr_type)&uip, &unit);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   Obtain the unit request lock, build a test unit ready command
/*   block and issue it to the target device. If this is the first
/*   command issued to the device after the SCSI bus reset other than
/*   the Inquiry command, the test unit ready will fail with a Check
/*   Condition status. The sense information will indicate that the
/*   unit attention flag for the unit is on. The attempted execution
/*   of the test unit ready command clears this condition and the
```

```
/*   command is reissued to determine if the unit is ready. If the
/*   device does not respond that it is ready after the second try,
/*   return the error.
/*
/*.End]-------------------------------------------------------------*/

for (i = 0; i < 2; i++)
    {
    io_sync_obtain_interleave_lock(&uip->request_lock);
    misc_dequeue_from_head(&uip->arb_free_queue,
                           (misc_queue_links_ptr_type *)&arb_ptr);
    scsi_cmd_blk_ptr = (dev_scsi_test_unit_ready_cmd_blk_ptr_type)
                                  &arb_ptr->scsi_cmd_blk;
    scsi_cmd_blk_ptr->op_code = DEV_SCSI_CMD_TEST_UNIT_READY;
    scsi_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
    scsi_cmd_blk_ptr->reserved1 = 0;
    scsi_cmd_blk_ptr->reserved2 = 0;
    scsi_cmd_blk_ptr->vendor_unique = 0;
    scsi_cmd_blk_ptr->reserved3 = 0;
    scsi_cmd_blk_ptr->link = FALSE;
    scsi_cmd_blk_ptr->flag = FALSE;
    arb_ptr->request_flags = (bit16e_type)0;
    io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                                    DEV_SD_NULL_BUFFER_PTR,
                                    (uint32_type)0);
    status = dev_sd_start_sync_request(uip, arb_ptr);
    misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
    if (io_assign_next_interleave_waiter(&uip->request_lock))
        {
        dev_sd_start_async_request(uip);
        }
    io_release_interleave_lock(&uip->request_lock);
    if (status == OK)
        {
        break;
        }
    }
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   Determine the device type and perform any initialization
/*   that is required on the unit.
/*
/*.End]-------------------------------------------------------------*/

status = dev_sd_determine_disk_type(uip, &disk_type);
uip->disk_type = disk_type;

if   ((status == OK) && (uip->open_count == 0))
   {
   switch (disk_type)
      {
      case DEV_SD_DISK_TYPE_RIGID:
         status = dev_sd_init_rigid_disk_unit(uip);
         break;

      case DEV_SD_DISK_TYPE_FLOPPY:
         status = dev_sd_init_floppy_disk_unit(uip);
         break;

      case DEV_SD_DISK_TYPE_WORM:
         status = dev_sd_init_worm_disk_unit(uip);
         break;

      case DEV_SD_DISK_TYPE_ERASABLE_OPTICAL:
      case DEV_SD_DISK_TYPE_READ_ONLY:
         status = dev_sd_init_optical_disk_unit(uip);
         break;
      }
   }
```

```
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   If the disk unit is not currently open, issue a read capacity
/*   command to get the sector size of the currently inserted medium.
/*   Set the allowable number of concurrent requests to the value
/*   appropriate for the sector size. If disk request buffering
/*   must be done (i.e. sector size > 512 bytes), only one request
/*   can be processed at a time. If open_count is zero, we are safe
/*   to initialize the interleave lock. If open_count is not zero,
/*   the concurrent request count should already be correct.
/*
/*.End]=-----------------------------------------------------------*/

if (uip->open_count == 0)
    {
    status =  dev_sd_read_disk_capacity(uip, &capacity_buffer);
    if (status == OK)
        {
        uip->sector_size = (uint16_type)capacity_buffer.block_len;
        uip->sector_count = capacity_buffer.highest_block_address+1;
        if (uip->sector_size > DEV_SD_STANDARD_SECTOR_SIZE)
            {
            io_initialize_interleave_lock(&uip->request_lock,
        (uint16_type)DEV_SD_MAX_BUFFERED_CONCURRENT_UNIT_REQUESTS);
            }
        else
            {
            io_initialize_interleave_lock(&uip->request_lock,
        (uint16_type)DEV_SD_MAX_CONCURRENT_UNIT_REQUESTS);
            }
        }
    else
        {
        goto done;
        }
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Having verified that the hardware is working, we check for
/*   conflicts with EXCLUDE_WRITERS intent. If there are no
/*   conflicts we increment the open count.
/*
/*.End]=-----------------------------------------------------------*/

if (channel_flags & IO_CHANNEL_WRITE_INTENT)
    {
    if (uip->exclude_writers_count != 0)
            {
            status = IO_EBUSY_OPEN_INTENT_CONFLICTS;
            goto done;
            }
    }
if (channel_flags & IO_CHANNEL_EXCLUDE_WRITERS_INTENT)
    {
    if (uip->writer_count != 0)
            {
            status = IO_EBUSY_OPEN_INTENT_CONFLICTS;
            goto done;
            }
    }
*device_handle_ptr = (io_device_handle_type)uip;

uip->open_count++;
if (channel_flags & IO_CHANNEL_WRITE_INTENT)
    {
    uip->writer_count++;
    }
if (channel_flags & IO_CHANNEL_EXCLUDE_WRITERS_INTENT)
    {
```

```
    uip->exclude_writers_count++;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Finally, disable removal of the device until the close
/*   operation is performed.
/*
/*.End]=-------------------------------------------------------------*/

dev_sd_control_medium_removal(uip, TRUE);

done:
lm_release_unsequenced_lock(&dev_sd_open_lock);

return(status);

}

/*.function */

               /*<--------*/
UNWIRED
status_type     dev_sd_close  (device_handle, channel_flags)
               /*>--------*/

io_device_handle_type               device_handle;/* READ ONLY  */
io_channel_flags_type               channel_flags;/* READ ONLY  */

/*.Summary[=---------------------------------------------------------------
/*
/*   This function performs the inverse of the open operation.
/*
/*.Parameters
/*
/* device_handle -- The device handle of the device that is
/*   to be closed.  This value must be a device handle that was
/*   returned by a successful call to dev_sd_open.
/*
/* channel_flags -- The channel flags with which the device
/*   was opened.
/*
/*.Functional_Description
/*
/*   The open_count of the specified device is decremented.  If the
/*   channel_flags specified any of the write intents, then we update
/*   the write intent management variables.
/*
/*.Return_Value
/*
/*  OK  -- Close was successful.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*

{

dev_sd_unit_info_ptr_type                       uip;


/*.Implementation[=--------------------------------------------------
/*
/*   Close performs the reverse of the open operation in
/*   managing the writer_count, exclude_writers flag and allowing
/*   medium removal.  The open_lock must be held to protect against
/*   simultaneous open operations.
/*
/*.End]=-------------------------------------------------------------*/
```

```
uip = (dev_sd_unit_info_ptr_type)device_handle;
dev_sd_control_medium_removal(uip, FALSE);

lm_obtain_unsequenced_lock(&dev_sd_open_lock);

if (channel_flags & IO_CHANNEL_EXCLUDE_WRITERS_INTENT)
    {
    uip->exclude_writers_count--;
    }
if (channel_flags & IO_CHANNEL_WRITE_INTENT)
    {
    uip->writer_count--;
    }
uip->open_count--;

lm_release_unsequenced_lock(&dev_sd_open_lock);

return(OK);

}

/*.function */

                /*<--------------*/
WIRED
status_type      dev_sd_read_write (request_info_ptr)
                /*>--------------*/

ic_request_info_ptr_typerequest_info_ptr;    /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------------
/*
/*   This function performs a synchronous I/O operation on a SCSI
/*   class disk (sd).
/*
/*.Parameters
/*
/* request_info_ptr -- A pointer to the structure specifying
/*   the operation to be performed, the device, the device address,
/*   the memory buffer address, and the length of the data
/*   transfer for the operation.
/*
/*.Functional_Description
/*
/*   This function performs a synchronous I/O operation on a SD
/*   class disk.  It allocates a generic adapter request block,
/*   which contains the information about a request, fills it in,
/*   and then issues the request through the supporting SCSI adapter
/*   to the disk.  The function then waits until the request
/*   completes, whereupon it checks the result statuses and returns
/*   an indication of the success or failure of the operation.
/*
/*   If the size of the caller's request is larger than the disk
/*   interface can handle, this function breaks the request into
/*   several smaller requests and returns only when the entire
/*   request has completed or an error occurs. Both hard and
/*   soft errors are    logged to the error log device. No direct
/*   indication of soft errors is given to the caller.
/*
/*   Additionally, this function checks to see whether the disk
/*   sector size is greater than 512-bytes/sector.  If so, a
/*   buffer is allocated which is a multiple of the sector size.
/*   If the request is a read, a read is issued, and the requested
/*   data is selected from the buffer when the read finishes.  If
/*   the request is a write, a read is first issued, the data is
/*   written into the sector buffer, and then the write is issued.
/*
/*.Return_Value
/*
/* OK -- The operation completed successfully.
/*
/* IO_EINVAL_ILLEGAL_REQUEST_SIZE -- The buffer vector
/*   contained a descriptor with a size that was not a multiple of
/*   the disk sector size.
/*
```

```
/* IO_EINVAL_ILLEGAL_BUFFER_ADDRESS -- The buffer vector
/*   contained an address that was not on an even byte boundary.
/*
/* IO_ENXIO_ILLEGAL_DEVICE_ADDRESS -- The byte granular
/*   device address is not a multiple of the disk sector size,
/*   in which case no data is transferred, or the request extended
/*   past the end of the disk media, in which case the buffer
/*   vector indicates the number of bytes actually transferred.
/*   In the latter case, all data up to the end of the disk will
/*   NOT necessarily be transferred. The end of the disk cannot be
/*   determined by reading off the end.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   Panic may be invoked with the following error codes:
/*
/* DEV_PANIC_SD_UNKNOWN_OPERATION -- An unknown operation was
/*   specified in the "op" field of the request_info record.  This is
/*   an internal software error.
/*

{

status_type             status;
dev_sd_unit_info_ptr_type                   uip;
misc_clock_value_type                   driver_entry_time;
misc_clock_value_type                   cur_time;
dev_adapter_request_block_ptr_type  arb_ptr;
io_buffer_vector_ptr_type                   buffer_vector_ptr;
uint32_type             current_size;
byte_address_type               current_buffer_ptr;
uint32_type             current_device_address;
uint32_type             bytes_transferred;
misc_clock_value_type                   busy_time;
dev_scsi_read_write_cmd_blk_ptr_type scsi_rw_cmd_blk_ptr;
dev_scsi_write_verify_cmd_blk_ptr_type      scsi_write_verify_cmd_blk_ptr;
uint32_type             current_device_offset;
boolean_type            sector_buffering;
boolean_type            sector_buffer_allocated;
pointer_to_any_type             sector_buffer_ptr;
uint32_type             sector_buffer_size;
pointer_to_any_type             sector_buffer_position_ptr;
uint32_type             allocated_size;

/*.Implementation[=----------------------------------------------
/*
/*   This function consists of a main outer loop that breaks up
/*   what may be a request for a very large number of bytes into
/*   pieces that are within the capacity of the driver. The
/*   capacity of the driver is limited to the maximum request size
/*   supported by the DMA interface and on the number of pages of
/*   the requestors's buffer we are willing to have wired at one time.
/*
/*   The current system time is recorded upon entry into this function
/*   for system activity reporting.
/*
/*.End]=-------------------------------------------------------------*/

status = OK;
vp_read_system_clock(&driver_entry_time);
uip = (dev_sd_unit_info_ptr_type)(request_info_ptr->device_handle);
MISC_CLOCK_VALUE_SET_TO_ZERO(&busy_time);
current_device_offset = request_info_ptr->device_offset;

if((current_device_offset % DEV_SD_STANDARD_SECTOR_SIZE) != 0)
    {
    return(IO_ENXIO_ILLEGAL_DEVICE_ADDRESS);
    }

sector_buffer_allocated = FALSE;
buffer_vector_ptr = &request_info_ptr->buffer_vector;
```

```
while((io_get_buffer_vector_residual(buffer_vector_ptr) != 0) && (status == OK))
   {

   current_device_address = current_device_offset / uip->sector_size;

/*.Implementation_Continued[=--------------------------------------
/*
/*   For each piece of the request, we first wire the requestor's
/*   buffer into memory.   Then we must construct a generic
/*   adapter request block to communicate our request through
/*   the supporting SCSI adapter to the disk.
/*
/*.End]=-----------------------------------------------------------*/

   io_get_buffer_vector_io_info(buffer_vector_ptr,
                        &current_buffer_ptr,
                        &current_size);

   if   ((current_size % DEV_SD_STANDARD_SECTOR_SIZE) != 0)
      {
      status = IO_EINVAL_ILLEGAL_REQUEST_SIZE;
      break;
      }

   if (ODD_BYTE_ADDRESS(current_buffer_ptr))
      {
      status = IO_EINVAL_ILLEGAL_BUFFER_ADDRESS;
      break;
      }

   current_size = MINIMUM(current_size, uip->max_request_size);

   if (request_info_ptr->op & IO_OPERATION_USER_BUFFER)
      {
      status = vm_wire_memory(current_buffer_ptr, TRUE,  current_size);
      if (status != OK)
         {
         break;
         }
      }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Determine if sector buffering is necessary. It may be
/*   necessary if the disk sector size is not 512.  Even then,
/*   sector buffering is only required if 1) the data is not
/*   aligned on a sector boundary, or 2) the amount of data to
/*   transfer is not a multiple of the sector size.
/*
/*.End]=-----------------------------------------------------------*/

   sector_buffering = FALSE;
   if (uip->sector_size != DEV_SD_STANDARD_SECTOR_SIZE)
       {
       if (current_device_offset % uip->sector_size != 0)
          {
          sector_buffering = TRUE;
          }
       if (current_size % uip->sector_size != 0)
          {
          sector_buffering = TRUE;
          }
       }

   if (!sector_buffering)
      {

/*.Implementation_Continued[=--------------------------------------
/*
/*   Sector buffering is not required.  Process the request
/*   normally.
/*
/*   Now obtain the unit request lock and fill in the adapter
/*   request block with request information. The unit request
/*   lock must be held to insure that an adapter request block is
```

```
/*   available.
/*
/*   The rezero unit command is the SCSI equivalent of the
/*   recalibrate command. Some disk models deviate from recalibrate
/*   in that the heads are positioned to logical block address zero
/*   instead of physical cylinder zero. This should not cause
/*   problems since SCSI disks automatically recalibrate when
/*   a seek error occurs.
/*
/*   The no retries option IO_CHANNEL_NO_RETRIES is currently not
/*   supported because it requires a mode selection operation
/*   which could affect other disk users. To support this option,
/*   a more complicated unit locking scheme would be needed.
/*
/*.End]-------------------------------------------------------------*/

      io_sync_obtain_interleave_lock(&uip->request_lock);
      misc_dequeue_from_head(&uip->arb_free_queue,
             (misc_queue_links_ptr_type *)&arb_ptr);

      if   (((uip->disk_type == DEV_SD_DISK_TYPE_ERASABLE_OPTICAL)
          || (uip->disk_type == DEV_SD_DISK_TYPE_WORM))
          && (request_info_ptr->op & IO_OPERATION_WRITE))
         {
         scsi_write_verify_cmd_blk_ptr =
             (dev_scsi_write_verify_cmd_blk_ptr_type)&arb_ptr->scsi_cmd_blk;
         scsi_write_verify_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE_VERIFY;
         scsi_write_verify_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
         scsi_write_verify_cmd_blk_ptr->reserved1 = 0;
         scsi_write_verify_cmd_blk_ptr->relative_addr = FALSE;
         scsi_write_verify_cmd_blk_ptr->logical_block_addr_high =
                     (uint16_type)((current_device_address >> 16) & 0xffff);
         scsi_write_verify_cmd_blk_ptr->logical_block_addr_low =
                           (uint16_type)(current_device_address & 0xffff);
         scsi_write_verify_cmd_blk_ptr->reserved2 = 0;
         scsi_write_verify_cmd_blk_ptr->transfer_length_high = 0;
         scsi_write_verify_cmd_blk_ptr->transfer_length_low = current_size /
                 uip->sector_size;
         scsi_write_verify_cmd_blk_ptr->reserved3 = 0;
         scsi_write_verify_cmd_blk_ptr->erase_control = FALSE;
         scsi_write_verify_cmd_blk_ptr->reserved4 = 0;
         scsi_write_verify_cmd_blk_ptr->link = FALSE;
         scsi_write_verify_cmd_blk_ptr->flag = FALSE;
         }
      else
         {
         if (request_info_ptr->op & (IO_OPERATION_READ | IO_OPERATION_WRITE))
               {
            scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                                  &arb_ptr->scsi_cmd_blk;
            if (request_info_ptr->op & IO_OPERATION_READ)
                  {
               scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ;
                  }
               else
                  {
               scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE;
                  }
            scsi_rw_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
                scsi_rw_cmd_blk_ptr->logical_block_address = current_device_address;
                scsi_rw_cmd_blk_ptr->transfer_length =  current_size /
                                              uip->sector_size;
            scsi_rw_cmd_blk_ptr->vendor_unique = 0;
            scsi_rw_cmd_blk_ptr->reserved1 = 0;
            scsi_rw_cmd_blk_ptr->link = FALSE;
            scsi_rw_cmd_blk_ptr->flag = FALSE;
                  }
         else
                  {
                  sc_panic(DEV_PANIC_SD_UNKNOWN_OPERATION);
                  }
         }

      io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                                      current_buffer_ptr,
```

```
                                        current_size);

      if (!(request_info_ptr->op & IO_OPERATION_USER_BUFFER))
         {
         arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
         }
      else
         {
         arb_ptr->request_flags = (bit16e_type)0;
         }

/*.Implementation_Continued[=---------------------------------------
/*
/*       Update the disk usage accounting variables used by sar and
/*       dg_sys_info().  Increment the number of read/write requests and
/*       blocks read/written on this unit. If the request is coming
/*       through the raw I/O interface (rdsk, rpsk), increment
/*       physical read or write counters. Note that if the operation
/*       is block-special, the buffer manager increments the block_read
/*       and block_write variables.
/*
/*.End]=-------------------------------------------------------------*/

      if (request_info_ptr->op & IO_OPERATION_READ)
         {
         misc_increment(&uip->read_request_count);
         misc_increment_by_value(&uip->read_block_count,
                                 (int32e_type)(current_size/DF_BYTES_PER_BLOCK));
         if (!(request_info_ptr->flags & IO_CHANNEL_BLOCK_SPECIAL))
            {
            misc_increment(&sc_sys_info.physical_read_request_count);
            }
         }
      else
         {
         misc_increment(&uip->write_request_count);
         misc_increment_by_value(&uip->write_block_count,
                                 (int32e_type)(current_size/DF_BYTES_PER_BLOCK));
         if (!(request_info_ptr->flags & IO_CHANNEL_BLOCK_SPECIAL))
            {
            misc_increment(&sc_sys_info.physical_write_request_count);
            }
         }

/*.Implementation_Continued[=---------------------------------------
/*
/*    Start the request. We will not return from
/*    dev_sd_start_sync_request until the request has completed.
/*
/*.End]=-------------------------------------------------------------*/

      status = dev_sd_start_sync_request(uip, arb_ptr);
      bytes_transferred =
                           io_get_buffer_vector_position(&arb_ptr->buffer_vector);
      io_add_to_buffer_vector_position(buffer_vector_ptr,
                                       (int32_type)bytes_transferred);

      } /* end if !sector_buffering */
   else
      {

/*.Implementation_Continued[=---------------------------------------
/*
/*       Sector buffering is required.
/*
/*       If a sector buffer has not been allocated, then allocate one.
/*       First, determine the size of the buffer needed.  The buffer
/*       should be as at least as large as the current request size
/*       and should be an integral of the disk sector size.
/*
/*.End]=-------------------------------------------------------------*/

      sector_buffer_size = uip->sector_size;

      if (current_size > DEV_SD_STANDARD_SECTOR_SIZE)
```

```
         {
         sector_buffer_size = sector_buffer_size +   ((current_size /
uip->sector_size)
                           * uip->sector_size) + uip->sector_size;
         }

      if (!sector_buffer_allocated)
         {
         sector_buffer_ptr = vm_get_wired_memory(sector_buffer_size,
                                                 VM_DEFAULT_ALIGNMENT);
         allocated_size = sector_buffer_size;
         sector_buffer_allocated = TRUE;
         }

/*.Implementation_Continued[=--------------------------------------
/*
/*      Determine the position within the sector buffer of the
/*      transfer data.
/*
/*.End]=-----------------------------------------------------------*/

      sector_buffer_position_ptr = sector_buffer_ptr +
                          (current_device_offset % uip->sector_size);

/*.Implementation_Continued[=--------------------------------------
/*
/*    Do a read.
/*
/*.End]=-----------------------------------------------------------*/

      io_sync_obtain_interleave_lock(&uip->request_lock);
      misc_dequeue_from_head(&uip->arb_free_queue,
                             (misc_queue_links_ptr_type *)&arb_ptr);
      scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                                    &arb_ptr->scsi_cmd_blk;
      scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ;
      scsi_rw_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
      scsi_rw_cmd_blk_ptr->logical_block_address = current_device_address;
      scsi_rw_cmd_blk_ptr->transfer_length = sector_buffer_size /
                                             uip->sector_size;
      scsi_rw_cmd_blk_ptr->vendor_unique = 0;
      scsi_rw_cmd_blk_ptr->reserved1 = 0;
      scsi_rw_cmd_blk_ptr->link = FALSE;
      scsi_rw_cmd_blk_ptr->flag = FALSE;
      arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
      io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                                      sector_buffer_ptr,
                                      sector_buffer_size);

      status = dev_sd_start_sync_request(uip, arb_ptr);
      bytes_transferred =
      io_get_buffer_vector_position(&arb_ptr->buffer_vector);

      misc_increment(&uip->read_request_count);
      misc_increment_by_value(&uip->read_block_count,
                          (int32e_type)(sector_buffer_size/DF_BYTES_PER_BLOCK));
      if (!(request_info_ptr->flags & IO_CHANNEL_BLOCK_SPECIAL))
         {
         misc_increment(&sc_sys_info.physical_read_request_count);
         }

/*.Implementation_Continued[=--------------------------------------
/*
/*      Check to see if the data buffer is a kernel buffer.  If so,
/*      we have to switch to kernel address space.
/*
/*.End]=-----------------------------------------------------------*/

      if (!(request_info_ptr->op & IO_OPERATION_USER_BUFFER))
         {
         sc_begin_kernel_access();
         }

/*.Implementation_Continued[=--------------------------------------
/*      If the operation is a write, update the sector buffer and
```

```
/*      do the write.  Otherwise, just copy the requested data to
/*      the input buffer.
/*
/*.End]=-------------------------------------------------------------*/

      if (request_info_ptr->op & IO_OPERATION_WRITE)
         {

         io_read_from_buffer_vector(buffer_vector_ptr,
                                    sector_buffer_position_ptr,
                                    &current_size);

         if   ((uip->disk_type == DEV_SD_DISK_TYPE_ERASABLE_OPTICAL)
            || (uip->disk_type == DEV_SD_DISK_TYPE_WORM))
            {
            scsi_write_verify_cmd_blk_ptr =
             (dev_scsi_write_verify_cmd_blk_ptr_type)&arb_ptr->scsi_cmd_blk;
            scsi_write_verify_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE_VERIFY;
            scsi_write_verify_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
            scsi_write_verify_cmd_blk_ptr->reserved1 = 0;
            scsi_write_verify_cmd_blk_ptr->relative_addr = FALSE;
            scsi_write_verify_cmd_blk_ptr->logical_block_addr_high =
                        (uint16_type)((current_device_address >> 16) & 0xffff);
            scsi_write_verify_cmd_blk_ptr->logical_block_addr_low =
                              (uint16_type)(current_device_address & 0xffff);
            scsi_write_verify_cmd_blk_ptr->reserved2 = 0;
            scsi_write_verify_cmd_blk_ptr->transfer_length_high = 0;
            scsi_write_verify_cmd_blk_ptr->transfer_length_low =
                                       sector_buffer_size / uip->sector_size;
            scsi_write_verify_cmd_blk_ptr->reserved3 = 0;
            scsi_write_verify_cmd_blk_ptr->erase_control = FALSE;
            scsi_write_verify_cmd_blk_ptr->reserved4 = 0;
            scsi_write_verify_cmd_blk_ptr->link = FALSE;
            scsi_write_verify_cmd_blk_ptr->flag = FALSE;
            }
             else
                {
            scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                                                  &arb_ptr->scsi_cmd_blk;
            scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE;
            scsi_rw_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
            scsi_rw_cmd_blk_ptr->logical_block_address =
                                                 current_device_address;
            scsi_rw_cmd_blk_ptr->transfer_length = sector_buffer_size /
                                                       uip->sector_size;
            scsi_rw_cmd_blk_ptr->vendor_unique = 0;
            scsi_rw_cmd_blk_ptr->reserved1 = 0;
            scsi_rw_cmd_blk_ptr->link = FALSE;
            scsi_rw_cmd_blk_ptr->flag = FALSE;
                }

         arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
         io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                                         sector_buffer_ptr,
                                         sector_buffer_size);
         status = dev_sd_start_sync_request(uip, arb_ptr);
         bytes_transferred =
                        io_get_buffer_vector_position(&arb_ptr->buffer_vector);
         misc_increment(&uip->write_request_count);
         misc_increment_by_value(&uip->write_block_count,
                          (int32e_type)(sector_buffer_size/DF_BYTES_PER_BLOCK));
         if (!(request_info_ptr->flags & IO_CHANNEL_BLOCK_SPECIAL))
            {
            misc_increment(&sc_sys_info.physical_write_request_count);
            }
         }
      else
         {
         io_write_to_buffer_vector(sector_buffer_position_ptr,
                                   buffer_vector_ptr, &current_size);
         }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Return to user space if necessary.
```

```
/*
/*.End]=-------------------------------------------------------------*/

      if (!(request_info_ptr->op & IO_OPERATION_USER_BUFFER))
         {
         sc_end_kernel_access();
         }

      } /* end else sector_buffering */

/*.Implementation_Continued[=----------------------------------------
/*
/*   If a sector buffer was allocated, release the memory.
/*
/*.End]=-------------------------------------------------------------*/

if (sector_buffer_allocated)
   {
   vm_release_wired_memory(sector_buffer_ptr, allocated_size);
   }

/*.Implementation_Continued[=----------------------------------------
/*
/*   One way or anther we have finished processing this request.
/*   Update the fields indicating the amount of data that was
/*   transferred, and unwire the caller's buffer if we wired it
/*   earlier.  We then go back around the main 'while' loop to
/*   see if more data needs to be transferred to completely
/*   satsify the original request.  If an error occurred on
/*   this portion of the caller's request, the non-OK status
/*   will kick us out of the 'while' loop and return the non-OK
/*   status.
/*
/*.End]=-------------------------------------------------------------*/

   MISC_CLOCK_VALUE_ADD(&arb_ptr->total_request_busy_time, &busy_time);

   misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
   if (io_assign_next_interleave_waiter(&uip->request_lock))
       {
       dev_sd_start_async_request(uip);
       }
   io_release_interleave_lock(&uip->request_lock);

   if (request_info_ptr->op & IO_OPERATION_USER_BUFFER)
      {
      if (request_info_ptr->op & IO_OPERATION_READ)
         {
         vm_mark_mod_and_ref_and_unwire_memory(
                        current_buffer_ptr, TRUE, current_size);
         }
      else
         {
         vm_mark_ref_and_unwire_memory(
                        current_buffer_ptr, TRUE, current_size);
         }
      }

    current_device_address += current_size;

    if (request_info_ptr->op & IO_OPERATION_CHECK_SELF_ID)
           {
           if (!fs_check_self_id(current_buffer_ptr,
           &request_info_ptr->self_id,
                          &current_size))
               {
               status = IO_EINVAL_BAD_SELF_ID;
               }
           }

    } /* End while */

/*.Implementation_Continued[=----------------------------------------
/*
/*    Calculate the response and busy times required to process this
```

```
/*    request and add them to the totals for the unit. The unit lock
/*    is held while the total is updated to protect against simultaneous
/*    access.
/*
/*.End]=-------------------------------------------------------------*/

vp_read_system_clock(&cur_time);
MISC_CLOCK_VALUE_SUBTRACT(&driver_entry_time, &cur_time);
lm_obtain_unsequenced_lock(&uip->unit_lock);
MISC_CLOCK_VALUE_ADD(&cur_time, &uip->total_response_time);
MISC_CLOCK_VALUE_ADD(&busy_time, &uip->total_busy_time);
lm_release_unsequenced_lock(&uip->unit_lock);

return(status);
}

/*.function */

       /*<----------*/
WIRED
void    dev_sd_select (device_handle, select, ec_ptr, intent_ptr)
       /*>----------*/

io_device_handle_type       device_handle;  /* READ ONLY  */
boolean_type                select;     /* READ ONLY  */
vp_ec_ptr_type              ec_ptr;     /* READ ONLY  */
io_select_intent_ptr_type   intent_ptr;             /* READ WRITE */

/*.Summary[=-----------------------------------------------------------
/*
/*   Indicate whether the specified disk is ready to perform I/O.
/*
/*
/*.Parameters
/*
/* device_handle -- The device handle of the device that is
/*   the target of select.  This handle must be a device handle that
/*   was returned by the open function of this driver.
/*
/* select -- If TRUE, this is the start of a select operation
/*   and conditions that are not immediately TRUE should be
/*   recorded so that the eventcounter can be advanced when they
/*   become TRUE.  If FALSE, this is the end of a select operation
/*   and any previously remembered conditions should be forgotten.
/*
/* ec_ptr -- Specifies the eventcounter to be advanced
/*   by the driver when the select is satisfied if it is not
/*   immediately satisfied.
/*
/* intent_ptr -- On input, specifies whether a select
/*   is to be instituted for a combination of read, write, or
/*   exceptional conditions.  On output, specifies the subset of
/*   the input conditions that are currently TRUE.
/*
/*.Functional_Description
/*
/*   Since disks always respond very quickly, this function
/*   always returns TRUE when selecting for READ or WRITE.  Since
/*   there are never any exceptions to report via the driver
/*   interface, this function always returns FALSE when selecting
/*   for EXCEPTION.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*
```

```
{

*intent_ptr &= IO_SELECT_INTENT_READ | IO_SELECT_INTENT_WRITE;

}

/*.function */

                /*<--------*/
WIRED
status_type      dev_sd_ioctl       (device_handle,
                /*>--------*/
                                    command,
                                    parameter,
                                    return_value_ptr)

io_device_handle_type   device_handle; /* READ ONLY  */
bit32e_type             command;       /* READ ONLY  */
bit32e_type             parameter;     /* READ/WRITE */
bit32e_ptr_type         return_value_ptr; /* WRITE ONLY */

/*.Summary[=----------------------------------------------------------------
/*
/*   This function performs SD disk specific 'ioctl' commands.
/*
/*.Parameters
/*
/* device_handle -- The device handle of the device that is
/*   the target of the ioctl operation.  This value must be a device
/*   handle that was returned by a successful call to dev_sd_open.
/*
/* command      -- A command to the device.  The
/*   interpretation of the command is specific to the driver.
/*
/* parameter    -- An argument to the command.  The
/*   interpretation of the parameter is specific to the driver and
/*   the command.  The parameter may be used to transfer information
/*   in either direction between the caller and the device.  In
/*   particular it may be a pointer to a buffer supplied by the
/*   caller.
/*
/* return_value_ptr -- A pointer to the value to be returned
/*   to the user.
/*
/*.Functional_Description
/*
/*   The SD disk ioctl command provides an entry point for issuing
/*   the following commands to the device.
/*
/* The DSKIOCGET command,  which returns information about
/*   the physical characteristics of the disk unit.
/* The DSKIOC_GENERIC_SCSI command provides a generic interface
/*   to a SCSI device. When invoked through this driver it provides
/*   direct (unrestricted) access to SCSI device.
/*
/*   This command interprets "parameter" as a pointer to type
/*   "dev_scsi_generic_parm_ptr_type", which provides the SCSI
/*   command buffer, a buffer for status and  sense data,
/*   the memory buffer address, and the number of byte to be
/*   transferred by the operation.
/*
/*   This function provides synchronous operations on a SD
/*   class device.  The unit information block of the target is
/*   filled in from the "parameter" data; appropriate access to
/*   any I/O buffer is verified and the buffer is wired.
/*   The command is then issued through the supporting SCSI adapter.
/*   The function then waits until the request
/*   completes, checks the result statuses and returns
/*   an indication of the success or failure of the operation.
/*   When a "CHECK CONDITION" status is received from the device
/*   the sense key and additional sense key are returned.
/*
/* The DSKIOC_READ_DISK_LABEL command reads the DG/UX disk
/*   label from block zero of the disk.
/* Th DSKIOC_WRITE_DISK_LABEL command writes a DG/UX disk
```

```
/*   label to block zero of the disk.
/* The DSKIOCUSAGE command returns sar disk activity
/*   information for the specified disk unit.
/*
/*
/*.Return_Value
/*
/* OK -- The ioctl command completed normally.
/*
/* IO_EINVAL_COMMAND_NOT_SUPPORTED_BY_DEVICE -- The ioctl
/*   was not one that is supported by the SD driver.
/*
/* IO_EINVAL_ILLEGAL_REQUEST_SIZE -- The I/O
/*   request exceeded the maximum request size (determined by the maximun
/*   amount of memory that's recommended to be wired by the driver.
/*
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*


{
status_type             status;
dev_sd_unit_info_ptr_type                       uip;
dev_adapter_request_block_ptr_type  arb_ptr;
struct dskget                       dskget;
dev_scsi_read_write_cmd_blk_ptr_type scsi_rw_cmd_blk_ptr;
dev_scsi_cmd_blk_ptr_type                   scsi_gen_cmd_ptr;
dev_scsi_generic_parm_ptr_type          scsi_gen_parm_ptr;
dev_scsi_generic_cmd_blk_type                   local_scsi_gen_cmd;
int32_type              data_buffer_byte_size;
uint32_type             data_bytes_transferred;
uint8e_type             sense_keys[3];
boolean_type            input_requested;
pointer_to_any_type                     user_data_ptr;
df_physical_disk_label_block_ptr_type   disk_label_ptr;
struct dskusage                         dskusage;

/*.Implementation[=-------------------------------------------------
/*
/*   Determine the command type and process the command.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
uip = (dev_sd_unit_info_ptr_type)device_handle;

switch( command )
    {
    case DSKIOCGET:

/*.Implementation_Continued[=--------------------------------------
/*
/*      The command is DSKIOCGET, return disk parameters to the
/*      caller.
/*
/*.End]=-----------------------------------------------------------*/

        dskget.total_sectors = uip->sector_count;
        dskget.bytes_per_sector = uip->sector_size;
        dskget.controller_id = 0;
        status = sc_check_access((word_address_ptr_type)&parameter,
                                 usizeof(struct dskget), SC_WRITE_ACCESS);
        if (status == OK)
            {
            status = sc_write_bytes_to_user((pointer_to_any_type)&dskget,
                           (pointer_to_any_type)(word_address_type)parameter,
                           usizeof(struct dskget));
            }
```

```
        break;

    case DSKIOC_GENERIC_SCSI:
            scsi_gen_parm_ptr = (dev_scsi_generic_parm_ptr_type)parameter;

/*.Implementation_Continued[=----------------------------------------
/*
/*    Validate user buffers and copy contents into kernel address space.
/*
/*.End]=-------------------------------------------------------------*/

            sc_read_bytes_from_user((pointer_to_any_type)&scsi_gen_parm_ptr->cmd,
      (pointer_to_any_type)&local_scsi_gen_cmd,
      usizeof(dev_scsi_generic_cmd_blk_type));
            sc_read_bytes_from_user(
      (pointer_to_any_type)&scsi_gen_parm_ptr->data_ptr,
      (pointer_to_any_type)&user_data_ptr,
      usizeof(pointer_to_any_type));

/*.Implementation_Continued[=----------------------------------------
/*
/*   When the request is for data transfer (i.e.
/*   user_data_ptr is not NULL) the sense of
/*   <data_buffer_byte_size> provides the direction of the data
/*   transfer: a negative value indicates an input operation and
/*   positive indicates output. Thus obviating any need to
/*   translate the SCSI command to determine whether to mark
/*   the page frame(s) as been modified or merely referenced when
/*   unwiring the user's i/o buffer after a dma operation.
/*
/*
/*.End]=-------------------------------------------------------------*/

            if (user_data_ptr != 0)
      {
      sc_read_bytes_from_user(
      (pointer_to_any_type)&scsi_gen_parm_ptr->data_size,
      (pointer_to_any_type)&data_buffer_byte_size,
      usizeof(int32_type));
      if (data_buffer_byte_size < 0)
      {
      input_requested = TRUE;
      data_buffer_byte_size = -data_buffer_byte_size;
      }
      else
      {
      input_requested = FALSE;
      }

      if (data_buffer_byte_size > uip->max_request_size)
      {
      status = IO_EINVAL_ILLEGAL_REQUEST_SIZE;
      break;
      }

            status = sc_check_access((word_address_ptr_type)&user_data_ptr,
      (uint32_type)data_buffer_byte_size,
      SC_READ_ACCESS|SC_WRITE_ACCESS);
      if (status != OK)
      {
      break;
      }

            vm_wire_memory(user_data_ptr,
             TRUE,
             (uint32_type)data_buffer_byte_size);
      }
        else
            {
            user_data_ptr = DEV_SD_NULL_BUFFER_PTR;
            data_buffer_byte_size = 0;
            }
            if (status == OK)
      {
            io_sync_obtain_interleave_lock(&uip->request_lock);
```

```
            misc_dequeue_from_head(&uip->arb_free_queue,
      (misc_queue_links_ptr_type *)&arb_ptr);
            scsi_gen_cmd_ptr  = (dev_scsi_cmd_blk_ptr_type)
       &arb_ptr->scsi_cmd_blk;
        local_scsi_gen_cmd.lun = arb_ptr->unit_spec.unit;
      misc_byte_copy((byte_address_type)&local_scsi_gen_cmd,
      (pointer_to_any_type)scsi_gen_cmd_ptr,
      (uint32_type)sizeof(dev_scsi_cmd_blk_type));

      arb_ptr->request_flags    = (bit16e_type)0;

            io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
       user_data_ptr,
                              (uint32_type)data_buffer_byte_size);
           status = dev_sd_start_sync_request(uip, arb_ptr);
            data_bytes_transferred = io_get_buffer_vector_position(
         &arb_ptr->buffer_vector);
            misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
            if (io_assign_next_interleave_waiter(&uip->request_lock))
                {
                dev_sd_start_async_request(uip);
                }
            io_release_interleave_lock(&uip->request_lock);
      sc_write_bytes_to_user(
      (pointer_to_any_type)&data_bytes_transferred,
                        (pointer_to_any_type)&scsi_gen_parm_ptr->data_size,
                        (uint32_type)sizeof(uint32_type));
                        if (status == OK)
                        {
                        sense_keys[0]= (uint8e_type)0;
                        sense_keys[1]= (uint8e_type)0;
                        sense_keys[2]= (uint8e_type)0;
                        }
                        else
                        {
                        sense_keys[0]= (uint8e_type)status;
                        sense_keys[1]=
                        (uint8e_type)arb_ptr->sense_buffer.sense_key;
                        sense_keys[2]=
                            arb_ptr->sense_buffer.additional_sense_byte_5;
                        }
                        sc_write_bytes_to_user( (pointer_to_any_type)sense_keys,
                        (pointer_to_any_type)&scsi_gen_parm_ptr->status,
                        (uint32_type)sizeof(sense_keys));
                        }

           if (user_data_ptr != 0 && input_requested)
                        {
                        vm_mark_mod_and_ref_and_unwire_memory(user_data_ptr,
                        TRUE, (uint32_type)data_buffer_byte_size);
                        }
           else if (user_data_ptr != 0)
                        {
                        vm_mark_ref_and_unwire_memory(user_data_ptr,
                        TRUE, (uint32_type)data_buffer_byte_size);
                        }
           break;

    case DSKIOC_READ_DISK_LABEL:
    case DSKIOC_WRITE_DISK_LABEL:
        /*.Implementation_Continued[=---------------------------------------
        /*
/*  The request is to read/write the disk label. Get the open lock
/*  the the label can't be changed while we are accessing it.
/*
/*  Allocate memory for the local disk label and determine the
/*  operation type requested.
/*
/*  If the operation is a disk label write, make sure that we are
/*  the only process that has the disk unit open and read the
/*  label from user address space into our local buffer.
        /*
        /*.End]=-------------------------------------------------------------*/

        lm_obtain_unsequenced_lock(&dev_sd_open_lock);
```

```
        disk_label_ptr = (df_physical_disk_label_block_ptr_type)
                          vm_get_wired_memory((uint32_type)uip->sector_size,
                                              VM_DEFAULT_ALIGNMENT);

        if (command == DSKIOC_WRITE_DISK_LABEL)
             {
            if (uip->open_count != 1)
                 {
                 status = IO_EBUSY_DEVICE_HAS_OPEN_UNITS;
                 goto disk_label_op_failed;
                 }
                status = sc_check_access_and_read_bytes_from_user(
                             (word_address_ptr_type)&parameter,
                                     (pointer_to_any_type)disk_label_ptr,
                                     usizeof(df_physical_disk_label_block_type));
                if (status != OK)
                       {
                    goto disk_label_op_failed;
                       }
                }

/*.Implementation_Continued[=---------------------------------------
/*
/*  Allocate a SCSI adapter parameter block and build a read/write
/*  command block.
/*
/*.End]=-------------------------------------------------------------*/

            io_sync_obtain_interleave_lock(&uip->request_lock);
            misc_dequeue_from_head(&uip->arb_free_queue,
                                            (misc_queue_links_ptr_type *)&arb_ptr);
            scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                          &arb_ptr->scsi_cmd_blk;
            scsi_rw_cmd_blk_ptr->lun = uip->unit_spec.unit;
            scsi_rw_cmd_blk_ptr->logical_block_address = 0;
            scsi_rw_cmd_blk_ptr->transfer_length =  1;
            scsi_rw_cmd_blk_ptr->vendor_unique = 0;
            scsi_rw_cmd_blk_ptr->reserved1 = 0;
            scsi_rw_cmd_blk_ptr->link = FALSE;
            scsi_rw_cmd_blk_ptr->flag = FALSE;
            arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
        io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                        (pointer_to_any_type)disk_label_ptr,
                                          (uint32_type)uip->sector_size);

        if (command == DSKIOC_READ_DISK_LABEL)
             {
/*.Implementation_Continued[=---------------------------------
/*
/*  The request is to read the label. Call start_sync_request
/*   to perform the read operation. If the read completes with
/*  a good status, copy the label out to user space.
/*
/*.End]=-------------------------------------------------------*/

                scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ;
                status = dev_sd_start_sync_request(uip, arb_ptr);
            misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
            if (io_assign_next_interleave_waiter(&uip->request_lock))
                {
                dev_sd_start_async_request(uip);
                }
            io_release_interleave_lock(&uip->request_lock);
                if (status == OK)
                    {
                status = sc_check_access((word_address_ptr_type)&parameter,
                                         usizeof(df_physical_disk_label_block_type),
                                SC_WRITE_ACCESS);
                if (status == OK)
                {
                    status = sc_write_bytes_to_user(
                           (pointer_to_any_type)disk_label_ptr,
                                          (pointer_to_any_type)parameter,
                                          usizeof(df_physical_disk_label_block_type));
                    }
```

```
                        }
                    }
                else
                    {
/*.Implementation_Continued[=--------------------------------
/*
/*  The request is to write the label. Note that the caller's
/*  label has already been copied from user space. Call
/*  start_sync_request to perform the write operation.
/*
/*.End]=-----------------------------------------------------*/

                    scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE;
                    status = dev_sd_start_sync_request(uip, arb_ptr);
                misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
                if (io_assign_next_interleave_waiter(&uip->request_lock))
                    {
                    dev_sd_start_async_request(uip);
                    }
                io_release_interleave_lock(&uip->request_lock);
                    }

 /*.Implementation_Continued[=-------------------------------------
 /*
 /*  Release the memory used as a local container for the disk label
 /*  and release the disk open lock.
 /*
 /*.End]=-----------------------------------------------------------*/

disk_label_op_failed:
        vm_release_wired_memory((pointer_to_any_type)disk_label_ptr,
                                (uint32_type)uip->sector_size);
        lm_release_unsequenced_lock(&dev_sd_open_lock);
        break;

    case DSKIOCUSAGE:
/*.Implementation_Continued[=-------------------------------------
/*
/*  The request is for sar type disk activity information. Fill in
/*  a local dskusage structure and copy it out to the callers
/*  buffer. Start by getting the various counters. These can
/*  be copied out atomically and don't require any locks.
/*
/*.End]=-----------------------------------------------------------*/

        misc_get_value(&uip->read_block_count, &dskusage.read_block_count);
        misc_get_value(&uip->write_block_count, &dskusage.write_block_count);
        misc_get_value(&uip->read_request_count, &dskusage.read_request_count);
        misc_get_value(&uip->write_request_count, &dskusage.write_request_count);

/*.Implementation_Continued[=-------------------------------------
/*
/*  Now get the total response and total busy times for the unit.
/*  The unit lock is is required to insure exclusive access.
/*
/*.End]=-----------------------------------------------------------*/

            lm_obtain_unsequenced_lock(&uip->unit_lock);
        misc_clock_value_to_timeval(&uip->total_response_time,
                                    &dskusage.response_time);
        misc_clock_value_to_timeval(&uip->total_busy_time,
                                    &dskusage.busy_time);
            lm_release_unsequenced_lock(&uip->unit_lock);

/*.Implementation_Continued[=-------------------------------------
/*
/* Verify the callers access to the supplied dskusage structure
/* and copy out the information.
/*
/*.End]=-----------------------------------------------------------*/

        status = sc_check_access((word_address_ptr_type)&parameter,
                                 usizeof(struct dskusage), SC_WRITE_ACCESS);
        if (status == OK)
            {
```

```
            status = sc_write_bytes_to_user((pointer_to_any_type)&dskusage,
                        (pointer_to_any_type)(word_address_type)parameter,
                        usizeof(struct dskusage));
            }
        break;

    default:
        status = IO_EINVAL_COMMAND_NOT_SUPPORTED_BY_DEVICE;
        break;
    }

*return_value_ptr =   ((status == OK) ? 0 : -1);
/* sc_write_bytes_to_user(((status == OK) ? 0 : -1),
                        return_value_ptr, sizeof(uint32_type)); */
return(status);
}

/*.function */

            /*<-----------*/
WIRED
status_type  dev_sd_start_io (op_record_ptr)
            /*>-----------*/

io_operation_record_ptr_type        op_record_ptr;/* READ ONLY  */

/*.Summary[=-------------------------------------------------------------
/*
/*   This function starts an asynchronous read or write operation
/*   on the specified device.
/*
/*.Parameters
/*
/* op_record_ptr -- A pointer to the operation record for
/*   the asynchronous request.  The operation record contains fields
/*   indicating the device handle of the device that is the target
/*   of the operation, the operation to be performed, the offset on
/*   the device from which the operation is to commence, the size of
/*   the transfer, the address of the main memory buffer, and the
/*   addressof the function that is to be called when the operation
/*   completes.
/*
/*.Functional_Description
/*
/*   This function attempts to obtain the specified unit's request
/*   lock. If the lock is obtained, dev_sd_start_async_request is
/*   called to start the request. Control is returned to the
/*   caller as soon as the request has been issued through the
/*   supporting adapter to the disk unit. The I/O daemon handles
/*   request completion and starts the next request in the queue
/*   if there is one.
/*
/*   If the unit request lock cannot be obtained, the request is
/*   added to the unit request queue and the function returns
/*   immediately. The enqueued request is started when the
/*   currently executing request and all requests ahead in the
/*   queue have been    executed.
/*
/*.Return_Value
/*
/* OK -- The request was successfully started.  This status
/*   does not indicate that the request has completed successfully.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*

{

dev_sd_unit_info_ptr_type                       uip;
```

```
/*.Implementation[=-----------------------------------------------
/*
/*    We queue the asynchronous request and then check to see if
/*    we can get the unit request lock.  If we can't get a lock,
/*    we increment the number of waiters in the current asynchronous
/*    batch and return immediately. The request will be started when
/*    the unit becomes free.
/*
/*    Note that the request had to be queued before we checked if
/*    the unit was free. If unit becomes free immediately after the
/*    check, the request must be in the queue so that it can be
/*    started by the process which just completed a request.
/*
/*    If the unit is immediately available, then we proceed to start
/*    a request by calling a common asynchronous start routine. Note
/*    that while we are guaranteed to be able to dequeue a request
/*    if the unit is available, it may not be the same request we queued
/*    at the beginning of the routine because other requests may have
/*    been queued in the interim.
/*
/*    Dev_sd_start_async_request always returns a good status.
/*    Any errors that occur in starting the request must be reported
/*    via the completion routine because the process that actually
/*    calls dev_sd_start_async_request may not be the process that
/*    originated the request.
/*
/*.End]=-------------------------------------------------------------*/

uip = (dev_sd_unit_info_ptr_type)op_record_ptr->ri.device_handle;

misc_enqueue_at_tail(&uip->async_request_queue, &op_record_ptr->links);

if(io_async_obtain_interleave_lock(&uip->request_lock))
    {
    dev_sd_start_async_request(uip);
    io_release_interleave_lock(&uip->request_lock);
    }

return(OK);

}

/*.function */

        /*<-------*/
INITIALIZATION
void     dev_sd_init  ()
        /*>-------*/

/*.Summary[=-----------------------------------------------------------
/*
/*    This function performs pre-configuration initialization
/*    required for the sd driver at system boot-time.
/*
/*.Parameters
/*
/*    None.
/*
/*.Functional_Description
/*
/*    See Summary.
/*
/*
/*.Return_Value
/*
/*    None.
/*
/*.Exceptions
/*
/*    None.
/*

{
lm_initialize_unsequenced_lock(&dev_sd_open_lock);
}
```

```
/*.function */

                /*<-------------*/
UNWIRED
status_type      dev_sd_configure  (device_name_ptr, major_number)
                /*>-------------*/

char_ptr_type                          device_name_ptr;/* READ ONLY  */
io_major_device_number_type            major_number;           /* READ ONLY  */

/*.Summary[=-------------------------------------------------------------
/*
/*      This function configures the specified device if it is a
/*      SCSI disk device.
/*
/*.Parameters
/*
/* device_name_ptr -- A pointer to the character string name
/*   of the device to be configured.
/*
/* major_number -- The major device number on which the
/*   device is to be configured.
/*
/*.Functional_Description
/*
/*   This function configures the specified device if it is a SCSI
/*   disk device. Configuration includes allocation and initialization
/*   of controlling data structures, configuration of supporting
/*   SCSI adapter, SCSI disk device initialization, minor device
/*   number allocation, and creation of the appropriate /dev entries.
/*
/*   Dev_sd_parse_device_name is called to extract a device mnemonic,
/*   SCSI adapter name, SCSI adapter address, SCSI adapter number,
/*   SCSI id and unit number from the name string specified by
/*   <device_name_ptr>. The name string speicfied by <device_name_ptr>
/*   is of the form:
/*
/*       sd(<SCSI adapter mnemonic>(<SCSI adapter address>),SCSI id).
/*
/*   The device mnemonic must be "sd" for the device to belong to
/*   this driver. The SCSI adapter name and address are used as the
/*   name string to call the supporting adapter's configure routine.
/*   Adapter address may be specified as an adapter number which
/*   corresponds to the adapter's bus position relative to
/*   other adapters present.        The SCSI id identifies the disk devices
/*   location on the SCSI bus. A SCSI id of "*" indicates that all
/*   disk devices on the bus should be configured. The unit number
/*   field is ignored since all units at the specified SCSI id are
/*   configured.
/*
/*.Return_Value
/*
/* OK -- The device was successfully configured.
/*
/* DEV_ENXIO_ADAPTER_CONFIG_FAILED -- The SCSI adapter which
/*   supports the specified disk could not be configured.
/*
/* IO_ENXIO_DEVICE_DOES_NOT_EXIST -- No disks were found
/*   on the SCSI bus.
/*
/* IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED -- All disks found
/*   on the bus were already configured.
/*
/* IO_ENXIO_ALL_MINOR_NUMBERS_IN_USE -- The call to allocate
/*   a minor device number for a disk unit failed.
/*
/*   Return values for dev_sd_parse_device_name.
/*

{
status_type             status;
uint8_type              unit;
char_type                   adapter_name[IO_DEV_ADAPT_MAX_SPEC_SIZE];
uint8_type              scsi_id;
dev_sd_unit_info_ptr_type                   uip;
```

```
uint16_type                     disk_type;
dev_adapter_request_block_ptr_type              arb_ptr;
vp_event_type                                   delay_event;
int32_type                                      result_index;
fs_dev_request_type                 dev_entry_info;
uint32_type             length;
uint32_type             i;
uint32_type             k;
dev_sd_unit_info_ptr_type uip_table[DEV_SCSI_MAX_SCSI_IDS][DEV_SCSI_MAX_UNITS];
dev_sd_unit_info_ptr_type uips_to_free[DEV_SCSI_MAX_UNITS];
io_device_number_type               adapter_device_number;
boolean_type            configure_all_disks;
boolean_type            disk_found;
boolean_type            disk_found_and_registered;
dev_scsi_adapter_unit_registration_blk_type     unit_reg_blk;
dev_scsi_adapter_unit_options_block_type        unit_opt_blk;
dev_scsi_inquiry_cmd_blk_ptr_type               scsi_cmd_blk_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Parse the device name to see if the specified device belongs
/*   to this driver. Dev_sd_parse_device_name takes the adapter
/*   address information provided and returns the adapter address
/*   and adapter number. Adapter number corresponds to the adapter's
/*   bus position relative to other adapters present and
/*   is assigned as the adapter's minor device number.
/*   Io_sd_parse_device_name also converts SCSI id "*" to
/*   DEV_SCSI_ID_ALL.
/*
/*.End]=-----------------------------------------------------------*/

status = dev_sd_parse_device_name(device_name_ptr,
                        adapter_name,
                        &scsi_id,
                        &unit);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*      Call dev_scsi_adapter_configure to configure the supporting
/*      SCSI adapter if it has not already been configured. If the
/*   adapter has already been configured the status
/*   IO_ENXIO_DEVICE_CODE_ALREADY_ASSIGNED is returned and we
/*      interpret this as an OK status.
/*
/*.End]=-----------------------------------------------------------*/

status = dev_scsi_adapter_configure(adapter_name);

if   ((status != OK) && (status != IO_ENXIO_DEVICE_CODE_ALREADY_ASSIGNED))
    {
    goto done;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*      Call the adapter name to device routine to get the device
/*      number of the adapter. The device number identifies the class
/*   of adapter and a particular instance of an adapter of the class.
/*
/*.End]=-----------------------------------------------------------*/

status = dev_scsi_adapter_name_to_device(adapter_name,
                                &adapter_device_number);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*
```

```
/*   Call the adapter "device to name" routine to get the
/*   complete adapter specification which will be used to
/*   create the /dev entries.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_scsi_adapter_device_to_name(
                        adapter_device_number,
                        adapter_name,
                        (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   If <device_name_ptr> specified that all SCSI disks on the
/*   bus should be configured then we will attempt to configure
/*   all possible disk units at all possible SCSI ids.
/*
/*.End]=-------------------------------------------------------------*/

configure_all_disks = FALSE;
if (scsi_id == DEV_SCSI_ID_ALL)
     {
     configure_all_disks = TRUE;
     scsi_id = 0;
     }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Initialize to uip table. The uip table is used to keep
/*   track of which disk units are sucessfully configured.
/*
/*.End]=-------------------------------------------------------------*/

for (i = 0; i < DEV_SCSI_MAX_SCSI_IDS; i++)
    {
    for (k = 0; k < DEV_SCSI_MAX_UNITS; k++)
        {
            uip_table[i][k] = DEV_SD_NULL_UNIT_INFO_PTR;
            }
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Get the SCSI disk open lock and configure the requested
/*   devices. The open lock is used globally to synchronize
/*   all configure, deconfigure, and open operations done on
/*   SCSI disks.
/*
/*   Initialize the "uips_to_free" table to null. There are
/*   several places in the configuration process where something
/*   can go wrong and we need to back out of the configuration of a
/*   unit. The "uips_to_free" table is used to keep track of which
/*   unit configurations failed so that the cleanup can be done in
/*   one place instead of duplicating the cleanup code in every
/*   failure path.
/*
/*.End]=-------------------------------------------------------------*/

disk_found = FALSE;
disk_found_and_registered = FALSE;
lm_obtain_unsequenced_lock(&dev_sd_open_lock);
for ( i = 0; i < DEV_SCSI_MAX_SCSI_IDS; i++)
    {
    if (i != scsi_id)
            {
            continue;
            }

for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
    {
    uips_to_free[unit] = DEV_SD_NULL_UNIT_INFO_PTR;
```

```
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Determine if one or more disk units are at the current SCSI
/*   id.
/*
/*   For each possible unit at the SCSI id a unit information structure
/*   is allocated and registered with the supporting adapter. The
/*   adapter registration routine does the following:
/*
/* Checks if a device has already been registered at
/*   the specified SCSI id and unit number.
/*
/*   Allocates an adapter specific parameter block for
/*   the unit. A pointer to the parameter block is saved
/*   in the adapter bus table which maps adapter number,
/*   scsi id, and unit number to parameter block and unit
/*   information structure.
/*   Note that the structure is alloctated with no page cross.
/*   The inquiry buffer within this structure may be used
/*   for a DMA operation by the SCSI adapter.
/*
/*   Returns the maximum number of bytes that can be
/*   transferred through the disk interface in a single
/*   operation.
/*
/*
/*   A SCSI inquiry command is performed on each unit. If the unit
/*   does not identify itself as a disk, the unit information
/*   structure is deregistered and deallocated. Otherwise, if the
/*   unit is a disk, its unit information block is initialized.
/*
/*   The SCSI adapter driver's set unit options routine is called for
/*   each unit. This function is called to specify request timeout
/*   and retry count values.
/*
/*   If the inquiry command fails, we delay ten milliseconds and
/*   retry the command. Up to DEV_SCSI_MAX_RETRIES_AFTER_RESET delays
/*   and retries are performed before giving up because some devices
/*   require time to settle after a bus reset before they can respond
/*   to commands. The adpater configure routine resets the SCSI bus
/*   the first time it is .
/*
/*.End]=-------------------------------------------------------------*/

    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
        {
        uip = (dev_sd_unit_info_ptr_type)vm_get_wired_memory(
                        usizeof(dev_sd_unit_info_type),
                        VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS);
            uip->adapter_device_number = adapter_device_number;
        misc_initialize_queue(&uip->arb_free_queue);
        uip->unit_spec.scsi_id = scsi_id;
        uip->unit_spec.unit = unit;
        misc_initialize_queue(&uip->async_request_queue);
        io_initialize_interleave_lock(&uip->request_lock,
                            (uint16_type)DEV_SD_MAX_CONCURRENT_UNIT_REQUESTS);

/*.Implementation_Continued[=----------------------------------------
/*
/*   Attempt to register with the SCSI interface. If a device
/*   is already registered for the SCSI id and unit number,
/*   release the resources allocated for the configuration.
/*
/*.End]=-------------------------------------------------------------*/

        uip->adapter_handle = DEV_SD_NULL_ADAPTER_HANDLE;
            unit_reg_blk.adapter_device_number = adapter_device_number;
            unit_reg_blk.unit_spec.scsi_id = scsi_id;
            unit_reg_blk.unit_spec.unit = unit;
            unit_reg_blk.driver_handle = (io_device_handle_type)uip;
            unit_reg_blk.device_type = DEV_SD_DEVICE_TYPES_SUPPORTED;
            unit_reg_blk.max_concurrent_requests =
                            (uint16_type)DEV_SD_MAX_CONCURRENT_UNIT_REQUESTS;
```

```
        status = dev_scsi_adapter_register_requester(
                        adapter_device_number.major,
                        &unit_reg_blk);
        if (status != OK)
            {
                uips_to_free[unit] = uip;
                uip_table[scsi_id][unit] = DEV_SD_NULL_UNIT_INFO_PTR;
                if (unit_reg_blk.device_type & DEV_SD_DEVICE_TYPES_SUPPORTED)
                        {
                        disk_found = TRUE;
                    continue;
                        }
                else
                        {
                        break;
                        }
                }
        uip->adapter_handle = unit_reg_blk.adapter_handle;
        uip->max_request_size = unit_reg_blk.max_request_size;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Allocate memory for the SCSI adapter request blocks,
/*   initialize them, and enqueue them to the free queue for
/*   the unit.
/*
        /*.End]=-----------------------------------------------------------*/

        for (k = 0; k < DEV_SD_MAX_CONCURRENT_UNIT_REQUESTS; k++)
            {
                arb_ptr = (dev_adapter_request_block_ptr_type)
                      vm_get_wired_memory(
                             usizeof(dev_adapter_request_block_type),
                            VM_DEFAULT_ALIGNMENT);
            arb_ptr->type = DEV_SCSI_ARB_TYPE_SCSI_I;
            arb_ptr->unit_spec.scsi_id = scsi_id;
            arb_ptr->unit_spec.unit = unit;
            arb_ptr->adapter_handle = unit_reg_blk.adapter_handle;
            (void)misc_enqueue_at_tail(&uip->arb_free_queue,
                              &arb_ptr->links);
            }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Do the inquiry command to determine if a disk device
/*   exists at the current SCSI id and unit number. Note that
/*   we will use the default unit options (timeout, retries, ...)
/*   to perform the inquiry. Note that the default unit options
/*   for the SCSI interface are used to perform the inquiry.
/*
/*      The inquiry command is the first command issued to the
/*      target device. For various reasons (i.e. first command after
/*      reset, sync negotiation attempted on first command ...)
/*      some devices misbehave on the first command. As a result,
/*      we retry the inquiry command a couple of times as long
/*   as the command does not timeout. A timeout indicates that
/*   there is no device out present to accept the command.
/*
/*.End]=-----------------------------------------------------------*/

        io_sync_obtain_interleave_lock(&uip->request_lock);
        misc_dequeue_from_head(&uip->arb_free_queue,
                        (misc_queue_links_ptr_type *)&arb_ptr);
        for (k = 0; k < DEV_SCSI_MAX_RETRIES_AFTER_RESET; k++)
                {
            scsi_cmd_blk_ptr = (dev_scsi_inquiry_cmd_blk_ptr_type)
                                                &arb_ptr->scsi_cmd_blk;
            scsi_cmd_blk_ptr->op_code = DEV_SCSI_CMD_INQUIRY;
            scsi_cmd_blk_ptr->lun = unit;
            scsi_cmd_blk_ptr->reserved1 = 0;
            scsi_cmd_blk_ptr->reserved2 = 0;
            scsi_cmd_blk_ptr->reserved3 = 0;
            scsi_cmd_blk_ptr->alloc_len = usizeof(dev_scsi_inquiry_buffer_type);
            scsi_cmd_blk_ptr->vendor_unique = 0;
            scsi_cmd_blk_ptr->reserved4 = 0;
```

```
            scsi_cmd_blk_ptr->link = FALSE;
            scsi_cmd_blk_ptr->flag = FALSE;
            arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
            io_init_one_entry_buffer_vector(
                        &arb_ptr->buffer_vector,
                        (pointer_to_any_type)&uip->inquiry_buffer,
                        (uint32_type)scsi_cmd_blk_ptr->alloc_len);
            status = dev_sd_start_sync_request(uip, arb_ptr);
                if (status == OK)
                    {
                if (!((1 << uip->inquiry_buffer.device_type) &
                                        DEV_SD_DEVICE_TYPES_SUPPORTED))
                        {
                            status = IO_ENXIO_DEVICE_NOT_SUPPORTED;
                            }
                    }
                if (status == OK)
                        {
                        break;
                        }
            if   ((status == IO_ENXIO_UNIT_NOT_READY) || (status ==
                        IO_EIO_PHYSICAL_UNIT_FAILURE))
                        {
/*.Implementation_Continued[=----------------------------------
/*   If the unit is telling us that it is not ready, delay
/*  for a second to give it time to recover from the last
/*  request (i.e. SCSI bus reset, sync negotiation).
/*
/*.End]=-------------------------------------------------------*/

                    vp_create_clock_event(&delay_event, &misc_one_second);
                        vp_await_ec(&delay_event, (int32_type)1, &result_index);
                            }
                    }

            misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
            if (io_assign_next_interleave_waiter(&uip->request_lock))
                {
                dev_sd_start_async_request(uip);
                }
            io_release_interleave_lock(&uip->request_lock);

/*.Implementation_Continued[=----------------------------------------
/*
/*   If the inquiry command failed, update the uips_to_free table
/*   so that all resources associated with the unit will be
/*   released.
/*
/*.End]=-------------------------------------------------------------*/

            if (status != OK)
                {
                    uips_to_free[unit] = uip;
                    uip_table[scsi_id][unit] = DEV_SD_NULL_UNIT_INFO_PTR;
                    if (status == IO_EIO_DEVICE_TIMED_OUT)
                            {
/*.Implementation_Continued[=------------------------------
/*
/*  We only expect a timeout to occur on an inquiry if no
                /*  units are present at a SCSI id. As a result, if a
                /*  timeout is detected here we don't bother to check for
                /*  any more units at the SCSI id. Doing this shortens the
                /*  time it takes to auto-configure the disks.
/*
/*.End]=-----------------------------------------------------*/
                            break;
                            }

                    else
                            {
                        continue;
                            }
                }

/*.Implementation_Continued[=--------------------------------------
```

```
/*
/*   A disk was found at the current unit number and SCSI id.
/*   Complete the initialization of the uip and record that a
/*      disk was actually found.
/*
/*.End]=-------------------------------------------------------------*/

          disk_found_and_registered = TRUE;
        uip_table[scsi_id][unit] = uip;
        uip->device_number.major = major_number;
        uip->open_count = 0;
        uip->writer_count = 0;
        uip->exclude_writers_count = 0;
          uip->inhibit_error_logging = FALSE;
        uip->sector_size = DEV_SD_STANDARD_SECTOR_SIZE;
        lm_initialize_unsequenced_lock(&uip->unit_lock);
        misc_initialize_counter(&uip->read_block_count, (int32e_type)0);
        misc_initialize_counter(&uip->write_block_count, (int32e_type)0);
        misc_initialize_counter(&uip->read_request_count, (int32e_type)0);
        misc_initialize_counter(&uip->write_request_count, (int32e_type)0);
        MISC_CLOCK_VALUE_SET_TO_ZERO(&uip->total_response_time);
        MISC_CLOCK_VALUE_SET_TO_ZERO(&uip->total_busy_time);

        } /* End of unit for loop */


/*.Implementation_Continued[=----------------------------------------
/*
/*   Set the unit options for each disk unit that was found.
/*   The timeouts selected for the device are based on the
/*   device type.  A relatively short timeout is used for
/*   conventional hard disks since they respond very quickly.
/*   A longer timeout value is needed for slower direct access
/*   devices like floppies, WORMS, and CD-ROMS. For now,
/*   fifteen seconds seems to be enough time for the slower
/*   devices. However, just to be extra safe we allow thirty
/*   seconds.
/*
/*   Unit options are set only after all the inquiries at the
/*   SCSI id have been done. This is done because some SCSI
/*   interfaces do not allow options for units to be set
/*   individually. A set_unit_options request for a particular
/*   unit applies to all units at the SCSI id. The set_unit_options
/*   must be done after all the inquiries so that we don't set a
/*   large timeout for all luns before all the units have been
/*   querried.
/*
/*.End]=-------------------------------------------------------------*/

    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
        {
            uip = uip_table[scsi_id][unit];
        if (uip != DEV_SD_NULL_UNIT_INFO_PTR)
                {
                unit_opt_blk.sense_bytes = usizeof(
                          dev_scsi_request_sense_buffer_type);
                status = dev_sd_determine_disk_type(uip, &disk_type);

            if   ((status == OK) && (disk_type == DEV_SD_DISK_TYPE_RIGID))
                {
                    unit_opt_blk.bus_request_timeout_ptr = &misc_two_seconds;
                    unit_opt_blk.disconnect_timeout_ptr  = &misc_five_seconds;
                        }
                else
                        {
                    unit_opt_blk.bus_request_timeout_ptr = &misc_fifteen_seconds;
                    unit_opt_blk.disconnect_timeout_ptr  = &misc_fifteen_seconds;
                        }
                unit_opt_blk.max_disconn_reconn_per_command = 4;
                unit_opt_blk.adapter_retries = 3;
                unit_opt_blk.synchronous_data_transfers = FALSE;
                unit_opt_blk.perform_request_sorting = TRUE;
            status = dev_scsi_adapter_set_unit_options(
                          adapter_device_number.major,
                          uip->adapter_handle,
```

```
                        &unit_opt_blk);
                if (status != OK)
                        {
                        uip_table[scsi_id][unit] = DEV_SD_NULL_UNIT_INFO_PTR;
                    uips_to_free[unit] = uip;
                        }
                }
        }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Allocate the minor device numbers for all the disk units located
/*   at the current SCSI id. If an error occurs during minor number
/*   allocation, back out by releasing all minor numbers allocated
/*   for devices at the current SCSI id.
/*
/*.End]=-----------------------------------------------------------*/

    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
        {
        if (uip_table[scsi_id][unit] == DEV_SD_NULL_UNIT_INFO_PTR)
            {
                continue;
                }
        status = io_allocate_device_number(
                            major_number,
                            (bit32e_type)uip_table[scsi_id][unit],
                            unit,
                            &uip_table[scsi_id][unit]->device_number.minor);
        if (status != OK)
                {
                while (unit != 0)
                    {
                    unit--;
                    if (uip_table[scsi_id][unit] != DEV_SD_NULL_UNIT_INFO_PTR)
                {
                            uip_table[scsi_id][unit] = DEV_SD_NULL_UNIT_INFO_PTR;
                        uips_to_free[unit] = uip;
                            }
                    }
                    goto config_next_scsi_id;
                }
        }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Add the disks found at the current SCSI id to the list of
/*   disks to be implicitly registered as part of system
/*   initialization.
/*
/*.End]=-----------------------------------------------------------*/

    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
        {
        if (uip_table[scsi_id][unit] != DEV_SD_NULL_UNIT_INFO_PTR)
                {
            io_add_to_register_list(uip_table[scsi_id][unit]->device_number);
            }
        }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Create appropriate /dev/pdsk and /dev/rpdsk entries for the
/*   units at this SCSI id.   We create block and character
/*   special entries with permissions set to 0640.
/*
/*.End]=-----------------------------------------------------------*/

    dev_entry_info.operation = Fs_Dev_Request_Operation_Create;
    length = usizeof(dev_entry_info.dirname);
    misc_string_copy("pdsk", dev_entry_info.dirname, &length);
    dev_entry_info.op.create.mode_bits = S_IFBLK | S_IRUSR | S_IWUSR;
    for (k=0; k <= 1; k++)
        {
        for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
```

```
            {
        if (uip_table[scsi_id][unit] == DEV_SD_NULL_UNIT_INFO_PTR)
            {
                continue;
                }
            dev_entry_info.op.create.device = uip_table[scsi_id][unit]->
                                              device_number;
            misc_format_line(dev_entry_info.filename,
                        usizeof(dev_entry_info.filename),
                        "sd(%s,%d,%d)",
                        (bit32e_type)adapter_name,
                        (bit32e_type)scsi_id,
                        (bit32e_type)unit);
            fs_submit_dev_request(&dev_entry_info);
            }
        length = usizeof(dev_entry_info.dirname);
        misc_string_copy("rpdsk", dev_entry_info.dirname, &length);
        dev_entry_info.op.create.mode_bits = S_IFCHR | S_IRUSR | S_IWUSR;
        }

config_next_scsi_id:

/*.Implementation_Continued[=---------------------------------------
/*
/*   Perform cleanup of unit information structures for units
/*   which could not be configured.
/*
/*.End]=-------------------------------------------------------------*/

    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
        {
        uip = uips_to_free[unit];
        if (uip == DEV_SD_NULL_UNIT_INFO_PTR)
            {
                continue;
                }
            if (uip->adapter_handle != DEV_SD_NULL_ADAPTER_HANDLE)
                {
            dev_scsi_adapter_deregister_requester(
                        adapter_device_number.major,
                        uip->adapter_handle);
                }
        misc_dequeue_from_head(&uip->arb_free_queue,
                               (misc_queue_links_ptr_type *)&arb_ptr);
        while   ((misc_queue_links_ptr_type)arb_ptr !=
MISC_QUEUE_NULL_LINKS_PTR)
            {

            vm_release_wired_memory((pointer_to_any_type)arb_ptr,
                            usizeof(*arb_ptr));
            misc_dequeue_from_head(&uip->arb_free_queue,
                                   (misc_queue_links_ptr_type *)&arb_ptr);
            }
        vm_release_wired_memory((pointer_to_any_type)uip, usizeof(*uip));
        }


    if (configure_all_disks)
        {
            scsi_id++;
        }
    } /* End SCSI ID for loop */

lm_release_unsequenced_lock(&dev_sd_open_lock);

/*.Implementation_Continued[=---------------------------------------
/*
/*   Report an error only if no disks were configured. Search the
/*   local uip table to see if any disks were configured. If the
/*   table is empty, one of three possible errors have occurred:
/*
/* IO_ENXIO_DEVICE_DOES_NOT_EXIST - No disk devices were
/*   found at the requested SCSI bus locations.
/* IO_ENXIO_DEVICE_ALREADY_CONFIGURED -  All disks devices
/*   found at the requested SCSI bus locations were already
```

```
/*   configured.
/* IO_ENXIO_ALL_MINOR_NUMBERS_IN_USE -  One or more
/*   configurable disks were found but minor numbers
/*   could not be allocated for them.
/*
/*
/*
/*.End]=-------------------------------------------------------------*/

status = IO_ENXIO_DEVICE_DOES_NOT_EXIST;
for (scsi_id = 0; scsi_id < DEV_SCSI_MAX_SCSI_IDS; scsi_id++)
    {
    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
        {
            if (uip_table[scsi_id][unit] != DEV_SD_NULL_UNIT_INFO_PTR)
                {
                status = OK;
                }
            }
    }
if (status != OK)
    {
    if (disk_found)
        {
            if (disk_found_and_registered)
                {
                status = IO_ENXIO_ALL_MINOR_NUMBERS_IN_USE;
                }
            else
                {
                status = IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED;
                }
        }
    }
done:
return(status);

}

/*.function */

               /*<---------------*/
UNWIRED
status_type     dev_sd_deconfigure   (device_name_ptr)
               /*>---------------*/

char_ptr_type                        device_name_ptr;/* READ ONLY  */

/*.Summary[=-----------------------------------------------------------
/*
/*   This function deconfigures the specified device if it is a
/*   SCSI disk device.
/*
/*.Parameters
/*
/* device_name_ptr -- A pointer to the null-terminated string
/*   specifying the device to be deconfigured.
/*
/*.Functional_Description
/*
/*   This function deconfigures all units associated with the specified
/*   device if it is a SCSI disk device.  The minor device numbers
/*   assigned to the units are made available for reuse. The device
/*   is deregistered with the SCSI adapter manager and all memory
/*   used to control the device is released.
/*
/*.Return_Value
/*
/* OK -- The device was successfully deconfigured.
/*
/* IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED -- The given
/*   device specification was not the name of a SCSI disk device.
/*
/* IO_EBUSY_DEVICE_HAS_OPEN_UNITS -- The device could
/*   not be deconfigured because one or more units are still
```

```
/*   open.
/*
/*.Exceptions
/*
/*   None.
/*

{

status_type                          status;
uint8_type                  unit;
uint8_type                  scsi_id;
dev_sd_unit_info_ptr_type                uip_table[DEV_SCSI_MAX_UNITS];
dev_sd_unit_info_ptr_type                uip;
char_type                           adapter_name[IO_DEV_ADAPT_MAX_SPEC_SIZE];
io_device_number_type               adapter_device_number;
dev_scsi_adapter_unit_spec_type     unit_spec;
dev_adapter_request_block_ptr_type  arb_ptr;


/*.Implementation[=-------------------------------------------------
/*
/*      Parse the device name to see if it belongs to this driver.
/*      If not, return the error.
/*
/*.End]=-----------------------------------------------------------*/

status = dev_sd_parse_device_name(device_name_ptr,
                                  adapter_name,
                                  &scsi_id,
                                  &unit);
if (status != OK)
    {
    return(status);
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Get the SCSI disk open lock so that no other operations can
/*   be performed on the device while it is being deconfigured.
/*   Call scsi adapter name_to_device routine to get the supporting
/*   SCSI adapter's device number.
/*
/*.End]=-----------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_sd_open_lock);
status = dev_scsi_adapter_name_to_device(adapter_name,
                                         &adapter_device_number);

if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Find all the disk units at the specified SCSI id. If any one
/*   of the units is still open, abort the deconfigure and return
/*   the error.
/*
/*.End]=-----------------------------------------------------------*/

for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
    {
    uip_table[unit] = DEV_SD_NULL_UNIT_INFO_PTR;
    status = dev_scsi_adapter_get_device_info(
                    adapter_name,
                    unit_spec,
                    DEV_SD_DEVICE_TYPES_SUPPORTED,
                    (io_device_handle_ptr_type)&uip);
     if (status == OK)
        {
        uip_table[unit] = uip;
        if (uip->open_count != 0)
                {
```

```
            status = IO_EBUSY_DEVICE_HAS_OPEN_UNITS;
                goto done;
                }
        }

    }

/*.Implementation_Continued[=------------------------------------
/*
/*   For each unit found, give back the memory used for the adapter
/*   request blocks and unit information structure. Also, deallocate
/*   the device number used for the unit.
/*
/*.End]=-------------------------------------------------------------*/

for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
    {
    if (uip_table[unit] != DEV_SD_NULL_UNIT_INFO_PTR)
        {
        dev_scsi_adapter_deregister_requester(adapter_device_number.major,
                                              uip->adapter_handle);
        misc_dequeue_from_head(&uip->arb_free_queue,
                               (misc_queue_links_ptr_type *)&arb_ptr);
        while   ((misc_queue_links_ptr_type)arb_ptr !=
MISC_QUEUE_NULL_LINKS_PTR)
            {
            vm_release_wired_memory((pointer_to_any_type)arb_ptr,
                            usizeof(*arb_ptr));
            misc_dequeue_from_head(&uip->arb_free_queue,
                                   (misc_queue_links_ptr_type *)&arb_ptr);
            }
        io_deallocate_device_number(uip->device_number);
        vm_release_wired_memory((pointer_to_any_type)uip, usizeof(*uip));
        }
    }

done:
lm_release_unsequenced_lock(&dev_sd_open_lock);
return(status);
}

/*.function */

            /*<------------------*/
UNWIRED
status_type  dev_sd_name_to_device (device_name_ptr, number_ptr)
            /*>------------------*/

char_ptr_type                       device_name_ptr;/* READ ONLY  */
io_device_number_ptr_type           number_ptr;/* WRITE ONLY */

/*.Summary[=---------------------------------------------------------------
/*
/*   This function translates the specified device_name into a
/*   device number, if <device_name_ptr> names a configured
/*   SCSI disk.
/*
/*.Parameters
/*
/* device_name_ptr -- A pointer to the null-terminated device
/*   name that is to be translated.
/*
/* number_ptr -- A pointer to where the corresponding
/*   device number is to be written.
/*
/*.Functional_Description
/*
/*   See Summary.
/*
/*.Return_Value
/*
/*
/* OK -- The device name was successfully translated.
/*   Return values from dev_sd_parse_device_name.
/*   Return values from the supporting adapter's get_device_info routine.
```

```
/*
/*.Exceptions
/*
/*   None.
/*


{

status_type                status;
char_type                  adapter_name[IO_DEV_ADAPT_MAX_SPEC_SIZE];
io_device_number_type                adapter_device_number;
dev_scsi_adapter_unit_spec_type   unit_spec;
dev_sd_unit_info_ptr_type             uip_ptr;

/*.Implementation[=----------------------------------------------
/*
/*   Parse the device name to see if it belongs to this driver.
/*   If so, validate that the SCSI id specifies a unique SCSI
/*   bus location.
/*
/*.End]=-------------------------------------------------------------*/

status = OK;
status = dev_sd_parse_device_name(device_name_ptr,
                          adapter_name,
                          &unit_spec.scsi_id,
                          &unit_spec.unit);
if (status != OK)
    {
    goto name_not_recognized;
    }
if (unit_spec.scsi_id == DEV_SCSI_ID_ALL)
    {
    status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
    goto name_not_recognized;
    }

/*.Implementation_Continued[=-----------------------------------
/*
/*   Get the SCSI disk open lock so the device cannot be deconfigured
/*   while we are mapping device name to number. Call scsi adapter
/*   name_to_device routine to get the supporting SCSI adapter's
/*   device number.
/*
/*.End]=-------------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_sd_open_lock);
status = dev_scsi_adapter_name_to_device(adapter_name,
                                  &adapter_device_number);

if (status != OK)
    {
    goto error_release_lock;
    }

/*.Implementation_Continued[=-----------------------------------
/*
/*   Call the supporting SCSI adapter's get_device_info routine
/*   to get a pointer to the physical disk's unit information
/*   structure. If the unit information structure pointer is
/*   successfully returned, get the disk unit's device number
/*   and return it to the caller.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_scsi_adapter_get_device_info(
                              adapter_name,
                              unit_spec,
                              DEV_SD_DEVICE_TYPES_SUPPORTED,
                              (io_device_handle_ptr_type)&uip_ptr);
if (status == OK)
    {
    *number_ptr = uip_ptr->device_number;
    }
```

```
error_release_lock:
lm_release_unsequenced_lock(&dev_sd_open_lock);
name_not_recognized:
return(status);
]

/*.function */

                /*<------------------*/
UNWIRED
status_type     dev_sd_device_to_name  (device_number, name_ptr, size)
                /*>------------------*/

io_device_number_type   device_number;/* READ ONLY  */
char_ptr_type           name_ptr;     /* WRITE ONLY */
uint32_type             size;         /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------------
/*
/*   This function returns the character string name associated with
/*   the specified device number.
/*
/*.Parameters
/*
/* device_number -- The device number for which is the
/*   character string name is wanted.
/*
/* name_ptr -- A pointer to where the null-terminated
/*   character string name is to be written.
/*
/* size -- The maximum number of bytes, including the terminating
/*   null, that is to be written to <name_ptr>.
/*
/*.Functional_Description
/*
/*   The given device number is mapped to a SCSI id, SCSI adapter and
/*   unit number, and a string of the form
/*
/*   sd(<SCSI adapter mnemonic>@<device code>(<SCSI adapter address>),
/*           <SCSI id>, <unit number>)
/*
/*   is returned.
/*
/*.Return_Value
/*
/* OK -- The translation was performed successfully.
/*
/* IO_ENXIO_DEVICE_IS_NOT_CONFIGURED -- The specified
/*   device number is not configured.
/*
/*.Exceptions
/*
/*   None.
/*

{

status_type                         status;
dev_sd_unit_info_ptr_type               uip;
char_type                           adapter_name[IO_DEV_ADAPT_MAX_SPEC_SIZE];
uint16_type                         unit;


/*.Implementation[=-------------------------------------------------
/*
/*   Get the SCSI disk open lock and map the specified device
/*   number to a unit information structure.
/*
/*.End]=-----------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_sd_open_lock);
status = io_map_device_number(device_number, (bit32e_ptr_type)&uip, &unit);
if (status != OK)
    {
    goto done;
```

```
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Call the supporting adapter's "device to name" routine to
/*   get the complete adapter specification.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_scsi_adapter_device_to_name(
                        uip->adapter_device_number,
                        adapter_name,
                                (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE);

/*.Implementation_Continued[=----------------------------------------
/*
/*   Produce a formatted device name from information in the disk
/*   uip and the adapter spec.
/*
/*.End]=-------------------------------------------------------------*/

if (status == OK)
    {
    misc_format_line(name_ptr,
                            size,
                            "sd(%s,%d,%d)",
                            (bit32e_type)adapter_name,
                    (bit32e_type)uip->unit_spec.scsi_id,
                            (bit32e_type)unit);
    }
done:
lm_release_unsequenced_lock(&dev_sd_open_lock);
return(status);

}

/*.function */

                /*<---------------------*/
WIRED
status_type     dev_sd_parse_device_name  (device_spec_ptr,
                            adapter_name_ptr,
                            scsi_id_ptr,
                            unit_ptr)
                /*>---------------------*/

char_ptr_type                           device_spec_ptr;      /* READ ONLY  */
char_ptr_type                           adapter_name_ptr;     /* WRITE ONLY */
uint8_ptr_type                          scsi_id_ptr;          /* WRITE ONLY */
uint8_ptr_type                          unit_ptr;             /* WRITE ONLY */

/*.Summary[=---------------------------------------------------------
/*
/*   This function parses the specified device name, determines
/*   whether it is the name of a scsi disk device, and if so, returns
/*   the parsed information about the device.
/*
/*.Parameters
/*
/* device_spec_ptr -- A pointer to the null-terminated
/*   character string identifying the device spec to be parsed.
/*
/* adapter_name_ptr -- A pointer to where the SCSI adapter
/*   name which is embedded in <device_name_ptr> is to be returned.
/*
/* scsi_id_ptr -- A pointer to where the SCSI id from
/*   <device_name_ptr> is to be returned.
/*
/* unit_ptr -- A pointer to where the unit number from
/*   <device_name_ptr> is to be returned.
/*
/*.Functional_Description
/*
/*   See Summary.
/*
```

```
/*.Return_Value
/*
/* OK -- The device was successfully configured.
/*
/* IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED -- The given
/*   device specification was not the name of a SCSI disk device.
/*
/*.Exceptions
/*
/*   None.
/*

{

status_type                 status;
uint32_type                 hex_number;
io_dev_adapt_info_type      adapter_info;
int32_type                  spec_size;

/*.Implementation[=-----------------------------------------------
/*
/*   Call the generic parse device spec routine to
/*   break the device spec into its components.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;

if ( !io_parse_device_spec(device_spec_ptr, &adapter_info, &spec_size))
    {
    status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*    See if the device mnemonic returned by the parse device
/*    spec routine matches the mnemonic that specifies a
/*    device under the juristiction of this driver. If not,
/*    return the error.
/*
/*.End]=-----------------------------------------------------------*/

if (misc_string_compare((byte_address_type)adapter_info.name,
                        "sd",
                        (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE) != 0)
    {
    status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*   Copy the scsi adapter spec parameter to <adapter_name_ptr>.
/*   If no adpater was specified, return an error.
/*
/*.End]=-----------------------------------------------------------*/

if (*adapter_info.params[0] == ' ')
    {
    status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
    goto done;
    }
spec_size = IO_DEV_ADAPT_MAX_SPEC_SIZE;
(void)misc_string_copy((pointer_to_any_type)adapter_info.params[0],
                       (pointer_to_any_type)adapter_name_ptr,
                    (uint32_ptr_type)&spec_size);

/*.Implementation_Continued[=-------------------------------------
/*
/*   Return the SCSI id of the disks device. If a SCSI id was not
/*   specified, use the default. If SCSI id is specified as '*',
/*   return the id code which indicates that all SCSI disks in
/*   the system are being named.
/*
```

```
/*.End]=-------------------------------------------------------------*/

if (*adapter_info.params[1] == ' ')
    {
    *scsi_id_ptr = DEV_SD_DEFAULT_SCSI_ID;
    }
else if (*adapter_info.params[1] == '*')
    {
    *scsi_id_ptr = DEV_SCSI_ID_ALL;
    }
else
    {
    if (io_hex_str_to_int(adapter_info.params[1],
                                &hex_number))
        {
            status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
            goto done;
            }
    *scsi_id_ptr = (uint8_type)hex_number;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Return the unit number of the disk device. If a unit number
/*   was not specified, return the default unit number of zero.
/*   Note that the unit number is assumed to be in the third
/*   parameter field of the device spec. The memo "Device
/*   Specifications for Industry Standard Machines" defines this
/*   field to be a file number. However, since file number does
/*   not apply to disk device configuration, this field is used
/*   to specify the unit number of the device. In the future,
/*   a fourth parameter to the device spec may be added to
/*   specifiy the unit number.
/*
/*.End]=-------------------------------------------------------------*/

if (*adapter_info.params[2] == ' ')
    {
    *unit_ptr = DEV_SD_DEFAULT_UNIT_NUMBER;
    }
else
    {
    if (io_hex_str_to_int(adapter_info.params[2],
                                &hex_number))
        {
            status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
            }
    *unit_ptr = (uint8_type)hex_number;
    }

done:
return(status);

}

/*.function */

                /*<--------------*/
WIRED
void             dev_sd_complete_io  (data, status)
                /*>--------------*/

bit32e_type             data;       /* READ ONLY  */
status_type         status;

/*.Summary[=---------------------------------------------------------
/*
/*   This function handles the completion of asynchronous requests
/*   that have been completed by the disk controller.
/*
/*.Parameters
/*
/* data -- The 32 bits of data that was in the message
/*   given to the driver demon.
/*
```

```
/*.Functional_Description
/*
/*   This function handles the completion of asynchronous I/O
/*   requests. The completion status of the I/O operation is
/*   determined. Additional information about the request status
/*   is obtained from sense information returned by the unit if
/*   necessary.
/*
/*   Also, if the request indicates that sector buffering is in
/*   progress, this function calls dev_sd_complete_async_sb_io
/*   to determine the proper action to take.
/*
/*   When the result of the operation is determined, this function
/*   calls the "complete_io" function specified in the operation
/*   record.  This is an upcall to notify the requestor that the
/*   asynchronous operation is now complete.
/*
/*.Return_Value
/*
/* OK -- The operation completed successfully.
/*
/*   Return values from dev_sd_evaluate_sense_info.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*

{

dev_adapter_request_block_ptr_type  arb_ptr;
io_operation_record_ptr_type                    op_record_ptr;
dev_sd_unit_info_ptr_type           uip;
uint32_type             bytes_requested;
pointer_to_any_type                 buffer_ptr;
misc_clock_value_type               busy_time;
misc_clock_value_type               cur_time;
boolean_type            done;

/*.Implementation[=-------------------------------------------------
/*
/*   An asynchronous I/O request has just completed. If the
/*   completion status indicates that the command completed
/*   with a check condition status, the sense information is
/*   evaluated.
/*
/*.End]=-----------------------------------------------------------*/

arb_ptr = (dev_adapter_request_block_ptr_type)data;
op_record_ptr = arb_ptr->op_record_ptr;
uip = (dev_sd_unit_info_ptr_type)(op_record_ptr->ri.device_handle);

if (status == DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION)
    {
/*.Implementation_Continued[=--------------------------------------
/*
/*   Sense information has been delivered. The sense information
/*   is interpreted and the appropriate status is returned to
/*   the caller.
/*
/*
/*.End]=-----------------------------------------------------------*/

    status = dev_sd_evaluate_sense_info(uip, arb_ptr);
    if (status == DEV_SCSI_SENSE_KEY_RECOVERED_ERROR)
       {
       status = OK;
       }
    done = TRUE;
    }
```

```
if (status == OK)
    {

    io_get_buffer_vector_io_info(&op_record_ptr->ri.buffer_vector,
                                 &buffer_ptr,
                                 &bytes_requested);

    if (arb_ptr->request_flags & DEV_SCSI_REQUEST_FLAGS_SB)
       {
       status = dev_sd_complete_async_sb_io(arb_ptr, &done);
       }

    if (op_record_ptr->ri.op & IO_OPERATION_CHECK_SELF_ID)
       {
       if (!fs_check_self_id(buffer_ptr,
                             &op_record_ptr->ri.self_id,
                             &bytes_requested))
          {
          status = IO_EINVAL_BAD_SELF_ID;
          }
       }
    }
else if (status == DEV_EIO_SD_UNIT_ATTENTION_ON)
    {
/*.Implementation_Continued[=---------------------------------------
/*
/*   Map any none exported statuses to something the buffer
/*   manager will understand.
/*
/*.End]=-------------------------------------------------------------*/

    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    done = TRUE;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   Return the adapter request block to the free queue, release the
/*   unit request lock, and get the next asynchronous request started
/*   if there is one. Note that after we release the lock we must
/*   not reference the adapter request block any more.
/*
/*.End]=-------------------------------------------------------------*/

if (done)
   {
   if (status == OK)
      {
      io_add_to_buffer_vector_position(&op_record_ptr->ri.buffer_vector,
                                       (int32_type)bytes_requested);
      }
   misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
   if (io_assign_next_interleave_waiter(&uip->request_lock))
      {
      dev_sd_start_async_request(uip);
      }

/*.Implementation_Continued[=---------------------------------------
/*
/*   One way or another, we are finished processing this request.
/*   Calculate the response and busy times required to process this
/*   request and add them to the totals for the unit. The unit
/*   lock is held while the total is updated to protect against
/*   simultaneous updates.
/*
/*   Upcall the completion routine for this request, passing the
/*   original operation record and the status.
/*
/*.End]=-------------------------------------------------------------*/

   vp_read_system_clock(&cur_time);
   MISC_CLOCK_VALUE_SUBTRACT(&arb_ptr->async_request_start_time, &cur_time);
   lm_obtain_unsequenced_lock(&uip->unit_lock);
   MISC_CLOCK_VALUE_ADD(&cur_time, &uip->total_response_time);
   MISC_CLOCK_VALUE_ADD(&busy_time, &uip->total_busy_time);
```

```
   lm_release_unsequenced_lock(&uip->unit_lock);

   (*(op_record_ptr->completion_routine))(op_record_ptr, status);
   }
return;
}

/*.function */

                /*<-----------------------*/
WIRED
void             dev_sd_start_async_request       (uip)
                /*>-----------------------*/

dev_sd_unit_info_ptr_type                         uip;        /* READ ONLY  */


/*.Summary[=-------------------------------------------------------------
/*
/*   Start an asynchronous I/O request.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit info structure for the
/*   device on which an asynchronous request is to be started.
/*
/*.Functional_Description
/*
/*   An operation record is removed from the head of the async queue
/*   and the unit's adpater request block is filled in with request
/*   information. A timeout value is established, and then the command
/*   is issued through the supporting SCSI adapter to the disk unit
/*   to start the request.
/*
/*   Also, this function checks to see whether the sector size of
/*   the disk is greater then 512-bytes/sector.  If so, a buffer
/*   is allocated which is a multiple of the sector size.  This
/*   buffer becomes the actual io buffer.  dev_sd_complete will
/*   then determine what needs to be done with this data.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Exceptions
/*
/*   None.
/*

{

io_operation_record_ptr_type                    op_record_ptr;
dev_adapter_request_block_ptr_type  arb_ptr;
pointer_to_any_type                 buffer_ptr;
uint32_type             count;
dev_scsi_read_write_cmd_blk_ptr_type    scsi_rw_cmd_blk_ptr;
dev_scsi_write_verify_cmd_blk_ptr_type          scsi_write_verify_cmd_blk_ptr;
uint32_type             sector_buffer_size;
uint32_type                         logical_block_addr;

/*.Implementation[=-----------------------------------------------
/*
/*   We start by filling in the adapter request block with
/*   information from the operation record.  Note that the buffer
/*   must be in global kernel memory because eventually it is
/*   mapped to physical pages in memory. The buffer may NOT be
/*   in per-process memory because the process executing this code
/*   may not be the same that originally made the request.
/*
/*   The number of read/write requests and number of blocks
/*   read/written are updated for system activity reporting.
/*
/*.End]=-----------------------------------------------------------*/

misc_dequeue_from_head(&uip->arb_free_queue,
```

```
                                (misc_queue_links_ptr_type *)&arb_ptr);

vp_read_system_clock(&arb_ptr->async_request_start_time);
misc_dequeue_from_head(&uip->async_request_queue,
                       (misc_queue_links_ptr_type *)&op_record_ptr);
io_get_buffer_vector_io_info(&op_record_ptr->ri.buffer_vector,
                             &buffer_ptr,
                             &count);
arb_ptr->op_record_ptr = op_record_ptr;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Determine if sector buffering is necessary. It may be
/*   necessary if the disk sector size is not 512.  Even then,
/*   sector buffering is only required if 1) the data is not
/*   aligned on a sector boundary, or 2) the amount of data to
/*   transfer is not a multiple of the sector size.
/*
/*.End]=-----------------------------------------------------------*/

arb_ptr->request_flags = 0;

if (uip->sector_size != DEV_SD_STANDARD_SECTOR_SIZE)
   {
   if (op_record_ptr->ri.device_offset % uip->sector_size != 0)
      {
      arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_SB;
       }
   if (count % uip->sector_size != 0)
      {
      arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_SB;
      }
   }

/*.Implementation_Continued[=--------------------------------------
/*
/*   If sector buffering is necessary, allocated a new buffer.
/*
/*.End]=-----------------------------------------------------------*/

if (arb_ptr->request_flags & DEV_SCSI_REQUEST_FLAGS_SB)
   {

   sector_buffer_size = uip->sector_size;

   if (count > DEV_SD_STANDARD_SECTOR_SIZE)
      {
      sector_buffer_size = sector_buffer_size +   ((count / uip->sector_size)
                           * uip->sector_size) + uip->sector_size;
      }

   buffer_ptr = vm_get_wired_memory(sector_buffer_size, VM_DEFAULT_ALIGNMENT);
   count = sector_buffer_size;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Set up the request to do a read.  If the original request
/*   is a read, indicate in the request flags that the request
/*   is done upon completion.
/*
/*.End]=-----------------------------------------------------------*/

   scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                         &arb_ptr->scsi_cmd_blk;
   scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ;
   scsi_rw_cmd_blk_ptr->lun = uip->unit_spec.unit;
   scsi_rw_cmd_blk_ptr->logical_block_address =
                        op_record_ptr->ri.device_offset /
                                                uip->sector_size;
   scsi_rw_cmd_blk_ptr->transfer_length =  count / uip->sector_size;
   scsi_rw_cmd_blk_ptr->vendor_unique = 0;
   scsi_rw_cmd_blk_ptr->reserved1 = 0;
   scsi_rw_cmd_blk_ptr->link = FALSE;
   scsi_rw_cmd_blk_ptr->flag = FALSE;
```

```
   if (op_record_ptr->ri.op == IO_OPERATION_READ)
      {
      arb_ptr->request_flags |= DEV_SCSI_REQUEST_FLAGS_SB_READ +
                        DEV_SCSI_REQUEST_FLAGS_SB_DONE;
      }

   misc_increment(&uip->read_request_count);
   misc_increment_by_value(&uip->read_block_count,
                           (int32e_type)(count/DF_BYTES_PER_BLOCK));
   }
else
   {

/*.Implementation_Continued[=-----------------------------------
/*
/*   No sector buffering is required.  Process the request
/*   normally.  First, determine if we should set up to do a
/*   write and verify.  We currently only do this for optical
/*   disks.
/*
/*.End]=-------------------------------------------------------*/

   if   (((uip->disk_type == DEV_SD_DISK_TYPE_ERASABLE_OPTICAL) ||
       (uip->disk_type == DEV_SD_DISK_TYPE_WORM)) &&
       (op_record_ptr->ri.op & IO_OPERATION_WRITE))
      {
      scsi_write_verify_cmd_blk_ptr =
            (dev_scsi_write_verify_cmd_blk_ptr_type)&arb_ptr->scsi_cmd_blk;
      scsi_write_verify_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE_VERIFY;
      scsi_write_verify_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
      scsi_write_verify_cmd_blk_ptr->reserved1 = 0;
      scsi_write_verify_cmd_blk_ptr->relative_addr = FALSE;
      logical_block_addr = op_record_ptr->ri.device_offset / uip->sector_size;
      scsi_write_verify_cmd_blk_ptr->logical_block_addr_high =
                  (uint16_type)((logical_block_addr>> 16) & 0xffff);
      scsi_write_verify_cmd_blk_ptr->logical_block_addr_low =
                        (uint16_type)(logical_block_addr & 0xffff);
      scsi_write_verify_cmd_blk_ptr->reserved2 = 0;
      scsi_write_verify_cmd_blk_ptr->transfer_length_high = 0;
      scsi_write_verify_cmd_blk_ptr->transfer_length_low =  count /
                                                            uip->sector_size;
      scsi_write_verify_cmd_blk_ptr->reserved3 = 0;
      scsi_write_verify_cmd_blk_ptr->erase_control = FALSE;
      scsi_write_verify_cmd_blk_ptr->reserved4 = 0;
      scsi_write_verify_cmd_blk_ptr->link = FALSE;
      scsi_write_verify_cmd_blk_ptr->flag = FALSE;
      misc_increment(&uip->read_request_count);
      misc_increment_by_value(&uip->read_block_count,
                              (int32e_type)(count/DF_BYTES_PER_BLOCK));
      }
   else
      {
      scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                           &arb_ptr->scsi_cmd_blk;
      if (op_record_ptr->ri.op == IO_OPERATION_READ)
         {
         scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ;
         misc_increment(&uip->read_request_count);
         misc_increment_by_value(&uip->read_block_count,
                              (int32e_type)(count/DF_BYTES_PER_BLOCK));
         }
      else
         {
         scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE;
         misc_increment(&uip->write_request_count);
         misc_increment_by_value(&uip->write_block_count,
                              (int32e_type)(count/DF_BYTES_PER_BLOCK));
         }
      scsi_rw_cmd_blk_ptr->lun = uip->unit_spec.unit;
      scsi_rw_cmd_blk_ptr->logical_block_address =
                        op_record_ptr->ri.device_offset /
                                             uip->sector_size;
      scsi_rw_cmd_blk_ptr->transfer_length =  count / uip->sector_size;
      scsi_rw_cmd_blk_ptr->vendor_unique = 0;
      scsi_rw_cmd_blk_ptr->reserved1 = 0;
```

```
      scsi_rw_cmd_blk_ptr->link = FALSE;
      scsi_rw_cmd_blk_ptr->flag = FALSE;
      ]
   ]

arb_ptr->request_flags |= DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                        buffer_ptr,
                        count);
arb_ptr->sync_io = FALSE;
arb_ptr->op_record_ptr = op_record_ptr;
arb_ptr->complete_io_routine = dev_sd_complete_io;

/*.Implementation_Continued[=----------------------------------------
/*
/*   Issue the command to the supporting adapter. Any status
/*   information generated by the request will be processed by
/*   the driver complete I/O routine.
/*
/*.End]=-------------------------------------------------------------*/

(void)dev_scsi_adapter_issue_async_command(uip->adapter_device_number.major,
                                    arb_ptr);
}

/*.function */

              /*<----------------------*/
WIRED
status_type             dev_sd_start_sync_request  (uip, arb_ptr)
              /*>----------------------*/

dev_sd_unit_info_ptr_type           uip;          /* READ ONLY */
dev_adapter_request_block_ptr_type  arb_ptr;      /* READ/WRITE */

/*.Summary[=---------------------------------------------------------
/*
/*   Issue an adapter request block (command) to the supporting
/*   adapter and return the status of the operation.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the device that is the target of the synchronous request.
/*
/* arb_ptr -- A pointer to the adapter request block which
/*   specifies the request.
/*
/*.Functional_Description
/*
/*   This function dispatches through the supporting adapter's
/*   routines vector to execute the requested command on the
/*   target device. If the command terminates with a Check
/*   Condition status, the Request Sense information is
/*   interpreted and the appropriate status is returned to the
/*   caller.
/*
/*.Assumptions
/*
/*   This function assumes that the unit request lock is held
/*   by the caller.
/*
/*.Return_Value
/*
/* OK -- The synchronous operation completed successfully.
/*
/*   Return values from dev_sd_evaluate_sense_info.
/*
/*.Exceptions
/*
/*   None.
/*
{

status_type             status;
```

```
/*.Implementation[=-----------------------------------------------
/*
/*   Issue the request to the supporting SCSI adapter. If the
/*   request completes with a Check Condition status, call the
/*   evaluate sense info routine to interpret the sense information.
/*
/*   Signal delivery is disabled for the call to the supporting
/*   adapter manager so the disk request cannot be
/*   terminated. Disk requests should always be allowed to
/*   complete or timeout. Since the adapter manager does not
/*   make any distinction between disks and other device types,
/*   signal delivery is disabled here.
/*
/*.End]=-------------------------------------------------------------*/

arb_ptr->sync_io = TRUE;
pm_disable_signal_delivery();
status = dev_scsi_adapter_issue_command(uip->adapter_device_number.major,
                                        arb_ptr);
pm_enable_signal_delivery();
if (status == DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION)
    {
    status = dev_sd_evaluate_sense_info(uip, arb_ptr);
    }

return(status);
}

/*.function */

                /*<----------------------*/
WIRED
status_type      dev_sd_evaluate_sense_info  (uip, arb_ptr)
                /*>----------------------*/

dev_sd_unit_info_ptr_type            uip;     /* READ ONLY */
dev_adapter_request_block_ptr_type   arb_ptr; /* READ ONLY */

/*.Summary[=-----------------------------------------------------------
/*
/*   Evaluate data returned from a Request Sense command.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the device that returned sense information.
/*
/* arb_ptr -- A pointer to the adapter request block which
/*   specifies a request that completed with a check condition status.
/*
/*.Functional_Description
/*
/*   This function is called to interpret sense information returned
/*   from a request to a device. Sense information is returned by
/*   the supporting SCSI adapter whenever a request completes with
/*   a Check Condition status. A Check Condition status indicates
/*   that an error, exception, or abnormal condition has occurred
/*   during command execution. Basically, the sense data error code
/*   is mapped to a DG/UX status code and the error logger is called
/*   if appropriate.
/*
/*.Return_Value
/*
/* IO_ENXIO_UNIT_NOT_READY -- The specified unit is not
/*   responding to commands.
/*
/* IO_EIO_HARD_IO_ERROR -- A hard I/O occurred in trying to
/*   read or write to the disk. A retries were performed but to no
/*   avail. The number of retries performed depends on the mode
/*   settings of the device.
/*
/* IO_EIO_PHYSICAL_UNIT_FAILURE -- A nonrecoverable hardware
/*   error occurred while the command was being executed.
/*
/* IO_ENXIO_ILLEGAL_DEVICE_ADDRESS -- The sum of the disk
```

```
/*   address and the number of blocks to be transferred was larger
/*   than the largest disk address on the specified unit.
/*
/* IO_EIO_SD_UNIT_ATTENTION_ON -- The command to the unit
/*   failed because of a Unit Attention condition.
/*
/* IO_ENXIO_NO_WRITE_RING -- A write operation was attempted
/*   on a write only device.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*

{

status_type             status;
dev_sd_request_sense_buffer_ptr_type    disk_sense_buffer_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Get the sense key from the buffer containing the sense
/*   data and take the appropriate action.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
switch (arb_ptr->sense_buffer.sense_key)
    {
    case DEV_SCSI_SENSE_KEY_NOT_READY:
/*.Implementation_Continued[=--------------------------------------
/*
/*   The addressed logical unit cannot be accessed. Either it
/*   has been turned off or it just stopped working.
/*
/*.End]=-----------------------------------------------------------*/
            status = IO_ENXIO_UNIT_NOT_READY;
        break;

    case DEV_SCSI_SENSE_KEY_RECOVERED_ERROR:
/*.Implementation_Continued[=--------------------------------------
/*
/*   The command completed successfully with some recovery
/*   action performed by the disk controller. Recovery
/*   action includes retries and application of ecc
/*   correction. The error is logged with the error logger
/*   and a good status is returned.
/*
/*.End]=-----------------------------------------------------------*/
            if (!uip->inhibit_error_logging)
                {
            dev_sd_log_error(arb_ptr, OK);
                }
            break;

    case DEV_SCSI_SENSE_KEY_MEDIUM_ERROR:
/*.Implementation_Continued[=--------------------------------------
/*
/*   The command terminated with a nonrecoverable error
/*   condition caused by a flaw in the medium or by an
/*   error in the recorded data. The error IO_EIO_HARD_ERROR
/*   is returned so that the file system will remap the
/*   offending disk block. The number of bytes transferred
/*   by the drive does not include any data from the bad
/*   block. The error is logged with the error logger.
/*
/*.End]=-----------------------------------------------------------*/
            status = IO_EIO_HARD_IO_ERROR;
            if (!uip->inhibit_error_logging)
                {
                dev_sd_log_error(arb_ptr, status);
```

```
            }
        break;

    case DEV_SCSI_SENSE_KEY_HARDWARE_ERROR:
/*.Implementation_Continued[=----------------------------------------
/*
/*    A nonrecoverable hardware error (e.g. controller failure,
/*    device failure, parity error, etc.) was detected while the
/*    target device was performing the command. Log the error
/*    and return IO_EIO_PHYSICAL_UNIT_FAILURE.
/*
/*.End]=-------------------------------------------------------------*/
        status = IO_EIO_PHYSICAL_UNIT_FAILURE;
        if (!uip->inhibit_error_logging)
            {
            dev_sd_log_error(arb_ptr, status);
            }
        break;

    case DEV_SCSI_SENSE_KEY_ILLEGAL_REQUEST:
    case DEV_SCSI_SENSE_KEY_BLANK_CHECK:
/*.Implementation_Continued[=----------------------------------------
/*
/*    An illegal parameter was detected in the command block.
/*    The original request specified an offset or size that
/*    the device can't handle, return the error.
/*
/*.End]=-------------------------------------------------------------*/
        status = IO_ENXIO_ILLEGAL_DEVICE_ADDRESS;
        break;

    case DEV_SCSI_SENSE_KEY_UNIT_ATTENTION:
/*.Implementation_Continued[=----------------------------------------
/*
/*    The unit attenion flag of the device is on. The unit
/*    attention flag is set for any of the following reasons:
/*
/* The unit has been reset by a SCIS bus reset.
/* A mode select has been issued which may affect
/*    the parameters of another initiator.
/* A removable medium has been changed.
/*
/*    Unit attention remains on until the drive returns a Check
/*    Condition status for a command sent from the initiator.
/*    The command resulting in the Check Condition status is not
/*    performed. Only the Inquiry and the Request Sense commands
/*    can be performed while unit attention is on. Inquiry
/*    executes normally and preserves the unit attenion
/*    condition. Request Sense reports unit attention and
/*    clears it.
/*
/*    Examine the error code from the sense buffer. If a
/*    medium change occurred, return the medium change error.
/*    Otherwise report unit attention.
/*
/*.End]=-------------------------------------------------------------*/

        disk_sense_buffer_ptr = (dev_sd_request_sense_buffer_ptr_type)
                                        &arb_ptr->sense_buffer;
            if (disk_sense_buffer_ptr->error_code ==
                        DEV_SCSI_SENSE_ERR_CODE_MEDIUM_CHANGE)
                {
                status = IO_EIO_MEDIUM_CHANGE_OCCURRED;
                }
            else
                {
                status = DEV_EIO_SD_UNIT_ATTENTION_ON;
                }
            break;

    case DEV_SCSI_SENSE_KEY_DATA_PROTECT:
/*.Implementation_Continued[=----------------------------------------
/*
/*    A write operation has been attempted on a device that is
/*    write protected.
```

```
/*
/*.End]=-------------------------------------------------------------*/
            status = IO_ENXIO_NO_WRITE_RING;
            break;

    default:
/*.Implementation_Continued[=-----------------------------------
/*
/*   No other command interpretation or execution errors are
/*   anticipated. If any occur, flag a physical unit failure.
/*
/*.End]=-------------------------------------------------------------*/
            status = IO_EIO_PHYSICAL_UNIT_FAILURE;
            break;
    }

return(status);

}

/*.function */

               /*<-------------*/
WIRED
void            dev_sd_log_error  (arb_ptr, op_status)
               /*>-------------*/

dev_adapter_request_block_ptr_type   arb_ptr; /* READ ONLY */
status_type                       op_status;     /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------
/*
/*   This function reports a SCSI disk I/O error to the error logger
/*   pseudo-device.
/*
/*.Parameters
/*
/* arb_ptr -- A pointer to the adapter request block
/*   which contains the command that caused the error.
/*
/* op_status -- The status from the operation that will
/*   be returned to the originator of the operation.
/*
/*.Functional_Description
/*
/*   The necessary values are assembled from the request block
/*   and passed to the error logger.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Exceptions
/*
/*   None.
/*

{
dev_scsi_read_write_cmd_blk_ptr_type scsi_rw_ptr;

scsi_rw_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                        &arb_ptr->scsi_cmd_blk;
/*.Implementation[=---------------------------------------------------
/*
/*   If the op_status == OK, then the error is logged as a soft error.
/*   Otherwise, the hard error message is logged with the sense key
/*   that describes the error.
/*
/*   The error logger returns a boolean indicating whether an error
/*   queue element was available for the message to be posted. This
/*   boolean is ignored since it is considered very unlikely that
/*   the queue is full and hence not worth the added complexity of
/*   insuring that the error is posted.
/*
/*.End]=-------------------------------------------------------------*/
```

```
if (op_status == OK)
    {
    (void)io_err_log_error((uint32e_type)LOG_NOTICE,
                           dev_sd_soft_error_message,
                           (bit32e_type)arb_ptr->unit_spec.scsi_id,
                           (bit32e_type)arb_ptr->unit_spec.unit,
                           (bit32e_type)scsi_rw_ptr->logical_block_address);

    }
else
    {
   (void)io_err_log_error((uint32e_type)LOG_WARNING,
                           dev_sd_hard_error_message,
                           (bit32e_type)arb_ptr->unit_spec.scsi_id,
                           (bit32e_type)arb_ptr->unit_spec.unit,
                           (bit32e_type)scsi_rw_ptr->logical_block_address,
                           (bit32e_type)op_status);
    }

return;

}

/*.function */

               /*<-------------------------*/
WIRED
status_type     dev_sd_init_rigid_disk_unit  (uip)
               /*>-------------------------*/

dev_sd_unit_info_ptr_type                   uip;  /* READ/WRITE */

/*.Summary[=-----------------------------------------------------------------
/*
/*   Initialize a rigid disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the initialization
/*   operation.
/*
/*.Functional_Description
/*
/*   This function is called to initialize a rigid disk unit.
/*   Initialization consists of setting the unit to operate
/*   in the desired mode.
/*
/*.Return_Value
/*
/* OK -- The unit initialization was successful.
/*
/* IO_ENXIO_DEVICE_NOT_SUPPORTED -- The modes desired
/*   for the device could not be selected.
/*

{
status_type                 status;
dev_scsi_da_mode_buffer_type                          mode_buffer;
dev_scsi_da_mode_buffer_error_param_page_ptr_type  error_page_ptr;


/*.Implementation[=-------------------------------------------------
/*
/*   Read the current error recovery parameters from the disk to
/*   determine if they are set the way are set the way we want
/*   them. The following settings are desired:
/*
/* Disabled Automatic Write Reallocation Enabled (AWRE)
/*   and Automatic Read Reallocation Enabled (ARRE) so
/*   that automatic reassigning (remapping) of bad blocks
/*   is disabled. Bad block remapping is handled through
/*   the DG/UX file system. See the URB memo "Disk Format
/*   and Error Recovery", 19 August 1988 for a discussion
/*   of the motivation for managing bad blocks through
```

```
/*   operating system software.
/*
/*   If the unit has never been formatted, our attempt to read
/*   the error recovery page will fail. If the read successful and
/*   the mode settings are not correct, a mode select is done.
/*   Doing the selection the causes the unit's current error recovery
/*   parameters to be updated.
/*
/*   The defaults are taken for all other mode selection parameters.
/*   See disk model specific documentation for a complete description
/*   of all selectable parameters and their default values.
/*
/*   Note - selection of error recovery parameters does not
/*   generate a Unit Attention condition. Unit Attention is
/*   generated when the Direct-Access Format Parameters or
/*   the Disk Driver Geometry Parameters are modified.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_sd_sense_unit_mode(uip,
                        DEV_SCSI_MODE_SENSE_ERROR_RECOVERY_PAGE,
                        DEV_SCSI_MODE_SENSE_CURRENT_VALUES,
                        &mode_buffer);
if (status == OK)
    {
    error_page_ptr = (dev_scsi_da_mode_buffer_error_param_page_ptr_type)
                                    &mode_buffer.page;
    if   ((error_page_ptr->arre != FALSE) ||
          (error_page_ptr->awre != FALSE))
        {
/*.Implementation_Continued[=----------------------------------------
/*
/*   Clear arre and awre. Note that all reserved fields and mode
/*   sense fields that do no apply to mode select must be
/*   zeroed out. The devices don't always clear the reserved
/*   fields on the mode sense but are sensitive to nonzero values
/*   on the mode select.
/*
/*.End]=-------------------------------------------------------------*/

        error_page_ptr->arre = FALSE;
        error_page_ptr->awre = FALSE;
        error_page_ptr->param_savable = FALSE;
        error_page_ptr->reserved1 = 0;
        mode_buffer.header.reserved1 = 0;
        mode_buffer.header.reserved2 = 0;
        mode_buffer.blk_descriptor.reserved_must_be_zero = 0;
        status = dev_sd_select_unit_mode(uip, &mode_buffer);
        if (status != OK)
            {
            status = IO_ENXIO_DEVICE_NOT_SUPPORTED;
            }
        }
    }
return(status);
}

/*.function */

                /*<-------------------------*/
UNWIRED
status_type      dev_sd_init_floppy_disk_unit  (uip)
                /*>-------------------------*/

dev_sd_unit_info_ptr_type                   uip; /* READ/WRITE */

/*.Summary[=---------------------------------------------------------------
/*
/*   Initialize a floppy disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the initialization
/*   operation.
```

```
/*
/*.Functional_Description
/*
/*   This function is called to initialize a floppy disk unit.
/*   Initialization consists of setting the unit to operate
/*   in a mode that is compatible with the currently inserted
/*   medium. Format information (mode sense) is not readable from
/*   the floppy disk unit itself. As a result, the driver must
/*   determine the correct mode settings through trial and error.
/*   Modes for a standard medium type are selected and an attempt
/*   is made to access the medium. If the access fails, another
/*   mode is tried. This procedure is repeated until one of the
/*   mode selects works or all supported modes have been tried.
/*
/*   Currently this driver supports the following floppy medium
/*   types:
/*
/* 5.25 inch 1.2 Mbyte (formatted) floppy
/* 5.25 inch .720 Mbyte (formatted) floppy
/* 5.25 inch .360 Mbyte (formatted) floppy
/* 3.50 inch 1.44 Mbyte (formatted) floppy
/* 3.50 inch .720 Mbyte (formatted) floppy
/*
/*   Note that the .360 Mbyte floppy is meant to be used
/*   with a 48 TPI drive. The 5.25 inch floppy unit that
/*   this driver supports is a 96 TPI drive, but is capable
/*   of reading/writing 48 TPI data. However, the heads
/*   on a 48 TPI unit are bigger than a 96 TPI unit's
/*   heads. The 96 TPI unit uses a narrower track. As a result,
/*   a 48 TPI unit may have trouble reading data written by
/*   a 96 TPI unit.
/*
/*.Return_Value
/*
/* OK -- The unit initialization was successful.
/*
/* DEV_EIO_UNIT_NOT_FORMATTED -- Mode information could
/*   not be read from the device. This usually means that the
/*   device has never been formatted.
/*
/* IO_ENXIO_DEVICE_NOT_SUPPORTED -- The modes desired
/*   for the device could not be selected.
/*

{
status_type                 status;
boolean_type                mixed_config;
dev_scsi_da_mode_buffer_type                          mode_buffer;
dev_scsi_da_mode_buffer_flexible_disk_geometry_page_ptr_type geom_page_ptr;

/*.Implementation[=-----------------------------------------------
/*
/*   Set the floppy controller to the correct mode. Since the floppy
/*   may have just been inserted and its format may differ from the
/*   current controller settings, we must set the controller to
/*   the mode that matches the floppy format.
/*
/*   The error recovery page is left at the default since
/*   Disabled Automatic Write Reallocation Enabled (AWRE) and
/*   Automatic Read Reallocation Enabled (ARRE) do not apply
/*   to floppy.  See dev_sd_init_rigid_disk_unit for a description
/*   of the desired error recovery options.
/*
/*   First read the disk geometry page to get a template for the
/*   mode select buffer.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_sd_sense_unit_mode(uip,
                            DEV_SCSI_MODE_SENSE_FLEXIBLE_DISK_GEOMETRY_PAGE,
                            DEV_SCSI_MODE_SENSE_CURRENT_VALUES,
                            &mode_buffer);
if (status != OK)
    {
    goto done;
```

```
    }

/*.Implementation_Continued[=-----------------------------------------
/*
/*   Fill in the mode buffer header, block descriptor,  and flexible
/*   disk geometry parameter page with any values which are common
/*   to all medium types.
/*
/*.End]=-------------------------------------------------------------*/

mode_buffer.header.reserved1 = 0;
mode_buffer.header.reserved2 = 0;

mode_buffer.blk_descriptor.num_logical_blks_msb = 0;
mode_buffer.blk_descriptor.num_logical_blks_mid = 0;
mode_buffer.blk_descriptor.num_logical_blks_lsb = 0;
mode_buffer.blk_descriptor.num_logical_blks_lsb = 0;

mode_buffer.blk_descriptor.logical_blk_len_msb =   ((uip->sector_size
                           & DEV_SCSI_CMD_COUNT_HIGH_8_BIT_MASK) >>
                           DEV_SCSI_CMD_COUNT_HIGH_8_BIT_SHIFT);
mode_buffer.blk_descriptor.logical_blk_len_mid =   ((uip->sector_size
                           & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
                           DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
mode_buffer.blk_descriptor.logical_blk_len_msb = uip->sector_size
                           & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr = (dev_scsi_da_mode_buffer_flexible_disk_geometry_page_ptr_type)
                        &mode_buffer.page;
geom_page_ptr->reserved1 = 0;

geom_page_ptr->data_bytes_per_physical_sector_high =   ((uip->sector_size &
DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
                DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
geom_page_ptr->data_bytes_per_physical_sector_low = uip->sector_size
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->write_precompensation_value =
               DEV_SD_FLOPPY_DEFLAULT_WRITE_PRECOMPENSATION_VALUE;

geom_page_ptr->reserved4 = 0;
geom_page_ptr->reserved5 = 0;
geom_page_ptr->reserved6 = 0;
geom_page_ptr->reserved7 = 0;

/*.Implementation_Continued[=-----------------------------------------
/*
/*   Determine the type of configuration present on the
/*   adapter card by examining the inquiry buffer. The configuration
/*   will be one of the following:
/*
/* All 5.25 inch units (LUNs 0-3) where present.
/*
/* All 3.5 inch units (LUNs 0-1) where present.
/*
/* A mixture wiht 3.5 inch units at LUNs 0-1 and 5.25
/*   units at LUNs 2-3.
/*
/*   See dev_sd_def.h for a complete description of how the
/*   configuration is determined from the inquiry data.
/*
/*   If the adapter card is jumpered to an unknown or unsupported
/*   configuration return the error.
/*
/*.End]=-------------------------------------------------------------*/

if (misc_string_compare((byte_address_type)&uip->inquiry_buffer.
             vendor_unique[DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_OFFSET],
             DEV_SD_FLOPPY_INQUIRY_CONFIG_5_25,
          (uint32_type)DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_SIZE) == 0)
    {
    mixed_config = FALSE;
    }

else if (misc_string_compare((byte_address_type)&uip->inquiry_buffer.
```

```
            vendor_unique[DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_OFFSET],
            DEV_SD_FLOPPY_INQUIRY_CONFIG_MIXED,
        (uint32_type)DEV_SD_FLOPPY_INQUIRY_CONFIG_INFO_SIZE) == 0)
    {
    mixed_config = TRUE;
    }
else
    {
    status = IO_ENXIO_DEVICE_NOT_SUPPORTED;
    goto done;
    }

/*.Implementation_Continued[=------------------------------------
/*
/*   Now fill in the values for the flexible disk geometry page,
/*   first assuming a 5.25 inch 1.2 Mbyte floppy.  We only modify
/*   the fields that are changable and vary between medium types.
/*
/*   Note that it is not possible to have a mixed configuration
/*   with a 5.25 inch 1.2 Mbyte floppy at unit number 0 or 1.
/*   See dev_sd_def.h for a detailed description of how the
/*   1.2 Mbyte 5.25 floppy is selected on a mixed configuration.
/*
/*.End]-----------------------------------------------------------*/

if   ((!mixed_config) || (mixed_config && uip->unit_spec.unit > 1))
    {
    if (mixed_config)
            {
        mode_buffer.header.medium_type =
                DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR_MIXED;
            }
    else
            {
        mode_buffer.header.medium_type =
                DEV_SD_FLOPPY_MEDIUM_TYPE_5_25INCH_96TPI_13262BPR;
            }
    geom_page_ptr->transfer_rate_high =
((DEV_SD_FLOPPY_5_25INCH_1200KB_TRANSFER_RATE
            & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
            DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
    geom_page_ptr->transfer_rate_low =
            DEV_SD_FLOPPY_5_25INCH_1200KB_TRANSFER_RATE
        & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

    geom_page_ptr->sectors_per_track =
            DEV_SD_FLOPPY_5_25INCH_1200KB_SECTORS_PER_TRACK;

    geom_page_ptr->number_of_cylinders_high =
((DEV_SD_FLOPPY_5_25INCH_1200KB_NUMBER_OF_CYLINDERS
            & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
            DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);

    geom_page_ptr->number_of_cylinders_low =
            DEV_SD_FLOPPY_5_25INCH_1200KB_NUMBER_OF_CYLINDERS
        & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

    geom_page_ptr->step_pulses_per_cylinder =
            DEV_SD_FLOPPY_5_25INCH_1200KB_STEP_PULSES_PER_CYLINDER;

/*.Implementation_Continued[=------------------------------------
/*
/*   Do the mode select. If the mode mode select succeeds,
/*   attempt to access the disk to see if the mode select
/*   parameters match the currently inserted medium. If
/*   the mode select failed, try selecting a different
/*   medium type.
/*
/*.End]-----------------------------------------------------------*/

    status = dev_sd_select_unit_mode(uip, &mode_buffer);
    if (status == OK)
        {
        status = dev_sd_test_mode_select(uip);
        if (status == OK)
```

```
            {
            goto done;
            }
        }
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   The medium is not a 5.25 inch 1.2 Mb floppy, do a mode select
/*   for a 5.25 inch .360 Mbyte (formatted) floppy.
/*
/*.End]=-----------------------------------------------------------*/

mode_buffer.header.medium_type = DEV_SD_FLOPPY_MEDIUM_TYPE_96_135TPI_7958BPR;

geom_page_ptr->transfer_rate_high =
((DEV_SD_FLOPPY_5_25INCH_360KB_TRANSFER_RATE
              & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
              DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);

geom_page_ptr->transfer_rate_low =
              DEV_SD_FLOPPY_5_25INCH_360KB_TRANSFER_RATE
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->sectors_per_track =
              DEV_SD_FLOPPY_5_25INCH_360KB_SECTORS_PER_TRACK;

geom_page_ptr->number_of_cylinders_high =
((DEV_SD_FLOPPY_5_25INCH_360KB_NUMBER_OF_CYLINDERS
              & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
              DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
geom_page_ptr->number_of_cylinders_low =
              DEV_SD_FLOPPY_5_25INCH_360KB_NUMBER_OF_CYLINDERS
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->step_pulses_per_cylinder =
              DEV_SD_FLOPPY_5_25INCH_360KB_STEP_PULSES_PER_CYLINDER;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Do the mode select. If the mode mode select succeeds,
/*   attempt to access the disk to see if the mode select
/*   parameters match the currently inserted medium. If
/*   the mode select failed, try selecting a different
/*   medium type.
/*
/*
/*.End]=-----------------------------------------------------------*/

status = dev_sd_select_unit_mode(uip, &mode_buffer);
if (status == OK)
    {
    status = dev_sd_test_mode_select(uip);
    if (status == OK)
        {
        goto done;
        }
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   The access failed. Do a mode select for a 3.5 inch
/*   1.44 Mbyte (formatted) floppy.
/*
/*.End]=-----------------------------------------------------------*/

mode_buffer.header.medium_type =
                        DEV_SD_FLOPPY_MEDIUM_TYPE_3_50INCH_135TPI_15916BPR;

geom_page_ptr->transfer_rate_high =
((DEV_SD_FLOPPY_3_50INCH_1440KB_TRANSFER_RATE
              & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
              DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);

geom_page_ptr->transfer_rate_low =
```

```
            DEV_SD_FLOPPY_3_50INCH_1440KB_TRANSFER_RATE
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->sectors_per_track =
            DEV_SD_FLOPPY_3_50INCH_1440KB_SECTORS_PER_TRACK;

geom_page_ptr->number_of_cylinders_high =
((DEV_SD_FLOPPY_3_50INCH_1440KB_NUMBER_OF_CYLINDERS
            & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
            DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
geom_page_ptr->number_of_cylinders_low =
            DEV_SD_FLOPPY_3_50INCH_1440KB_NUMBER_OF_CYLINDERS
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->step_pulses_per_cylinder =
            DEV_SD_FLOPPY_3_50INCH_1440KB_STEP_PULSES_PER_CYLINDER;

/*.Implementation_Continued[=---------------------------------------
/*
/*   Do the mode select. If the mode mode select succeeds,
/*   attempt to access the disk to see if the mode select
/*   parameters match the currently inserted medium. If
/*   the mode select failed, try selecting a different
/*   medium type.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_sd_select_unit_mode(uip, &mode_buffer);
if (status == OK)
    {
    status = dev_sd_test_mode_select(uip);
    if (status == OK)
        {
        goto done;
        }
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   The access failed. Finally, do a mode select for a .720 Mbyte
/*   floppy. Note that the mode selection parameters are the
/*   same for 5.25 and 3.50 inch .720 Mbyte floppies.
/*
/*.End]=-------------------------------------------------------------*/

mode_buffer.header.medium_type = DEV_SD_FLOPPY_MEDIUM_TYPE_96_135TPI_7958BPR;

geom_page_ptr->transfer_rate_high =   ((DEV_SD_FLOPPY_720KB_TRANSFER_RATE
            & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
            DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
geom_page_ptr->transfer_rate_low =
            DEV_SD_FLOPPY_720KB_TRANSFER_RATE
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->sectors_per_track = DEV_SD_FLOPPY_720KB_SECTORS_PER_TRACK;

geom_page_ptr->number_of_cylinders_high =
((DEV_SD_FLOPPY_720KB_NUMBER_OF_CYLINDERS
            & DEV_SCSI_CMD_COUNT_MID_8_BIT_MASK) >>
            DEV_SCSI_CMD_COUNT_MID_8_BIT_SHIFT);
geom_page_ptr->number_of_cylinders_low =
            DEV_SD_FLOPPY_720KB_NUMBER_OF_CYLINDERS
          & DEV_SCSI_CMD_COUNT_LOW_8_BIT_MASK;

geom_page_ptr->step_pulses_per_cylinder =
            DEV_SD_FLOPPY_720KB_STEP_PULSES_PER_CYLINDER;

/*.Implementation_Continued[=---------------------------------------
/*
/*   Do the mode select. If the mode mode select succeeds,
/*   attempt to access the disk to see if the mode select
/*   parameters match the currently inserted medium. If
/*   the mode select failed, try selecting a different
/*   medium type.
/*
```

```
/*.End]-------------------------------------------------------------*/

status = dev_sd_select_unit_mode(uip, &mode_buffer);
if (status == OK)
    {
    status = dev_sd_test_mode_select(uip);
    if (status == OK)
        {
        goto done;
        }
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   ********* NOTE - If none of the mode select operations were
/*   successful, the floppy disk is probably not formatted. For
/*   now, we return a good status if the disk has not been
/*   hardware formatted so that the ioctl command DSKIOC_GENERIC_SCSI
/*   can be used to format the disk. Eventually DSKIOC_GENERIC_SCSI
/*   is going to be moved to the SCSI adapter driver, and an error
/*   will be return here if the floppy cannot be accessed.
/*
/*.End]-------------------------------------------------------------*/

status = OK;

done:
return(status);
}

/*.function */

              /*<----------------------*/
UNWIRED
status_type     dev_sd_init_worm_disk_unit  (uip)
              /*>----------------------*/

dev_sd_unit_info_ptr_type                   uip; /* READ/WRITE */

/*.Summary[=--------------------------------------------------------
/*
/*   Initialize a Write Once Read Many optical disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the initialization
/*   operation.
/*
/*.Functional_Description
/*
/*   This function is called to initialize a WORM optical disk
/*   unit. Initialization consists of setting the unit to operate
/*   in the desired mode.
/*
/*.Return_Value
/*
/* OK -- The unit initialization was successful.
/*
/* IO_ENXIO_DEVICE_NOT_SUPPORTED -- Either the mode sense
/*   or mode select failed indicating that the target did accept
/*   or understand the mode requested.
/*
/*   Return values from dev_sd_sense_unit_mode.
/*

{
status_type                 status;
dev_scsi_da_mode_buffer_type                          mode_buffer;
dev_scsi_da_mode_buffer_error_param_page_ptr_type     error_page_ptr;
dev_scsi_da_mode_buffer_error_param_page_ptr_type     select_error_page_ptr;
dev_scsi_worm_mode_header_ptr_type                    mode_header_ptr;
dev_sd_worm_optimem_mode_buffer_ptr_type              optimem_mode_buffer_ptr;

/*.Implementation[=-------------------------------------------------
```

```
/*
/*    Read the current error recovery parameters from the unit to
/*    get a template for the mode select operation. The
/*    enable_blank_check flag is set in the mode select header
/*    so that an error will be returned if a write is attempted to
/*    a block that has already been written to. Also, Automatic Write
/*    Reallocation Enabled (AWRE) and Automatic Read Reallocation
/*    Enabled (ARRE) are enabled so that automatic reassigning
/*    (remapping) of bad blocks is done. Since the WORM is a write
/*    once device, bad block remapping can not be managed by the
/*    usual file system facilities.
/*
/*    The defaults are taken for all other mode selection parameters.
/*    See model specific documentation for a complete description
/*    of all selectable parameters and their default values.
/*
/*    Note that one of the WORM devices that we support, the Optimem
/*    1000, has a vendor unique mode select buffer format and must
/*    be handled as a special case. The ANSI SCSI standard specifies
/*    that mode sense/select information is specified by a mode
/*    select header followed by a block descriptor followed by a
/*    parameter page. The Optimem 1000 simply uses a hybrid version
/*    of the mode sense/select header. The Optimem 1000 format
/*    is recognized if the page returned by the device does not
/*    identify itself as an Error Recovery Page.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_sd_sense_unit_mode(uip,
                        DEV_SCSI_MODE_SENSE_ERROR_RECOVERY_PAGE,
                        DEV_SCSI_MODE_SENSE_CURRENT_VALUES,
                        &mode_buffer);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*    Get a pointer to Error Recovery Paramter Page. If the device
/*    did not return a block descriptor with the mode select data,
/*    the Error Recovery Parameter Page will be at the offset
/*    usually occupied by the block descriptor.
/*
/*.End]=-------------------------------------------------------------*/

if (mode_buffer.header.blk_desc_len != 0)
    {
    /*.Implementation_Continued[=------------------------------------
    /*
    /*    A block descriptor was returned, get the page from the usual
    /*    place in the mode buffer.
    /*
    /*.End]=---------------------------------------------------------*/

    error_page_ptr = (dev_scsi_da_mode_buffer_error_param_page_ptr_type)
                                &mode_buffer.page;
    }
else
    {
/*.Implementation_Continued[=----------------------------------------
/*
/*   A block descriptor was not returned, get the page from the
/*   position in the mode buffer usually occupied by the block
/*   descriptor.
/*
/*.End]=-------------------------------------------------------------*/

    error_page_ptr = (dev_scsi_da_mode_buffer_error_param_page_ptr_type)
                                &mode_buffer.blk_descriptor;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*    If the page code field of the Error Recovery Page does not
```

```
/*   specify DEV_SCSI_MODE SENSE_ERROR_RECOVERY_PAGE, assume that
/*   the device is an Optimem 1000. The Optimem 1000 uses a vendor
/*   unique mode buffer format.
/*
/*.End]=-------------------------------------------------------------*/

if (error_page_ptr->page_code == DEV_SCSI_MODE_SENSE_ERROR_RECOVERY_PAGE)
    {
/*.Implementation_Continued[=----------------------------------------
/*
/*  The page is in the ANSI SCSI format, turn on the desired bits.
/*  At least one WORM vendor (toshiba) returns a block descriptor
/*  in the mode sense but does not accept one in the mode select.
/*  As a result, we never use one in the mode select (it's not
/*  needed). Because there is no block descriptor, the mode select
/*  buffer must be built where the block descriptor normally is
/*  in the mode buffer.
/*
/*.End]=-------------------------------------------------------------*/

    select_error_page_ptr = (dev_scsi_da_mode_buffer_error_param_page_ptr_type)
                            &mode_buffer.blk_descriptor;
    *select_error_page_ptr = *error_page_ptr;
    select_error_page_ptr->arre = TRUE;
    select_error_page_ptr->awre = TRUE;
    select_error_page_ptr->param_savable = FALSE;
    select_error_page_ptr->reserved1 = 0;
    mode_buffer.header.reserved1 = 0;
    mode_buffer.header.reserved2 = 0;
    }
else
    {
/*.Implementation_Continued[=----------------------------------------
/*
/* The device is an Optimem 1000 WORM, use its vendor unique mode
/* buffer to turn on sector relocation.
/*
/*.End]=-------------------------------------------------------------*/

    optimem_mode_buffer_ptr = (dev_sd_worm_optimem_mode_buffer_ptr_type)
                        &mode_buffer;
    optimem_mode_buffer_ptr->enable_sector_relocation = TRUE;
    optimem_mode_buffer_ptr->blk_desc_len = 0;
    optimem_mode_buffer_ptr->enable_physical_read = FALSE;
    optimem_mode_buffer_ptr->delay_error_reporting = FALSE;
    optimem_mode_buffer_ptr->disable_seek_immediate = FALSE;
    optimem_mode_buffer_ptr->reserved4 = 0;
    optimem_mode_buffer_ptr->disable_retry_times = FALSE;
    optimem_mode_buffer_ptr->error_detection_level = FALSE;
    optimem_mode_buffer_ptr->reserved5 = 0;
    optimem_mode_buffer_ptr->disable_error_detection_and_correction = FALSE;
    optimem_mode_buffer_ptr->parity_enable = TRUE;
    optimem_mode_buffer_ptr->enable_diagnostics = FALSE;
    optimem_mode_buffer_ptr->reserved6 = 0;
    optimem_mode_buffer_ptr->reserved7 = 0;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Set the fields that are common to the Optimem device and
/*   ANSI SCSI compliant and issue the mode select command to
/*   the device.
/*
/*.End]=-------------------------------------------------------------*/

mode_header_ptr = (dev_scsi_worm_mode_header_ptr_type)
                        &mode_buffer.header;
mode_header_ptr->blk_desc_len = 0;
mode_header_ptr->enable_blank_check = TRUE;
mode_header_ptr->reserved1 = 0;
mode_header_ptr->reserved2 = 0;
mode_header_ptr->reserved3 = 0;
(void)dev_sd_select_unit_mode(uip, &mode_buffer);

/*.Implementation_Continued[=----------------------------------------
```

```
/*
/*   The mode select may fail for a number of reasons:
/*
/* The currently inserted medium may not have been
/*   formatted to support the desired options.
/*
/* The Optimem 1000 rejects all mode select requests
/*   after the first successful one. A SCSI bus reset must
/*   precede each subsequent mode select.
/*
/*   This driver allows access to the device if the mode select
/*   fails as long as the mode sense buffer indicates that
/*   the Blank Check option is enabled. Blank Check prevents the
/*   user from inserting a previously written to medium and
/*   destroying the data by attempting to write to it.
/*
/*   A mode sense operation is done to insure that either
/*   the mode select of the Blank Check worked or the
/*   device already has Blank Check on. If Blank Check is not
/*   enabled the error IO_ENXIO_DEVICE_NOT_SUPPORTED is returned.
/*
/*.End]-------------------------------------------------------------*/

status = dev_sd_sense_unit_mode(uip,
                        DEV_SCSI_MODE_SENSE_ERROR_RECOVERY_PAGE,
                        DEV_SCSI_MODE_SENSE_CURRENT_VALUES,
                        &mode_buffer);
if (status == OK)
    {
    mode_header_ptr = (dev_scsi_worm_mode_header_ptr_type)
                              &mode_buffer.header;
    if (!mode_header_ptr->enable_blank_check)
            {
        status = IO_ENXIO_DEVICE_NOT_SUPPORTED;
        }
    }

done:
return(status);
}

/*.function */

               /*<-------------------------*/
UNWIRED
status_type     dev_sd_init_optical_disk_unit  (uip)
               /*>-------------------------*/

dev_sd_unit_info_ptr_type                   uip; /* READ/WRITE */

/*.Summary[=--------------------------------------------------------------
/*
/*   Initialize a optical disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the initialization
/*   operation.
/*
/*.Functional_Description
/*
/*   This function is called to initialize a optical disk unit.
/*   Initialization consists of setting the unit to operate
/*   in the desired mode.
/*
/*.Return_Value
/*
/* OK -- The unit initialization was successful.
/*
/*   Return values for dev_sd_start_sync_request.
/*

{
status_type                 status;
```

```
dev_scsi_da_mode_buffer_type                        mode_buffer;
dev_scsi_da_mode_buffer_error_param_page_ptr_type   error_page_ptr;


/*.Implementation[=-------------------------------------------------
/*
/*   If the disk is an erasable optical disk, set the awre bit
/*   in the error recovery parameters page.  This will allow the
/*   controller to do automatic bad block remapping when write
/*   and verifies fail.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
if (uip->disk_type == DEV_SD_DISK_TYPE_ERASABLE_OPTICAL)
    {
    status = dev_sd_sense_unit_mode(uip,
                                    DEV_SCSI_MODE_SENSE_ERROR_RECOVERY_PAGE,
                                    DEV_SCSI_MODE_SENSE_CURRENT_VALUES,
                                    &mode_buffer);
    if (status == OK)
       {
        error_page_ptr = (dev_scsi_da_mode_buffer_error_param_page_ptr_type)
                                                    &mode_buffer.page;
/*.Implementation_Continued[=---------------------------------------
/*
/*    Set awre. Note that all reserved fields and mode
/*    sense fields that do no apply to mode select must be
/*    zeroed out. The devices don't always clear the reserved
/*    fields on the mode sense but are sensitive to nonzero values
/*    on the mode select.
/*
/*.End]=-----------------------------------------------------------*/

        error_page_ptr->awre = TRUE;
        error_page_ptr->param_savable = FALSE;
        error_page_ptr->reserved1 = 0;
        mode_buffer.header.reserved1 = 0;
        mode_buffer.header.reserved2 = 0;
        mode_buffer.blk_descriptor.reserved_must_be_zero = 0;
        status = dev_sd_select_unit_mode(uip, &mode_buffer);
        if (status != OK)
            {
            status = IO_ENXIO_DEVICE_NOT_SUPPORTED;
            }
        }
    }

return(status);
}

/*.function */

               /*<--------------------*/
WIRED
status_type     dev_sd_sense_unit_mode  (uip,
                         page_code,
                         page_control,
                         mode_buffer_ptr)
               /*>--------------------*/

dev_sd_unit_info_ptr_type        uip;          /* READ/WRITE */
bit8e_type                       page_code;    /* READ ONLY  */
bit8e_type                       page_control;/* READ ONLY  */
dev_scsi_da_mode_buffer_ptr_type mode_buffer_ptr; /* WRITE ONLY */

/*.Summary[=-------------------------------------------------------------
/*
/*   Get an operating mode parameter page of a specified disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the operation.
/*
```

```
/* page_code -- Specifies the type of mode information (page
/*   type) requested.
/*
/* page_code -- Specifies the type of page values to be
/*   returned: current, changeable, default, or saved.
/*
/* mode_buffer_ptr -- A pointer to the data buffer which
/*   the mode information is to be returned in.
/*
/*.Functional_Description
/*
/*   This function performs a mode sense command on the specified
/*   unit and returns the mode information in the supplied buffer.
/*   This function allows a single page of mode information to be
/*   read.
/*
/*.Return_Value
/*
/* OK -- The mode select operation was successful.
/*
/*   Return values from dev_sd_start_sync_request.
/*

[
status_type              status;
dev_adapter_request_block_ptr_type  arb_ptr;
dev_scsi_da_mode_sense_cmd_blk_ptr_type mode_sense_cmd_blk_ptr;

/*.Implementation[=-----------------------------------------------
/*
/*   Obtain the unit request lock and allocate an adapter request
/*   block.
/*
/*.End]=-------------------------------------------------------------*/

io_sync_obtain_interleave_lock(&uip->request_lock);
misc_dequeue_from_head(&uip->arb_free_queue,
                       (misc_queue_links_ptr_type *)&arb_ptr);

/*.Implementation_Continued[=-------------------------------------
/*
/*   Build the SCSI command block needed to request the mode
/*   information.
/*
/*.End]=-------------------------------------------------------------*/

mode_sense_cmd_blk_ptr = (dev_scsi_da_mode_sense_cmd_blk_ptr_type)
                         &arb_ptr->scsi_cmd_blk;
mode_sense_cmd_blk_ptr->op_code = DEV_SCSI_CMD_MODE_SENSE;
mode_sense_cmd_blk_ptr->lun = uip->unit_spec.unit;
mode_sense_cmd_blk_ptr->reserved1 = 0;
mode_sense_cmd_blk_ptr->page_control = page_control;
mode_sense_cmd_blk_ptr->page_code = page_code;
mode_sense_cmd_blk_ptr->reserved2 = 0;
mode_sense_cmd_blk_ptr->alloc_len = usizeof(dev_scsi_da_mode_buffer_type);
mode_sense_cmd_blk_ptr->vendor_unique = 0;
mode_sense_cmd_blk_ptr->reserved3 = 0;
mode_sense_cmd_blk_ptr->link = 0;
mode_sense_cmd_blk_ptr->flag = 0;

/*.Implementation_Continued[=-------------------------------------
/*
/*   Complete the adapter request block and issue the request.
/*   Return the status of the operation to the caller.
/*
/*.End]=-------------------------------------------------------------*/

arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;


io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                                (pointer_to_any_type)mode_buffer_ptr,
                                usizeof(dev_scsi_da_mode_buffer_type));
```

```
status = dev_sd_start_sync_request(uip, arb_ptr);
misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
if (io_assign_next_interleave_waiter(&uip->request_lock))
    {
    dev_sd_start_async_request(uip);
    }
io_release_interleave_lock(&uip->request_lock);
return(status);
}

/*.function */

             /*<-------------------*/
WIRED
status_type     dev_sd_select_unit_mode  (uip, mode_buffer_ptr)
             /*>-------------------*/

dev_sd_unit_info_ptr_type           uip;             /* READ/WRITE */
dev_scsi_da_mode_buffer_ptr_type    mode_buffer_ptr; /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------
/*
/*   Set an operating mode parameter page on a specified disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the operation.
/*
/* mode_buffer_ptr -- A pointer to the data buffer which
/*   contains the mode information that is to be selected.
/*
/*.Functional_Description
/*
/*   This function performs a mode select command on the specified
/*   unit. The mode select updates the current operating mode of the
/*   unit. This function allows a single page of mode information
/*   to be selected. The page type is specified in the data specified
/*   by <mode_buffer_ptr>.
/*
/*.Return_Value
/*
/* OK -- The mode select operation was successful.
/*
/*   Return values from dev_sd_start_sync_request.
/*

{
status_type                 status;
dev_adapter_request_block_ptr_type    arb_ptr;
dev_scsi_da_mode_select_cmd_blk_ptr_type  mode_select_cmd_blk_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Obtain the unit request lock and allocate an adapter request
/*   block.
/*
/*.End]=-----------------------------------------------------------*/

io_sync_obtain_interleave_lock(&uip->request_lock);
misc_dequeue_from_head(&uip->arb_free_queue,
                       (misc_queue_links_ptr_type *)&arb_ptr);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Build the SCSI command block needed to select the requested
/*   mode.
/*
/*.End]=-----------------------------------------------------------*/

mode_select_cmd_blk_ptr = (dev_scsi_da_mode_select_cmd_blk_ptr_type)
                          &arb_ptr->scsi_cmd_blk;
mode_select_cmd_blk_ptr->op_code = DEV_SCSI_CMD_MODE_SELECT;
mode_select_cmd_blk_ptr->lun = uip->unit_spec.unit;
mode_select_cmd_blk_ptr->reserved1 = 0;
```

```
mode_select_cmd_blk_ptr->save_mode_params = 0;
mode_select_cmd_blk_ptr->reserved2 = 0;
mode_select_cmd_blk_ptr->param_list_len =
     usizeof(dev_scsi_mode_header_type) +
     mode_buffer_ptr->header.blk_desc_len +
     DEV_SCSI_MODE_BUFFER_PAGE_PAGE_HEADER_SIZE +
     mode_buffer_ptr->page.page_header.page_length;
mode_select_cmd_blk_ptr->vendor_unique = 0;
mode_select_cmd_blk_ptr->reserved3 = 0;
mode_select_cmd_blk_ptr->link = 0;
mode_select_cmd_blk_ptr->flag = 0;

/*.Implementation_Continued[=---------------------------------------
/*
/*   Complete the adapter request block and issue the request.
/*   Return the status of the operation to the caller.
/*
/*.End]=-------------------------------------------------------------*/

arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
io_init_one_entry_buffer_vector(
                  &arb_ptr->buffer_vector,
                         (pointer_to_any_type)mode_buffer_ptr,
                         (uint32_type)mode_select_cmd_blk_ptr->param_list_len);
status = dev_sd_start_sync_request(uip, arb_ptr);
misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
if (io_assign_next_interleave_waiter(&uip->request_lock))
    {
    dev_sd_start_async_request(uip);
    }
io_release_interleave_lock(&uip->request_lock);
return(status);
}

/*.function */

               /*<--------------------*/
WIRED
status_type     dev_sd_test_mode_select  (uip)
               /*>--------------------*/

dev_sd_unit_info_ptr_type                  uip;  /* READ/WRITE */

/*.Summary[=---------------------------------------------------------------
/*
/*   Test a disk unit for accessibility after a mode select
/*   operation.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the initialization
/*   operation.
/*
/*.Functional_Description
/*
/*   This function is called to insure that the correct modes
/*   have been selected on a disk unit. Some types of disk
/*   units support multiple medium types. The disk driver
/*   must set the disk controller to the mode that matches the
/*   current medium type before it can access the disk. In
/*   some cases, the disk driver must determine the medium type
/*   through trial and error. This function is called to verify
/*   that the disk controller mode settings match the medium
/*   type.
/*
/*.Return_Value
/*
/* OK -- Access to the medium was successful.
/*
/*   Return values from dev_sd_start_sync_request.
/*

{
status_type                status;
```

```
dev_scsi_read_write_cmd_blk_ptr_type          scsi_rw_cmd_blk_ptr;
pointer_to_any_type                       read_buffer_ptr;
dev_adapter_request_block_ptr_type                arb_ptr;


/*.Implementation[=-------------------------------------------------
/*
/*   Allocate a buffer to receive data in. Dequeue a generic
/*   adapter request block.
/*
/*.End]=-----------------------------------------------------------*/


read_buffer_ptr = vm_get_wired_memory((uint32_type)uip->sector_size*2,
                                      VM_DEFAULT_ALIGNMENT);
io_sync_obtain_interleave_lock(&uip->request_lock);
misc_dequeue_from_head(&uip->arb_free_queue,
                       (misc_queue_links_ptr_type *)&arb_ptr);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Build a SCSI read command. We will attempt to read two
/*   blocks at an offset that is guaranteed to fail if the mode
/*   selected does not match the inserted medium. See dev_sd_def.h
/*   for a complete description of how this offset was selected.
/*
/*.End]=-----------------------------------------------------------*/

scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                      &arb_ptr->scsi_cmd_blk;
scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ;
scsi_rw_cmd_blk_ptr->lun = uip->unit_spec.unit;
scsi_rw_cmd_blk_ptr->logical_block_address =
                        DEV_SD_FLOPPY_MODE_SELECT_TEST_SECTOR;
scsi_rw_cmd_blk_ptr->transfer_length =  2;
scsi_rw_cmd_blk_ptr->vendor_unique = 0;
scsi_rw_cmd_blk_ptr->reserved1 = 0;
scsi_rw_cmd_blk_ptr->link = FALSE;
scsi_rw_cmd_blk_ptr->flag = FALSE;
arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                        (pointer_to_any_type)read_buffer_ptr,
                        (uint32_type)uip->sector_size*2);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Make the request, release the buffer memory and return the
/*   status of the request. Error logging is turned off for the
/*   request since an error in this code path is expected
/*   as part of trail and error mode selection.
/*
/*.End]=-----------------------------------------------------------*/

uip->inhibit_error_logging = TRUE;
status = dev_sd_start_sync_request(uip, arb_ptr);
misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
if (io_assign_next_interleave_waiter(&uip->request_lock))
    {
    dev_sd_start_async_request(uip);
    }
io_release_interleave_lock(&uip->request_lock);

uip->inhibit_error_logging = FALSE;

vm_release_wired_memory(read_buffer_ptr, (uint32_type)uip->sector_size*2);
return(status);
}

/*.function */

                /*<-------------------------*/
WIRED
void             dev_sd_control_medium_removal  (uip, inhibit_removal_flag)
                 /*>-------------------------*/
```

```
dev_sd_unit_info_ptr_type    uip;                    /* READ/WRITE */
boolean_type                 inhibit_removal_flag;   /* READ ONLY  */

/*.Summary[=-------------------------------------------------------------
/*
/*   Enable or disable removal of the medium on a removable medium
/*   device.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the command.
/*
/* inhibit_removal_flag -- Boolean, if TRUE, medium removal is
/*   disabled for the device. If false, medium removal is enabled.
/*
/*.Functional_Description
/*
/*   A SCSI Prevent/Allow Medium Removal command is issued to the
/*   target device. <inhibit_removal_flag> indicates whether medium
/*   removal is to be allowed or inhibited. If <inhibit_removal_flag>
/*   is TRUE, the medium eject button on the device is disabled. If
/*   <inhibit_removal_flag> is FALSE, the eject button is enabled.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Remarks
/*
/*   This function is used on direct access removable medium devices
/*   to prevent the user from removing the medium while a file system
/*   on the device is active (mounted). This function is a no-op
/*   if issued on a device that does not support the Prevent/Allow
/*   Medium Removal command.
/*

{

dev_scsi_control_medium_removal_cmd_blk_ptr_type   scsi_cmd_blk_ptr;
dev_adapter_request_block_ptr_type                 arb_ptr;


/*.Implementation[=-----------------------------------------------
/*
/*   Allocate a generic adapter request block to use in issuing
/*   the command.
/*
/*.End]=-----------------------------------------------------------*/

io_sync_obtain_interleave_lock(&uip->request_lock);
misc_dequeue_from_head(&uip->arb_free_queue,
                       (misc_queue_links_ptr_type *)&arb_ptr);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Build a SCSI Prevent/Allow Medium Removal command. Use the
/*   calllers control flag to set the prohibit flag in the
/*   command block.
/*
/*.End]=-----------------------------------------------------------*/

scsi_cmd_blk_ptr = (dev_scsi_control_medium_removal_cmd_blk_ptr_type)
                   &arb_ptr->scsi_cmd_blk;
scsi_cmd_blk_ptr->op_code = DEV_SCSI_CMD_CONTROL_MEDIUM_REMOVAL;
scsi_cmd_blk_ptr->lun = uip->unit_spec.unit;
scsi_cmd_blk_ptr->reserved1 = 0;
scsi_cmd_blk_ptr->reserved2 = 0;
scsi_cmd_blk_ptr->reserved3 = 0;
scsi_cmd_blk_ptr->prohibit = inhibit_removal_flag;
scsi_cmd_blk_ptr->vendor_unique = 0;
scsi_cmd_blk_ptr->reserved4 = 0;
scsi_cmd_blk_ptr->link = FALSE;
scsi_cmd_blk_ptr->flag = FALSE;
arb_ptr->request_flags = (bit16e_type)0;
```

```
io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                                DEV_SD_NULL_BUFFER_PTR,
                                (uint32_type)0);

/*.Implementation_Continued[=----------------------------------------
/*
/*   Issue the request to the target device. Ignore any status
/*   generated. If the command failed we assume it failed because
/*   the device does not support the command (i.e. non-removable
/*   medium device). After the command completes, release all
/*   resources used to execute the command.
/*
/*.End]=-------------------------------------------------------------*/

(void)dev_sd_start_sync_request(uip, arb_ptr);
misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
if (io_assign_next_interleave_waiter(&uip->request_lock))
    {
    dev_sd_start_async_request(uip);
    }
io_release_interleave_lock(&uip->request_lock);

return;
}

/*.function */

              /*<----------------------*/
UNWIRED
status_type     dev_sd_read_disk_capacity  (uip, capacity_buffer_ptr)
              /*>----------------------*/

dev_sd_unit_info_ptr_type             uip;                  /* READ/WRITE */
dev_sd_read_capacity_buffer_ptr_type  capacity_buffer_ptr; /* WRITE ONLY */

/*.Summary[=---------------------------------------------------------------
/*
/*   Read capacity of a disk unit.
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk device which is the target of the read capacity
/*   operation.
/*
/* capacity_buffer_ptr -- A pointer to the buffer to which
/*   the capacity information is to be written.
/*
/*.Functional_Description
/*
/*   This function is called to issue a SCSI Read Capacity command
/*   to a disk unit and return capacity information to the buffer
/*   specified by <capacity_buffer_ptr>.
/*
/*.Return_Value
/*
/* OK -- The read capacity operation was successful.
/*
/*   Return values for dev_sd_start_sync_request.
/*

{
status_type             status;
dev_adapter_request_block_ptr_type  arb_ptr;
dev_sd_read_capacity_cmd_blk_ptr_type           scsi_cmd_blk_ptr;

/*.Implementation[=--------------------------------------------------
/*
/*   Build a read capacity command and issue it to the target
/*   device.
/*
/*.End]=-------------------------------------------------------------*/

io_sync_obtain_interleave_lock(&uip->request_lock);
misc_dequeue_from_head(&uip->arb_free_queue,
```

```
                                (misc_queue_links_ptr_type *)&arb_ptr);
scsi_cmd_blk_ptr = (dev_sd_read_capacity_cmd_blk_ptr_type)
                            &arb_ptr->scsi_cmd_blk;
scsi_cmd_blk_ptr->op_code = DEV_SCSI_CMD_READ_CAPACITY;
scsi_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
scsi_cmd_blk_ptr->reserved1 = 0;
scsi_cmd_blk_ptr->relative_addr = FALSE;
scsi_cmd_blk_ptr->logical_block_addr_high = 0;
scsi_cmd_blk_ptr->logical_block_addr_low = 0;
scsi_cmd_blk_ptr->reserved2 = 0;
scsi_cmd_blk_ptr->reserved3 = 0;
scsi_cmd_blk_ptr->vendor_unique1 = 0;
scsi_cmd_blk_ptr->reserved4 = 0;
scsi_cmd_blk_ptr->pmi = FALSE;
scsi_cmd_blk_ptr->vendor_unique2 = 0;
scsi_cmd_blk_ptr->reserved5 = 0;
scsi_cmd_blk_ptr->flag = FALSE;
scsi_cmd_blk_ptr->link = FALSE;
arb_ptr->request_flags = DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER;
io_init_one_entry_buffer_vector(&arb_ptr->buffer_vector,
                        (pointer_to_any_type)capacity_buffer_ptr,
                        usizeof(dev_sd_read_capacity_buffer_type));
status = dev_sd_start_sync_request(uip, arb_ptr);

/*.Implementation_Continued[=---------------------------------------
/*
/*   Free all resources used to issue the command and return the
/*   status of the operation.
/*
/*.End]=-------------------------------------------------------------*/

misc_enqueue_at_tail(&uip->arb_free_queue, &arb_ptr->links);
if (io_assign_next_interleave_waiter(&uip->request_lock))
    {
    dev_sd_start_async_request(uip);
    }
io_release_interleave_lock(&uip->request_lock);
return(status);
}

/*.function */

               /*<-------------------------*/
WIRED
uint32_type     dev_scsi_get_bytes_requested  (cmd)
               /*>-------------------------*/

dev_scsi_generic_cmd_ptr_type       cmd;        /* READ ONLY */

/*.Summary[=---------------------------------------------------------
/*
/*   Convert commands data buffer size to bytes.
/*
/*.Parameters
/*
/* cmd -- A pointer to the generic command block structure
/*   containing the command code and data size information.
/*
/*.Functional_Description
/*
/*   This function is called to interpret the length designator field
/*   of the SCSI command block based of the command type.
/*   The unit and translation of special value of the data length (byte 4)
/*   field is not consistent between commands, this function converts the
/*   each length to bytes.
/*
/*.Return_Value
/*
/* The maximun number of bytes to by provided or expected
/*   by the command initiator.
/*
/*.Exceptions
/*
/*   None.
/*
```

```
/*.Abort_Conditions
/*
/*    None
/*

{
uint32_type num_bytes;
uint32_type bytes_per_sector;
uint32_type size;

/*.Implementation[=-------------------------------------------------
/*
/*    Convert length according to command type.
/*    NOTE: bytes-per-sector log2(number-of-bytes) - 7
/*    Ex:   bytes-per-sector = 1 ==>represents 256 bytes/sector
/*
/*.End]=-----------------------------------------------------------*/

size = (uint32_type)cmd->alloc_len;
switch (cmd->op_code)
    {
    case DEV_SCSI_CMD_REQUEST_SENSE:
            if( size == 0 )
                        {
                        num_bytes = 4;
                        }
            else
                        {
                        num_bytes = size;
                        }
        break;

    case DEV_SCSI_CMD_INQUIRY:
    case DEV_SCSI_CMD_MODE_SELECT:
    case DEV_SCSI_CMD_MODE_SENSE:
            if( size == 0 )
                        {
                        num_bytes = 0;
                        }
            else
                        {
                        num_bytes = size;
                        }
            break;

    case DEV_SCSI_CMD_READ:
    case DEV_SCSI_CMD_WRITE:
            if( size == 0 )
                        {
                        num_bytes = 0;
                        }
            else
                        {
                        bytes_per_sector = (uint32_type)cmd->log2_bytes_sector;
                        num_bytes = size * (1 << (bytes_per_sector + 7));
                        }
            break;

    default:
            num_bytes = 0;
    }

return(num_bytes);

}

/*.function */

              /*<-----------------------*/
UNWIRED
status_type     dev_sd_determine_disk_type  (uip, disk_type_ptr)
              /*>-----------------------*/

dev_sd_unit_info_ptr_type             uip;              /* READ ONLY */
uint16_ptr_type                       disk_type_ptr;    /* WRITE ONLY */
```

```
/*.Summary[=-----------------------------------------------------------
/*
/*   Determine the type of a disk (i.e. rigid, floppy, erasable
/*   optical, etc).
/*
/*.Parameters
/*
/* uip -- A pointer to the unit information structure of
/*   the disk.
/*
/* disk_type_ptr -- A pointer to where the disk type is to
/*   be returned.
/*
/*.Functional_Description
/*
/*   This function determines the type of a disk, using information
/*   found in the inquiry buffer and mode sense buffer.  It assumes
/*   that an inquiry has already been performed, and that the uip
/*   contains a pointer to the inquiry buffer.
/*
/*.Return_Value
/*
/* OK -- The type of the disk was determined successfully.
/*
/* Any error returned by dev_sd_sense_unit_mode.
/*
/* IO_ENXIO_DEVICE_NOT_SUPPORTED -- The type of the disk
/*      could not be determined from the information given.
/*

{

uint8_type              device_type;
dev_scsi_da_mode_buffer_type            mode_buffer;
uint32_type             status;

/*.Implementation[=-------------------------------------------------
/*
/*   Get the device type from the inquiry buffer.  Use this to
/*   narrow down the disk type.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
device_type = uip->inquiry_buffer.device_type;

switch (device_type)
   {
   case DEV_SCSI_DEVICE_DIRECT_ACCESS:
      if (uip->inquiry_buffer.removable_medium)
         {

/*.Implementation[=-----------------------------------------
/* The disk is either a floppy or an erasable optical disk.
/* It is necessary to do a mode sense to determine which type
/* it is.  If the medium type is 0, then the disk is an
/* erasable optical disk; otherwise, it is a floppy.
/*.End]=-----------------------------------------------------*/

         status = dev_sd_sense_unit_mode(uip,
                                DEV_SCSI_MODE_SENSE_VENDOR_UNIQUE_PAGE,
                                DEV_SCSI_MODE_SENSE_CURRENT_VALUES,
                                &mode_buffer);

            if (status == OK)
               {
               if (mode_buffer.header.medium_type == 0)
                  {
              *disk_type_ptr = DEV_SD_DISK_TYPE_ERASABLE_OPTICAL;
                  }
               else
                  {
                  *disk_type_ptr = DEV_SD_DISK_TYPE_FLOPPY;
               }
                }
```

```
        }
      else
        {
        *disk_type_ptr = DEV_SD_DISK_TYPE_RIGID;
        }
      break;

   case DEV_SCSI_DEVICE_WRITE_ONCE_READ_MULTIPLE:
      *disk_type_ptr = DEV_SD_DISK_TYPE_WORM;
      break;

   case DEV_SCSI_DEVICE_DIRECT_ACCESS_READ_ONLY:
      *disk_type_ptr = DEV_SD_DISK_TYPE_READ_ONLY;
      break;

   default:
      status = IO_ENXIO_DEVICE_NOT_SUPPORTED;
      break;
   }

return(status);

}

/*.function */

              /*<-------------------------*/
WIRED
status_type     dev_sd_complete_async_sb_io  (arb_ptr, done_ptr)
              /*>-------------------------*/

dev_adapter_request_block_ptr_type  arb_ptr;/* READ/WRITE */
boolean_ptr_type                    done_ptr;/* READ/WRITE */


/*.Summary[=-------------------------------------------------------------
/*
/*   Determines the proper actions to take when an io completes
/*   which is part of a sector buffered io.  Sector buffering is
/*   used when the disk sector size is greater than 512-bytes/
/*   sector.
/*
/*.Parameters
/*
/* arb_ptr -- A pointer to an adapter request block.
/* done_ptr -- A pointer to a boolean flag which indicates
/*   whether the sector buffered io is completely finished.
/*
/*.Functional_Description
/*
/*   This function determines the proper actions to take, based
/*   on the values of the request flags in the adapter request
/*   block, when an io completes which is part of a sector
/*   buffered io.  If the flags indicate that the request is
/*   done, then if the operation was a read, the data is transferred
/*   from the sector buffer to the original io buffer.  The
/*   memory for the buffer is then released.  If the original
/*   operation is a write, the original buffer is copied into
/*   the sector buffer, and a new asynchronous io is started.
/*
/*.Return_Value
/*
/* OK -- /*

{

io_operation_record_ptr_type                    op_record_ptr;
dev_sd_unit_info_ptr_type                       uip;
dev_scsi_read_write_cmd_blk_ptr_type    scsi_rw_cmd_blk_ptr;
dev_scsi_write_verify_cmd_blk_ptr_type          scsi_write_verify_cmd_blk_ptr;
pointer_to_any_type                 sector_buffer_ptr;
uint32_type             sector_buffer_size;
pointer_to_any_type                 io_buffer_ptr;
uint32_type             io_buffer_size;
pointer_to_any_type                 sector_buffer_position_ptr;
```

```
uint32_type             logical_block_addr;

/*.Implementation[=-------------------------------------------------
/*
/*   Set up the required structures and get the necessary buffer
/*   info.
/*
/*.End]=-----------------------------------------------------------*/

op_record_ptr = arb_ptr->op_record_ptr;
uip = (dev_sd_unit_info_ptr_type)(op_record_ptr->ri.device_handle);

io_reset_buffer_vector_position(&arb_ptr->buffer_vector);
io_get_buffer_vector_io_info(&arb_ptr->buffer_vector,
                             &sector_buffer_ptr,
                             &sector_buffer_size);

io_get_buffer_vector_io_info(&op_record_ptr->ri.buffer_vector,
                             &io_buffer_ptr,
                             &io_buffer_size);

*done_ptr = FALSE;

/*.Implementation[=-------------------------------------------------
/*
/*   If the request is done, determine if the original request
/*   was a read.  If so, copy the requested data into the
/*   original buffer.  Then, release the memory for the buffer.
/*
/*.End]=-----------------------------------------------------------*/

if (arb_ptr->request_flags & DEV_SCSI_REQUEST_FLAGS_SB_DONE)
   {
   if (arb_ptr->request_flags & DEV_SCSI_REQUEST_FLAGS_SB_READ)
      {
      sector_buffer_position_ptr = sector_buffer_ptr +
                                                   (op_record_ptr->ri.device_offset %
                           uip->sector_size);

      misc_byte_copy(sector_buffer_position_ptr,
                             io_buffer_ptr,
                             io_buffer_size);
      }

   vm_release_wired_memory(sector_buffer_ptr, sector_buffer_size);
   *done_ptr = TRUE;

   }
else
   {

/*.Implementation[=-------------------------------------------------
/*
/*   Set up to do an asyncchronous write.  First, copy the
/*   original buffer into the sector buffer.  Then, do the
/*   write.
/*
/*.End]=-----------------------------------------------------------*/

   sector_buffer_position_ptr = sector_buffer_ptr +
                                                 (op_record_ptr->ri.device_offset %
                         uip->sector_size);

   misc_byte_copy(io_buffer_ptr,
                          sector_buffer_position_ptr,
                          io_buffer_size);

   if   ((uip->disk_type == DEV_SD_DISK_TYPE_ERASABLE_OPTICAL)
      || (uip->disk_type == DEV_SD_DISK_TYPE_WORM))
      {
      scsi_write_verify_cmd_blk_ptr =
               (dev_scsi_write_verify_cmd_blk_ptr_type)&arb_ptr->scsi_cmd_blk;
      scsi_write_verify_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE_VERIFY;
      scsi_write_verify_cmd_blk_ptr->lun = arb_ptr->unit_spec.unit;
      scsi_write_verify_cmd_blk_ptr->reserved1 = 0;
```

```
      scsi_write_verify_cmd_blk_ptr->relative_addr = FALSE;
      logical_block_addr = op_record_ptr->ri.device_offset / uip->sector_size;
      scsi_write_verify_cmd_blk_ptr->logical_block_addr_high =
                  (uint16_type)((logical_block_addr>> 16) & 0xffff);
      scsi_write_verify_cmd_blk_ptr->logical_block_addr_low =
                           (uint16_type)(logical_block_addr & 0xffff);
      scsi_write_verify_cmd_blk_ptr->reserved2 = 0;
      scsi_write_verify_cmd_blk_ptr->transfer_length_high = 0;
      scsi_write_verify_cmd_blk_ptr->transfer_length_low =  sector_buffer_size /
                                                             uip->sector_size;
      scsi_write_verify_cmd_blk_ptr->reserved3 = 0;
      scsi_write_verify_cmd_blk_ptr->erase_control = FALSE;
      scsi_write_verify_cmd_blk_ptr->reserved4 = 0;
      scsi_write_verify_cmd_blk_ptr->link = FALSE;
      scsi_write_verify_cmd_blk_ptr->flag = FALSE;
      }
   else
      {
      scsi_rw_cmd_blk_ptr = (dev_scsi_read_write_cmd_blk_ptr_type)
                           &arb_ptr->scsi_cmd_blk;

      scsi_rw_cmd_blk_ptr->op_code = DEV_SCSI_CMD_WRITE;
      scsi_rw_cmd_blk_ptr->lun = uip->unit_spec.unit;
      scsi_rw_cmd_blk_ptr->logical_block_address =
                         op_record_ptr->ri.device_offset /
                                              uip->sector_size;
      scsi_rw_cmd_blk_ptr->transfer_length =  sector_buffer_size /
                        uip->sector_size;
      scsi_rw_cmd_blk_ptr->vendor_unique = 0;
      scsi_rw_cmd_blk_ptr->reserved1 = 0;
      scsi_rw_cmd_blk_ptr->link = FALSE;
      scsi_rw_cmd_blk_ptr->flag = FALSE;
      }

   misc_increment(&uip->write_request_count);
   misc_increment_by_value(&uip->write_block_count,
                     (int32e_type)(sector_buffer_size/DF_BYTES_PER_BLOCK));
   arb_ptr->request_flags |= DEV_SCSI_REQUEST_FLAGS_SB_DONE;
   arb_ptr->sync_io = FALSE;
   arb_ptr->op_record_ptr = op_record_ptr;
   arb_ptr->complete_io_routine = dev_sd_complete_io;

/*.Implementation_Continued[=---------------------------------------
/*
/*   Issue the command to the supporting adapter. Any status
/*   information generated by the request will be processed by
/*   the driver complete I/O routine.
/*
/*.End]=-------------------------------------------------------------*/

   (void)dev_scsi_adapter_issue_async_command(
                        uip->adapter_device_number.major,
                                  arb_ptr);
   }

return(OK);
}
```

System File Entries

This section shows a partial listing of a system file showing the **sd** driver's entry.

```
#
# System file
#

# drivers
sd(cisc(),0)
hken()
loop()
```

```
#syac()
prf()
meter()
```

Master File Entries

The following section shows a partial listing of the master file:

```
#------------------------------------------------------------------
# Disks:
#
#
#
#                          Maximum  #
#    Name       Major       of units per    Restriction
#    Prefix     Number(s)   Controller      Flags
#    --------   ---------   -------------   -----------
#
     sd         6           7               n
     cird       7           7               n
#
#------------------------------------------------------------------
```

End of Appendix

Appendix B
A Sample SCSI Adapter Driver

This appendix gives the sample code for an SCSI adapter driver. We include the type definitions, global data definitions, driver supplied I/O routines, an example system file, and an example master file. For this example, **xxx** is replaced by **cisc**.

NOTE: The code provided here is only a sample. It is not guaranteed to be either complete or operational.

Data Definitions: dev_cisc_def.h

```
/*<-------------------------------------------------------*/
/*              dev_cisc_def.h                           */
/*>-------------------------------------------------------*/

/*.Contents[=-------------------------------------------------------------
/*
/* DEV_CISC_SCSI_TARGET_ID -- subsystem
/* DEV_CISC_SCSI_HOST_ID -- subsystem
/* DEV_CISC_SCSI_HOST_UNIT -- subsystem
/* DEV_CISC_STATUS_REGISTER_MASK -- subsystem
/* DEV_CISC_STATUS_READY_AFTER_RESET -- subsystem
/* DEV_CISC_PRIMARY_DEVICE_CODE_DEFAULT -- subsystem
/* DEV_CISC_SECONDARY_DEVICE_CODE_DEFAULT -- subsystem
/* DEV_CISC_PRIMARY_ADDRESS_DEFAULT -- subsystem
/* DEV_CISC_SECONDARY_ADDRESS_DEFAULT -- subsystem
/* DEV_CISC_MAX_ADAPTERS -- subsystem
/* DEV_CISC_MAX_REQUEST_SIZE -- subsystem
/* DEV_CISC_MAX_PARAM_BLOCKS -- subsystem
/* DEV_CISC_MAX_STATUS_BLOCKS -- subsystem
/* DEV_CISC_MAX_SCATTER_GATHER_BLOCKS -- subsystem
/* DEV_CISC_MAX_CONCURRENT_REQUESTS  -- subsystem
/* DEV_CISC_ISR_POSITION_VME_INTERRUPT_LEVEL_2 -- subsystem
/* DEV_CISC_INTERRUPT_LEVEL -- subsystem
/* DEV_CISC_UNIT_OPTIONS_TIMEOUT_DISABLE -- subsystem
/* DEV_CISC_UNIT_OPTIONS_SELECT_TIMEOUT -- subsystem
/* DEV_CISC_UNIT_OPTIONS_DEFAULT_DISCON_TIMEOUT -- subsystem
/* DEV_CISC_SCATTER_GATHER_TERMINAL_LINK -- subsystem
/* DEV_CISC_CMD_START_LIST -- subsystem
/* DEV_CISC_CMD_STOP_LIST -- subsystem
/* DEV_CISC_CMD_IDENTIFY -- subsystem
/* DEV_CISC_CMD_GET_BOARD_STATUS -- subsystem
/* DEV_CISC_CMD_SET_CONTROLLER_OPTIONS -- subsystem
/* DEV_CISC_CMD_SET_UNIT_OPTIONS -- subsystem
/* DEV_CISC_CMD_SELF_TEST -- subsystem
/* DEV_CISC_SELF_TEST_STATIC_RAM_TEST -- subsystem
/* DEV_CISC_SELF_TEST_PROM_TEST -- subsystem
/* DEV_CISC_RETRY_CONTROL_NONE -- subsystem
/* DEV_CISC_RETRY_CONTROL_INT -- subsystem
/* DEV_CISC_RETRY_CONTROL_ISB -- subsystem
/* DEV_CISC_RETRY_CONTROL_RPE -- subsystem
/* DEV_CISC_RETRY_CONTROL_RCE -- subsystem
/* DEV_CISC_RETRY_CONTROL_RBE -- subsystem
/* DEV_CISC_RETRY_CONTROL_NONE -- subsystem
```

```
/* DEV_CISC_UNIT_FLAGS_IDI  -- subsystem
/* DEV_CISC_UNIT_FLAGS_SYN -- subsystem
/* DEV_CISC_UNIT_FLAGS_IAT -- subsystem
/* DEV_CISC_UNIT_FLAGS_COM -- subsystem
/* DEV_CISC_UNIT_FLAGS_SOR -- subsystem
/* DEV_CISC_UNIT_FLAGS_ISE -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_NONE -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_SGO -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_DIR -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_DAT -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_IRS -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_ISC -- subsystem
/* DEV_CISC_PARAM_BLK_FLAGS_DBV -- subsystem
/* DEV_CISC_STATUS_BLK_ERROR_NO_ERROR -- subsystem
/* DEV_CISC_STATUS_BLK_ERROR_BUS_TIMEOUT -- subsystem
/* DEV_CISC_STATUS_BLK_ERROR_SELECT_TIMEOUT  -- subsystem
/* DEV_CISC_STATUS_BLK_ERROR_DISCONNECT_TIMEOUT -- subsystem
/* DEV_CISC_STATUS_BLK_ERROR_BAD_SCSI_STATUS -- subsystem
/* DEV_CISC_OPTIONS_THROTTLE_TRANSFER_COUNT -- subsystem
/* DEV_CISC_OPTIONS_THROTTLE_BYTE_COUNT  -- subsystem
/* dev_cisc_param_block_type -- subsystem
/* dev_cisc_param_block_ptr_type -- subsystem
/* DEV_CISC_REQUEST_SENSE_DATA_SIZE -- subsystem
/* dev_cisc_status_block_type -- subsystem
/* dev_cisc_status_block_ptr_type -- subsystem
/* dev_cisc_command_list_type -- subsystem
/* dev_cisc_command_list_type -- subsystem
/* dev_cisc_command_list_ptr_type -- subsystem
/* dev_cisc_set_up_cmd_list_pb_type -- subsystem
/* dev_cisc_set_up_cmd_list_pb_ptr_type  -- subsystem
/* dev_cisc_run_diagnostics_pb_type -- subsystem
/* dev_cisc_run_diagnostics_pb_ptr_type -- subsystem
/* dev_cisc_set_unit_options_pb_type -- subsystem
/* dev_cisc_unit_options_pb_ptr_type -- subsystem
/* dev_cisc_general_options_pb_type -- subsystem
/* dev_cisc_general_options_pb_ptr_type -- subsystem
/* dev_cisc_identify_pb_type -- subsystem
/* dev_cisc_identify_pb_ptr_type -- subsystem
/* dev_cisc_type_0_pb_type -- subsystem
/* dev_cisc_type_0_pb_ptr_type -- subsystem
/* dev_cisc_device_info_type -- subsystem
/* dev_cisc_device_info_ptr_type -- subsystem
/* dev_cisc_unit_table_entry_type -- subsystem
/* dev_cisc_unit_table_entry_ptr_type -- subsystem
/* dev_cisc_unit_table_type -- subsystem
/* dev_cisc_unit_table_ptr_type -- subsystem
/* dev_cisc_request_blk_type -- subsystem
/* dev_cisc_request_blk_ptr_type -- subsystem
/* dev_cisc_scatter_gather_blk_type -- subsystem
/* dev_cisc_scatter_gather_blk_ptr_type -- subsystem
/* dev_cisc_physical_device_info_type -- subsystem
/* "dev_cisc_physical_device_info_ptr_type -- subsystem
/*
/*.Description
/*
/*   This module contains definitions that support the Ciprico
/*   Rimfire 3500 Host SCSI bus adapter driver modules.
/*
/*   Some of the definitions herein describe host-side only data
/*   structures that are used by the driver to keep track of the
/*   state of the outstanding requests, and are subject to change
/*   as the driver changes.  Other definitions describe
/*   the host-controller Ciprico Rimfire 3500 interface and as
/*   such are understood by the Ciprico controller.  These
/*   definitions should obviously not be changed.
/*

/*.Literal_Section[=-----------------------------------------------------
/*
/*      Miscellaneous CISC literals.
/*

#define DEV_CISC_SCSI_TARGET_ID   ((uint8e_type)0xff)
    /*
     * The SCSI id value that is used when a general board command is
```

```
     * issued to the cisc controller.
     */

#define DEV_CISC_SCSI_HOST_ID   ((uint8e_type)0x0007)
    /*
     * The SCSI id that is reserved for the host system on the cisc SCSI
     * interface.
     */

#define DEV_CISC_SCSI_HOST_UNIT   ((uint8e_type)0x0000)
    /*
     * The unit number that is used to access the cisc unit table
     * entry reservced for the host system.
     */

#define DEV_CISC_STATUS_REGISTER_MASK   ((bit32e_type)0x03ff)
    /*
     * Mask used to extract the board type and status from the
     * Ciprico 3500 Board Status Register.
     */

#define DEV_CISC_STATUS_READY_AFTER_RESET   ((bit32e_type)0x0202)
    /*
     * Masked Board Status Register value that indicates the board is
     * a Rimfire 3500 and the board is ready to accept commands.
     */

#define     DEV_CISC_PRIMARY_DEVICE_CODE_DEFAULT   0x0028
    /*
     * The default VME interrupt vector (device code) that is used for
     * a Rimfire 3500 if a device code is not present in a device spec.
     */

#define     DEV_CISC_SECONDARY_DEVICE_CODE_DEFAULT   0x0029
    /*
     * The default VME interrupt vector (device code) that is used for
     * a Rimfire 3500 if a device code is not present in a device spec
     * but the controller resides at the secondary control register
     * address.
     */

#define DEV_CISC_PRIMARY_ADDRESS_DEFAULT        0xfffff300
    /*
     * The default control register starting address that is used for
     * a Rimfire 3500 if a control register starting address is not
     * present in a device spec.
     *
     */
#define DEV_CISC_SECONDARY_ADDRESS_DEFAULT      0xfffff500
    /*
     * The default control register starting address that is used for
     * the second Rimfire 3500 controller in a system if a starting
     * address is not present in a device spec.
     */

#define DEV_CISC_MAX_ADAPTERS   0x0003
    /*
     * The maximum number of Rimfire 3500 SCSI adapters that are
     * supported in a single system configuration.
     */

#define DEV_CISC_MAX_REQUEST_SIZE               0x8000
    /*
     * The maximum request size in bytes that can be specified in
     * a single Rimfire 3500 I/O operation. This size is limited
     * to the number of bytes the kernel is willing to allow a user
     * to have wired and not by a physical restriction in the controller.
     */

#define DEV_CISC_MAX_PARAM_BLOCKS   ((uint16_type)0x0040)
    /*
     * The number of Ciprico parameter blocks allocated for a Ciprico
     * Rimfire 3500 controller.
     */
```

```
#define DEV_CISC_MAX_STATUS_BLOCKS          2*DEV_CISC_MAX_PARAM_BLOCKS
    /*
     * The number of Ciprico status blocks allocated for a Ciprico
     * Rimfire 3500 controller. Two status block are allocated for each
     * parameter block. Execution of a single parameter block may result
     * in multiple status blocks being used. As a result, more status
     * blocks than parameter blocks are needed.
     */

#define DEV_CISC_MAX_SCATTER_GATHER_BLOCKS
       DEV_CISC_MAX_PARAM_BLOCKS*((DEV_CISC_MAX_REQUEST_SIZE/32768)+1)
    /*
     * The number of scatter gather blocks allocated for a Ciprico Rimfire
     * 3500 controller. Enough blocks are allocated so that the maximum
     * data transfer size can be specified by each parameter block
     * simultaneously. 32K bytes can be specified by each scatter gather
     * block. If a 32K buffer is not page aligned, 2 scatter gather arrays
     * are required to perform the transfer.
     */

#define DEV_CISC_MAX_CONCURRENT_REQUESTS   ((uint16_type)
   DEV_CISC_MAX_PARAM_BLOCKS-1)
/*
 * The maximum number of requests (parameter blocks) that the ESDI/SMD
 * disk driver allows to be concurrently issued to the disk controller.
 * Note that this number must be one less than the actual number of
 * parameter blocks in the command list. This is required for the command
 * list circular queue mechanism to work correctly. We must prevent the
 * index from rolling over and making "list full" appear to be "list empty".
 */

#define DEV_CISC_ISR_POSITION_VME_INTERRUPT_LEVEL_2   ((uint16_type)0x0006)
    /*
     * The bit number in the Interrupt Status Register which corresponds
     * to VME devices set at interrupt level 2.
     */

#define DEV_CISC_INTERRUPT_LEVEL   ((uint16_type)0x00002)
    /*
     * The interrupt level used by the cisc controller.
     */

#define DEV_CISC_UNIT_OPTIONS_TIMEOUT_DISABLE   ((uint16_type)0x000000)
    /*
     * Value used in the Set Unit Options parameter block to disable
     * timeouts.
     */

#define DEV_CISC_UNIT_OPTIONS_SELECT_TIMEOUT   ((uint16_type)0x0000fa)
    /*
     * Value used in the Set Unit Options parameter block to request
     * a select timeout of 250 milliseconds. This field is specified in
     * units of milliseconds.
     */

#define DEV_CISC_UNIT_OPTIONS_DEFAULT_DISCON_TIMEOUT   ((uint16_type)0x00014)
    /*
     * Default value used in in the Set Unit Options parameter block
     * disconnect timeout field (2 seconds). This field is specified in
     * units of .1 seconds.
     */


#define DEV_CISC_SCATTER_GATHER_TERMINAL_LINK ((dev_cisc_scatter_gather_blk_ptr_type)0xffffff1
    /*
     * Used in "next sg" field of a scatter/gather array to indicate that
     * no other arrays are in the chain.
     *
     */

#define DEV_CISC_SYNC_SUPPORT_FIRMWARE_REVISION   ((uint8e_type)11)
    /*
     * Controllers with a firmware revision greater than or equal to this
     * value are assumed to support the SCSI synchronous data transfer
     * protocol.
```

```
     */

/*.Literal_Section[=-----------------------------------------------------
/*
/*   CISC command codes. See the Rimfire 3500 Product Specification
/*   for a complete description of these commands.
/*

#define DEV_CISC_CMD_START_LIST   ((uint8e_type)0x0001)
#define DEV_CISC_CMD_STOP_LIST    ((uint8e_type)0x0002)
#define DEV_CISC_CMD_IDENTIFY     ((uint8e_type)0x0005)
#define DEV_CISC_CMD_GET_BOARD_STATS    ((uint8e_type)0x0006)
#define DEV_CISC_CMD_SET_CONTROLLER_OPTIONS    ((uint8e_type)0x0007)
#define DEV_CISC_CMD_SET_UNIT_OPTIONS    ((uint8e_type)0x0008)
#define DEV_CISC_CMD_SELF_TEST    ((uint8e_type)0x0009)
#define DEV_CISC_SELF_TEST_STATIC_RAM_TEST    ((bit8e_type)0x0001)
#define DEV_CISC_SELF_TEST_PROM_TEST    ((bit8e_type)0x0002)

/*.Literal_Section[=-----------------------------------------------------
/*
/*   CISC Retry Control Definitions.
/*
/*.Description
/*
/*   These constants define the selectable options in the Retry
/*   Control byte of a Ciprico Rimfire 3500 Unit Options
/*   parameter block. The Retry Control options tell the controller
/*   which types of errors to retry and how to report retries
/*   to the host.
/*

#define DEV_CISC_RETRY_CONTROL_NONE    ((bit8e_type)0x0000)
    /*
     * This retry control constant is used if none of
     * the options are being selected.
     */

#define DEV_CISC_RETRY_CONTROL_INT    ((bit8e_type)0x0001)
    /*
     * Issue interrupt bit - if set, tells the controller to issue an
     * interrupt for each retry performed.
     */

#define DEV_CISC_RETRY_CONTROL_ISB    ((bit8e_type)0x0002)
    /*
     * Issue Status Block bit, if set, tells the controller to issue a
     * status block for each retry performed.
     */

#define DEV_CISC_RETRY_CONTROL_RPE    ((bit8e_type)0x0004)
    /*
     * Retry parity errors bit - if set, tells the controller to
     * retry parity errors.
     */

#define DEV_CISC_RETRY_CONTROL_RCE    ((bit8e_type)0x0008)
    /*
     * Retry command errors bit - if set, tells the controller to
     * retry command errors (SCSI device-reported errors)
     */

#define DEV_CISC_RETRY_CONTROL_RBE    ((bit8e_type)0x0010)
    /*
     * Retry SCSI bus errors bit - if set, tells the controller
     * to retry SCSI bus errors (e.g. selection timeouts).
     */

#define DEV_CISC_RETRY_CONTROL_NONE    ((bit8e_type)0x0000)
    /*
     * This retry control constant is used if none of
     * the options are being selected.
     */

/*.Literal_Section[=-----------------------------------------------------
/*
```

```
/*      CISC Unit Flags Definitions.
/*
/*.Description
/*
/*   These constants define the selectable options in the Unit
/*   Flags field of a Ciprico Rimfire 3500 Unit Options
/*   parameter block. The Unit Flags allow selection of various
/*   modes of data transfer between the unit and the Ciprico
/*   controller.
/*

#define DEV_CISC_UNIT_FLAGS_IDI   ((bit8e_type)0x0001)
    /*
     * Inihibit disconnect bit - if set, prevents a device from
     * disconnecting while a command is taking place.
     */

#define DEV_CISC_UNIT_FLAGS_SYN   ((bit8e_type)0x0002)
    /*
     * Synchronous transfer bit - if set, enables SCSI synchronous
     * data transfers to the device.
     */

#define DEV_CISC_UNIT_FLAGS_IAT   ((bit8e_type)0x0004)
    /*
     * Inhibit assert attenion bit - if set, causes the adapter
     * to refrain from asserting the ATN signal when selecting
     * the device. This option is used with targets that
     * do not respond or can't handle ATN.
     */

#define DEV_CISC_UNIT_FLAGS_SOR   ((bit8e_type)0x0008)
    /*
     * Sort Commands bit - if set, enables command sorting for
     * SCSI disk commands.
     */

#define DEV_CISC_UNIT_FLAGS_COM   ((bit8e_type)0x0008)
    /*
     * Command Combining bit - if set, enables command combining for
     * SCSI disk commands.
     */
#define DEV_CISC_UNIT_FLAGS_ISE   ((bit8e_type)0x0020)
    /*
     * Ignore SCSI Soft Errors bit - if set, SCSI soft errors
     * (Error Class. 7, Sense Key 1) will not be retries.
     */

/*.Literal_Section[=-------------------------------------------------
/*
/*   CISC Parameter Block Flag Definitions. See the Rimfire 3500
/*   Product Specification for a complete description of these flags.
/*

#define DEV_CISC_PARAM_BLK_FLAGS_NONE   ((bit8e_type)0x0000)
#define DEV_CISC_PARAM_BLK_FLAGS_SGO   ((bit8e_type)0x0001)
#define DEV_CISC_PARAM_BLK_FLAGS_DIR   ((bit8e_type)0x0002)
#define DEV_CISC_PARAM_BLK_FLAGS_DAT   ((bit8e_type)0x0004)
#define DEV_CISC_PARAM_BLK_FLAGS_IRS   ((bit8e_type)0x0008)
#define DEV_CISC_PARAM_BLK_FLAGS_ISC   ((bit8e_type)0x0040)
#define DEV_CISC_PARAM_BLK_FLAGS_DBV   ((bit8e_type)0x0080)

/*.Literal_Section[=-------------------------------------------------
/*
/*   CISC Status Block Error Code Definitions. See the Rimfire 3500
/*   Product Specification for a complete description of these error
/*   codes.
/*

#define DEV_CISC_STATUS_BLK_ERROR_NO_ERROR   ((bit8e_type)0x0000)
#define DEV_CISC_STATUS_BLK_ERROR_BUS_TIMEOUT   ((bit8e_type)0x0014)
#define DEV_CISC_STATUS_BLK_ERROR_SELECT_TIMEOUT   ((bit8e_type)0x001E)
#define DEV_CISC_STATUS_BLK_ERROR_DISCONNECT_TIMEOUT   ((bit8e_type)0x001F)
#define DEV_CISC_STATUS_BLK_ERROR_BAD_SCSI_STATUS   ((bit8e_type)0x0023)
```

```
/*.Literal_Section[=-------------------------------------------------------
/*
/*   CISC Controller Options Definitions. See the Rimfire 3500
/*   Product Specification for a complete description of these options.
/*


#define DEV_CISC_OPTIONS_THROTTLE_TRANSFER_COUNT   ((uint8e_type)0x0000)

#define DEV_CISC_OPTIONS_THROTTLE_BYTE_COUNT   ((uint8e_type)0x0001)


/*.Section[=-----------------------------------------------------------
/*
/*      Ciprico Rimfire 3500 Data Structures.
/*

/*.type */

typedef struct
    {
    uint32e_type                    param_blk_id;
    uint8e_type                     reserved_1_must_be_zero;
    bit8e_type                      flags;
    uint8e_type                     address_modifier;
    uint8e_type                     target_id;
    uint32e_type                    vme_memory_address;
    uint32e_type                    transfer_count;
    dev_scsi_cmd_blk_type                       scsi_cmd_blk;

    /*<-------------------------*/
}      dev_cisc_param_block_type   ;
    /*>-------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*   This is the standard command descriptor (parameter) block
/*   used when sending a command to Ciprico Rimfire 3500 controller.
/*   See the Rimfire 3500 Product Specification for a complete description
/*   of the members of this structure.
/*

/*.type */

typedef dev_cisc_param_block_type  *

                    /*<-----------------------------*/
                       dev_cisc_param_block_ptr_type  ;
                    /*>-----------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*   A pointer to a parameter block.
/*

#define DEV_CISC_REQUEST_SENSE_DATA_SIZE   ((uint32_type)0x0008)
    /*
     *  The number of bytes of request sense data that are returned
     *  in a single cisc status block.
     */

/*.type */

typedef struct
    {
    uint32e_type        command_id;
    skip_type                       reserved_1_must_be_zero : 8;
    bit8e_type                      scsi_status;
    bit8e_type                      error_code;
    bit8e_type                      flags;
    byte8e_type                     sense_data[DEV_CISC_REQUEST_SENSE_DATA_SIZE];

    /*<-----------------------*/
}    dev_cisc_status_block_type ;
    /*>-----------------------*/
```

```
/*.Description[=-------------------------------------------------------
/*
/*   This type defines the status block returned by a Ciprico
/*   Rimfire 3500 controller. See the Rimfire 3500 Product Specification
/*   for a complete description of the members of this structure.
/*

/*.type */

typedef dev_cisc_status_block_type  *

                    /*<-----------------------------*/
                       dev_cisc_status_block_ptr_type  ;
                    /*>-----------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*   A pointer to a status block.
/*

/*.type */

typedef struct
    {
    uint32e_type        command_id;
    uint8e_type                     firmware_revision;
    uint8e_type                     engineering_revision;
    bit8e_type                      error_code;
    bit8e_type                      flags;
    skip_type                       reserved_1_must_be_zero : 7;
    field_type                      floppy_disk_option_present_flag;
    uint8e_type                     day;
    uint8e_type                     month;
    uint8e_type                     year;
    uint32e_type        reserved_2_must_be_zero;

    /*<--------------------------------*/
}    dev_cisc_identify_status_block_type ;
    /*>--------------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*   This type defines the status block returned by a Ciprico
/*   Rimfire 3500 controller in response to the identify command.
/*   See the Rimfire 3500 Product Specification for a complete
/*   description of the members of this structure.
/*

/*.type */

typedef dev_cisc_identify_status_block_type  *

                    /*<-------------------------------------*/
                       dev_cisc_identify_status_block_ptr_type  ;
                    /*>-------------------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*   A pointer to an identify status block.
/*

/*.type */

typedef struct
    {
    uint32e_type                    param_blk_in_index;
    uint32e_type                    param_blk_out_index;
    uint32e_type                    status_blk_in_index;
    uint32e_type                    status_blk_out_index;
    uint32e_type                    param_blk_area_size;
    uint32e_type                    status_blk_area_size;
    uint32e_type                    reserved_1_must_be_zero;
    uint32e_type                    reserved_2_must_be_zero;
    dev_cisc_param_block_type       param_blk[DEV_CISC_MAX_PARAM_BLOCKS];
```

```
    dev_cisc_status_block_type      status_blk[DEV_CISC_MAX_STATUS_BLOCKS];

      /*<-------------------------*/
    }    dev_cisc_command_list_type     ;
      /*>-------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This type defines a command list into which commands descriptors
/*   (parameter blocks) are placed and status blocks retrieved.
/*   The queues of param/status blocks are maintained by in and
/*   out indices.  The param_block_in_index is the first free
/*   entry on the queue.  The param_block_out_index should be
/*   modified on
/*
/*.Members
/*
/*  reserved_n_must_be_zero  --
/*   Reserved by Ciprico, must be zero.  The first six of these
/*   are the high 16 bits of a 32-bit quantity.  Since the
/*   indices for the array will never go above 2**16, the
/*   high-order word is not used.
/*
/*  param_block_in_index  --
/*   The first free entry on the circular list of parameter blocks.
/*   If this value is equal to param_block_out_index, all parameter
/*   blocks are free.  If this value is equal to one less than
/*   param_block_out_index, there are no free parameter blocks.
/*   The controller will not alter this index.
/*
/*  param_block_out_index  --
/*   The first in-use entry on the circular list of parameter blocks.
/*   If this value is equal to param_block_out_index, there are
/*   no free parameter blocks.  This value is NOT to be changed
/*   by the os, only by the controller.
/*
/*  status_block_in_index  --
/*   The first free entry on the circular list of status blocks.
/*   If this value is equal to param_block_out_index, all status
/*   blocks are free.  If this value is equal to one less than
/*   status, there are no free status blocks.
/*   This value is NOT to be changed by the os, only by the controller.
/*
/*  status_block_out_index  --
/*   The first in-use entry on the circular list of status blocks.
/*   If this value is equal to status_block_out_index, there are
/*   no free status blocks.
/*   The controller will not alter this index.
/*
/*

/*.type */

typedef dev_cisc_command_list_type  *

    /*<-----------------------------*/
       dev_cisc_command_list_ptr_type    ;
    /*>-----------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a command list.
/*

/*.type */

typedef struct
{
    uint32e_type                    param_block_id;
    uint16e_type                    reserved_1_must_be_zero;
    uint8e_type                     address_modifier;
    uint8e_type                     target_id;
    uint32e_type                    command_list_ptr;
    uint16e_type                    reserved_2_must_be_zero;
    uint8e_type                     interrupt_level;
```

```
    uint8e_type                     interrupt_vector;
    uint8e_type                     command;
    skip_type                                   reserved_3_must_be_zero : 24;
    uint32e_type                    reserved_4_must_be_zero;
    uint32e_type                                reserved_5_must_be_zero;

    /*<--------------------------------*/
}      dev_cisc_set_up_cmd_list_pb_type      ;
    /*>--------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure defines a Ciprico Rimfire 3500 Start Command List
/*   parameter block. See the Rimfire 3500 Product Specification for
/*   a complete description of the members of this structure.
/*

/*.type */

typedef dev_cisc_set_up_cmd_list_pb_type *

    /*<-------------------------------------*/
       dev_cisc_set_up_cmd_list_pb_ptr_type                ;
    /*>-------------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a start-command-list type.
/*

/*.type */

typedef struct
    {
    uint32e_type                        param_block_id;
    field_type                          reserved_1_must_be_zero : 24;
    uint8e_typ                          target_id;
    uint32e_type                        reserved_2_must_be_zero;
    uint32e_type                        reserved_3_must_be_zero;
    uint8e_type                         command;
    bit8e_type                          test_flags;
    uint16e_type                        reserved_4_must_be_zero;
    uint32e_type                        reserved_5_must_be_zero;
    uint32e_type                        reserved_6_must_be_zero;

      /*<--------------------------------*/
    }    dev_cisc_run_diagnostics_pb_type    ;
      /*>--------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure defines a Ciprico Rimfire 3500 Run Diagnostics
/*   parameter block. See the Rimfire 3500 Product Specification for
/*   a complete description of the members of this structure.
/*
/*

typedef dev_cisc_run_diagnostics_pb_type *

      /*<-------------------------------------*/
         dev_cisc_run_diagnostics_pb_ptr_type    ;
      /*>-------------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a run-diagnostics param block.
/*

/*.type */

typedef struct
    {
    uint32e_type                        param_block_id;
    uint16e_type                    disconnect_timeout;
    uint8e_type                     unit_id;
```

```
    uint8e_type                     target_id;
    uint16e_type                    select_timeout;
    uint8e_type                     retry_control;
    uint8e_type                     retry_limit;
    bit16e_type                     reserved_1;
    uint8e_type                     sense_bytes;
    uint8e_type                     unit_flags;
    uint8e_type                     command;
    field_type                      reserved_2 : 24;
    uint32e_type                    reserved_3;
    uint32e_type                    reserved_4;

      /*<-----------------------------------*/
    }    dev_cisc_set_unit_options_pb_type    ;
      /*>-----------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure defines a Ciprico Rimfire 3500 Set Unit Options
/*   parameter block. See the Rimfire 3500 Product Specification for
/*   a complete description of the members of this structure.
/*
/*

typedef dev_cisc_set_unit_options_pb_type *

      /*<-----------------------------------*/
         dev_cisc_unit_options_pb_ptr_type    ;
      /*>-----------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a Set Unit Options param block.
/*
/*.type */

typedef struct
    {
    uint32e_type                        param_block_id;
    skip_type                           reserved_1_must_be_zero : 5;
    field_type                                  block_mode_flag : 1;
    field_type                                  parity_check_flag : 1;
    field_type                                  allow_disconnect_flag : 1;
    field_type                      throttle_type : 1;
    field_type                          throttle_count : 7;
    uint8e_type                     host_id;
    uint8e_type                     target_id;
    uint32e_type                        reserved_2_must_be_zero;
    uint32e_type                        reserved_3_must_be_zero;
    uint8e_type                         command;
    skip_type                       reserved_4_must_be_zero : 24;
    uint32e_type                        reserved_5_must_be_zero;
    uint32e_type                        reserved_6_must_be_zero;


      /*<-----------------------------------*/
    }    dev_cisc_general_options_pb_type       ;
      /*>-----------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure defines a Ciprico Rimfire 3500 General Options
/*   parameter block. See the Rimfire 3500 Product Specification for
/*   a complete description of the members of this structure.
/*
/*

/*.type */

typedef dev_cisc_general_options_pb_type *

      /*<-----------------------------------*/
         dev_cisc_general_options_pb_ptr_type    ;
      /*>-----------------------------------*/
```

```
/*.Description[=-------------------------------------------------------
/*
/*    A pointer to a general options param block.
/*

/*.type */

typedef struct
    [
    uint32e_type                        param_block_id;
    skip_type                           reserved_1_must_be_zero : 24;
    uint8e_type                         target_id;
    uint32e_type                        reserved_2_must_be_zero;
    uint32e_type                        reserved_3_must_be_zero;
    uint8e_type                         command;
    skip_type                           reserved_4_must_be_zero : 24;
    uint32e_type                        reserved_5_must_be_zero;
    uint32e_type                        reserved_6_must_be_zero;

      /*<-------------------------*/
    ]    dev_cisc_identify_pb_type      ;
      /*>-------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*    This structure defines a Ciprico Rimfire 3500 Identify command
/*    parameter block. See the Rimfire 3500 Product Specification for
/*    a complete description of the members of this structure.
/*
/*

/*.type */

typedef dev_cisc_identify_pb_type *

      /*<-----------------------------*/
         dev_cisc_identify_pb_ptr_type    ;
      /*>-----------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*    A pointer to a general options param block.
/*
/*.type */

typedef struct
    {
    dev_cisc_param_block_type           std_param_block;
    uint16e_type                        reserved_1_must_be_zero;
    uint8e_type                         interrupt_level;
    uint8e_type                         interrupt_vector;
    uint32e_type                        reserved_2_must_be_zero;
    dev_cisc_status_block_type          status_block;

    /*<---------------------*/
}      dev_cisc_type_0_pb_type   ;
    /*>---------------------*/

/*.Description[=-------------------------------------------------------
/*
/*    A type zero parameter block.  Type zero parameter blocks
/*    are used to issue command without the use of the command list
/*    facilities. See the Rimfire 3500 Product Specification for
/*    a complete description of the members of this structure.
/*
/*
/*.Members
/*
/* std_param_block --
/*       A standard parameter block is embedded in a type zero
/*       parameter block.
/*
/* reserved_1_must_be_zero --
/*       Reserved by Ciprico.  Must be zero.
/*
```

```
/* interrupt_level --
/*      The interrupt level to be used when an interrupt is posted
/*      on the command's completion.  If zero, interrupts are
/*      disabled.  Other valid values are 1-7, those being the
/*      permissable VME interrupt levels.
/*
/* interrupt_vector --
/*      The interrupt vector to be returned when an interrupt is
/*      acknowledged on the command's completion.  This should
/*      be the controller's vector number.
/*
/* reserved_2_must_be_zero --
/*      Reserved by Ciprico.  Must be zero.
/*
/* status_block --
/*      A status block is embedded in a type zero
/*      parameter block.
/*

/*.type */

typedef dev_cisc_type_0_pb_type  *

                    /*<-------------------------*/
                       dev_cisc_type_0_pb_ptr_type    ;
                    /*>-------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*      A pointer to a type zero parameter block.
/*
/*.type */

typedef struct
    {
    io_service_interrupt_routine_ptr_type  service_interrupt_routine_ptr;
    io_interleave_lock_type                request_lock;
    misc_queue_header_type                 async_request_queue;
    misc_queue_header_type                 request_blk_queue;
    misc_queue_header_type                 scatter_gather_blk_queue;
    misc_spin_lock_type                    controller_lock;
    dev_ciprico_register_ptr_type      cisc_reg_ptr;
    dev_cisc_command_list_type         cmd_list;
    struct dev_cisc_unit_table_entry_tag * ute_ptr;
    uint8e_type                        interrupt_level;
    io_device_code_type                    device_code;
    io_device_number_type                  device_number;
    bit16e_type                        vme_address_modifier;
    boolean_type                       sync_scsi_supported;

    /*<-------------------------*/
]      dev_cisc_device_info_type   ;
    /*>-------------------------*/

/*.Description[=-------------------------------------------------------
/*
/*   This data structure contains all of the per-device information
/*   for a Ciprico Rimfire 3500 controller that is under the
/*   jurisdiction of the CISC device driver modules.
/*
/*   Dev_cisc_device_info_type must be smaller than a page because
/*   the command_list structure contained within is accessed by the
/*   Ciprico controller. The command_list location is specified
/*   to the controller by a single physical address. A
/*   dev_cisc_device_info_type is allocated with the specification
/*   that it not cross a page boundary.
/*
/*.Members
/*
/* service_interrupt_routine_ptr -- This pointer identifies
/*   the routine that is called by the system interrupt handler to
/*   service interrupts for this controller.  THIS FIELD MUST BE
/*   THE FIRST FIELD IN THE STRUCTURE AS IT IS ASSUMED SUCH BY THE
/*   SYSTEM INTERRUPT HANDLER.
/*
```

```
/* request_lock -- Controls access to the parameter blocks
/*   in the command list. The lock may be held concurrently as many
/*   times as there are entries in the request_blk_queue.
/*
/* async_request_queue -- The queue header for the queue of
/*   asynchronous requests that could not immediately be assigned a
/*   parameter block from the command list. As request blocks are
/*   freed up, requests are removed from this queue and assigned a
/*   request block.
/*
/* request_blk_queue -- This structure is the queue header
/*   for the queue of free request blocks. These request blocks are
/*   assigned to incoming I/O requests and are put back on the
/*   free queue when the request has been completely serviced. The
/*   number of request blocks allocated corresponds directly to the
/*   number of Ciprico parameter blocks allocated in the command list
/*   parameter block array.
/*
/* scatter_gather_blk_queue -- The queue header for the queue
/*   of free scatter gather blocks. Scatter gather blocks are allocated
/*   from the free queue on a per-request basis to specify all
/*   information needed by the controller to perform DMA. Scatter gather
/*   blocks are returned to the free queue after a request has
/*   completed.
/*
/* controller_lock -- This spin lock is obtained to insure
/*   exclusive access to some of the data structures in the
/*   dev_cisc_device_info_type structure. A spin lock must be used
/*   instead of the standard sequenced lock because Ciprico controller
/*   interrupts are cleared by the time the cisc interrupt service
/*   routine is called. The interrupt service routine can't simply
/*   return and assume the interrupt will remain posted if the
/*   controller lock can't be obtained.
/*
/* cisc_reg_ptr -- A pointer to the control space in system
/*   memory which is used to communicate with the Ciprico controller.
/*   The address of the control space is determined by jumpers on
/*   the board. If a control space address specified in the DG/UX
/*   System file for the controller, it must match the address
/*   jumpered on the board.
/*
/* cmd_list -- A Ciprico Rimfire 3500 command list structure.
/*   The command list consists of an array of parameter blocks, an
/*   array of status block, and a set of indices which specifiy the
/*   state of each array. Once controller initialization has been
/*   done, all commands to the controller are issued through this
/*   structure. In addition, the controller reports all request status
/*   information through this structure. Note that this structure must
/*   not cross a page boundry since it is accessed by the controller.
/*   As a result, the device info structure is allocated with the
/*   "no page cross" specification.
/*
/* ute_ptr -- Unit table entry reserved for the
/*   the SCSI interface. Used during controller initialization
/*   so that common support functions can be used.
/*
/* interrupt_level -- The VME interrupt priority for the
/*   Ciprico controller. The interrupt level specifies which bit of the
/*   eight VME bits in the system interrupt status register will be set
/*   when Ciprico controller requests a host interrupt. The interrupt
/*   level is assigned by software. The Ciprico controller is told
/*   what interrupt level to use when it is configured. The interrupt
/*   level assigned to the Ciprico 3500 is DEV_CISC_INTERRUPT_LEVEL.
/*   Interrupt level assignments for DG/UX are based on 88k memo #129,
/*   "VME Device Specifications".
/*
/* device_code -- The VME interrupt vector for the Ciprico
/*   controller. The interrupt vector identifies a particular device
/*   amoung devices that interrupt at the same level. The interrupt
/*   vector of the interrupting device with the lowest vector is
/*   returned when a VME acknowledge is done at a particular level.
/*   The interrupt vector number is used by the system interrupt handler
/*   to index the Device Interrupt Table. The interrupt vector is assigned
/*   by software. The Ciprico controller is told what interrupt vector
/*   to use when it is configured. The interrupt level assigned to the
```

```
/*   Ciprico 3500 is determined by the device specification entered in
/*   the DG/UX System file. Default interrupt vector assignments for
/*   DG/UX are based on 88k memo #129, "VME Device Specifications".
/*
/* device_number -- The major and minor device number
/*   of this Ciprico Rimfire 3500 controller.
/*
/* vme_address_modifier -- The vme address modifier used to
/*   perform dma through the controller. Either Extended Supervisory
/*   Block Transfer or Extended Supervisory Data Access is used.
/*   If the machine architecture supports block mode vme, Extended
/*   Supervisory Block Transfer is used. Otherwise Extended Supervisory
/*   Data Access is used. Uc_get_config_info is called to determine
/*   what the machine supports.
/*
/* sync_scsi_supported -- The controller was determined to
/*   be at a firmware revision level that allows use of synchronous SCSI.
/*
/*.type */

typedef dev_cisc_device_info_type  *

                        /*<-----------------------------*/
                           dev_cisc_device_info_ptr_type    ;
                        /*>-----------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*      A pointer to a cisc device information structure.
/*
/*.type */

typedef struct dev_cisc_unit_table_entry_tag
    {
    boolean_type                    in_use;
    dev_scsi_adapter_unit_spec_type     unit_spec;
    dev_cisc_device_info_ptr_type   dip;
    bit32e_type                         driver_handle;
    bit8e_type                          device_type;
    misc_clock_value_type                       controller_dead_timeout;
    misc_clock_value_type               start_busy_time;
    misc_counter_type                   outstanding_request_count;


        /*<--------------------------*/
    }   dev_cisc_unit_table_entry_type  ;
        /*>--------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*
/*      This structure defines an entry in a cisc unit table. The
/*      unit table is used map between a SCSI device and the SCSI
/*      interface used to access the device.
/*
/*.Members
/*
/* in_use -- Flag, when set indicates that a device is
/*      currently registered at this entry in the unit table.
/*
/* unit_spec -- The SCSI id and unit number of the device
/*      registered at the entry.
/*
/* dip -- A pointer to the device information structure
/*      used to control the cisc interface that is used to
/*      to reference the device.
/*
/* driver_handle -- An opaque handle that the device
/*      driver registers with the SCSI interface. The handle
/*      is returned to the driver by the get_device_info
/*      function.
/*
/* unit_options_blk -- The unit options block which
/*      describes the various device control parameters of the device
/*      registered at this table entry.
```

```
/*
/* device_type -- Specifies the class of device which is
/*      registered at this entry. The device class is obtained from
/*      the inquiry buffer returned from the device during device
/*      configuration.
/*
/* controller_dead_timeout -- Maximum amount of time to
/*   wait for a request to complete before assuming that the
/*   controller has failed. The ciprico controller has its own
/*   internal timeout mechanism. This timeout is used as a backup
/*   in case the controller fails.
/*
/* start_busy_time -- The system time when the disk unit
/*   last went from the idle to the active state. This field is
/*   used to collect system activity data.  See urb memo 052
/*   for a complete description of how system activity data is
/*   collected for intelligent disk controllers.
/*
/* outstanding_request_count -- The number of requests that
/*      have been issued to the disk unit that are still outstanding.
/*      This counter is used to determine when the disk unit toggles
/*      between the idle and busy states.
/*

/*.type */

typedef dev_cisc_unit_table_entry_type  *

        /*<--------------------------------*/
         dev_cisc_unit_table_entry_ptr_type  ;
        /*>--------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a cisc unit table entry.
/*

/*.type */

typedef dev_cisc_unit_table_entry_type

    /*<---------------------*/
     dev_cisc_unit_table_type [DEV_SCSI_MAX_SCSI_IDS][DEV_SCSI_MAX_UNITS];
    /*>---------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*
/*      An cisc unit table. The unit table is indexed by SCSI id and
/*      unit number of a device. It is used to map between a device
/*      and its SCSI interface.
/*
/*
/*.type */

typedef dev_cisc_unit_table_type  *

        /*<-------------------------*/
         dev_cisc_unit_table_ptr_type  ;
        /*>-------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*
/*   A pointer to a cisc unit table.
/*
/*.type */

typedef struct
    {
    misc_queue_links_type                       links;
    dev_adapter_request_block_ptr_type                  arb_ptr;
    dev_cisc_status_block_type                  status_blk_array[2];
    uint8_type                  status_blk_index;
    misc_queue_header_type                      used_scatter_gather_queue;
```

```
    vp_ec_type                          sync_io_ec;
    dev_cisc_device_info_ptr_type               dip;
    dev_cisc_unit_table_entry_ptr_type                  ute_ptr;
    dev_cisc_param_block_type                   param_block;
    uint8_type                              retries_started;
    uint8_type                              retries_acknowledged;
    opaque32_type                       async_timeout_id;
    misc_clock_value_ptr_type                   async_timeout_ptr;
    misc_clock_value_type                   total_request_busy_time;
    boolean_type                        request_aborted;
    boolean_type                        request_timed_out;
    boolean_type                        async_request_handled;
    boolean_type                        sync_io;

        /*<---------------------*/
    }   dev_cisc_request_blk_type  ;
        /*>---------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure is the per I/O request data structure that contains
/*   all of the information necessary to issue a request through the
/*   the Ciprico SCSI interface, over the SCSI bus and maintain the
/*   request through all the steps of execution.
/*
/*   A fixed number of request blocks are allocated to manage devices
/*   on the SCSI bus. The number allocated is based on the maximum number
/*   of simultaneous requests that the interface/device can handle.
/*   When an I/O request is initiated by a SCSI device driver, a request
/*   block is allocated in the CISC manager from the request block free
/*   queue associated with the device's Ciprico controller. When the
/*   request completes, the request block is put back on the free queue.
/*
/*.Members
/*
/* links -- This field is used by the queue manager to maintain
/*   the request block on the various queues it may end up on as it
/*   is being processed.
/*
/* arb_ptr -- A pointer to the generic adapter request block
/*   passed down by the calling SCSI device driver. The adapter request
/*   block specifies the request to be made to a SCSI device.
/*
/* status_blk_array -- Array of Ciprico status blocks returned
/*   upon request completion. At least one request block is returned for
/*   each request. Two are returned if the request completes with a
/*   Check Condition status and sense information is returned.
/*
/* status_blk_index -- Index into the status block array,
/*   indicates the currnet position in the array.
/*
/* used_scatter_gather_queue -- List of scatter gather arrays
/*   used to perform a DMA transfer on the current request. The list
/*   is maintained so that the arrays can be returned to the free
/*   queue upon request completion.
/*
/* sync_io_ec -- Event counter used by the CISC mananger to
/*   synchronize with the Ciprico interface while waiting for an I/O
/*   event.
/*
/* dip -- A pointer to the device information structure of
/*   the Ciprico controller that the request block is being
/*   executed on.
/*
/* ute_ptr -- A pointer to the unit table entry for the
/*   device that is the target of the request.
/*
/* param_block -- Ciprico paramter block that is built to
/*   issue the request specified by this request block.
/*
/* retries_started -- The number of retry interrupts that
/*   were received while the request was being processed.
/*
/* retries_acknowledged -- The number of retries that base
/*   level code woke up and noticed. Base level acknowledges retries
```

```
/*   when it wakes up after a timeout. If a timeout occurs and
/*   retries_started is less than retries_acknowledged, base level
/*   increments retries_acknowledged and goes back to sleep. If
/*   a timeout occurs and retries_started is equal to
/*   retries_acknowledged, base level performs timeout processing for
/*   the request.
/*
/* async_timeout_id -- Variable to hold the timeout ID returned
/*   by the timeout manager. The timeout manager is used to implement
/*   timeouts on asynchronous requests.
/*
/* async_timeout_ptr -- Pointer to a clock value which
/*   specifies the timeout value for an asynchronous request.
/*
/* total_request_busy_time -- The total amount of time that
/*   the target device spent processing the current request
/*   specified by this request block.
/*
/* request_aborted -- Boolean, when TRUE indicates that the
/*   request specified by this request block has been terminated
/*   by a process termination signal.
/*
/* request_timed_out -- Boolean, when TRUE indicates that the
/*   hardware request issued on behalf of this request block has
/*   timed out.
/*
/* async_request_handled -- Used in async timeout processing.
/*   This flag indicates whether the real interrupt has already been
/*   received for the current outstanding hardware request made on behalf
/*   of this request block. The flag is used to prevent a timeout
/*   from queueing a message to the driver demon for a request that has
/*   already gotten a real interrupt.
/*
/* sync_io -- Boolean, when TRUE indicates that the request
/*   specified by this request block is a synchronous I/O request.
/*   When sync_io is FALSE, the request is asynchronous.
/*
/*.type */

typedef dev_cisc_request_blk_type  *

        /*<--------------------------*/
         dev_cisc_request_blk_ptr_type  ;
        /*>--------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a cisc request block.
/*
/*.type */

typedef struct
    {

    misc_queue_links_type                       links;
    dev_ciprico_scatter_gather_header_type      header;

        /*<----------------------------*/
    }   dev_cisc_scatter_gather_blk_type  ;
        /*>----------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This structure defines a scatter gather queue element. It is used
/*   to link Ciprico scatter gather arrays into the various queues
/*   used to manage them.
/*
/*.Members
/*
/* links -- Link field used by the Queue Manager to maintain
/*   the scatter gather block in a queue.
/*
/* header -- Scatter gather array used to specify a DMA
/*   operation.
/*
```

```
/*.type */

typedef dev_cisc_scatter_gather_blk_type  *

        /*<--------------------------------*/
         dev_cisc_scatter_gather_blk_ptr_type  ;
        /*>--------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a scatter gather block.
/*
/*.type */

typedef struct
    {
    dev_ciprico_register_ptr_type   cisc_reg_ptr;
    dev_cisc_type_0_pb_type                    type_0_param_blk;
    bit16e_type                              vme_address_modifier;

    /*<--------------------------------*/
}      dev_cisc_physical_device_info_type   ;
    /*>--------------------------------*/

/*.Description[=-------------------------------------------------------------
/*
/*   This data structure is used to access and control the Ciprico
/*   Rimfire 3500 interface when the system is in shutdown mode.
/*
/*.Members
/*
/* cisc_reg_ptr -- A pointer the start of the cisc control
/*   register structure.
/*
/* type_0_param_blk -- A Ciprico 3500 type 0 parameter block
/*   used to issue requests to devices on the SCSI bus.
/*
/* vme_address_modifier -- The vme address modifier used to
/*   perform dma through the controller. Either Extended Supervisory
/*   Block Transfer or Extended Supervisory Data Access is used.
/*   If the machine architecture supports block mode vme, Extended
/*   Supervisory Block Transfer is used. Otherwise Extended Supervisory
/*   Data Access is used. Uc_get_config_info is called to determine
/*   what the machine supports.
/*
/*.type */

typedef dev_cisc_physical_device_info_type  *

                    /*<--------------------------------------*/
                       dev_cisc_physical_device_info_ptr_type    ;
                    /*>--------------------------------------*/


/*.Description[=-------------------------------------------------------------
/*
/*   A pointer to a cisc physical device information structure.
/*
```

Static Global Data: dev_cisc_global_data.c

```
/*<-------------------------------------------------*/
/*              dev_cisc_global_data.c               */
/*>-------------------------------------------------*/

/*.Contents[=----------------------------------------------------------------
/*
/* dev_cisc_open_lock -- subsystem
/* cfv_cisc_routines_vector -- exported
/* dev_cisc_physical_dip -- subsystem
/*
```

```
/*
/*.Description
/*
/*   This module contains global data for the Ciprico Rimfire 3500
/*   device driver. The Ciprico Rimfire 3500 is a SCSI host bus
/*   adapter.
/*

/*.variable */
                        /*<---------------*/
UNWIRED
lm_unsequenced_lock_type dev_cisc_open_lock  = {0};
                        /*>---------------*/

/*.Description[=-------------------------------------------------------
/*
/*   This locks protects all operations that involve
/*   configuring/deconfiguring, opening/closing, and mapping device
/*   numbers for devices under the jurisdiction of this driver.
/*

/*.variable */
                                        /*<---------------------*/
WIRED
dev_scsi_adapter_routines_vector_type   cfv_cisc_routines_vector  =
                                        /*>---------------------*/
        {
        {1,                     /* Version 1 of this structure */
         0,                     /* Flags -- currently unused */
         io_nodevice_open,
         io_nodevice_close,
         io_nodevice_read_write,
         io_nodevice_select,
         io_nodevice_ioctl,
         io_nodevice_start_io,
         dev_cisc_init,
         dev_cisc_configure,
         io_nodevice_deconfigure,
         dev_cisc_device_to_name,
         dev_cisc_name_to_device,
         dev_cisc_open_dump,
         io_nodevice_write_dump,
         io_nodevice_read_dump,
         io_nodevice_close_dump,
         io_nodevice_powerfail,
             io_nodevice_mmap,
             io_nodevice_munmap,
             io_nodevice_maddmap},
        {1,                     /* Version 1 of this structure */
         0,                     /* Flags -- currently unused */
         dev_cisc_register_requester,
         dev_cisc_set_unit_options,
         dev_cisc_deregister_requester,
         dev_cisc_issue_command,
         dev_cisc_issue_async_command,
         dev_cisc_get_device_info,
         dev_cisc_issue_command_physical_mode}
        };

/*.Description[=-------------------------------------------------------
/*
/*   This variable contains pointers to each of the externally
/*   referencable functions provided by this device driver.  Note
/*   that since this is a SCSI adapter driver, it contains a set
/*   SCSI interface routines in addition to the standard device
/*   driver interfaces that are used by higher levels of the kernel.
/*   The SCSI interface routines are used by SCSI device drivers to
/*   issue requests through this driver to the SCSI bus.
/*

/*.variable */
                    /*<------------------*/
WIRED
uint32_type                 dev_cisc_physical_dip  [60] =
                    /*>------------------*/
```

```
        {
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        };

/*.Description[=-------------------------------------------------------
/*
/*   This array is allocated for global storage for use as the cisc
/*   device information table when the system is in shutdown mode and
/*   the Ciprico Rimfire 3500 is being used to interface to the
/*   system dump destination.
/*
```

Main Driver C Code: dev_cisc_driver.c

```
/*<-------------------------------------------------------*/
/*                    dev_cisc_driver.c                  */
/*>-------------------------------------------------------*/

/*.Contents[=-----------------------------------------------------------
/*
/* dev_cisc_configure -- subsystem
/* dev_cisc_name_to_device -- subsystem
/* dev_cisc_device_to_name -- subsystem
/* dev_cisc_parse_device_name -- internal
/* dev_cisc_init -- subsystem
/* dev_cisc_open_dump -- subsystem
/*
/*.Description
/*
/*   This module is the main part of the device driver for
/*   Ciprico Rimfire 3500 Host SCSI bus adapter used in the
/*   Topgun I/O architecture. The driver interfaces provided
/*   here are used by the SCSI device drivers which require
/*   support from the adapter.
/*
/*   Currently, direct user access to these interfaces is not
/*   supported. However, the interfaces could be easily expanded,
/*   to allow users to open the adapter and issue requests to it.
/*   For example, the user may wish to open the adapter and
/*   issue an ioctl command which returns a list of devices
/*   on the SCSI bus.
/*

/*.function */

                /*<---------------*/
UNWIRED
status_type      dev_cisc_configure  (device_spec_ptr, major_number)
                /*>---------------*/

char_ptr_type                   device_spec_ptr;        /* READ ONLY  */
io_major_device_number_type     major_number;           /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------
/*
/*      This function configures the specified device if it is a
/*      Ciprico SCSI Adapter device.
/*
/*.Parameters
/*
/* device_spec_ptr -- A pointer to the character string name
/*      of the device to be configured.
/*
/* major_number -- The major device number on which the
/*      device is to be configured.
/*
/*.Functional_Description
/*
/*   This function configures the specified device if it is a Ciprico
```

```
/*   SCSI Adapter device. Configuration includes allocation and
/*   initialization of controlling data structures, device registration
/*   in the system device interrupt table, device initializaition.
/*
/*   Dev_cisc_parse_device_name is called to extract the adapter
/*   control register address and interrupt vector from the
/*   name string specified by <device_spec_ptr>. The name string
/*   speicfied by <device_spec_ptr>
/*   is of the form:
/*
/*          cisc@<interrupt vector>(<SCSI adapter address>).
/*
/*.Return_Value
/*
/* OK -- The device was successfully configured.
/*
/*   Return values from dev_cisc_parse_device_name.
/*
/*   Return values from io_register_device_info.
/*   Return values from dev_cisc_reset.
/*

{

status_type             status;
dev_cisc_device_info_ptr_type                   dip;
io_device_code_type                 device_code;
uint32e_type                        adapter_address;
uint16_type             i;
dev_cisc_scatter_gather_blk_ptr_type            scatter_gather_blk_ptr;
dev_cisc_unit_table_ptr_type                    unit_table_ptr;
dev_cisc_unit_table_entry_ptr_type              ute_ptr;
uint16_type             unit;
uint16_type             scsi_id;
dev_cisc_request_blk_ptr_type                   request_blk_ptr;
dev_cisc_request_blk_ptr_type                   request_blk_memory;
bit32e_type             dummy_container;

/*.Implementation[=-----------------------------------------------
/*
/*   First, if the device name is not a cisc device name with the
/*   proper syntax, return an error.  If the device name is OK,
/*   then we get the device code and the adapter address from
/*   the name.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_cisc_parse_device_name(device_spec_ptr,
                                    &adapter_address,
                            &device_code);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*   Allocate and initialize the device info for the new cisc
/*   device.  The device info structure must not cross a page
/*   boundary because it contains the parameter and status
/*   blocks which are used to communicate with the controller.
/*
/*.End]=-------------------------------------------------------------*/

dip = (dev_cisc_device_info_ptr_type)vm_get_wired_memory(
                        usizeof(dev_cisc_device_info_type),
                        VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS);
dip->service_interrupt_routine_ptr = (io_service_interrupt_routine_ptr_type)
                                dev_cisc_service_interrupt;
dip->device_code = device_code;
dip->interrupt_level = DEV_CISC_INTERRUPT_LEVEL;

dip->cisc_reg_ptr = (dev_ciprico_register_ptr_type)adapter_address;
dip->device_number.major = major_number;
io_initialize_interleave_lock(&dip->request_lock,
```

```
                                DEV_CISC_MAX_CONCURRENT_REQUESTS);
misc_initialize_queue(&dip->async_request_queue);
misc_initialize_queue(&dip->scatter_gather_blk_queue);
misc_initialize_queue(&dip->request_blk_queue);
dip->controller_lock = MISC_SPIN_LOCK_INITIAL_VALUE;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Allocate and initialize the scatter/gather arrays to be used
/*   to do DMA and enqueue them at the dip scatter/gather queue.
/*
/*.End]=-----------------------------------------------------------*/

for (i = 0; i < DEV_CISC_MAX_SCATTER_GATHER_BLOCKS; i++)
    {
    scatter_gather_blk_ptr = (dev_cisc_scatter_gather_blk_ptr_type)
              vm_get_wired_memory(usizeof(dev_cisc_scatter_gather_blk_type),
                                  VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS);
    misc_zero_fill((pointer_to_any_type)scatter_gather_blk_ptr,
                   usizeof(dev_cisc_scatter_gather_blk_type));
    (void)misc_enqueue_at_tail(&dip->scatter_gather_blk_queue,
                               &scatter_gather_blk_ptr->links);
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Allocate and initialize the controller request blocks for the
/*   controller and enqueue them at the dip request block queue.
/*
/*.End]=-----------------------------------------------------------*/

request_blk_memory = (dev_cisc_request_blk_ptr_type)
              vm_get_wired_memory(DEV_CISC_MAX_PARAM_BLOCKS*
                                  usizeof(dev_cisc_request_blk_type),
                                  VM_DEFAULT_ALIGNMENT);
request_blk_ptr = request_blk_memory;
for (i = 0; i < DEV_CISC_MAX_PARAM_BLOCKS; i++)
    {
    misc_initialize_queue(&request_blk_ptr->used_scatter_gather_queue);
    vp_initialize_ec(&request_blk_ptr->sync_io_ec);
    request_blk_ptr->dip = dip;
    (void)misc_enqueue_at_tail(&dip->request_blk_queue,
                               &request_blk_ptr->links);
    request_blk_ptr++;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Mask interrupt for the device to initialize the controller.
/*   Now claim the device code for use by this device. If the device
/*   code is already in use, exit via the error path and the configure
/*   operation fails. If the registration succeeds, allocate space
/*   in the timeout and demon message queues. Also enable interrupts
/*   for the device.
/*
/*   At this point we must get the cisc open lock to protect against
/*   a deconfigure operation while we are configuring the device.
/*
/*.End]=-----------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_cisc_open_lock);
io_mask_interrupt_variety( Uc_Level_2_VME_Interrupt );
status = io_register_device_info(device_code, Uc_Vme188_Device_Class,
                         (word_address_type)dip);
if (status != OK)
    {
    status = OK;
    io_unmask_interrupt_variety(Uc_Level_2_VME_Interrupt);
    goto config_failed_release_mem;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Make sure the specified controller address is not already
/*   in use by a previously configured device.
```

```
/*
/*.End]=-------------------------------------------------------------*/

status = io_check_device_spec((opaque_ptr_type)adapter_address, device_code);

if (status != OK)
    {
    io_unmask_interrupt_variety(Uc_Level_2_VME_Interrupt);
    goto config_failed_release_mem;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Perform a lazy_wire read of the adapter board status register.
/*   If this returns IO_ENXIO_DEVICE_NOT_PRESENT, then the adapter
/*   is not present.  We use the lazy_wire evaluation to prevent the
/*   system from hanging due to access of a non-existent address
/*   space.
/*
/*.End]=-------------------------------------------------------------*/

status = io_do_first_long_board_access(
                        &dip->cisc_reg_ptr->controller_status,
                            &dummy_container,
                            FALSE);

if (status != OK)
    {
    goto config_failed_board_not_present;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Reserve space in the I/O demon message queue for the
/*   maximum possible number of simultaneous asynchronous requests.
/*
/*.End]=-------------------------------------------------------------*/

io_specify_max_demon_messages((uint32_type)DEV_CISC_MAX_CONCURRENT_REQUESTS);
vp_specify_max_timeouts((uint32_type)DEV_CISC_MAX_CONCURRENT_REQUESTS);

/*.Implementation_Continued[=----------------------------------------
/*
/*   Unmask device interrupts for the cisc controller.
/*
/*.End]=-------------------------------------------------------------*/

io_unmask_interrupt_variety(Uc_Level_2_VME_Interrupt);

/*.Implementation_Continued[=----------------------------------------
/*
/*   Allocate and initailize an adapter unit table. Assign the
/*   unit table entry at [MAX_SCSI_ID[MAX_UNIT] to the controller.
/*
/*.End]=-------------------------------------------------------------*/

unit_table_ptr = (dev_cisc_unit_table_ptr_type)
                vm_get_wired_memory(usizeof(dev_cisc_unit_table_type),
                                                VM_DEFAULT_ALIGNMENT);
for (scsi_id = 0; scsi_id < DEV_SCSI_MAX_SCSI_IDS; scsi_id++)
    {
    for (unit = 0; unit < DEV_SCSI_MAX_UNITS; unit++)
            {
            ute_ptr = (dev_cisc_unit_table_entry_ptr_type)
                        &((*unit_table_ptr)[scsi_id][unit]);
            ute_ptr->in_use = FALSE;
            ute_ptr->unit_spec.scsi_id = scsi_id;
            ute_ptr->unit_spec.unit = unit;
            ute_ptr->dip = dip;
            ute_ptr->controller_dead_timeout = misc_ten_seconds;
        misc_initialize_counter(&ute_ptr->outstanding_request_count,
                                                (int32e_type)0);
            }
    }
dip->ute_ptr = ute_ptr;
```

```
/*.Implementation_Continued[=----------------------------------------
/*
/*   Allocate a minor device number for the adapter. A pointer to
/*   to the unit table is saved in the device table entry. The minor
/*   device number is used to distinguish between adapters in
/*   a multi-adapter configuration.
/*
/*.End]=-------------------------------------------------------------*/

dip->device_number.major = major_number;
status = io_allocate_device_number(
                            major_number,
                            (bit32e_type)unit_table_ptr,
                            (uint16_type)0,
                            &dip->device_number.minor);
if (status != OK)
    {
    goto config_failed_deregister_device_info;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Get the appropriate vme address modifier to use for dma
/*   transfers from the uc subsystem. Uc_is_feature_present will
/*   indicate whether the system supports block mode. If the
/*   system supports block mode we use the Extended Supervisory
/*   Block Transfer address modifier. Otherwise we use Extended
/*   Supervisory Data Access address modifier.
/*
/*.End]=-------------------------------------------------------------*/

if (uc_is_feature_present(Uc_Feature_VME_Block_Transfer))
    {
    dip->vme_address_modifier = DEV_CIPRICO_VME_ADDR_MOD_SUP_BLK_32;
    }
else
    {
    dip->vme_address_modifier = DEV_CIPRICO_VME_ADDR_MOD_SUP_32;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Do a reset of the ciprico controller. The reset function
/*   will perform all controller initialization required. If the
/*   reset fails, deregister the device and deallocate the device
/*   info structure.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_cisc_reset(dip);
if (status != OK)
    {
    goto config_failed_deallocate_device_number;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   If the SCSI controller doesn't support sync, log a warning.
/*
/*.End]=-------------------------------------------------------------*/
if ( !dip->sync_scsi_supported)
    {
    (void)io_err_log_error((uint32e_type)LOG_WARNING,
            "Firmware in SCSI controller
           (bit32e_type)device_spec_ptr);
    }


lm_release_unsequenced_lock(&dev_cisc_open_lock);
done:
return(status);


config_failed_deallocate_device_number:
io_deallocate_device_number(dip->device_number);
```

```
config_failed_deregister_device_info:
vm_release_wired_memory((pointer_to_any_type)unit_table_ptr,
                                usizeof(dev_cisc_unit_table_type));
config_failed_board_not_present:
io_deregister_device_info(dip->device_code,Uc_Vme188_Device_Class);
config_failed_release_mem:
lm_release_unsequenced_lock(&dev_cisc_open_lock);
misc_dequeue_from_head(&dip->scatter_gather_blk_queue,
                      (misc_queue_links_ptr_type *)&scatter_gather_blk_ptr);
while   ((misc_queue_links_ptr_type)scatter_gather_blk_ptr !=
       MISC_QUEUE_NULL_LINKS_PTR)
    {
    vm_release_wired_memory((pointer_to_any_type)scatter_gather_blk_ptr,
                            usizeof(*scatter_gather_blk_ptr));
    misc_dequeue_from_head(&dip->scatter_gather_blk_queue,
                      (misc_queue_links_ptr_type *)&scatter_gather_blk_ptr);
    }
vm_release_wired_memory((pointer_to_any_type)dip, usizeof(*dip));
vm_release_wired_memory((pointer_to_any_type)request_blk_memory,
                            DEV_CISC_MAX_PARAM_BLOCKS*
                                usizeof(dev_cisc_request_blk_type));
return(status);

]

/*.function */

                    /*<--------------------*/
UNWIRED
status_type dev_cisc_name_to_device  (device_name_ptr, device_number_ptr)
                    /*>--------------------*/

char_ptr_type                       device_name_ptr;/* READ ONLY  */
io_device_number_ptr_type           device_number_ptr;/* WRITE ONLY */

/*.Summary[=-----------------------------------------------------------
/*
/*   This function translates the specified device name into a
/*   device number, if <device_name> names a configured cisc device.
/*
/*.Parameters
/*
/* device_name_ptr -- A pointer to the null-terminated device
/*   name that is to be translated.
/*
/* device_number_ptr -- A pointer to where the corresponding
/*
/*.Functional_Description
/*
/*   See Summary.
/*
/*.Return_Value
/*
/* OK -- The device name was successfully translated.
/*   Return values from dev_cisc_parse_device_name.
/*   Return values from io_get_device_info.
/*

{
status_type             status;
dev_cisc_device_info_ptr_type                   dip;
uint32e_type            adapter_address;
io_device_code_type                 adapter_device_code;

/*.Implementation[=-------------------------------------------------
/*
/*   Parse the device name to see if it belongs to this driver.
/*   If so, get a pointer to the device information structure
/*   for the device.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
status = dev_cisc_parse_device_name(device_name_ptr,
                                    &adapter_address,
```

```
                                &adapter_device_code);
if (status != OK)
    {
    goto done;
    }

lm_obtain_unsequenced_lock(&dev_cisc_open_lock);
status = io_get_device_info(adapter_device_code, Uc_Vme188_Device_Class,
                        dev_cisc_service_interrupt,(word_address_ptr_type)&dip);
if (status == OK)
    {
/*.Implementation_Continued[=---------------------------------------
/*
/*   Return the device number for the SCSI adapter.
/*
/*.End]=-------------------------------------------------------------*/

    *device_number_ptr = dip->device_number;
    }
lm_release_unsequenced_lock(&dev_cisc_open_lock);
done:
return(status);
}

/*.function */

             /*<-----------------------*/
UNWIRED
status_type     dev_cisc_device_to_name   (device_number, name_ptr, size)
             /*>-----------------------*/

io_device_number_type   device_number;  /* READ ONLY  */
char_ptr_type           name_ptr;       /* WRITE ONLY */
uint32_type             size;           /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------
/*
/*      This function returns the character string name associated with
/*      the specified device number.
/*
/*.Parameters
/*
/* device_number -- The device number for which is the
/*      character string name is wanted.
/*
/* name_ptr -- A pointer to where the null-terminated
/*      character string name is to be written.
/*
/* size -- The maximum number of bytes, including the terminating
/*      null, that is to be written to <name_ptr>.
/*
/*.Functional_Description
/*
/*      The given device number is mapped to a device code and
/*      unit number, and a string of the form
/*
/*      cisc@<device_code>(<device registers ptr>)
/*
/*      is returned.
/*
/*.Return_Value
/*
/* OK -- The translation was performed successfully.
/*
/* IO_ENXIO_DEVICE_IS_NOT_CONFIGURED -- The specified
/*      device number is not configured.
/*
/*.Exceptions
/*
/*      None.
/*

{

status_type                     status;
```

```
dev_cisc_unit_table_ptr_type            unit_table_ptr;
dev_cisc_device_info_ptr_type       dip;
uint16_type                         unit;

lm_obtain_unsequenced_lock(&dev_cisc_open_lock);

status = io_map_device_number(device_number,
                              (bit32e_ptr_type)&unit_table_ptr,
                              &unit);

dip = (*unit_table_ptr)[DEV_CISC_SCSI_HOST_ID][DEV_CISC_SCSI_HOST_UNIT].dip;
if (status == OK)
    {
    misc_format_line(name_ptr, size, "cisc@%x(%x)",
                              (bit32e_type)dip->device_code,
                     (bit32e_type)dip->cisc_reg_ptr);
    }
lm_release_unsequenced_lock(&dev_cisc_open_lock);
return(status);

}

/*.function */

               /*<-----------------------*/
WIRED
status_type     dev_cisc_parse_device_name  (device_spec_ptr,
                             adapter_address_ptr,
                             device_code_ptr)
               /*>-----------------------*/

char_ptr_type                       device_spec_ptr; /* READ ONLY   */
uint32_ptr_type                     adapter_address_ptr; /* WRITE ONLY  */
io_device_code_ptr_type             device_code_ptr;/* WRITE ONLY  */

/*.Summary[=-----------------------------------------------------------------
/*
/*   This function parses the specified device spec, determines
/*   whether it is the name of a ciprico scsi controller, and if so,
/*   returns the parsed information about the device.
/*
/*.Parameters
/*
/* device_spec_ptr -- A pointer to the device specification.
/*
/* adatper_address_ptr -- A pointer to where the adapter
/*   control registers starting address is to be returned.
/*
/* device_code_ptr -- A pointer to where the adapter
/*   device code is to be returned. The device code is really the
/*   the interrupt vector of the adapter.
/*
/*.Functional_Description
/*
/*   See Summary.
/*
/*.Return_Value
/*
/* OK -- The device was successfully configured.
/*
/* IO_ENXIO_DEVICE_NOT_RECOGNIZED -- The given device name was
/*   not the name of a Ciprico SCSI controller.
/*
/*.Exceptions
/*
/*   None.
/*

{

status_type                 status;
io_dev_adapt_info_type      adapter_info;
int32_type                  spec_size;
```

```
/*.Implementation[=-----------------------------------------------
/*
/*   Call the generic parse device spec routine to
/*   break the device spec into its components.
/*
/*.End]=-------------------------------------------------------------*/

status = OK;

if ( !io_parse_device_spec(device_spec_ptr, &adapter_info, &spec_size))
    {
    status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*   See if the device mnemonic returned by the parse device
/*   spec routine matches the mnemonic that specifies a
/*   device under the juristiction of this driver. If not,
/*   return the error.
/*
/*.End]=-------------------------------------------------------------*/

if (misc_string_compare((byte_address_type)adapter_info.name,
                        "cisc",
                                (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE) != 0)
    {
    status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*   Get the adapter address from the first parameter returned.
/*   If an actual address was specified, convert it from ascii
/*   to a hex number. If no adapter address was specified, use
/*   the primary default value.
/*
/*.End]=-------------------------------------------------------------*/

if (misc_string_compare((byte_address_type)adapter_info.params[0],
                        "",
                                (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE) == 0)
    {
    *adapter_address_ptr = DEV_CISC_PRIMARY_ADDRESS_DEFAULT;
    }
else if (misc_string_compare((byte_address_type)adapter_info.params[0],
                        "0",
                                (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE) == 0)
    {
    *adapter_address_ptr = DEV_CISC_PRIMARY_ADDRESS_DEFAULT;
    }
else if (misc_string_compare((byte_address_type)adapter_info.params[0],
                        "1",
                                (uint32_type)IO_DEV_ADAPT_MAX_SPEC_SIZE) == 0)
    {
    *adapter_address_ptr = DEV_CISC_SECONDARY_ADDRESS_DEFAULT;
    }
else
    {
    if (io_hex_str_to_int(adapter_info.params[0],
                          adapter_address_ptr))
            {
        status = IO_ENXIO_DEVICE_NAME_NOT_RECOGNIZED;
            goto done;
            }
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*   Get the device code of the adapter.
/*   If no device code is specified, return the default.
/*
/*.End]=-------------------------------------------------------------*/
```

```
if (adapter_info.device_code == IO_INVALID_DEVICE_CODE)
    {
    if (*adapter_address_ptr == DEV_CISC_SECONDARY_ADDRESS_DEFAULT)
        {
        *device_code_ptr = DEV_CISC_SECONDARY_DEVICE_CODE_DEFAULT;
        }
    else
        {
        *device_code_ptr = DEV_CISC_PRIMARY_DEVICE_CODE_DEFAULT;
        }
    }
else
    {
    *device_code_ptr = adapter_info.device_code;
    }

done:
return(status);
}

/*.function */

        /*<---------*/
INITIALIZATION
void     dev_cisc_init  ()
        /*>---------*/

/*.Summary[=-------------------------------------------------------------
/*
/*   Perform preconfiguration initialization at for the cisc driver
/*   at system boot-time.
/*
/*.Parameters
/*
/*      None.
/*
/*.Functional_Description
/*
/*      See summary.
/*
/*.Return_Value
/*
/*      None.
/*

{

lm_initialize_unsequenced_lock(&dev_cisc_open_lock);

}

/*.function */

               /*<---------------*/
WIRED
status_type     dev_cisc_open_dump  (device_name_ptr)
               /*>---------------*/

char_ptr_type                   device_name_ptr;        /* READ ONLY  */

/*.Summary[=-------------------------------------------------------------
/*
/*   This function opens the Ciprico SCSI adapter device for use as
/*   the interface to the destination of a system dump.
/*
/*.Parameters
/*
/* device_name_ptr -- A pointer to the null-terminated
/*      character string identifying the device to be opened as
/*      a dump device.
/*
/*.Functional_Description
/*
/*   This function initializes the Ciprico SCSI interface so
/*   that a system dump can be written through the adapter to a
```

```
/*   device on the SCSI bus. For a detailed description of the
/*   Ciprico initialization, see the routine dev_cisc_configure.
/*
/*.Return_Value
/*
/* OK -- The open completed successfully.
/*
/* IO_STATUS_DUMP_NOT_SUPPORTED -- The device identified
/*      by the device_name string is not a device that is supported
/*      as a dump device by this driver.
/*

{

status_type             status;
dev_cisc_physical_device_info_ptr_type                    dip;
uint32_type             i;
dev_ciprico_register_ptr_type                  adapter_address;
io_device_code_type                adapter_device_code;
bit32e_type             controller_status;
dev_cisc_general_options_pb_ptr_type           general_options_pb_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Parse the specified device name and see if the device is
/*   an Ciprico SCSI adapter.
/*
/*.End]=-----------------------------------------------------------*/

status = dev_cisc_parse_device_name(device_name_ptr,
                                    (uint32_ptr_type)&adapter_address,
                                &adapter_device_code);
if (status != OK)
    {
    status = IO_STATUS_DUMP_NOT_SUPPORTED;
    goto done;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   Initialize the "physical" device information structure.
/*
/*.End]=-----------------------------------------------------------*/

dip = (dev_cisc_physical_device_info_ptr_type)dev_cisc_physical_dip;
dip->cisc_reg_ptr = adapter_address;

/*.Implementation_Continued[=---------------------------------------
/*
/*   Get the appropriate vme address modifier to use for dma
/*   transfers from the uc subsystem. Uc_is_feature_present will
/*   indicate whether the system supports block mode. If the
/*   system supports block mode we use the Extended Supervisory
/*   Block Transfer address modifier. Otherwise we use Extended
/*   Supervisory Data Access address modifier.
/*
/*.End]=-----------------------------------------------------------*/

if (uc_is_feature_present(Uc_Feature_VME_Block_Transfer))
    {
    dip->vme_address_modifier = DEV_CIPRICO_VME_ADDR_MOD_SUP_BLK_32;
    }
else
    {
    dip->vme_address_modifier = DEV_CIPRICO_VME_ADDR_MOD_SUP_32;
    }


/*.Implementation_Continued[=---------------------------------------
/*
/*      Do a reset of the cisc chip, delay for a couple of seconds to
/*      allow the reset to complete, then check the status register
/*      for on-line and ready. The delay is done in pieces since
/*      sc_busy_wait cannot handle very large integers.
/*
```

```
/*.End]=-------------------------------------------------------------*/

dip->cisc_reg_ptr->controller_reset = (bit32e_type)TRUE;
for (i=0; i<1000; i++)
    {
    sc_busy_wait_microseconds((uint32e_type)5000);
    }

controller_status = dip->cisc_reg_ptr->controller_status;
controller_status &= (bit32e_type)DEV_CISC_STATUS_REGISTER_MASK;
if (controller_status != DEV_CISC_STATUS_READY_AFTER_RESET)
    {
    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    goto done;
    }

/*.Implementation_Continued[=----------------------------------
/*
/*   Set up the controller general options. After the general
/*   options have been set up, the controller is ready to
/*   accept commands.
/*
/*.End]=-------------------------------------------------------------*/

general_options_pb_ptr = (dev_cisc_general_options_pb_ptr_type)
                                                &dip->type_0_param_blk;
general_options_pb_ptr->param_block_id = (uint32e_type)dip;
general_options_pb_ptr->block_mode_flag = FALSE;
general_options_pb_ptr->parity_check_flag = FALSE;
general_options_pb_ptr->allow_disconnect_flag = FALSE;
general_options_pb_ptr->throttle_type = 1;
general_options_pb_ptr->throttle_count = 31;
general_options_pb_ptr->host_id = DEV_CISC_SCSI_HOST_ID;
general_options_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
general_options_pb_ptr->command = DEV_CISC_CMD_SET_CONTROLLER_OPTIONS;
general_options_pb_ptr->reserved_1_must_be_zero = 0;
general_options_pb_ptr->reserved_2_must_be_zero = 0;
general_options_pb_ptr->reserved_3_must_be_zero = 0;
general_options_pb_ptr->reserved_4_must_be_zero = 0;
general_options_pb_ptr->reserved_5_must_be_zero = 0;
general_options_pb_ptr->reserved_6_must_be_zero = 0;
status = dev_cisc_send_type_0_cmd_physical_mode(dip);

done:
return(status);
]
```

Adapter Management Code: dev_cisc_mgr.c

```
/*<-------------------------------------------------------*/
/*                    dev_cisc_mgr.c                     */
/*>-------------------------------------------------------*/

/*.Contents[=--------------------------------------------------------------
/*
/* dev_cisc_reset -- subsystem
/* dev_cisc_service_interrupt -- subsystem
/* dev_cisc_start_command_list_request -- subsystem
/* dev_cisc_await_sync_event -- subsystem
/* dev_cisc_lock_controller -- subsystem
/* dev_cisc_unlock_controller -- subsystem
/* dev_cisc_service_async_timeout -- subsystem
/* dev_cisc_send_type_0_cmd_physical_mode -- subsystem
/*
/*.Description
/*
/*   This module manages the Ciprico Rimfire 3500 SCSI Host Bus
/*   Adapter (CISC). The Rimfire 3500 is used to interface the
/*   VME bus on Topgun class computers to a SCSI I/O bus.
/*
```

```
/*   The Rimfire 3500 provides a relatively high level SCSI bus
/*   interface.  The host system issues requests through parameter
/*   blocks which are placed in a command queue. The parameter
/*   blocks contain a device specification, a SCSI command block,
/*   and a data buffer specification. The Rimfire 3500 fetches
/*   requests from the command queue, and upon request
/*   completion enqueues a status block to a status block queue.
/*   A single host interrupt is generated when a command completes
/*   and a status block is ready for interpretation.
/*
/*   Device drivers make SCSI device requests by means of parameter
/*   blocks which are submitted to the SCSI Adapter Manager. The
/*   SCSI Adapter Manager dispatches to one of adapter interface
/*   routines in the module dev_cisc_util. Dev_cisc_util translates
/*   the generic request into a cisc request and submits the
/*   request to this module which is responsible for interactions
/*   with the Ciprico Rimfire 3500 hardware. Functions in this
/*   module make no interpretation of SCSI commands and have no
/*   knowledge of target device types.
/*
/*   For more information on the Topgun SCSI implementation and SCSI
/*   in general, refer to the following documents:
/*
/* Topgun System Board Design Specification (Revision 0.2)
/* Ciprico Rimfire 3500 Product Specification
/* Motorola VMEbus Specification (Revision C.1)
/* ANSI SCSI-1 Specification (X3.131-1986)
/*
/*

/*.function */

            /*<-------------*/
WIRED
status_type  dev_cisc_reset   (dip)
            /*>-------------*/

dev_cisc_device_info_ptr_type       dip;        /* READ/WRITE */

/*.Summary[=-----------------------------------------------------------
/*
/*   Reset and initialize Ciprico Rimfire 3500 SCSI adapter.
/*
/*.Parameters
/*
/* dip -- A pointer to the device information structure of
/*   the cisc SCSI adapter.
/*
/*.Functional_Description
/*
/*   This function does all initailization required to get the
/*   Ciprico Rimfire 3500 SCSI adapter into normal operating mode
/*   during system initialization or after a board reset has
/*   occurred.
/*
/*.Return_Value
/*
/* OK -- The specified adapter was successfully initialized.
/*
/* IO_EIO_PHYSICAL_UNIT_FAILURE -- The adapter initialization
/*   failed.

{

status_type             status;
bit32e_type             controller_status;
vp_event_type                       events[2];
int32_type                          result_index;
dev_cisc_type_0_pb_ptr_type                     type_0_param_blk_ptr;
dev_cisc_set_up_cmd_list_pb_ptr_type            set_up_cmd_list_pb_ptr;
dev_cisc_run_diagnostics_pb_ptr_type            run_diagnostics_pb_ptr;
dev_cisc_request_blk_ptr_type                   request_blk_ptr;
dev_cisc_general_options_pb_ptr_type            general_options_pb_ptr;
dev_cisc_identify_pb_ptr_type                   identify_pb_ptr;
uint32e_type                        physical_address;
```

```
uint16_type             scsi_id;
dev_cisc_identify_status_block_ptr_type         identify_status_blk_ptr;
dev_cisc_unit_options_pb_ptr_type       unit_options_pb_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Set the reset bit in the Rimfire 3500 reset port and wait
/*   for two seconds for the reset to complete. After three
/*   seconds the Rimfire status port should indicate that the
/*   board is ready to accpet commands. If the board isn't ready,
/*   return the error.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
dip->cisc_reg_ptr->controller_reset = (bit32e_type)TRUE;
vp_create_clock_event(&events[0], &misc_three_seconds);
vp_await_ec(events, (int32_type)1, &result_index);
controller_status = dip->cisc_reg_ptr->controller_status;
controller_status &= DEV_CISC_STATUS_REGISTER_MASK;
if (controller_status != DEV_CISC_STATUS_READY_AFTER_RESET)
    {
    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    goto reset_failed;
    }

/*.Implementation_Continued[=---------------------------------------
/*
/*   Do all the setup necessary to use the Ciprico command list
/*   mechanism. We will issue the start command list command
/*   in a type zero parameter block structure to get the command
/*   list stuff working. All subsequent commands to the controller
/*   will be issued through the command list mechanism. Start by
/*      initializing the command list structure.
/*
/*.End]=-----------------------------------------------------------*/

dip->cmd_list.param_blk_in_index = 0;
dip->cmd_list.param_blk_out_index = 0;
dip->cmd_list.status_blk_in_index = 0;
dip->cmd_list.status_blk_out_index = 0;
dip->cmd_list.param_blk_area_size = DEV_CISC_MAX_PARAM_BLOCKS;
dip->cmd_list.status_blk_area_size = DEV_CISC_MAX_STATUS_BLOCKS;
dip->cmd_list.reserved_1_must_be_zero = 0;
dip->cmd_list.reserved_2_must_be_zero = 0;

/*.Implementation_Continued[=---------------------------------------
/*
/*   Allocate and initialize a type zero parameter block which will
/*   be used to issue the start command command. The parameter block
/*   must not cross a page boundry since it is accessed by the
/*   controller.
/*
/*.End]=-----------------------------------------------------------*/

type_0_param_blk_ptr = (dev_cisc_type_0_pb_ptr_type)
            vm_get_wired_memory(usizeof(dev_cisc_type_0_pb_type),
            VM_DEFAULT_ALIGNMENT_NO_PAGE_CROSS);
misc_zero_fill((pointer_to_any_type)type_0_param_blk_ptr,
                  usizeof(dev_cisc_type_0_pb_type));

/*.Implementation_Continued[=---------------------------------------
/*
/*   Fill in the Start Command List parameter block.
/*
/*.End]=-----------------------------------------------------------*/

set_up_cmd_list_pb_ptr = (dev_cisc_set_up_cmd_list_pb_ptr_type)
                                    &type_0_param_blk_ptr->std_param_block;
set_up_cmd_list_pb_ptr->address_modifier = dip->vme_address_modifier;
set_up_cmd_list_pb_ptr->command = DEV_CISC_CMD_START_LIST;
set_up_cmd_list_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
vm_get_physical_address(
            (word_address_type)&dip->cmd_list,
                FALSE,
```

```
            (word_address_ptr_type)&set_up_cmd_list_pb_ptr->command_list_ptr);
set_up_cmd_list_pb_ptr->interrupt_level = dip->interrupt_level;
set_up_cmd_list_pb_ptr->interrupt_vector = dip->device_code;

/*.Implementation_Continued[=----------------------------------------
/*
/*   Tell the controller about the location of the parameter block
/*   by loading the address buffer port with data transfer control
/*   information and the physical address of the parameter block.
/*   The address buffer port consists of three 16 bit words. Three
/*   consecutive writes to the base address of the address buffer
/*   port allows each word to be loaded with data.
/*
/*.End]=-------------------------------------------------------------*/

dip->cisc_reg_ptr->address_buffer =
                        DEV_CIPRICO_ADDRESS_BUFFER_SET_CONTROL_BITS |
                            dip->vme_address_modifier;

vm_get_physical_address(
            (word_address_type)type_0_param_blk_ptr,
                FALSE,
                (word_address_ptr_type)&physical_address);
dip->cisc_reg_ptr->address_buffer = (bit32e_type)physical_address >> 16;
dip->cisc_reg_ptr->address_buffer = (bit32e_type)physical_address & 0xFFFF;

/*.Implementation_Continued[=----------------------------------------
/*
/*   Start command execution by writing a zero to the channel
/*   attention register. Zero indicates that this is a type
/*   zero command. Give the command one second to complete. If
/*   it doesn't complete in one second or completes with an
/*   error, return the status.
/*
/*.End]=-------------------------------------------------------------*/

type_0_param_blk_ptr->interrupt_level = 0;
type_0_param_blk_ptr->interrupt_vector = 0;
dip->cisc_reg_ptr->channel_attention = 0;
vp_create_clock_event(&events[0], &misc_two_seconds);
vp_await_ec(events, (int32_type)1, &result_index);
if   ((!(type_0_param_blk_ptr->status_block.flags &
    DEV_CIPRICO_COMMAND_COMPLETE_STATUS_FLAG)) ||
    (type_0_param_blk_ptr->status_block.flags & DEV_CIPRICO_ERROR_STATUS_FLAG))
    {
    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    }
vm_release_wired_memory((pointer_to_any_type)type_0_param_blk_ptr,
                        usizeof(dev_cisc_type_0_pb_type));
if (status != OK)
    {
    goto reset_failed;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Dequeue a request block and set up a command to run the
/*   controller diagnostics. From this point on, all commands will
/*   be issued through the command list mechanism. If the diagnostics
/*   fail, return the error.
/*
/*.End]=-------------------------------------------------------------*/

misc_dequeue_from_head(&dip->request_blk_queue,
                       (misc_queue_links_ptr_type *)&request_blk_ptr);
request_blk_ptr->sync_io = TRUE;
request_blk_ptr->ute_ptr = dip->ute_ptr;
run_diagnostics_pb_ptr = (dev_cisc_run_diagnostics_pb_ptr_type)
                                        &request_blk_ptr->param_block;

run_diagnostics_pb_ptr->param_block_id = (uint32e_type)request_blk_ptr;
run_diagnostics_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
run_diagnostics_pb_ptr->command = DEV_CISC_CMD_SELF_TEST;
run_diagnostics_pb_ptr->test_flags =
            (DEV_CISC_SELF_TEST_STATIC_RAM_TEST & DEV_CISC_SELF_TEST_PROM_TEST);
```

```
run_diagnostics_pb_ptr->reserved_1_must_be_zero = 0;
run_diagnostics_pb_ptr->reserved_2_must_be_zero = 0;
run_diagnostics_pb_ptr->reserved_3_must_be_zero = 0;
run_diagnostics_pb_ptr->reserved_4_must_be_zero = 0;
run_diagnostics_pb_ptr->reserved_5_must_be_zero = 0;
run_diagnostics_pb_ptr->reserved_6_must_be_zero = 0;

events[0].name = &request_blk_ptr->sync_io_ec;
vp_get_next_ec_value(&request_blk_ptr->sync_io_ec, &events[0].value);
dev_cisc_start_command_list_request(request_blk_ptr);
dev_cisc_await_sync_event(request_blk_ptr,
                          &events[0],
                           &misc_ten_seconds);
if   ((request_blk_ptr->request_aborted) ||
    (request_blk_ptr->request_timed_out) ||
    (request_blk_ptr->status_blk_array[0].flags &
     DEV_CIPRICO_ERROR_STATUS_FLAG))
    {
    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    goto reset_failed_release_parameter_blk;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Initialize the controller options. See the Ciprico Rimfire
/*   3500 Product Specification manual for a complete description
/*   of the controller options available. We will use the same
/*   request block allocated to run the diagnostics.
/*
/*.End]=-----------------------------------------------------------*/

general_options_pb_ptr = (dev_cisc_general_options_pb_ptr_type)
                                    &request_blk_ptr->param_block;
general_options_pb_ptr->param_block_id = (uint32e_type)request_blk_ptr;
if (dip->vme_address_modifier == DEV_CIPRICO_VME_ADDR_MOD_SUP_BLK_32)
    {
    general_options_pb_ptr->block_mode_flag = TRUE;
    }
else
    {
    general_options_pb_ptr->block_mode_flag = FALSE;
    }
general_options_pb_ptr->parity_check_flag = FALSE;
general_options_pb_ptr->allow_disconnect_flag = TRUE;
general_options_pb_ptr->throttle_type = 1;
general_options_pb_ptr->throttle_count = 31;
general_options_pb_ptr->host_id = DEV_CISC_SCSI_HOST_ID;
general_options_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
general_options_pb_ptr->command = DEV_CISC_CMD_SET_CONTROLLER_OPTIONS;
general_options_pb_ptr->reserved_1_must_be_zero = 0;
general_options_pb_ptr->reserved_2_must_be_zero = 0;
general_options_pb_ptr->reserved_3_must_be_zero = 0;
general_options_pb_ptr->reserved_4_must_be_zero = 0;
general_options_pb_ptr->reserved_5_must_be_zero = 0;
general_options_pb_ptr->reserved_6_must_be_zero = 0;

events[0].name = &request_blk_ptr->sync_io_ec;
request_blk_ptr->sync_io = TRUE;
request_blk_ptr->ute_ptr = dip->ute_ptr;
vp_get_next_ec_value(&request_blk_ptr->sync_io_ec, &events[0].value);
dev_cisc_start_command_list_request(request_blk_ptr);
dev_cisc_await_sync_event(request_blk_ptr,
                          &events[0],
                          &misc_ten_seconds);
if (request_blk_ptr->request_aborted ||
    request_blk_ptr->request_timed_out ||
    (request_blk_ptr->status_blk_array[0].flags &
     DEV_CIPRICO_ERROR_STATUS_FLAG))
    {
    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    goto reset_failed_release_parameter_blk;
    }

/*.Implementation_Continued[=--------------------------------------
/*
```

```
/*   Get the firmware revision level.  Use the level value to
/*   determine if SCSI synchronous data transfers are supported.
/*
/*.End]=-------------------------------------------------------------*/

identify_pb_ptr = (dev_cisc_identify_pb_ptr_type) &request_blk_ptr->param_block;
identify_pb_ptr->param_block_id = (uint32e_type)request_blk_ptr;
identify_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
identify_pb_ptr->command = DEV_CISC_CMD_IDENTIFY;
identify_pb_ptr->reserved_1_must_be_zero = 0;
identify_pb_ptr->reserved_2_must_be_zero = 0;
identify_pb_ptr->reserved_3_must_be_zero = 0;
identify_pb_ptr->reserved_4_must_be_zero = 0;
identify_pb_ptr->reserved_5_must_be_zero = 0;
identify_pb_ptr->reserved_6_must_be_zero = 0;

events[0].name = &request_blk_ptr->sync_io_ec;
request_blk_ptr->sync_io = TRUE;
request_blk_ptr->ute_ptr = dip->ute_ptr;
vp_get_next_ec_value(&request_blk_ptr->sync_io_ec, &events[0].value);
dev_cisc_start_command_list_request(request_blk_ptr);
dev_cisc_await_sync_event(request_blk_ptr,
                          &events[0],
                          &misc_ten_seconds);

identify_status_blk_ptr = (dev_cisc_identify_status_block_ptr_type)
    request_blk_ptr->status_blk_array;

if (request_blk_ptr->request_aborted ||
    request_blk_ptr->request_timed_out ||
    (identify_status_blk_ptr->flags & DEV_CIPRICO_ERROR_STATUS_FLAG))
    {
    status = IO_EIO_PHYSICAL_UNIT_FAILURE;
    goto reset_failed_release_parameter_blk;
    }

if (identify_status_blk_ptr->firmware_revision >=
            DEV_CISC_SYNC_SUPPORT_FIRMWARE_REVISION)
    {
    dip->sync_scsi_supported = TRUE;
    }
else
    {
    dip->sync_scsi_supported = FALSE;
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Set the unit options for each possible unit on the SCSI bus
/*   to some initial values. Note that the options are set on a
/*   per SCSI ID basis, i.e. options for each unit at a
/*   particular SCSI ID cannot be set individually.
/*
/*.End]=-------------------------------------------------------------*/

unit_options_pb_ptr = (dev_cisc_unit_options_pb_ptr_type)
                          &request_blk_ptr->param_block;
unit_options_pb_ptr->param_block_id = (uint32e_type)request_blk_ptr;
unit_options_pb_ptr->disconnect_timeout =
                         DEV_CISC_UNIT_OPTIONS_DEFAULT_DISCON_TIMEOUT;
unit_options_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
unit_options_pb_ptr->select_timeout = DEV_CISC_UNIT_OPTIONS_SELECT_TIMEOUT;
unit_options_pb_ptr->retry_control = 0;
unit_options_pb_ptr->retry_limit = 0;
unit_options_pb_ptr->reserved_1 = 0;
unit_options_pb_ptr->sense_bytes = DEV_CISC_REQUEST_SENSE_DATA_SIZE;
if (dip->sync_scsi_supported)
    {
    unit_options_pb_ptr->unit_flags = DEV_CISC_UNIT_FLAGS_SYN;
    }
else
    {
    unit_options_pb_ptr->unit_flags = 0;
    }
unit_options_pb_ptr->command = DEV_CISC_CMD_SET_UNIT_OPTIONS;
```

```
unit_options_pb_ptr->reserved_2 = 0;
unit_options_pb_ptr->reserved_3 = 0;
unit_options_pb_ptr->reserved_4 = 0;
events[0].name = &request_blk_ptr->sync_io_ec;
request_blk_ptr->sync_io = TRUE;
request_blk_ptr->ute_ptr = dip->ute_ptr;
for (scsi_id = 0; scsi_id < DEV_SCSI_MAX_SCSI_IDS; scsi_id++)
    {
    if (scsi_id == DEV_CISC_SCSI_HOST_ID)
            {
            continue;
            }
    unit_options_pb_ptr->unit_id = scsi_id;
    vp_get_next_ec_value(&request_blk_ptr->sync_io_ec, &events[0].value);
    dev_cisc_start_command_list_request(request_blk_ptr);
    dev_cisc_await_sync_event(request_blk_ptr,
                              &events[0],
                              &misc_ten_seconds);
    if (request_blk_ptr->request_aborted ||
        request_blk_ptr->request_timed_out ||
        (identify_status_blk_ptr->flags & DEV_CIPRICO_ERROR_STATUS_FLAG))
        {
        status = IO_EIO_PHYSICAL_UNIT_FAILURE;
            break;
        }
    }

reset_failed_release_parameter_blk:
misc_enqueue_at_tail(&dip->request_blk_queue, &request_blk_ptr->links);
reset_failed:
return(status);
}

/*.function */

            /*<-----------------------*/
WIRED
void         dev_cisc_service_interrupt  (dip)
            /*>-----------------------*/

dev_cisc_device_info_ptr_type       dip;        /* READ/WRITE */

/*.Summary[=-------------------------------------------------------------
/*
/*   This function services interrupts issued by a Ciprico Rimfire
/*   3500 SCSI adapter. It is called from the system interrupt
/*   handler.
/*
/*.Parameters
/*
/* dip -- A pointer to the device information structure of
/*   the interrupting cisc SCSI adapter.
/*
/*.Functional_Description
/*
/*   Each pending status block is extracted from the command list and
/*   the status information is evaluated.
/*   status information is copied into the original request block
/*   (the status block id is the request block pointer).
/*   If the request was issued synchronously, the awaiting process is
/*   awakened via the ec in the request block. Asynchronous requests
/*   are completed through the I/O demon. After the status block has
/*   been saved, the out pointer for the status list updated.
/*
/*   Note that the interrupt is cleared by the vme acknowledge before
/*   this routine is called. As a result, a spin lock must be used to
/*   lock the data structures which manage the controller.
/*
/*.Return_Value
/*
/*   None.
/*

{
dev_cisc_status_block_ptr_type                  status_blk_ptr;
```

```
dev_cisc_request_blk_ptr_type                    request_blk_ptr;
dev_cisc_unit_table_entry_ptr_type  ute_ptr;
misc_clock_value_type               cur_time;
misc_clock_value_type               current_time;
bit8e_type              status_flags;

/*.Implementation[=-------------------------------------------------
/*
/*   Get the controller lock so that the status block section of
/*   the command list can be accessed exclusively.
/*
/*.End]=-----------------------------------------------------------*/

dev_cisc_lock_controller(dip, FALSE);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Remove all pending status blocks from the queue and examine
/*   the request status. Depending on how the unit options were
/*   set and whether any errors occurred during command execution,
/*   one or more status blocks are returned for each request. Status
/*   blocks are returned for each retry and when the command completes.
/*   Sense information is returned in the status block when a retry
/*   interrupt occurs and when a command completes with an error.
/*   Eight bytes of sense information can be returned in each
/*   status block. Multiple status blocks are returned if more
/*   than eight bytes of sense information were requested. This
/*   driver supports a maximum of sixteen bytes of sense data (two
/*   status blocks). If more than two status blocks are generated for
/*   a command, only the last two are saved because they indicate
/*   the actual completion status of the command.
/*
/*   If the status block indicates that this is a retry interrupt,
/*   the retry count in the request block is incremented. Base level
/*   is not directly notified of retries. However if base level
/*   times out, it looks at the retry count to determine if it
/*   should wait some more. Status blocks resulting from command
/*   errors that are going to be retried are not saved.
/*
/*   Base level is directly notified of an I/O event only when
/*   the status block indicates that the command has completed. If
/*   the command has completed but the request specified by the
/*   status block has been aborted by a process termination signal,
/*   the request block is returned to the free queue with no base
/*   level notification.
/*
/*   The action taken with each status block depends on the state of
/*   Command Complete, Continued, Retry, and Error bits in the flags
/*   field of the status block. The Continued bit is set only when
/*   a retry is reported and more than eight bytes of sense data
/*   are returned. The Continued bit is on in the second status
/*   block of the pair used to report the retry. No matter how many
/*   status blocks are generated for a particular command, the
/*   Command Complete flag will only be set in the very last status
/*   block. See the Ciprico Rimfire 3500 Product Specification for
/*   a general description of the meaning and interpretation of
/*   status block flags. A couple of observed examples are included
/*   here for clarification:
/*
/* With the unit options set so that 3 retries would be
/*   be performed and 16 bytes of sense information returned,
/*   a hard error resulted in 3 interrupts being generated with
/*   a total of 8 status blocks. The flags field of the 8
/*   respective status blocks contained the following: 0x60,
/*   0x64, 0x60, 0x64, 0x60, 0x64, 0x40, 0xC0. The error field
/*   of each of these status blocks was 0x23.
/* With the unit options set so that 3 retries would be
/*   be performed and 8 bytes of sense information returned,
/*   a hard error resulted in 3 interrupts being generated with
/*   a total of 4 status blocks. The flags field of the 4
/*   respective status blocks contained the following: 0x60,
/*   0x60, 0x60, 0x60, 0xC0. The error field of each of these
/*   status blocks was 0x23.
/*
/*
```

```
/*
/*.End]=-------------------------------------------------------------*/

while (dip->cmd_list.status_blk_in_index != dip->cmd_list.status_blk_out_index)
    {
    status_blk_ptr = &dip->cmd_list.status_blk
                                    [dip->cmd_list.status_blk_out_index];
    dev_cisc_physical_dip[0] = (uint32_type)status_blk_ptr;
    request_blk_ptr = (dev_cisc_request_blk_ptr_type)status_blk_ptr->command_id;
    status_flags = status_blk_ptr->flags;
    if (status_flags & DEV_CIPRICO_COMMAND_COMPLETE_STATUS_FLAG)
        {
/*.Implementation_Continued[=----------------------------------------
/*
/*  The command complete bit is set, perform completion processing.
/*
/*.End]=-------------------------------------------------------------*/

            request_blk_ptr->status_blk_array[request_blk_ptr->status_blk_index++]
                        = *status_blk_ptr;

/*.Implementation_Continued[=----------------------------------------
/*
/*   Record the total amount of time that the target device spent
/*   processing the request. Note that the start time is not
/*   necessarily the starting time of this particular request but
/*   is the time when the disk went from idle to active. This
/*   does not really matter since all the timings will even
/*   out as requests complete. Further, system activity
/*   evaluation is not done at a per-request granularity.
/*
/*.End]=-------------------------------------------------------------*/

        vp_read_system_clock(&current_time);
            cur_time = current_time;
        ute_ptr = request_blk_ptr->ute_ptr;
        misc_decrement(&ute_ptr->outstanding_request_count);
        MISC_CLOCK_VALUE_SUBTRACT(&ute_ptr->start_busy_time, &cur_time);
        MISC_CLOCK_VALUE_ASSIGN(&cur_time,
                        &request_blk_ptr->total_request_busy_time);
        MISC_CLOCK_VALUE_ASSIGN(&current_time, &ute_ptr->start_busy_time);

/*.Implementation_Continued[=----------------------------------------
/*
/*  If Command Complete is set but the request has been aborted,
/*  enqueue a request to the I/O demon to release all resources
/*  associated with the request.  Note that a request can't be
/*  aborted if DMA is being done, so there are no scatter gather
/*  arrays to be returned to the free queue.
/*
/*.End]=-------------------------------------------------------------*/

        if (request_blk_ptr->request_aborted)
                {
            (void)io_queue_message_to_driver_demon(
                                dev_cisc_complete_aborted_request,
                                (bit32e_type)request_blk_ptr,
                                TRUE);
                }

/*.Implementation_Continued[=----------------------------------------
/*
/* The command has completed and base level is still interested, do
/*  the notification.
/*
/*.End]=-------------------------------------------------------------*/

        else
                {
            if (request_blk_ptr->sync_io)
                {
                vp_advance_ec(&request_blk_ptr->sync_io_ec);
                }
            else
                {
```

```
                (void)io_queue_message_to_driver_demon(
                                dev_cisc_complete_io,
                                (bit32e_type)request_blk_ptr,
                                TRUE);
                }
            }
        }

    else
        {
/*.Implementation_Continued[=--------------------------------------
/*
/*  The Command Complete flag is not set. Either a retry is being
/*  performed or this is the first status block of a pair reporting
/*  command complete with an error. In either case, set the status
/*  block index to zero and save the status information. Status
/*  information previously received for the command is over-written.
/*  However, the over-written information is no longer relevant.
/*
/*  If the Retry flag is on and the Continued Status block flag is
/*  off, advance the retry count so that the timeout path will
/*  take the retry into account. A retry status block with
/*  Continued flag on is the second status block of a pair sent
/*  for retry notification. Hence the retry count is incremented
/*  only when the first of the pair is received.
/*
/*.End]=-----------------------------------------------------------*/

            request_blk_ptr->status_blk_index = 0;
            request_blk_ptr->status_blk_array[request_blk_ptr->status_blk_index++]
                        = *status_blk_ptr;
        if   ((status_flags & DEV_CIPRICO_RETRY_REQUIRED_STATUS_FLAG) &&
           !(status_flags & DEV_CIPRICO_CONTINUED_BLOCK_STATUS_FLAG))
                {
                request_blk_ptr->retries_started++;
                }
        }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Update the index into the status block queue. If the index exceeds
/*  the last element in the list, reset it to the first element.
/*
/*.End]=-----------------------------------------------------------*/

    dip->cmd_list.status_blk_out_index++;
    if (dip->cmd_list.status_blk_out_index >=
                                dip->cmd_list.status_blk_area_size)
        {
        dip->cmd_list.status_blk_out_index %=
                                dip->cmd_list.status_blk_area_size;
        }
    }

dev_cisc_unlock_controller(dip, FALSE);
return;
}

/*.function */

/*<--------------------------------*/
WIRED
void     dev_cisc_start_command_list_request     (request_blk_ptr)
/*>--------------------------------*/

dev_cisc_request_blk_ptr_type                    request_blk_ptr;  /* READ/WRITE */

/*.Summary[=-------------------------------------------------------------
/*
/*   Start a command list request.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that
/*   specifies a Ciprico Rimfire 3500 request.
```

```
/*
/*.Functional_Description
/*
/*   This function notifies the controller that a parameter block
/*   in the command list structure is ready for execution.
/*
/*.Return_Value
/*
/*   None.
/*

{
uint16e_type            param_blk_in_index;
dev_cisc_param_block_ptr_type                   param_blk_ptr;
dev_cisc_device_info_ptr_type                   dip;
dev_cisc_unit_table_entry_ptr_type  ute_ptr;


/*.Implementation[=-------------------------------------------------
/*
/*   Initialize the request block retry counters and the various
/*   request status flags.
/*
/*.End]=-----------------------------------------------------------*/

dip = request_blk_ptr->dip;
request_blk_ptr->status_blk_index = 0;
request_blk_ptr->retries_started = 0;
request_blk_ptr->retries_acknowledged = 0;
request_blk_ptr->request_aborted = FALSE;
request_blk_ptr->async_request_handled = FALSE;
request_blk_ptr->request_timed_out = FALSE;
ute_ptr = request_blk_ptr->ute_ptr;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Get the controller lock and allocate a parameter block from
/*   the command list. Copy the Ciprico parameter block from the
/*   request block to the parameter block allocated from the command
/*   list. The request interleave lock that had to be obtained to get
/*   to this function guarantees us that at least one command list
/*   parameter block is available.
/*
/*   Update the parameter block in index so that the controller
/*   will know where to start looking for the parameter block.
/*   Set the controller channel attention flag to notify the
/*   controller of the parameter block.
/*
/*   If the target unit is currently idle, the current time is
/*   saved as the start_busy_time.
/*
/*.End]=-----------------------------------------------------------*/

dev_cisc_lock_controller(dip, TRUE);
param_blk_ptr = &dip->cmd_list.param_blk[dip->cmd_list.param_blk_in_index];
*param_blk_ptr = request_blk_ptr->param_block;
param_blk_in_index = dip->cmd_list.param_blk_in_index + 1;
if (param_blk_in_index >= dip->cmd_list.param_blk_area_size)
    {
    param_blk_in_index %= dip->cmd_list.param_blk_area_size;
    }
dip->cmd_list.param_blk_in_index = param_blk_in_index;

if (misc_is_zero_and_increment(&ute_ptr->outstanding_request_count))
    {
    vp_read_system_clock(&ute_ptr->start_busy_time);
    }
dip->cisc_reg_ptr->channel_attention = 1;
dev_cisc_unlock_controller(request_blk_ptr->dip, TRUE);
return;
}

/*.function */

                /*<----------------------*/
```

```
WIRED
void          dev_cisc_await_sync_event  (request_blk_ptr,
                        completion_event_ptr,
                        timeout_ptr)
                  /*>---------------------*/

dev_cisc_request_blk_ptr_type         request_blk_ptr;      /* READ ONLY */
vp_event_ptr_type                     completion_event_ptr; /* READ ONLY */
misc_clock_value_ptr_type             timeout_ptr;                /* READ ONLY */

/*.Summary[=-------------------------------------------------------------
/*
/*   Wait for command complete, timeout, or termination signal.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to the cisc request block
/*   which is being used to execute the command.
/*
/* completion_event_ptr -- A pointer to an event structure which
/*   specifies when the command will be considered complete.
/*
/* timeout_ptr -- A pointer to a clock value which specifies
/*   the amount of the command has to complete before it is
/*   considered "timed-out".
/*
/*.Functional_Description
/*
/*   This function pends and waits until an operation completes,
/*   a timeout occurs, or the operation is interrupted by a
/*   signal. If a timeout occurs and no controller retries have
/*   been done, the cisc adpater is reset and a timeout error is
/*   recorded. If the operation is interrupted by a terminated signal,
/*   the request block is marked as aborted and the error flag is
/*   set in the request block.
/*
/*.Assumptions
/*
/*   This function assumes that the operation associated with the
/*   event has been started and that the caller's event structure
/*   contains a current ec value which will be realized if the
/*   operation completes.
/*
/*   This function also assumes that signal delivery has been disabled
/*   by the caller if it is not appropriate for the operation to
/*   be terminated by a signal.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Remarks
/*
/*   The Ciprico 3500 manages timeouts internally. The timeout
/*   processing done here is used as a backup in case the
/*   controller fails. If the timeout we set up here expires,
/*   the controller is considered dead and it is reset to
/*   insure that further communication with the host is not
/*   attempted. After the reset, the SCSI devices under the
/*   jurisdiction of the controller are inaccessible to the host.
/*   This method of recovery is rather severe but there no
/*   graceful way to recover if we timeout waiting on the controller.
/*   The Ciprico interface does not provide a way to check a
/*   command's execution status. Further, there is no way to abort
/*   an individual command once it has been entered into the
/*   active command list.
/*

{
vp_event_type                         events[3];
int32_type              result_index;

/*.Implementation[=-----------------------------------------------
/*
/*   Set up the timeout event, then pend and wait for one of the
```

```
/*   three events to occur.
/*
/*   If the command is terminated or times-out, the controller
/*   lock is held while the appropriate action is taken. The
/*   controller lock insures that the command complete interrupt
/*   won't come in while we are processing the error.
/*
/*.End]=-------------------------------------------------------------*/

events[0] = *completion_event_ptr;
do
    {
    vp_create_clock_event(&events[1], timeout_ptr);
    do
        {
        if (pm_is_terminated(&events[2]))
            {
            dev_cisc_lock_controller(request_blk_ptr->dip, TRUE);
                if (!vp_has_event_occurred(&events[0]))
                    {
                    request_blk_ptr->request_aborted = TRUE;
                    }
            dev_cisc_unlock_controller(request_blk_ptr->dip, TRUE);
                goto done;
            }
        vp_await_ec(events, (int32_type)3, &result_index);
        }
    while   ((result_index == 2) && !vp_has_event_occurred(&events[0]) &&
                        !vp_has_event_occurred(&events[1]));

    if   ((result_index == 1) && !vp_has_event_occurred(&events[0]))
        {
        dev_cisc_lock_controller(request_blk_ptr->dip, TRUE);
        if (!vp_has_event_occurred(&events[0]))
                {
/*.Implementation_Continued[=----------------------------------
/*
/*   The command has timed-out. If a controller retry is being
/*   performed, loop around and wait some more. Otherwise, reset
/*   the adapter and return the error.
/*
/*.End]=-------------------------------------------------------------*/

                if (request_blk_ptr->retries_started !=
                        request_blk_ptr->retries_acknowledged)
                    {
                    request_blk_ptr->retries_acknowledged++;
                        }
                else
                        {
                request_blk_ptr->dip->cisc_reg_ptr->controller_reset =
                            (bit32e_type)TRUE;
                    request_blk_ptr->request_timed_out = TRUE;
                    }
                }
        dev_cisc_unlock_controller(request_blk_ptr->dip, TRUE);
        }
    else  /* The completion event was satisfied */
        {
            break;
            }
    }
while (!request_blk_ptr->request_timed_out);

done:
return;
}

/*.function */

      /*<--------------------*/
WIRED
void    dev_cisc_lock_controller  (dip, base_level)
      /*>--------------------*/
```

```
dev_cisc_device_info_ptr_type       dip;        /* READ ONLY */
boolean_type            base_level;     /* READ ONLY */

/*.Summary[=-----------------------------------------------------------
/*
/*      This function locks the cisc data structures for single threaded
/*      access.
/*
/*.Parameters
/*
/* dip -- A pointer to a device information structure
/*   associated with the cisc SCSI adapter to be locked.
/*
/* base_level -- Boolean variable, when true, indicates
/*   that this function was called from base level.
/*
/*.Functional_Description
/*
/*   The controller lock is obtained to insure exclusive access
/*   to the data structures used to control the cisc adapter.
/*   Since the controller lock is a spin lock, cisc adapter
/*   interrupts are disabled while the lock is held to reduce
/*   the possibility that the cisc will interrupt and busy-wait
/*   for the lock.
/*
/*   If the request to lock the controller is coming from a base
/*   level code path, interrupts are disabled on the current
/*   processor. Interrupt are disabled to insure that the
/*   executing process is not descheduled while holding the spin lock
/*
/*   Note that it is possible for an interrupt to come in before
/*   we have a chance to mask the device. If this happens, either
/*   this code path or the interrupt code path is going to busy-wait
/*   for the lock. This type of collision is not expected to occur
/*   frequently so for simplicity it is allowed. Base level
/*   locks the controller only in an error path when a requesting
/*   process has been terminated or a request has timed-out. Further,
/*   simultaneous interrupts from the same device on different
/*   processors should occur very infrequently.
/*
/*.Return_Value
/*
/*      None.
/*

{

io_mask_interrupt_variety(Uc_Level_2_VME_Interrupt);
if (base_level)
    {
    vp_disable_interrupts();
    }
misc_obtain_spin_lock(&dip->controller_lock);
return;

}

/*.function */

        /*<-----------------------*/
WIRED
void      dev_cisc_unlock_controller  (dip, base_level)
        /*>-----------------------*/

dev_cisc_device_info_ptr_type                   dip;                /* READ ONLY */
boolean_type            base_level;     /* READ ONLY */

/*.Summary[=-----------------------------------------------------------
/*
/*   Release the cisc controller lock.
/*
/*.Parameters
/*
/* dip -- A pointer to a device information structure
/*   associated with the cisc SCSI adapter to be unlocked.
```

```
/*
/* base_level -- Boolean variable, when true, indicates
/*   that this function was called from base level.
/*
/*.Functional_Description
/*
/*   This function  releases the controller lock, enables interrupts
/*   on the current processor if we are executing at base level, and
/*   enables interrupts from the cisc controller.
/*
/*.Return_Value
/*
/*   None.
/*

{

misc_release_spin_lock(&dip->controller_lock);
if (base_level)
    {
    vp_enable_interrupts();
    }
io_unmask_interrupt_variety(Uc_Level_2_VME_Interrupt);
return;

}

/*.function */

               /*<---------------------------*/
WIRED
vp_ec_ptr_type  dev_cisc_service_async_timeout  (request_blk_ptr)
               /*>---------------------------*/

dev_cisc_request_blk_ptr_type                    request_blk_ptr;    /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------------
/*
/*   This function services timeouts of asynchronous cisc requests.
/*   It is called by the VP subsystem timeout manager when a
/*   previously established timeout occurs.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to the cisc request block
/*   that has timed out.
/*
/*.Functional_Description
/*
/*   The cisc manager establishes a timeout by calling the VP
/*   supplied timeout manager with a time interval, a pointer
/*   to this function, and the request block pointer as arguments.
/*   If the time specified elapses before the timeout is cancelled,
/*   this function is called with the request block pointer as
/*   an argument.
/*
/*   If a timeout has truly occurred, this function resets the
/*   cisc adapter and places a message in the driver demon's message
/*   queue, simulating the action of the interrupt service routine.
/*   The cisc complete_io function will be called by the demon to
/*   complete the processing of the request. See the remarks in
/*   the function dev_cisc_await_sync_event for a discussion of
/*   why the adapter is reset after a timeout.
/*
/*.Remarks
/*
/*   Remember that this function runs with interrupts disabled!  It
/*   must be very careful about what functions it calls.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Exceptions
/*
```

```
/*   None.
/*
{

vp_ec_ptr_type                          ec_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   We first get the controller lock and check the state of the
/*   'request_handled' flag in the request block. If it is set,
/*   the real interrupt occurred while we were getting to this
/*   handler and we should therefore do nothing.
/*
/*.End]=-----------------------------------------------------------*/

ec_ptr = VP_NULL_EC_PTR;
dev_cisc_lock_controller(request_blk_ptr->dip, TRUE);
if (!request_blk_ptr->async_request_handled)
    {
/*.Implementation_Continued[=--------------------------------------
/*
/*   The command complete interrupt for the request has not come in.
/*  Check to see if any retries have been done by the controller.
/*  If a retry has been started, restart the timeout and return
/*  to the waiting state.
/*
/*.End]=-----------------------------------------------------------*/

    if (request_blk_ptr->retries_started !=
                        request_blk_ptr->retries_acknowledged)
        {
            request_blk_ptr->retries_acknowledged++;
            request_blk_ptr->async_timeout_id =
                            vp_establish_timeout(request_blk_ptr->async_timeout_ptr,
                                                 dev_cisc_service_async_timeout,
                        (bit32e_type)request_blk_ptr);
            }
    else
            {
/*.Implementation_Continued[=-------------------------------------
/*
/*
/* The command complete interrupt for the request has not come in
/* and there are no outstanding retries in progress. Reset the
/* adapter and put a message in the I/O demon queue so the
/* cisc complete I/O routine will be called to process the timeout.
/*
/*.End]=-----------------------------------------------------------*/

        request_blk_ptr->async_request_handled = TRUE;
        request_blk_ptr->request_timed_out = TRUE;
        request_blk_ptr->dip->cisc_reg_ptr->controller_reset =
                            (bit32e_type)TRUE;
            ec_ptr = io_queue_message_to_driver_demon(
                        dev_cisc_complete_io,
                                    (bit32e_type)request_blk_ptr,
                        FALSE);
             }
    }
dev_cisc_unlock_controller(request_blk_ptr->dip, TRUE);
return(ec_ptr);
}

/*.function */

            /*<-----------------------------------*/
WIRED
status_type      dev_cisc_send_type_0_cmd_physical_mode      (dip)
            /*>-----------------------------------*/

dev_cisc_physical_device_info_ptr_type          dip;          /* READ/WRITE */

/*.Summary[=-------------------------------------------------------------
/*
/*   Perform a "physcial" Ciprico Rimfire 3500 operation.
```

```
/*
/*.Parameters
/*
/* dip -- A pointer to the device information structure
/*      for the Ciprico Rimfire 3500 SCSI adapter. Note that this
/*   is a special version of the dip and is not the same as the
/*   dip used during normal system operation.
/*
/*.Functional_Description
/*
/*   This function is called to execute a Ciprico Rimfire 3500
/*   command without the use of the normal operating system
/*   facilities. Synchronization is done without the use of
/*   event counters or interrupts. All buffer addresses are
/*   assummed to be physical. The system is assumed to be
/*   running a single thread of control so no lock management is
/*   required.
/*
/*   This function was originally developed to serve as the CISC
/*   prototype driver before vm, lock management, and interrupt handling
/*   were implemented on the 88k. They are now used to perform system
/*   dumps when the system is in shutdown mode. In the future, they may
/*   used by diagnostics software to access devices on the SCSI bus.
/*
/*.Return_Value
/*
/* OK -- A normal completion event was detected.
/*
/* IO_EIO_PHYSICAL_UNIT_FAILURE -- The controller did
/*   not respond within the timeout interval.
/*
{

status_type             status;
dev_cisc_type_0_pb_ptr_type         param_blk_ptr;
uint32_type             timeout;

/*.Implementation[=-----------------------------------------------
/*
/*      Clear the status portion of the type 0 parameter block.
/*
/*.End]=-------------------------------------------------------------*/

status = OK;
param_blk_ptr = &dip->type_0_param_blk;
param_blk_ptr->status_block.command_id = 0;
param_blk_ptr->status_block.error_code = 0;
param_blk_ptr->status_block.flags = 0;
param_blk_ptr->reserved_1_must_be_zero = 0;
param_blk_ptr->reserved_2_must_be_zero = 0;

/*.Implementation_Continued[=-------------------------------------
/*
/*   Load the address buffer port with data transfer control
/*   information and the physical address of the parameter block.
/*   The address buffer port consists of three 16 bit words. Three
/*   consecutive writes to the base address of the address buffer
/*   port allows each word to be loaded with data.
/*
/*.End]=-------------------------------------------------------------*/

dip->cisc_reg_ptr->address_buffer =
                        DEV_CIPRICO_ADDRESS_BUFFER_SET_CONTROL_BITS |
                            dip->vme_address_modifier;

dip->cisc_reg_ptr->address_buffer = (bit32e_type)param_blk_ptr >> 16;
dip->cisc_reg_ptr->address_buffer = (bit32e_type)param_blk_ptr & 0xFFFF;

/*.Implementation_Continued[=-------------------------------------
/*
/*   Start command execution by writing a zero to the channel
/*   attention register. Zero indicates that this is a type
/*   zero command. Busy-wait for the command complete bit to
/*   to come on in the status block flags field. If the
/*   operation does not complete in 120 seconds, timeout and
```

```
/*   return the error.
/*
/*.End]=-------------------------------------------------------------*/

param_blk_ptr->interrupt_level = 0;
param_blk_ptr->interrupt_vector = 0;
dip->cisc_reg_ptr->channel_attention = 0;
timeout = 120*1000;
do  {
    timeout--;
    if (timeout == 0)
        {
        status = IO_EIO_PHYSICAL_UNIT_FAILURE;
        break;
        }
            sc_busy_wait_microseconds((uint32e_type)1000);
    } while(!(param_blk_ptr->status_block.flags &
                              DEV_CIPRICO_COMMAND_COMPLETE_STATUS_FLAG));

return(status);

}
```

Driver Utility Code: dev_cisc_util.c

```
/*<-----------------------------------------------------*/
/*                   dev_cisc_util.c                   */
/*>-----------------------------------------------------*/

/*.Contents[=-------------------------------------------------------------
/*
/* dev_cisc_register_requester -- subsystem
/* dev_cisc_set_unit_options -- subsystem
/* dev_cisc_deregister_requester -- subsystem
/* dev_cisc_issue_command -- subsystem
/* dev_cisc_issue_async_command -- subsystem
/* dev_cisc_get_device_info -- subsystem
/* dev_cisc_complete_io -- subsystem
/* dev_cisc_issue_command_physical_mode -- subsystem
/* dev_cisc_start_async_request -- internal
/* dev_cisc_build_command_list_request -- internal
/* dev_cisc_check_status -- internal
/* dev_cisc_build_scatter_gather_arrays -- internal
/* dev_cisc_build_single_buffer -- internal
/* dev_cisc_free_scatter_gather_arrays -- internal
/* dev_cisc_complete_aborted_request -- subsystem
/*
/*.Description
/*
/*   The functions in this module provide the interface between
/*   the SCSI device drivers and the cisc manager. The cisc
/*   manager controls the Ciprico Rimfire 3500 Host SCSI bus
/*   adapter which is used to interface to the SCSI bus on
/*   Topgun class computers. The SCSI device drivers have no
/*   knowledge of SCSI bus interfaces and issue requests to
/*   SCSI devices by means of generic parameter blocks. The
/*   functions in this module convert the generic requests to
/*   a format used by the cisc manager and dispatch the
/*   requests to the cisc manager.
/*
/*   Since SCSI bus interfaces vary between system architectures,
/*   multiple SCSI bus interface managers exist within DG/UX. Each
/*   SCSI bus interface manager must provide a standard SCSI device
/*   driver interface which conforms to the one implemented here.
/*   The standard driver interface allows the SCSI drivers to work
/*   on the various architectures without modification.
/*
/*   The entry points to the functions in this module are packaged
/*   in a routines vector defined in dev_scsi_adapter_def.h.  SCSI
/*   device drivers obtain pointers to the routines vector during
/*   device configuration.
```

```
/*
/*.function */

            /*<-------------------------*/
UNWIRED
status_type  dev_cisc_register_requester  (rb_ptr)
            /*>-------------------------*/

dev_scsi_adapter_unit_registration_blk_ptr_type rb_ptr;       /* READ/WRITE */

/*.Summary[=-----------------------------------------------------------------
/*
/*   This function associates the specified device with a cisc
/*   SCSI adapter, thereby establishing a link between the device
/*   driver and the adapter service routines.
/*
/*.Parameters
/*
/* rb_ptr -- A pointer to a scsi adapter registration block.
/*
/*.Functional_Description
/*
/*   This function adds an entry to the unit table associated with
/*   the specified scsi id and unit number. The unit table entry
/*   consists of a device type specifier and an opaque unit handle
/*   which is meaningful only to the device driver. The unit table
/*   entry provides a bridge between the device driver and the cisc
/*   management routines.
/*
/*   If the unit table entry specified by the SCSI id and unit
/*   number is already occupied, an error is returned. Also the
/*   device type of the device occupying the entry is returned
/*   so that the caller can distinguish betweeen
/*   IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED and
/*   IO_ENXIO_DEVICE_DOES_NOT_EXIST.
/*
/*.Return_Value
/*
/* OK -- The specified device was successfully registered
/*   with the cisc adapter.
/*
/* IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED -- A device is
/*   already registered at the location specified by
/*   <rb_ptr>.
/*
/*   Return values from io_map_device_number.
/*

{
status_type                          status;
dev_cisc_unit_table_ptr_type         unit_table_ptr;
dev_cisc_unit_table_entry_ptr_type   ute_ptr;
uint16_type                  unit;

/*.Implementation[=-----------------------------------------------
/*
/*   Get the cisc open lock to insure that we have exclusive
/*   access to the cisc unit table data structures. Translate
/*   adapter major and minor number to a specific adapter unit
/*   table.
/*
/*.End]=-------------------------------------------------------------*/

status = OK;
lm_obtain_unsequenced_lock(&dev_cisc_open_lock);
status = io_map_device_number(rb_ptr->adapter_device_number,
                              (bit32e_ptr_type)&unit_table_ptr,
                              &unit);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
```

```
/*   Using the scsi id and unit number from the registration
/*   packet, get the unit table entry for the device being
/*   registered. If a device is already registered for the
/*   entry, return the error. Otherwise, mark the entry as in
/*   use and return the adapter handle.
/*
/*.End]=-------------------------------------------------------------*/

ute_ptr = &((*unit_table_ptr)[rb_ptr->unit_spec.scsi_id]
                             [rb_ptr->unit_spec.unit]);
if (ute_ptr->in_use)
    {
    rb_ptr->device_type = ute_ptr->device_type;
    status = IO_ENXIO_DEVICE_IS_ALREADY_CONFIGURED;
    }
else
    {
    ute_ptr->in_use = TRUE;
    ute_ptr->driver_handle = rb_ptr->driver_handle;
    ute_ptr->device_type = rb_ptr->device_type;
    misc_initialize_counter(&ute_ptr->outstanding_request_count,
                            (int32e_type)0);
    rb_ptr->max_request_size = DEV_CISC_MAX_REQUEST_SIZE;
    rb_ptr->adapter_handle = (io_device_handle_type)ute_ptr;
    }
done:
lm_release_unsequenced_lock(&dev_cisc_open_lock);
return(status);
}

/*.function */

            /*<----------------------*/
UNWIRED
status_type  dev_cisc_set_unit_options  (adapter_handle,
                                unit_options_block_ptr)
            /*>----------------------*/

io_device_handle_type                                adapter_handle;  /* READ ONLY */
dev_scsi_adapter_unit_options_block_ptr_type unit_options_block_ptr;
                                /* READ ONLY */

/*.Summary[=-----------------------------------------------------------
/*
/*   Set the unit options of a registered device.
/*
/*.Parameters
/*
/* adapter_handle -- The device handle of the physical unit
/*   the is the target of the set unit options operation. This handle
/*   must be the device handle that was returned by the
/*   register_requester routine of the adapter manager.
/*
/* unit_options_block_ptr -- Pointer to a unit options block
/*   which specifies the options to be selected for the unit.
/*
/*.Functional_Description
/*
/*   This function is called to set the various unit options which
/*   describe how the SCSI adatper driver manages a request that has
/*   been issued over the SCSI bus to a physical unit. See the
/*   definition of the  dev_scsi_adapter_unit_options_block in the file
/*   dev_scsi_adapter_def.h for a complete description of the unit
/*   options supported.
/*
/*.Return_Value
/*
/* OK -- The requested options were selected successfully.
/*
/* DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE -- An illegal
/*   option value was detected in the callers Set Unit Options
/*   Block.
/*
/* IO_EIO_PHYSICAL_UNIT_FAILURE -- The Set Unit Options
/*   command issued to the Ciprico controller resulted in an
```

```
/*   error.
/*

{

status_type                              status;
dev_cisc_unit_table_entry_ptr_type  ute_ptr;
struct timeval                           timeout_timeval;
long                        timeout_in_milliseconds;
uint16_type                 disconnect_timeout;
dev_cisc_unit_options_pb_ptr_type        unit_options_pb_ptr;
dev_cisc_device_info_ptr_type                    dip;
vp_event_type                            request_completion_event;
dev_adapter_request_block_type                   arb;
dev_cisc_request_blk_ptr_type            request_blk_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Get the cisc open lock and convert the adapter handle to
/*   a unit table entry for the device.
/*
/*.End]=-----------------------------------------------------------*/

status = OK;
lm_obtain_unsequenced_lock(&dev_cisc_open_lock);
ute_ptr = (dev_cisc_unit_table_entry_ptr_type)adapter_handle;

/*.Implementation_Continued[=--------------------------------------
/*
/*   The Ciprico controller handles disconnect timeouts
/*   automatically. We just have to tell it what the timeout
/*   value is. The disconnect timeout value in the Ciprico set
/*   unit options parameter block is specified in units of .1
/*   seconds. Convert the disconnect timeout value from the caller's
/*   unit options block to milliseconds, verify that the timeout
/*   is in a range supported by interface, and then conver the
/*   timeout value to .1 seconds units.
/*
/*   If disconnect timeout value pointer in the caller's set unit
/*   options block is DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR, disconnect
/*   timeouts are disabled for the device
/*
/*.End]=-----------------------------------------------------------*/

if (unit_options_block_ptr->disconnect_timeout_ptr !=
                            DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR)
    {
    misc_clock_value_to_timeval(
                            unit_options_block_ptr->disconnect_timeout_ptr,
                            &timeout_timeval);
    misc_timeval_to_milliseconds(&timeout_timeval, &timeout_in_milliseconds);
    if   ((timeout_in_milliseconds < DEV_SCSI_ADAPTER_MIN_TIMEOUT_VALUE) ||
        (timeout_in_milliseconds > DEV_SCSI_ADAPTER_MAX_TIMEOUT_VALUE))
        {
        status = DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE;
        goto done;
        }
    disconnect_timeout = timeout_in_milliseconds/100;
    }
else
    {
    disconnect_timeout = DEV_CISC_UNIT_OPTIONS_TIMEOUT_DISABLE;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Validate the caller's bus request timeout. If bus request
/*   timeout value pointer in the caller's set unit options block
/*   is DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR, no timeout is used.
/*   The Ciprico controller manages timeouts internally, the
/*   bus request timeout value is used as a backup timeout
/*   mechanism in case the Ciprico controller fails.
/*
/*.End]=-----------------------------------------------------------*/
```

```
if (unit_options_block_ptr->bus_request_timeout_ptr !=
                            DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR)
    {
    misc_clock_value_to_timeval(
                            unit_options_block_ptr->bus_request_timeout_ptr,
                            &timeout_timeval);
    misc_timeval_to_milliseconds(&timeout_timeval, &timeout_in_milliseconds);
    if   ((timeout_in_milliseconds < DEV_SCSI_ADAPTER_MIN_TIMEOUT_VALUE) ||
        (timeout_in_milliseconds > DEV_SCSI_ADAPTER_MAX_TIMEOUT_VALUE))
        {
        status = DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE;
        goto done;
        }
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*      Verify that the adapter retries field of the caller's set
/*      unit options block is within the supported range. If it
/*      is not, return an error status.
/*
/*.End]=-----------------------------------------------------------*/

if   ((unit_options_block_ptr->adapter_retries <
    DEV_SCSI_ADAPTER_MIN_ADAPTER_RETRIES) ||
    (unit_options_block_ptr->adapter_retries >
    DEV_SCSI_ADAPTER_MAX_ADAPTER_RETRIES))
    {
    status = DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE;
    goto done;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*      Verify that the sense bytes field of the caller's set
/*      unit options block is within the supported range. If it
/*      is not, return an error status.
/*
/*.End]=-----------------------------------------------------------*/

if   ((unit_options_block_ptr->sense_bytes < DEV_SCSI_ADAPTER_MIN_SENSE_BYTES)
    || (unit_options_block_ptr->sense_bytes > DEV_SCSI_ADAPTER_MAX_SENSE_BYTES))
    {
    status = DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE;
    goto done;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Verify that the maximum disconnect/reconnect per request field
/*   of the caller's set unit options block is within the supported
/*   range. If it is not, return an error status.
/*
/*.End]=-----------------------------------------------------------*/

if   ((unit_options_block_ptr->max_disconn_reconn_per_command <
    DEV_SCSI_ADAPTER_MIN_DISCON_RECON) ||
    (unit_options_block_ptr->max_disconn_reconn_per_command >
    DEV_SCSI_ADAPTER_MAX_DISCON_RECON))
    {
    status = DEV_STATUS_SCSI_ILLEGAL_UNIT_OPTIONS_VALUE;
    goto done;
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Calculate the timeout value that will be used to determine
/*   if the Ciprico controller has failed. This timeout is
/*   managed by the driver and is used as a backup in case the
/*   Ciprico stops working. The timeout is calculated as:
/*
/*   (disconnect timeout value + active bus request timeout) X
/*   maximum number of disconnect/reconnect cycles per command.
/*
/*.End]=-----------------------------------------------------------*/
```

```
if   ((unit_options_block_ptr->disconnect_timeout_ptr ==
    DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR) ||
   (unit_options_block_ptr->bus_request_timeout_ptr ==
    DEV_SCSI_ADAPTER_NULL_TIMEOUT_PTR))
    {
/*.Implementation_Continued[=--------------------------------------
/*
/*    If timeouts are to be disabled at the device, disable them
/*     driver timeouts as well.
/*
/*.End]=-----------------------------------------------------------*/

    ute_ptr->controller_dead_timeout = misc_maximum_clock_value;
    }
else if (unit_options_block_ptr->max_disconn_reconn_per_command == 0)
    {
/*.Implementation_Continued[=--------------------------------------
/*
/*    If disconnect/reconnect is disabled for the device, the driver
/*     timeout becomes the bus request timeout value.
/*
/*.End]=-----------------------------------------------------------*/

    ute_ptr->controller_dead_timeout =
                        *unit_options_block_ptr->bus_request_timeout_ptr;
    }
else
    {
    ute_ptr->controller_dead_timeout =
                        *unit_options_block_ptr->disconnect_timeout_ptr;
    MISC_CLOCK_VALUE_ADD(unit_options_block_ptr->bus_request_timeout_ptr,
                         &ute_ptr->controller_dead_timeout);

    MISC_CLOCK_VALUE_MULTIPLY((uint32_type)
                  unit_options_block_ptr->max_disconn_reconn_per_command,
              &ute_ptr->controller_dead_timeout);
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   Obtain the request lock to insure that a request block is
/*   available. Dequeue a request block and set up a parameter
/*   block to set the options for the unit.
/*
/*   Since the command is issued through the general adapter
/*   request mechanism, we need a local adapter request block
/*   to specify the command.
/*
/*   The cisc retry control bits are set so that a status block is
/*   generated for each retry. This is done so that the timeout
/*   path can informed of retries.
/*
/*.End]=-----------------------------------------------------------*/

dip = ute_ptr->dip;
io_sync_obtain_interleave_lock(&dip->request_lock);
misc_dequeue_from_head(&dip->request_blk_queue,
                       (misc_queue_links_ptr_type *)&request_blk_ptr);
request_blk_ptr->sync_io = TRUE;
io_init_one_entry_buffer_vector(
                  &arb.buffer_vector,
                  (pointer_to_any_type)DEFAULT_NULL_LINK,
                        (uint32_type)0);
arb.request_flags = 0;
arb.unit_spec = ute_ptr->unit_spec;
request_blk_ptr->arb_ptr = &arb;
request_blk_ptr->ute_ptr = ute_ptr;
unit_options_pb_ptr = (dev_cisc_unit_options_pb_ptr_type)
                                  &request_blk_ptr->param_block;
unit_options_pb_ptr->param_block_id = (uint32e_type)request_blk_ptr;
unit_options_pb_ptr->disconnect_timeout = disconnect_timeout;
unit_options_pb_ptr->unit_id = ute_ptr->unit_spec.scsi_id;
unit_options_pb_ptr->target_id = DEV_CISC_SCSI_TARGET_ID;
unit_options_pb_ptr->select_timeout = DEV_CISC_UNIT_OPTIONS_SELECT_TIMEOUT;
unit_options_pb_ptr->retry_control = DEV_CISC_RETRY_CONTROL_RCE |
```

```
                    DEV_CISC_RETRY_CONTROL_INT | DEV_CISC_RETRY_CONTROL_ISB;

unit_options_pb_ptr->retry_limit = unit_options_block_ptr->adapter_retries;
unit_options_pb_ptr->reserved_1 = 0;
unit_options_pb_ptr->sense_bytes = unit_options_block_ptr->sense_bytes;
if (unit_options_block_ptr->perform_request_sorting)
    {
    unit_options_pb_ptr->unit_flags = DEV_CISC_UNIT_FLAGS_SOR;
    }
else
    {
    unit_options_pb_ptr->unit_flags = 0;
    }
if (unit_options_block_ptr->max_disconn_reconn_per_command == 0)
    {
    unit_options_pb_ptr->unit_flags |= DEV_CISC_UNIT_FLAGS_IDI;
    }
if (dip->sync_scsi_supported)
    {
    unit_options_pb_ptr->unit_flags |= DEV_CISC_UNIT_FLAGS_SYN;
    }
unit_options_pb_ptr->command = DEV_CISC_CMD_SET_UNIT_OPTIONS;
unit_options_pb_ptr->reserved_2 = 0;
unit_options_pb_ptr->reserved_3 = 0;
unit_options_pb_ptr->reserved_4 = 0;

/*.Implementation_Continued[=--------------------------------------
/*
/*   Set up to pend and wait for the request to complete, start
/*   the request, then wait for the request to complete.
/*
/*.End]=-----------------------------------------------------------*/

request_completion_event.name = &request_blk_ptr->sync_io_ec;
vp_get_next_ec_value(&request_blk_ptr->sync_io_ec,
                     &request_completion_event.value);
dev_cisc_start_command_list_request(request_blk_ptr);
dev_cisc_await_sync_event(request_blk_ptr,
                          &request_completion_event,
                          &misc_three_seconds);

/*.Implementation_Continued[=--------------------------------------
/*
/*   The Set Unit Options request has completed. Call the
/*   check_status routine to see if any errors occurred. Note
/*   that the request block will be deallocated and the interleave
/*   lock will be reassigned by the check_status function.
/*
/*.End]=-----------------------------------------------------------*/

status = dev_cisc_check_status(request_blk_ptr);
io_release_interleave_lock(&dip->request_lock);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Release the open lock and return the status of the
/*   set unit options operation.
/*
/*.End]=-----------------------------------------------------------*/

done:
lm_release_unsequenced_lock(&dev_cisc_open_lock);
return(status);
}

/*.function */

            /*<-----------------------------*/
UNWIRED
void                    dev_cisc_deregister_requester  (adapter_handle)
            /*>-----------------------------*/

io_device_handle_type                       adapter_handle;  /* READ ONLY */

/*.Summary[=-------------------------------------------------------------
```

```
/*
/*   This function terminates the link between the cisc SCSI adapter
/*   manager and the device referenced by <adapter_handle>.
/*
/*.Parameters
/*
/* adapter_handle -- The device handle of the physical unit
/*   that is to be deregistered. This handle must be the device handle
/*   that was returned by the register_requester routine of the
/*   adapter manager.
/*
/*.Functional_Description
/*
/*   See Summary.
/*
/*.Return_Value
/*
/*   None.
/*
{

dev_cisc_unit_table_entry_ptr_type   ute_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Get the cisc open lock and free the unit table entry for
/*   the device.
/*
/*.End]=-----------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_cisc_open_lock);
ute_ptr = (dev_cisc_unit_table_entry_ptr_type)adapter_handle;
ute_ptr->in_use = FALSE;

lm_release_unsequenced_lock(&dev_cisc_open_lock);
return;
}

/*.function */

            /*<-------------------*/
WIRED
status_type  dev_cisc_issue_command  (arb_ptr)
            /*>-------------------*/

dev_adapter_request_block_ptr_type  arb_ptr;/* READ/WRITE */

/*.Summary[=-------------------------------------------------------
/*
/*   Issue a SCSI command synchronously through the adapter to a
/*   target device.
/*
/*.Parameters
/*
/* arb_ptr -- A pointer to a generic adapter request block
/*   that holds all information which decribes the request.
/*
/*.Functional_Description
/*
/*   This function transfers request information from the generic
/*   adapter request block to the cisc specific parameter
/*   block and calls the cisc manager to execute the request.
/*
/*   If the request completes with a Check Condition Status, sense
/*   information from the device is automatically returned in the
/*   adapter request block.
/*
/*.Return_Value
/*
/* OK -- A synchronous request completed successfully.
/*
/*   Return values from dev_cisc_check_status.
/*
{
```

```
dev_cisc_unit_table_entry_ptr_type      ute_ptr;
dev_cisc_device_info_ptr_type                   dip;
dev_cisc_request_blk_ptr_type                   request_blk_ptr;
vp_event_type                       request_completion_event;
status_type             status;

/*.Implementation[=-----------------------------------------------
/*
/*   Get a pointer to the device information structure and get
/*   the controller request lock.
/*
/*.End]=-------------------------------------------------------------*/

ute_ptr = (dev_cisc_unit_table_entry_ptr_type)arb_ptr->adapter_handle;
dip = ute_ptr->dip;
io_sync_obtain_interleave_lock(&dip->request_lock);

/*.Implementation_Continued[=-------------------------------------
/*
/*   Allocate a request block and build a Ciprico Rimfire 3500
/*   parameter block.
/*
/*.End]=-------------------------------------------------------------*/

misc_dequeue_from_head(&dip->request_blk_queue,
                       (misc_queue_links_ptr_type *)&request_blk_ptr);
request_blk_ptr->arb_ptr = arb_ptr;
request_blk_ptr->ute_ptr = ute_ptr;
request_blk_ptr->sync_io = TRUE;
dev_cisc_build_command_list_request(request_blk_ptr);

/*.Implementation_Continued[=-------------------------------------
/*
/*   Set up to pend for command completion, start the command,
/*   then pend and wait for the command to complete.
/*
/*.End]=-------------------------------------------------------------*/

request_completion_event.name = &request_blk_ptr->sync_io_ec;
vp_get_next_ec_value(&request_blk_ptr->sync_io_ec,
                             &request_completion_event.value);
dev_cisc_start_command_list_request(request_blk_ptr);
dev_cisc_await_sync_event(
             request_blk_ptr,
             &request_completion_event,
             &ute_ptr->controller_dead_timeout);

/*.Implementation_Continued[=-------------------------------------
/*
/*   The command list request has completed. Do the request completion
/*   processing and examine the status block returned. Note that
/*   request block will be deallocated and the interleave lock
/*   will be reassigned by the check_status function.
/*
/*.End]=-------------------------------------------------------------*/

status = dev_cisc_check_status(request_blk_ptr);
io_release_interleave_lock(&dip->request_lock);

return(status);
}

/*.function */

            /*<-------------------------*/
WIRED
status_type  dev_cisc_issue_async_command  (arb_ptr)
            /*>-------------------------*/

dev_adapter_request_block_ptr_type  arb_ptr;/* READ/WRITE */

/*.Summary[=-----------------------------------------------------------
/*
/*   Issue a SCSI command asynchronously through the adapter to a
/*   target device.
```

```
/*
/*.Parameters
/*
/* arb_ptr -- A pointer to a generic adapter request block
/*   that holds all information which decribes the request.
/*
/*.Functional_Description
/*
/*   The adapter request block is added to the asynchronous request
/*   queue and an attempt is made to obtain the specified controller's
/*   command list request lock. If the lock is obtained,
/*   dev_cisc_start_async_request is called to start the request.
/*   Control is returned to the caller as soon as the request has
/*   been issued through the adapter to the physical unit. The I/O
/*   daemon handles request completion and starts the next request
/*   in the queue if there is one.
/*
/*   If the command list request lock cannot be obtained, the request
/*   left on the request queue and the function returns immediately.
/*   The enqueued request is started when the currently executing
/*   request and all requests ahead in the queue have been executed.
/*
/*   Note that the requested block must be enqueued before we
/*   attempt to obtain the request lock. If the request lock becomes
/*   free immediately after the check, the request must be in the
/*   queue so that it can be started by the process which just
/*   completed a request.
/*
/*.Return_Value
/*
/* OK -- The request was successfully started.  This status
/*      does not indicate that the request has completed successfully.
/*
{

dev_cisc_unit_table_entry_ptr_type      ute_ptr;
dev_cisc_device_info_ptr_type                   dip;

/*.Implementation[=-------------------------------------------------
/*
/*   Add the adapter request block to the asynchronous request
/*   queue. If the request lock can be obtained, start the
/*   request. Otherwise, return immediately.
/*
/*.End]=-----------------------------------------------------------*/

ute_ptr = (dev_cisc_unit_table_entry_ptr_type)arb_ptr->adapter_handle;
dip = ute_ptr->dip;
misc_enqueue_at_tail(&dip->async_request_queue, &arb_ptr->links);
if (io_async_obtain_interleave_lock(&dip->request_lock))
    {
    dev_cisc_start_async_request(dip);
    io_release_interleave_lock(&dip->request_lock);
    }
return(OK);
}

/*.function */

              /*<----------------------*/
UNWIRED
status_type      dev_cisc_get_device_info  (adapter_device_number,
                           unit_spec,
                           device_type,
                                   driver_handle_ptr)
           /*>----------------------*/

io_device_number_type                   adapter_device_number; /* READ ONLY */
dev_scsi_adapter_unit_spec_type         unit_spec;                      /* READ ONLY */
bit8_type                       device_type;            /* READ ONLY */
bit32e_ptr_type                         driver_handle_ptr;      /* WRITE ONLY */

/*.Summary[=-------------------------------------------------------------
/*
/*   This function retrieves device information associated with
```

```
/*   specified registered device.
/*
/*.Parameters
/*
/* adapter_device_number -- The major and minor device
/*   number of the SCSI adapter used to access the target unit.
/*
/* unit_spec -- The SCSI id and unit number of the
/*   target device.
/*
/* device_type -- Device type of device expected to be
/*      registered for unit number and SCSI id.
/*
/* driver_handle_ptr -- A pointer to where the device
/*   information is to be returned.
/*
/*.Functional_Description
/*
/*   Return the opaque driver handle that was registered with the
/*   device. This function takes the place of io_get_device_info
/*   for SCSI devices which don't have DIT entries.
/*
/*.Return_Value
/*
/* OK -- The opaque driver handle was successfully retrieved
/*   and returned.
/*
/*   Return values from io_map_device_number.
/*
/* IO_ENXIO_DEVICE_IS_NOT_CONFIGURED -- A device of the
/*      specified type is not registered at the SCSI id and unit
/*      number slot.
/*
{

status_type                          status;
dev_cisc_unit_table_ptr_type         unit_table_ptr;
dev_cisc_unit_table_entry_ptr_type   ute_ptr;
uint16_type              unit;

/*.Implementation[=-----------------------------------------------
/*
/*   Get the cisc open lock to insure that we have exclusive
/*   access to the cisc unit table data structures. Translate
/*   adapter major and minor number to a specific adapter unit
/*   table.
/*
/*.End]=-----------------------------------------------------------*/

lm_obtain_unsequenced_lock(&dev_cisc_open_lock);
status = io_map_device_number(adapter_device_number,
                              (bit32e_ptr_type)&unit_table_ptr,
                              &unit);
if (status != OK)
    {
    goto done;
    }

/*.Implementation_Continued[=-------------------------------------
/*
/*   Using the scsi id and unit number arguments, get the unit
/*   table entry for the device being referenced. Extract the
/*   driver handle from the unit table entry and return it to
/*   the caller.
/*
/*.End]=-----------------------------------------------------------*/

ute_ptr = &((*unit_table_ptr)[unit_spec.scsi_id]
                             [unit_spec.unit]);
if (!ute_ptr->in_use)
    {
    status = IO_ENXIO_DEVICE_IS_NOT_CONFIGURED;
    }
else if (!(ute_ptr->device_type & device_type))
    {
```

```
    status = IO_ENXIO_DEVICE_IS_NOT_CONFIGURED;
    ]
else
    {
    *driver_handle_ptr = ute_ptr->driver_handle;
    ]
done:
lm_release_unsequenced_lock(&dev_cisc_open_lock);
return(status);
}

/*.function */

                /*<-----------------*/
WIRED
void             dev_cisc_complete_io  (data)
                /*>-----------------*/

bit32e_type             data;   /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------------
/*
/*      This function handles the completion of asynchronous requests
/*      that have been completed by the cisc controller.
/*
/*.Parameters
/*
/* data -- The 32 bits of data that was in the message
/*      given to the driver demon.
/*
/*.Functional_Description
/*
/*   This function handles the completion of asynchronous I/O requests.
/*   It calls the common status_check routine to determine the
/*   results of the request.
/*
/*   When the result of the operation is determined, this function
/*   calls the "complete_io" function specified in the original
/*   SCSI adpater request block. This is an upcall to notify the
/*   requestor that the asynchronous operation is now complete.
/*
/*.Return_Value
/*
/*   Return values from dev_cisc_check_status.
/*
/*.Exceptions
/*
/*   None.
/*
/*.Abort_Conditions
/*
/*   None.
/*
{
status_type             status;
dev_adapter_request_block_ptr_type      arb_ptr;
dev_cisc_request_blk_ptr_type                   request_blk_ptr;

/*.Implementation[=-----------------------------------------------
/*
/*   The command list request has completed. First cancel the
/*   timeout so we don't continue to tie up space in the timeout
/*   table. Examine the status returned and take the appropriate
/*   action. Note that request block will be deallocated and the
/*   interleave lock will be released by the check_status function.
/*
/*.End]=-----------------------------------------------------------*/

request_blk_ptr = (dev_cisc_request_blk_ptr_type)data;
vp_cancel_timeout(request_blk_ptr->async_timeout_id);
arb_ptr = request_blk_ptr->arb_ptr;

status = dev_cisc_check_status(request_blk_ptr);

/*.Implementation_Continued[=--------------------------------------
```

```
/*
/*   Upcall the orignal caller's complete I/O routine.
/*
/*.End]=-------------------------------------------------------------*/

(*(arb_ptr->complete_io_routine))(arb_ptr, status);
return;
]

/*.function */

            /*<--------------------------------*/
WIRED
status_type  dev_cisc_issue_command_physical_mode  (request_blk_ptr)
            /*>--------------------------------*/

dev_adapter_physical_request_blk_ptr_type    request_blk_ptr; /* READ/WRITE */

/*.Summary[=-----------------------------------------------------------
/*
/*   Issue a "physical" I/O request through the SCSI adapter to a
/*   target device.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that holds
/*   information which specifies the request. Note that this is a
/*   special version of the adapter request block and is not the
/*   same as the request block used during normal system operation.
/*
/*.Functional_Description
/*
/*   This function is called to issue a synchronous I/O request over
/*   the SCSI bus without the use of the normal operating system
/*   facilities. Synchronization is done without the use of
/*   event counters or interrupts. All buffer addresses are assumed
/*   to be physical. The system is assumed to be running a single
/*   thread of control so no lock management is required.
/*
/*.Return_Value
/*
/* OK -- A synchronous request completed successfully or
/*   an asynchronous request was started.
/*
/* DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION -- The
/*   command completed with a check condition status and sense
/*   information is available in the caller's sense buffer.
/*
/* DEV_STATUS_SCSI_DEVICE_IS_BUSY -- The
/*   command completed with a busy status indicating the device
/*   is probably performing it's power-on/reset initialization.
/*
/* IO_EIO_HARD_IO_ERROR -- The command completed with a
/*   check condition status and the subsequent request sense
/*   command failed.
/*
/*   Return values from dev_cisc_issue_type_0_cmd_physical_mode.
/*
[

status_type                     status;
dev_cisc_type_0_pb_ptr_type                     param_blk_ptr;
dev_cisc_physical_device_info_ptr_type          dip;
bit8e_type              scsi_status;


/*.Implementation[=---------------------------------------------------
/*
/*   Issue the request to the adapter manager. If the request
/*   fails with a check condition status, copy sense information
/*   from the status block into the request block.
/*
/*   Note that the ciprico 3500 will span contiguous physical
/*   pages on a dma transfer, so scatter gather is not
/*   required if the request is for more than a page.
```

```
/*
/*.End]=-------------------------------------------------------------*/

dip = (dev_cisc_physical_device_info_ptr_type)dev_cisc_physical_dip;
param_blk_ptr = &dip->type_0_param_blk;
param_blk_ptr->std_param_block.flags = (bit8e_type)0;
io_get_buffer_vector_io_info(
  &request_blk_ptr->buffer_vector,
  (pointer_to_any_ptr_type)&param_blk_ptr->std_param_block.vme_memory_address,
  &param_blk_ptr->std_param_block.transfer_count);

param_blk_ptr->std_param_block.address_modifier = dip->vme_address_modifier;
param_blk_ptr->std_param_block.target_id = request_blk_ptr->unit_spec.scsi_id;
param_blk_ptr->std_param_block.scsi_cmd_blk = request_blk_ptr->scsi_cmd_blk;

status = dev_cisc_send_type_0_cmd_physical_mode(dip);
if (status == OK)
    {
    scsi_status = param_blk_ptr->status_block.scsi_status;
    scsi_status >>= DEV_SCSI_STATUS_BYTE_SHIFT;
    scsi_status &= DEV_SCSI_STATUS_BYTE_MASK;
    if (scsi_status == DEV_SCSI_STATUS_BUSY)
            [
            status = DEV_STATUS_SCSI_DEVICE_IS_BUSY;
            ]
    else if (scsi_status != 0)
        {
            misc_byte_copy(
                (pointer_to_any_type)param_blk_ptr->status_block.sense_data,
                (pointer_to_any_type)&request_blk_ptr->sense_buffer,
                DEV_CISC_REQUEST_SENSE_DATA_SIZE);
        status =  DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION;
        }
    ]

return(status);
}

/*.function */

               /*<-------------------------*/
WIRED
void            dev_cisc_start_async_request  (dip)
               /*>-------------------------*/

dev_cisc_device_info_ptr_type   dip;            /* READ ONLY  */

/*.Summary[=-----------------------------------------------------------
/*
/*   This function fills in the request block for an asynchronous
/*   I/O request and sends it to the through the Ciprico SCSI interface
/*   to the specified physical unit.
/*
/*.Parameters
/*
/* dip -- A pointer to the device info structure of the
/*   cisc controller on which an asynchronous request is to be started.
/*
/*.Functional_Description
/*
/*   An adapter request block is removed from the head of the async
/*   queue and a request block is removed from the request block free
/*   queue. The request block is filled in with the information from the
/*   adapter request block and is initialized to indicate an asynchronous
/*   request.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Exceptions
/*
/*   None.
/*
[
```

```
dev_adapter_request_block_ptr_type  arb_ptr;
dev_cisc_request_blk_ptr_type                    request_blk_ptr;
dev_cisc_unit_table_entry_ptr_type      ute_ptr;

/*.Implementation[=-------------------------------------------------
/*
/*   Get the next generic adapter request block from the queue.
/*
/*.End]=-----------------------------------------------------------*/

misc_dequeue_from_head(&dip->async_request_queue,
                       (misc_queue_links_ptr_type *)&arb_ptr);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Allocate a controller request block and build a Ciprico
/*   Rimfire 3500 parameter block.
/*
/*.End]=-----------------------------------------------------------*/

misc_dequeue_from_head(&dip->request_blk_queue,
                     (misc_queue_links_ptr_type *)&request_blk_ptr);
request_blk_ptr->arb_ptr = arb_ptr;
request_blk_ptr->sync_io = FALSE;
dev_cisc_build_command_list_request(request_blk_ptr);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Start the async timeout with the vp timeout manager.
/*
/*.End]=-----------------------------------------------------------*/

ute_ptr = (dev_cisc_unit_table_entry_ptr_type)arb_ptr->adapter_handle;
request_blk_ptr->async_timeout_ptr = &ute_ptr->controller_dead_timeout;
request_blk_ptr->ute_ptr = ute_ptr;
request_blk_ptr->async_timeout_id =
            vp_establish_timeout(request_blk_ptr->async_timeout_ptr,
                                 dev_cisc_service_async_timeout,
                                 (bit32e_type)request_blk_ptr);

/*.Implementation_Continued[=--------------------------------------
/*
/*   Start the command and return immediately. The I/O demon will
/*   perform request completion processing.
/*
/*.End]=-----------------------------------------------------------*/

dev_cisc_start_command_list_request(request_blk_ptr);
return;
}

/*.function */

/*<--------------------------------*/
WIRED
void     dev_cisc_build_command_list_request (request_blk_ptr)
/*>--------------------------------*/

dev_cisc_request_blk_ptr_type    request_blk_ptr; /* READ/WRITE */

/*.Summary[=-------------------------------------------------------------
/*
/*   Build a Ciprico Rimfire 3500 request block.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that holds
/*   information which specifies the I/O request.
/*
/*.Functional_Description
/*
/*   A Ciprico Rimfire 3500 parameter block is constructed from
/*   information in original caller's generic adapter request
/*   block. If the request is going to require DMA between the
```

```
/*   host and controller, scatter gather array are allocated and
/*   set up.
/*
/*   Process signal delivery is disabled for the current process
/*   if the request requires DMA and the request is synchronous.
/*   DMA requests cannot be aborted because there is no way to
/*   tell the Ciprico controller to abort a request once it has
/*   started. If a DMA request is aborted, we can't just return
/*   to the caller like is done with non-DMA requests because
/*   the controller is going to read/write data to the user's
/*   buffer.
/*
/*
/*.Return_Value
/*
/*   None.
/*
{

dev_cisc_param_block_ptr_type                    param_blk_ptr;
dev_adapter_request_block_ptr_type  arb_ptr;

/*.Implementation[=----------------------------------------------
/*
/*   Fill the parameter block in with information needed to
/*   specify the request.
/*
/*.End]=-------------------------------------------------------------*/

param_blk_ptr = &request_blk_ptr->param_block;
param_blk_ptr->param_blk_id = (uint32e_type)request_blk_ptr;
param_blk_ptr->reserved_1_must_be_zero = 0;
arb_ptr = request_blk_ptr->arb_ptr;

/*.Implementation_Continued[=-----------------------------------
/*
/*   If the adapter request block specifies a data transfer,
/*   build the scatter gather arrays needed and disable signal
/*   delivery.
/*
/*   If the command is an INQUIRY command, perform the command
/*   without using scatter/gather.  The Ciprico firmware can't
/*   do scatter/gather on this command when synchronous SCSI
/*   is used, and since the non scatter gather operation is likely to be a
/*   little more efficient for this command, don't bother to see
/*   if this controller is actually using synchronous SCSI.
/*
/*.End]=-------------------------------------------------------------*/

if (io_get_buffer_vector_byte_count(&arb_ptr->buffer_vector) != 0)
    {
    if (arb_ptr->scsi_cmd_blk.op_code == DEV_SCSI_CMD_INQUIRY)
            {
            dev_cisc_build_single_buffer(request_blk_ptr, param_blk_ptr);
            }
    else
            {
            dev_cisc_build_scatter_gather_arrays(request_blk_ptr, param_blk_ptr);
            }
    if (request_blk_ptr->arb_ptr->sync_io)
        {
        pm_disable_signal_delivery();
        }
    }
else
    {
    param_blk_ptr->flags = (bit8e_type)0;
    }
param_blk_ptr->target_id = arb_ptr->unit_spec.scsi_id;
param_blk_ptr->scsi_cmd_blk = arb_ptr->scsi_cmd_blk;
return;
}

/*.function */
```

```
                /*<-------------------*/
WIRED
status_type dev_cisc_check_status  (request_blk_ptr)
                /*>-------------------*/

dev_cisc_request_blk_ptr_type    request_blk_ptr; /* READ/WRITE */

/*.Summary[=-----------------------------------------------------------------
/*
/*   Perform command list request completion processing.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that holds
/*   information which specifies the I/O request that has completed.
/*
/*.Functional_Description
/*
/*   This function is called to perform the following command
/*   completion processing:
/*
/* Deallocate the scatter/gather arrays used to perform DMA.
/*
/* Interpret the completion status of the request and take
/*   the appropriate action.
/*
/* Release the command list request lock and get the next
/*   asynchronous request started if there is one.
/*
/*
/*.Return_Value
/*
/* OK -- The command executed successfully with no errors
/*      or exceptions.
/*
/* DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION -- The
/*   command completed with a Check Condition status and sense
/*   information has been returned.
/*
/* IO_ENXIO_UNIT_NOT_READY -- The device was busy and could
/*   not accept the command.
/*
/* IO_EIO_PHYSICAL_UNIT_FAILURE -- An unexpected and
/*   unrecoverable error occured during command execution.
/*
/* IO_EINTR_INTERRUPTED_BY_SIGNAL -- A process termination
/*   signal was received while waiting for command completion.
/*
/* IO_EIO_DEVICE_TIMED_OUT -- The timeout interval
/*   expired while waiting for command completion.
/*
[
status_type             status;
bit8e_type              error_code;
bit8e_type              scsi_status;
uint8_type              status_blk_index;
dev_adapter_request_block_ptr_type      arb_ptr;
uint32_type             byte_count;
dev_cisc_status_block_ptr_type          status_blk_ptr;
pointer_to_any_type                  sense_buffer_ptr;


/*.Implementation[=-------------------------------------------------
/*
/*   Release any scatter/gather arrays that were used to process
/*   the request. If the request was synchronous with DMA,
/*   turn signal delivery back on.
/*
/*   If the command was an INQUIRY, no scatter/gather array is involved.
/*
/*   If DMA was performed, the caller's buffer vector is updated
/*   to reflect the data transfer. Note that there is no way
/*   to verify with the hardware how much data was actually
/*   transferred. As a result,  we always return the number of bytes
/*   requested as the number of bytes transferred. Presumably,
```

```
/*   if an error occurred during the transfer, the caller will
/*   be able to determine how much data was actually transferred by
/*   examining the Request Sense buffer.
/*
/*.End]=-------------------------------------------------------------*/

arb_ptr = request_blk_ptr->arb_ptr;
byte_count = io_get_buffer_vector_byte_count(&arb_ptr->buffer_vector);
if (byte_count != 0)
    {
    if (arb_ptr->scsi_cmd_blk.op_code != DEV_SCSI_CMD_INQUIRY)
            {
            dev_cisc_free_scatter_gather_arrays(request_blk_ptr);
            }
    if (request_blk_ptr->arb_ptr->sync_io)
        {
        pm_enable_signal_delivery();
        }
    io_add_to_buffer_vector_position(&arb_ptr->buffer_vector,
                                (int32_type)byte_count);
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Update the adapter request block with the total amount of
/*   time that the target device spent on the request.
/*
/*.End]=-------------------------------------------------------------*/

arb_ptr->total_request_busy_time = request_blk_ptr->total_request_busy_time;

/*.Implementation_Continued[=----------------------------------------
/*
/*   If the request timed out release all resources held and return
/*   the status. If the request was aborted, just return the status,
/*   the async path will release the associated resources when the
/*   controller completes the request.
/*
/*.End]=-------------------------------------------------------------*/

if (request_blk_ptr->request_timed_out)
    {
    status = IO_EIO_DEVICE_TIMED_OUT;
    }
else if (request_blk_ptr->request_aborted)
    {
    return(IO_EINTR_INTERRUPTED_BY_SIGNAL);
    }
else
    {
/*.Implementation_Continued[=----------------------------------------
/*
/*  The request was not aborted or timed out, get the error code
/*   from the cisc status block interpret it. The interrupt service
/*  routine always increments the status block index after saving
/*  a status block so the index is one greater than the actual number
/*   saved.
/*
/*.End]=-------------------------------------------------------------*/

    status_blk_index = request_blk_ptr->status_blk_index - 1;
    status_blk_ptr = &request_blk_ptr->status_blk_array[status_blk_index];
    error_code = status_blk_ptr->error_code;
    if (error_code == DEV_CISC_STATUS_BLK_ERROR_NO_ERROR)
            {
        status = OK;
            }
    else if   ((error_code == DEV_CISC_STATUS_BLK_ERROR_BUS_TIMEOUT) ||
        (error_code == DEV_CISC_STATUS_BLK_ERROR_SELECT_TIMEOUT) ||
        (error_code == DEV_CISC_STATUS_BLK_ERROR_DISCONNECT_TIMEOUT))
            {
        status = IO_EIO_DEVICE_TIMED_OUT;
            }
    else if (error_code == DEV_CISC_STATUS_BLK_ERROR_BAD_SCSI_STATUS)
            {
```

```
/*.Implementation_Continued[=----------------------------------------
/*
/*  The SCSI device reported a command completion status that
/*  was something other than GOOD (0). Get the SCSI completion
/*  status of the request and interpret it. If the command completed
/*  with a Check Condition status copy the sense information into the
/*  caller's request packet.
/*
/*.End]=-------------------------------------------------------------*/

        scsi_status = status_blk_ptr->scsi_status;
        scsi_status >>= DEV_SCSI_STATUS_BYTE_SHIFT;
        scsi_status &= DEV_SCSI_STATUS_BYTE_MASK;
        switch (scsi_status)
            {
            case DEV_SCSI_STATUS_GOOD_STATUS:
                status = OK;
                break;
            case DEV_SCSI_STATUS_CHECK_CONDITION:
/*.Implementation_Continued[=--------------------------------
/*
/*  The command completed with a Check Condition status.
/*  Copy the sense data in the caller's request sense buffer.
/*  If the unit was set up to send more than eight bytes of
/*  sense data, we should have received two status blocks
/*  with eight bytes each.
/*
/*.End]=-----------------------------------------------------*/

                status = DEV_STATUS_SCSI_CMD_COMPLETE_CHECK_CONDITION;
                sense_buffer_ptr = (pointer_to_any_type)
                        &arb_ptr->sense_buffer;
                    misc_byte_copy((pointer_to_any_type)
                    request_blk_ptr->status_blk_array[0].sense_data,
                    sense_buffer_ptr,
                    DEV_CISC_REQUEST_SENSE_DATA_SIZE);
                    if (status_blk_index != 0)
                        {
                        misc_byte_copy((pointer_to_any_type)
                    request_blk_ptr->status_blk_array[1].sense_data,
                    (pointer_to_any_type)((uint32_type)sense_buffer_ptr
                     + DEV_CISC_REQUEST_SENSE_DATA_SIZE),
                    DEV_CISC_REQUEST_SENSE_DATA_SIZE);
                        }
                break;
            case DEV_SCSI_STATUS_BUSY:
                status = IO_ENXIO_UNIT_NOT_READY;
                break;
            default:
                status = IO_EIO_PHYSICAL_UNIT_FAILURE;
                break;
            }
            }
    else
            {
        status = IO_EIO_PHYSICAL_UNIT_FAILURE;
            }
    }

/*.Implementation_Continued[=----------------------------------------
/*
/*   Return the request block to the free queue and release the
/*   command list request lock. If there is an asynchronous request
/*   waiting on the queue, get the request started.
/*
/*.End]=-------------------------------------------------------------*/

(void)misc_enqueue_at_tail(&request_blk_ptr->dip->request_blk_queue,
                           &request_blk_ptr->links);
if (io_assign_next_interleave_waiter(&request_blk_ptr->dip->request_lock))
    {
    dev_cisc_start_async_request(request_blk_ptr->dip);
    }
return(status);
}
```

```
/*.function */

/*<---------------------------------*/
WIRED
void         dev_cisc_build_scatter_gather_arrays (request_blk_ptr, param_blk_ptr)
/*>---------------------------------*/

dev_cisc_request_blk_ptr_type    request_blk_ptr; /* READ/WRITE */
dev_cisc_param_block_ptr_type    param_blk_ptr;   /* READ/WRITE */

/*.Summary[=-------------------------------------------------------------
/*
/*   Set up the scatter/gather arrays required to perform DMA through
/*   the cisc.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that holds
/*   information which specifies the I/O request.
/*
/* param_blk_ptr -- A pointer to the Ciprico Rimfire 3500
/*   parameter block that is being used to issue the I/O request
/*   to the controller.
/*
/*.Functional_Description
/*
/*   This function allocates the scatter/gather arrays needed to
/*   specify an I/O operation and fills them in with the required
/*   information. A queue of the scatter/gather arrays used for
/*   the transfer is maintained so that they can be returned to
/*   the free pool after the operation has completed.
/*
/*.Return_Value
/*
/*   None.
/*
{

dev_adapter_request_block_ptr_type  arb_ptr;
boolean_type            is_user_buffer;
dev_cisc_scatter_gather_blk_ptr_type scatter_gather_blk_ptr;
dev_cisc_scatter_gather_blk_ptr_type previous_scatter_gather_blk_ptr;
uint32e_type            physical_address;
pointer_to_any_type                 buffer_ptr;
uint32_type                         buffer_end;
uint32_type                         page;
uint32_type                         data_length;

/*.Implementation[=-----------------------------------------------
/*
/*   Set up the scatter/gather arrays needed to specifiy the
/*   host physical pages that data is to be transferred to/from.
/*   One array entry is required for each physical page that is
/*   to be referenced. Each arrray can specify up to 8 physical
/*   pages in host memory.
/*
/*.End]=-----------------------------------------------------------*/

arb_ptr = request_blk_ptr->arb_ptr;
if (arb_ptr->request_flags & DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER)
    {
    is_user_buffer = FALSE;
    }
    else
    {
    is_user_buffer = TRUE;
    }
param_blk_ptr->flags = DEV_CISC_PARAM_BLK_FLAGS_SGO;
param_blk_ptr->address_modifier = request_blk_ptr->dip->vme_address_modifier;

io_get_buffer_vector_io_info(&arb_ptr->buffer_vector,
                             &buffer_ptr,
                             &param_blk_ptr->transfer_count);

buffer_end = (uint32_type)buffer_ptr + param_blk_ptr->transfer_count;
```

```
previous_scatter_gather_blk_ptr = DEV_CISC_SCATTER_GATHER_TERMINAL_LINK;
while   ((uint32_type)buffer_ptr < buffer_end)
    {
    misc_dequeue_from_head(&request_blk_ptr->dip->scatter_gather_blk_queue,
                    (misc_queue_links_ptr_type *)&scatter_gather_blk_ptr);
    vm_get_physical_byte_address((pointer_to_any_type)
                                 &scatter_gather_blk_ptr->header,
                                 FALSE,
                                 (byte_address_ptr_type)&physical_address);
    if   ((byte_address_type)physical_address ==
VM_INVALID_PHYSICAL_ADDRESS_PTR)
        {
        sc_panic(DEV_PANIC_ADDRESS_TRANSLATION_FAILED);
        }

    if (previous_scatter_gather_blk_ptr ==
            DEV_CISC_SCATTER_GATHER_TERMINAL_LINK)
            {
        param_blk_ptr->vme_memory_address = physical_address;
            }
    else
            {
        previous_scatter_gather_blk_ptr->header.next_sg_header =
                            physical_address;
            }
    (void)misc_enqueue_at_tail(&request_blk_ptr->used_scatter_gather_queue,
                               &scatter_gather_blk_ptr->links);
    previous_scatter_gather_blk_ptr = scatter_gather_blk_ptr;
    for (page = 0;   ((page < DEV_CIPRICO_MAX_SCATTER_GATHER_ARRAY_ENTRIES) &&
((uint32_type)buffer_ptr < buffer_end)); page++)
        {
        data_length = NUM_BYTES_PER_PAGE -   ((uint32_type)buffer_ptr &
                      PAGE_OFFSET_MASK);
        data_length = MINIMUM(data_length, (buffer_end - (bit32e_type)
                      buffer_ptr));
        scatter_gather_blk_ptr->header.sg_desc[page].address_modifier =
                                        request_blk_ptr->dip->vme_address_modifier;
        vm_get_physical_byte_address((pointer_to_any_type)buffer_ptr,
                                     is_user_buffer,
                                (byte_address_ptr_type)&physical_address);
        if   ((byte_address_type)physical_address ==
                          VM_INVALID_PHYSICAL_ADDRESS_PTR)
            {
            sc_panic(DEV_PANIC_ADDRESS_TRANSLATION_FAILED);
            }
        scatter_gather_blk_ptr->header.sg_desc[page].data_address =
                                    physical_address;
        scatter_gather_blk_ptr->header.sg_desc[page].data_length = data_length;
        buffer_ptr = (pointer_to_any_type)((uint32_type)buffer_ptr +
                     data_length);
        }
    }

/*.Implementation_Continued[=--------------------------------------
/*
/*   If the current scatter/gather array has not been completely
/*   used, mark the first free entry to indicate the end of
/*   data. Also, terminate the scatter/gather array chain by
/*   putting DEV_CIPRICO_SCATTER_GATHER_TERMINAL_LINK in the
/*   link field of the last scatter/gather array used.
/*
/*.End]=-----------------------------------------------------------*/

if (page < DEV_CIPRICO_MAX_SCATTER_GATHER_ARRAY_ENTRIES)
    {
    scatter_gather_blk_ptr->header.sg_desc[page].data_length = 0;
    }
    scatter_gather_blk_ptr->header.next_sg_header = (uint32e_type)
                            DEV_CISC_SCATTER_GATHER_TERMINAL_LINK;

return;
}

/*.function */
```

```
/*<-------------------------*/
WIRED
void         dev_cisc_build_single_buffer (request_blk_ptr, param_blk_ptr)
/*>-------------------------*/

dev_cisc_request_blk_ptr_type    request_blk_ptr; /* READ/WRITE */
dev_cisc_param_block_ptr_type    param_blk_ptr;   /* READ/WRITE */

/*.Summary[=-----------------------------------------------------------------
/*
/*   Set up for DMA to a single data buffer, i.e. a non scatter/gather
/*   DMA operation.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that holds
/*   information which specifies the I/O request.
/*
/* param_blk_ptr -- A pointer to the Ciprico Rimfire 3500
/*   parameter block that is being used to issue the I/O request
/*   to the controller.
/*
/*.Functional_Description
/*
/*   The physical address of the data buffer the flags controlling
/*   the operation, are set up in the command parameter block.
/*
/*.Remarks
/*
/*   The caller must assure that the data buffer doesn't cross a page
/*   boundary.
/*
/*.Return_Value
/*
/*   None.
/*
{
dev_adapter_request_block_ptr_type  arb_ptr;
boolean_type             is_user_buffer;
uint32e_type             physical_address;
pointer_to_any_type                 buffer_ptr;

/*.Implementation[=-----------------------------------------------
/*
/*   Get an arb pointer, and see if the buffer is in user or kernel
/*   space.
/*
/*   Get the buffer logical location and length, and translate
/*   the address to a physical location.
/*
/*   Set up the parameter block.
/*
/*.End]=-------------------------------------------------------------*/

arb_ptr = request_blk_ptr->arb_ptr;
if (arb_ptr->request_flags & DEV_SCSI_REQUEST_FLAGS_KERNEL_BUFFER)
    {
    is_user_buffer = FALSE;
    }
    else
    {
    is_user_buffer = TRUE;
    }

io_get_buffer_vector_io_info(&arb_ptr->buffer_vector,
                             &buffer_ptr,
                             &param_blk_ptr->transfer_count);

vm_get_physical_byte_address((pointer_to_any_type)buffer_ptr,
                         is_user_buffer,
                         (byte_address_ptr_type)&physical_address);
if   ((byte_address_type)physical_address ==
                         VM_INVALID_PHYSICAL_ADDRESS_PTR)
    {
    sc_panic(DEV_PANIC_ADDRESS_TRANSLATION_FAILED);
```

```
    }

param_blk_ptr->flags = DEV_CISC_PARAM_BLK_FLAGS_NONE;
param_blk_ptr->address_modifier = request_blk_ptr->dip->vme_address_modifier;
param_blk_ptr->vme_memory_address = physical_address;

return;
}

/*.function */

/*<-----------------------------------*/
WIRED
void         dev_cisc_free_scatter_gather_arrays (request_blk_ptr)
/*>-----------------------------------*/

dev_cisc_request_blk_ptr_type    request_blk_ptr; /* READ/WRITE */

/*.Summary[=-------------------------------------------------------------
/*
/*   Deallocate scatter/gather arrays used to perform DMA through
/*   the cisc.
/*
/*.Parameters
/*
/* request_blk_ptr -- A pointer to a request block that holds
/*   information which specifies the I/O request.
/*
/*.Functional_Description
/*
/*   This function returns the scatter/gather arrays used to
/*   specfiy an I/O operation to the free pool.
/*
/*.Return_Value
/*
/*   None.
/*
{

dev_cisc_scatter_gather_blk_ptr_typescatter_gather_blk_ptr;

/*.Implementation[=-----------------------------------------------------
/*
/*   Dequeue scatter/gather arrays from the "used" queue and
/*   return them to the "free" until a null queue element is
/*   dequeued from retured from the queue manager.
/*
/*.End]=-----------------------------------------------------------------*/

while (TRUE)
    {
    misc_dequeue_from_head(&request_blk_ptr->used_scatter_gather_queue,
                    (misc_queue_links_ptr_type *)&scatter_gather_blk_ptr);
    if   ((misc_queue_links_ptr_type)scatter_gather_blk_ptr ==
                        MISC_QUEUE_NULL_LINKS_PTR)
        {
            break;
        }
    (void)misc_enqueue_at_tail(&request_blk_ptr->dip->scatter_gather_blk_queue,
                        &scatter_gather_blk_ptr->links);
    }
return;
}

/*.function */

              /*<-------------------------------*/
WIRED
void           dev_cisc_complete_aborted_request  (data)
              /*>-------------------------------*/

bit32e_type             data;   /* READ ONLY  */

/*.Summary[=-------------------------------------------------------------
/*
```

```
/*   This function handles the completion of an aborted synchronous
/*   request that has been completed by the cisc controller.
/*
/*.Parameters
/*
/* data -- The 32 bits of data that was in the message
/*      given to the driver demon.
/*
/*.Functional_Description
/*
/*   This function handles the completion of aborted synchronous I/O
/*   requests. The ciprico controller cannot be told to abort commands.
/*   If a process executing a synchronous command receives a process
/*   termination signal, this driver returns to the caller immediately
/*   and completes the execution of the command through the I/O demon.
/*   Synchronous requests that do not require dma may be aborted. All
/*   other requests (async, sync with dma) are allowed to complete
/*   regardless of what signals are received.
/*
/*.Return_Value
/*
/*   None.
/*
/*.Exceptions
/*
/*      None.
/*
/*.Abort_Conditions
/*
/*      None.
/*
{
dev_cisc_request_blk_ptr_type                    request_blk_ptr;

/*.Implementation[=-----------------------------------------------
/*
/*   The controller has completed an aborted synchronous command.
/*   Return the request block to the free queue and release the
/*   command list request lock. If there is an asynchronous request
/*   waiting on the queue, get the request started.
/*
/*.End]=-------------------------------------------------------------*/

request_blk_ptr = (dev_cisc_request_blk_ptr_type)data;
(void)misc_enqueue_at_tail(&request_blk_ptr->dip->request_blk_queue,
                           &request_blk_ptr->links);
if (io_assign_next_interleave_waiter(&request_blk_ptr->dip->request_lock))
    {
    dev_cisc_start_async_request(request_blk_ptr->dip);
    }

return;
}
```

System File Entries

This section shows a partial listing of a system file with the cisc adapter driver used as part of the sd entry..

```
#
# System file
#

# drivers
sd(cisc(),0)
hken()
loop()
#syac()
prf()
meter()
```

Master File Entries

The following section shows a partial listing of the master file:

```
#-------------------------------------------------------------------
# Adapters:
# Your system must have at least one scsi adapter.
#
#
    insc        1          7        z
    cisc        5          7        z
#
#-------------------------------------------------------------------
```

End of Appendix

Appendix C
Standard Peripherals and Their Defaults

This appendix lists the default values for memory-mapped I/O addresses, interrupt levels, interrupt vectors, and SCSI IDs for AViiON system and AViiON station devices. It also describes the conventions for selecting default values for these variables on your new device.

AViiON System I/O Defaults

Table C-1 shows the device mnemonic for various standard devices with their default memory mapped I/O address, interrupt level, and interrupt vector for an AViiON system.

Table C-1 AViiON System I/O Address and Interrupt Level/Vector Defaults

Device	Base Address (in bytes)	Interrupt Level and Vector	Description
cied(0)	a16-0xffffef00 (512)	2 / 0x18	1st Ciprico ESDI Disk Controller
cied(1)	a16-0xffffff100 (512)	2 / 0x19	2nd "
cied(2)	a16-0xffffffb00 (512)	2 / 0x1A	3rd "
cied(3)	a16-0xffffffd00 (512)	2 / 0x1B	4th "
cimd(0) *	a16-0xffffef00 (512)	2 / 0x18	1st Ciprico SMD Disk Controller
cird(0) *	a16-0xffffef00 (512)	2 / 0x18	1st Ciprico ESDI or SMD Disk Controller
cisc(0)	a16-0xffffff300 (512)	2 / 0x28	1st Ciprico SCSI Adapter
cisc(1)	a16-0xffffff500 (512)	2 / 0x29	2nd "
cisc(2)	a16-0xffffff700 (512)	2 / 0x2A	2nd "
cisc(3)	a16-0xffffff900 (512)	2 / 0x2B	2nd "
hken(0)**	a16-0xffff4000 (4K)	3 / 0x15	1st Hawk LAN (A16 address)
	a32-0x55900000(512K)		(A32 address)
hken(1)**	a16-0xffff5000 (4K)	3 / 0x10	2nd Hawk LAN (A16 address)
	a32-0x55980000(512K)		(A32 address)
sdcp(0)	a32-0x55b00000 (4K)	3 / 0x50	1st Systech Synchronous Controller
sdcp(1)	a32-0x55b10000 (4K)	3 / 0x51	2nd "
syac(0)	a32-0x60000000 (128K)	4 / 0x60	1st Systech Asynchronous Controller
syac(1)	a32-0x60020000 (128K)	4 / 0x61	2nd "
syac(2)	a32-0x60040000 (128K)	4 / 0x62	3rd "
syac(3)	a32-0x60060000 (128K)	4 / 0x63	4th "
syac(4)	a32-0x60080000 (128K)	4 / 0x64	5th "

NOTE: * **cimd** (SMD disk) and **cied** (ESDI disk) devices share the same default interrupt vectors and base addresses. The default values for the first and second instances of each type are shown under the **cied** entries above. The

cird driver handles either SMD (**cimd**) or ESDI (**cied**) devices. Therefore, **cird** nodes use the default values shown under **cied** above. If you have both SMD and ESDI devices, use the **cird** mnemonic and treat the two types of disks as first and second instances of a **cird** device.

NOTE: ** Some devices, such as the Hawk LAN, require two I/O address areas.

The following conventions and restrictions apply to selecting your memory-mapped I/O address, interrupt level, and interrupt vector:

- The *a16* and *a32* notes in the Base Address column indicate in which VME data width address space this address falls. Addresses of the *a16*, *a24*, and *a32* data width areas are fixed by the kernel. Figure C-1 shows the location of these areas.

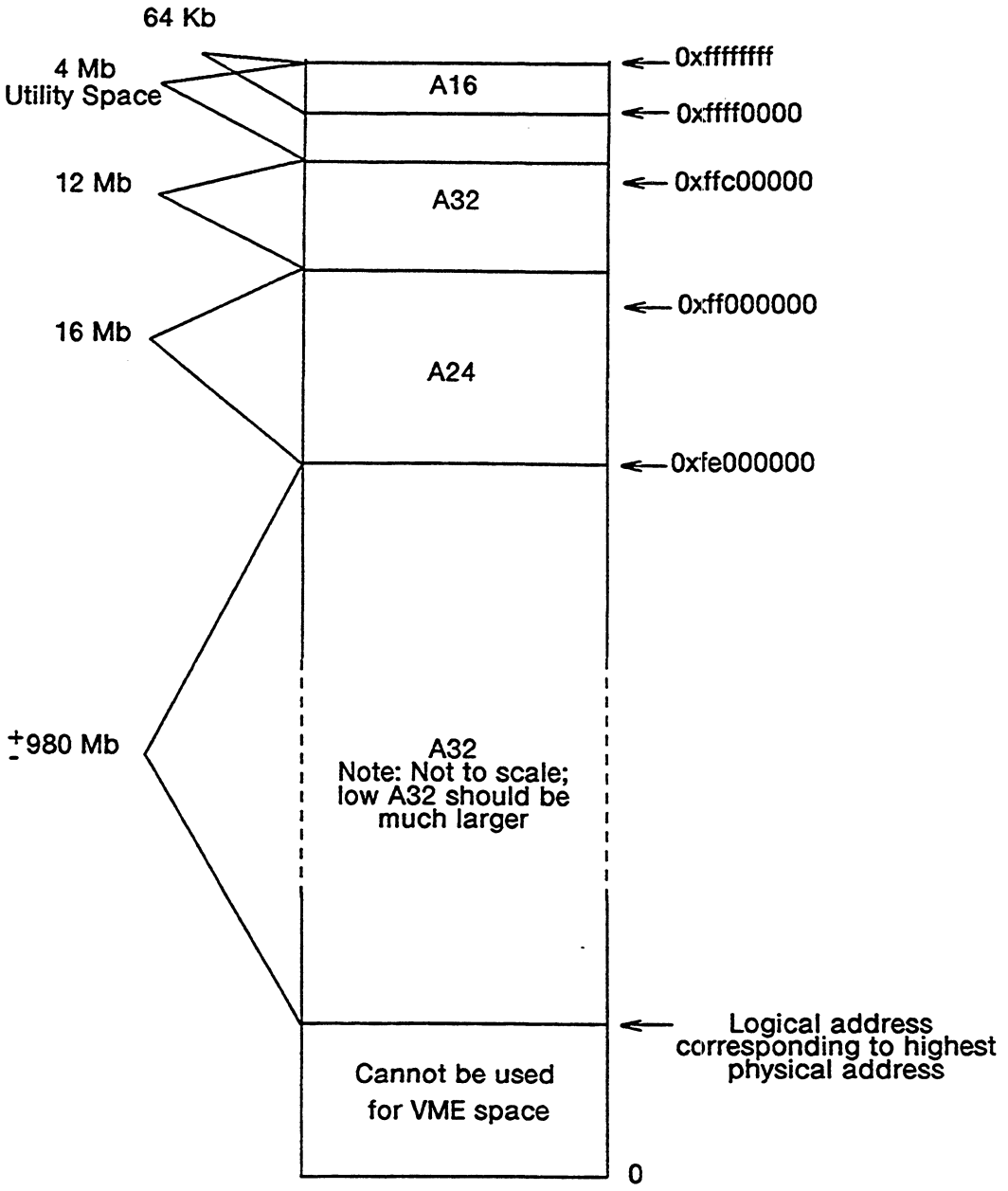

Figure C-1 AViiON System Memory-Mapped I/O Addresses and Data Width Areas

- In Table C-1 the parentheses following each base address show the number of bytes that the device uses. The base address plus the number of bytes equals the entire memory area reserved for the device.

- To select your base memory-mapped I/O address, you simply find an unreserved area of memory in the correct data width area. We recommend that you use the highest data width area possible to maximize device speed.

Thus, use Extended Addressing (a32) mode if possible and Standard Addressing (a24) mode as the first alternative. Use Short Addressing (a16) mode only if your device does not support any higher data width.

NOTE:

The a24 and a32 logical address space is used by the DG/UX kernel. Therefore, if you are using a memory mapped address in a24 or a32, you must map the a24/a32 physical address you want to another logical address. You do this using **mv_get_unwired_memory** to get a logical address and then using **vm_map_physical_memory** to map this address to the desired physical address.

- In setting your VME address modifiers, always use the Supervisory mode.

- On DG/UX systems, the standard interrupt levels for different devices are as follows:

```
2  for disks
3  for networks
4  for terminal controllers
2  for SCSI adapters
```

We recommend that you follow these defaults if you have one of the devices listed above. If you have a non-standard device, you may choose whichever interrupt level you want. Bear in mind that when you mask your device, you will be masking all others using the same interrupt level.

- The VME vector number uniquely identifies a controller or adapter for the whole VME bus. As with I/O addresses, vector numbers for Data General-supplied devices are pre-assigned and usually come correctly jumpered from the factory or are set by driver software. Unlike memory-mapped I/O addresses, the vector numbers of some devices can be set by the device driver at configuration time.

  On the AViiON system, you can select VME vector numbers from 0 through 255. To get a vector number for your device, simply refer to Table C-1 and select an unused vector number less than 255.

AViiON Station I/O Defaults

Table C-2 shows the device mnemonics for various standard devices with their default memory-mapped I/O address.

Table C-2 AViiON Station I/O Address Defaults

Mnemonic	Base Address (bytes)	Description
-	N/A	Power Fail
-	N/A	Parity Error
-	N/A	Z8536 C10 Interrupt
kbd	0xfff82800 (1K)	Keyboard
duart	0xfff82000 (255)	DUART
lp	0xfff82400 (1K)	Parallel Port
inen	0xfff8c000 (4K)	Ethernet Controller
insc	0xfff8a000 (4K)	SCSI Controller
-	N/A	DMA Terminal Count Reached
-	N/A	DMA Write Protect Error
-	N/A	DMA Valid Bit
grfx	0xfff89000 (4K)	Graphics
-	N/A	Software Interrupt

The following conventions and restrictions apply to selecting your memory-mapped I/O address.

- The parentheses following each base address show the number of bytes that the device uses. The base address plus the number of bytes equals the entire memory area reserved for the device.
- To select your base memory-mapped I/O address, you simply find an unreserved area of memory in the correct data width area, if appropriate for your machine.

SCSI IDs

Table C-3 shows the default SCSI IDs for various standard devices. These values are suggested defaults, not requirements. The kernel will allow an SCSI device to be configured at any SCSI ID between 0 and 6.

Table C-3 Default SCSI IDs

Device	SCSI ID
Disk 0	0
Disk 1	1
Disk 2	2
Disk 3	3
Tape 0	4
Tape 1	5
Tape 2	6
Reserved	7

If your system contains a cartridge tape, set the first instance of the tape to SCSI ID 4.

Device Specifications

A device specification contains a device driver name followed by a parenthesized list of optional parameters. The device name identifies the type of controller; the parameters provide additional information required to fully specify the device.

Table C-4 shows the device names and parameters used in device specification on the AViiON System.

Table C-4 AViiON System Device Specification Parameters

Device	Description	1st Parm.	2nd Parm.	3rd Parm.
VME Devices				
cied	Ciprico ESDI disk	controller number or specification	unit number (decimal)	N/A
cimd	Ciprico SMD disk	controller number or specification	unit number (decimal)	N/A
cird	Ciprico ESDI or SMD disk	controller number or specification	unit number (decimal)	N/A
hken	Interphase Hawk Ethernet	controller number or specification	host Ethernet address	N/A
sdcp	Systech synch board	controller number or specification	line	N/A
syac	Systech host adapter	adapter's device specification	line	N/A
SCSI Devices				
sd	SCSI disk	adapter's device specification	SCSI ID	unit #
st	SCSI tape	adapter's device specification	SCSI ID	file
cisc	Ciprico SCSI adapter	controller number or specification	N/A	N/A

Table C-5 AViiON Station Device Specification Parameters

Device	Description	1st Parm.	2nd Parm.	3rd Parm.
Integrated Bus Devices				
inen	integrated Ethernet	N/A	host Ethernet address	N/A
SCSI Devices				
sd	SCSI disk	adapter's device specification (insc)	SCSI ID	unit #
st	SCSI tape	adapter's device specification (insc)	SCSI ID	file
insc	integrated SCSI adapter	N/A	N/A	N/A

NOTE: N/A = reserved parameter that is taken to be zero.

NOTE: Ethernet device drivers use the "host" parameters during diskless booting.

By convention, most device names are four characters. For DG/UX drivers, the first two letters of the device mnemonics are generally two letters from the manufacturer's name (for example, *ci* for Ciprico) or manufacturer's trade name for the device (for example, *eg* for Eagle). The second two letters indicate the type of device. For example, *ed* is used for ESDI disk and *en* for ethernet. Thus, the Ciprico ESDI disk mnemonic is *cied*.

While you can give your device any unique mnemonic, for consistency we recommend that you follow similar naming conventions.

The parameter values are all zero-based. Therefore, cied(1,3) specifies the fourth drive of the second Ciprico ESDI controller in the system. The only drive on a system's only ESDI controller would be cied(0,0).

Parameters supply any information that may be required for a particular type of device. Default values are used if a parameter is null or missing, such as the first parameter in the specification **abcd(,1)** or the second and subsequent parameters in **abcd(0)**. The defaults are interpreted by the device driver itself.

By convention, all numerical parameters default to zero. An asterisk as a parameter represents all possible values for the parameter and is meaningful only in a few contexts. As with the defaults, the asterisk is interpreted by the driver.

Disk and Tape Command Set Compatibility

In order for a tape device to be compatible with the DG/UX SCSI tape driver (**st**), it must conform to the ANSI SCSI-1 Command Set specification (X3.131- 1986). In addition, it must also support the following commands listed as optional in the ANSI specification

- **Inquiry Command**
- **Space Command**
- **Test Unit Ready Command**
- **Mode Sense/Select Commands**

In order for a disk device to be compatible with the DG/UX SCSI disk driver (**sd**), it must conform to the ANSI SCSI-1 Command Set specification (X3.131- 1986). In addition, it must also support the following commands listed as optional in the ANSI specification

- **Inquiry Command**
- **Mode Sense/Select Commands**
- **Present/Allow Medium Removal Commands**
- **Read Capacity Command**
- **Test Unit Ready Command**

End of Appendix

Appendix D
Glossary

Device handle
The area defined by the driver that is beyond the interrupt service routine pointer.

Device Interrupt Table (DIT)
A list of addresses for each possible device code. When an interrupt occurs, the pointers in this table channel control to the correct interrupt service routine.

Device information structure
A structure that holds information relating to a specific device. The information can include timer values, unit numbers, block addresses, status, and word counts.

ESDI
An acronym for Enhanced Small Device Interface; the ANSI defined standard for computer and peripheral interconnection for magnetic and optical disks.

Host memory
Your main system's memory, which is not a part of a device controller (see *Local memory*).

Job processor (JP)
The actual physical processor (see *virtual processor*).

Jumper
A connecting switch that is manually set on a printed circuit board. For example, you will use jumpers to set your board's device code for CPU interrupts of a peripheral.

Kernel address space
Address space that requires Supervisory Access Privileges to access. This space is accessible to the kernel and not, as a rule, to user processes.

Local memory
Local memory is the memory in an intelligent board. This memory is a part of and is located on a controller board.

Logical addresses
: Logical addresses are the addresses that you see. They span the logical address space which is 4 gigabytes on AViiON machines. These addresses are mapped to physical addresses (by the system).

Medium Term Scheduler (MTS)
: The Medium Term Scheduler schedules processes onto the virtual processors, so that the processes can execute.

Page
: A page refers to an area of 4096 bytes of memory. Generally this is considered to be a piece of logical address space.

Page frame
: A page frame is the physical area on which a page is loaded or mapped; a page frame has a physical address.

Page Table Entry (PTE)
: The Page Table Entry is a list of pages used to translate logical addresses into physical addresses.

Physical addresses
: Physical addresses refer to unmapped, absolute addresses. Physical addresses are sometimes referred to as hardware addresses.

SCSI
: An acronym for Small Computer System Interface, which is an ANSI defined standard for computer and peripheral interconnection. It is a standard used where low-cost I/O interfacing is necessary.

Wired memory
: Wired memory is memory that cannot be paged out to disk (see *wired page*).

Wired page
: A wired page is a page that is bound to a page frame. Wired pages cannot be paged out to disk until they are unbound from the frame (unwired).

Unwired memory
: Unwired memory is memory that can be paged out to disk. A page fault must occur before the memory can be accessed after it has been paged out to disk.

User address space
: User address space refers to memory accessible to the owning user process. The kernel can also access this memory, but in general, other user processes cannot.

Virtual processor (VP)
An emulation of a physical processor, not the actual processor (see *job processor* (JP)).

End of Appendix

Documentation Set

This section lists documents relevant to the AViiON product line. The titles of Data General manuals are followed by nine-digit numbers used for ordering; you can order any of these manuals via mail or telephone (see the TIPS Order Form in the back of this manual). Following the list of Data General manuals are relevant documents published by other organizations (no ordering information is provided). Documents specifically referred to in the text of this manual are also listed in the "Related Documents" section of the Preface. *88open Binary Compatibility Standard* (069-701043)

Specifies a Binary Compatibility Standard (BCS).

88open Object Compatibility Standard (069-701044)

Specifies an Object Compatibility Standard (OCS) for operating systems based on Motorola MC88100 as well as future related microprocessors. Provides for portability of application-level software at the linkable level by specifying interfaces between the object file and the operating system libraries.

C: A Reference Manual (069-100226)

Describes lexical structure, the preprocessor, declarations, types, expressions, statements, functions, programs, and the run-time libraries.

Documenter's Tool Kit Technical Summary for the DG/UX™ System (069-701041)

Provides technical details about the tools supplied with the Documenter's Tool Kit; specifically, the **mm** macroinstruction package, the **tbl** text processor, and the nroff/troff formatter.

Green Hills Software User's Manual C-88000 (069-100230)

Describes the differences in the C programming language when run on an 88000 system.

Green Hills Software User's Manual Fortran-88000 (069-100232)

Describes differences in the FORTRAN programming language when run on an 88000 system.

Green Hills Software User's Manual Pascal-88000 (069-100231)

Describes differences in the Pascal programming language when run on an 88000 system.

IEEE Standard Portable Operating System Interface for Computer Environments (POSIX.1) (069-701045)

Specifies a POSIX standard.

Installing and Managing the DG/UX™ System (093-701052)

Shows how to install and manage the DG/UX operating system on AViiON hosts that will run as stand-alone, server, or client systems. Aimed at system administrators who are familiar with the UNIX operating system.

Learning the UNIX® Operating System (069-701042)

Helps beginners learn UNIX fundamentals through a step-by-step tutorial. (UNIX is a U.S. registered trademark of American Telephone and Telegraph Company.)

Managing NFS® and Its Facilities on the DG/UX™ System (093-701049)

Shows how to install, manage, and use the DG/UX ONC™/NFS product. Contains information on the Network File System (NFS), the Yellow Pages (YP), Remote Procedure Calls (RPC), and External Data Representation (XDR). (NFS is a U.S. registered trademark of Sun Microsystems, Inc. ONC is a trademark of Sun Microsystems, Inc.)

OSF/Motif™ Application Environment Specification (069-100326)

Specifies the interfaces that support the development of portable programs for OSF/Motif platforms.

OSF/Motif™ Programmer's Guide (069-100324)

A guide to programming using the various components of the OSF/Motif environment: the toolkit, window manager, and user interface language.

OSF/Motif™ Style Guide (069-100323)

Provides a framework for behavior specifications to guide application developers, widget developers, and window manager developers in the design of new products consistent with Presentation Manager and the OSF/Motif user interface.

Porting Applications to the DG/UX™ System (069-701059)

Describes how to port UNIX application programs to the DG/UX system.

POSIX.1 Conformance Document (069-701078)

Gives definitions and general requirements for conforming to the IEEE POSIX.1 standard.

Programmer's Reference for the DG/UX™ System (093-701055 and 093-701056)

Alphabetical listing of manual pages for programming commands on the DG/UX system. This two-volume set includes information on system calls, file formats, subroutines, and libraries.

Programmer's Reference for the X.25 Provider Interface on the DG/UX™ System (093-701082)

Describes how to use the data structures and messages of the X.25 Provider Interface in application programs.

Programming in the DG/UX™ System Application Environment (093-701076)

Discusses libraries, interprocess communications, programming interface, common object file format, and other programming-related topics.

Programming with TCP/IP on the DG/UX™ System (093-701024)

Describes how to program with the TCP and IP protocols and UDP interfaces.

Setting Up and Managing PAD on the DG/UX™ System (093-701073)

Tells you how to set up and manage the Packet Assembler/Disassembler (PAD) for AViiON Systems package. Also contains manual pages for the PAD package.

Setting Up and Managing TCP/IP on the DG/UX™ System (093-701051)

Explains how to prepare for the installation of Data General's TCP/IP (DG/UX) package on AViiON computer systems. Contains information on tailoring the software for your site, managing the system, and troubleshooting system problems.

Setting Up and Managing X.25 on the DG/UX™ System (093-701071)

This manual is for X.25 wide area network system administrators. It describes how to set up and manage the X.25 for AViiON Systems software. It also contains manual pages for the X.25 package.

STREAMS Primer for the DG/UX™ System (069-701033)

Defines STREAMS, a set of tools for developing DG/UX system communications services; explains how to build a stream; and discusses user-level and kernel-level functions.

STREAMS Programmer's Guide for the DG/UX™ System (069-701034)

Describes the development methods and design philosophy of STREAMS.

System Manager's Reference for the DG/UX™ System (093-701050)

Contains an alphabetical listing of manual pages for commands relating to system administration or operation.

User's Reference for the DG/UX™ System (093-701054)

Contains an alphabetical listing of manual pages for commands relating to general system operation.

Using API LU0,1,2,3 for AViiON™ Systems (093-000679)

Explains how to use the application program interface (API) for Logical Unit (LU) types 0, 1, 2, and 3 of IBM's System Network Architecture (SNA).

Using API LU6.2 for AViiON™ Systems (093-000680)

Explains how to use the application program interface (API) for Logical Unit (LU) type 6.2 of IBM's System Network Architecture (SNA).

Using PAD on the DG/UX™ System (069-701079)

Describes the user interface to the X.25 Packet Assembler/Disassembler (PAD) for AViiON Systems package.

Using SNA 3270 for AViiON™ Systems (093-000677)

Explains how to use the 3278 display and 3287 printer emulation capabilities within a multi-user environment.

Using SNA for AViiON™ Systems (093-000676)

Explains how to activate the SNA link to the host, establish node processes, and create configurations.

Using SNA/RJE for AViiON™ Systems (093-000678)

Explains how to use the 3776 emulation capabilities to submit jobs to and receive output from the host.

Using TCP/IP on the DG/UX™ System (093-701023)

Introduces Data General's implementation of the TCP/IP family of protocols and describes how to use the package.

Using the DG/UX™ Editors (069-701036)

Describes the text editors **vi** and **ed**, the batch editor **sed**, and the command line editor **editread**.

Using the DG/UX™ Kernel Debugger (093-701075)

Explains how to use the DG/UX kernel debugger to analyze the state of the kernel's internal data structures and the state of the underlying hardware's registers and memory.

Using the DG/UX™ Software Development Tools (093-701078)

Discusses programming support tools (**awk, nawk, lex, yacc, ld, lint,** and **as**), archiving, the C language, and SCCS.

Using the DG/UX™ System (069-701035)

Describes the DG/UX system and its major features, including **mailx**, the C shell, the Bourne shell, and the filing system.

Using the Documenter's Tool Kit on the DG/UX™ System (069-701039)

Provides a series of tutorials about the tools included in the Documenter's Tool Kit package. Describes the **mm** and **mv** macroinstruction packages; the **tbl, eqn, pic,** and **grap** preprocessors; the tools **checkmm, diffmk, hyphen, ndx,** and **subj**, and the nroff/troff formatter.

Writing a Device Driver for the DG/UX™ System (093-701053)

Describes how to write a device driver for a DG/UX system running on an AViiON computer. Describes the drivers written to address specific devices or adapters that manage secondary bus access to specific devices.

xlib Programming Manual (069-100227)

Explains programming concepts and techniques for the X library, which is the lowest level programming interface to the X Window System.

xlib Reference Manual (069-100228)

Provides a programmer's reference to the X library, including information about functions, event types, macroinstructions, and structures.

X Window System User's Guide (069-100229)

Explains the X Window System and common client applications, and describes how to customize the X environment.

To obtain any of the following documents, contact the indicated organization directly.

AIC-6250 High-Performance Protocol Chip data sheet (Adaptec)

Brooktree® Product Databook (Brooktree Corporation)

Local Area Controller Am7990 (LANCE) Technical Manual (Advance Micro Devices)

Memory Products Databook (SGS-Thompson Microelectronics)

Microprocessor Data Manual (Signetics)

The VMEbus Specification (Motorola)

uPD72120 Advanced Graphics Display Controller User's Manual (NEC, Inc.)

Z8536 Z-CIP/Z8536 CIO Counter/Timer and Parallel I/O Unit (Zilog, Inc.)

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

A

Adapter request block 3-14
Adapter-specific parameter block **3-10**, 3-14
Adding configurable parameters 2-4
Aliases 2-5

B

Buffer descriptors 4-7
Buffer vectors 3-13, 7-28
Building a new system image 2-6

C

Conf.c 2-4, 2-7
Config program 2-7
Configuration list 2-9
Constants and data structures
  for buffer vectors 7-29
  for eventcounters 6-3
  for system clock values 6-39
  for wired and unwired memory 7-3
  include file for 3-1
Creating a dev entry 8-9

D

dev_scsi_adapter_configure 5-3
dev_scsi_adapter_deregister_requester 5-9
dev_scsi_adapter_device_to_name 5-4
dev_scsi_adapter_get_device_info 5-12
dev_scsi_adapter_issue_async_command 5-11
dev_scsi_adapter_issue_command 5-10
dev_scsi_adapter_issue_command_physical_mode 5-13
dev_scsi_adapter_name_to_device 5-5
dev_scsi_adapter_open_dump 5-6
dev_scsi_adapter_register_requester 5-7
dev_scsi_adapter_set_unit_options 5-8
dev_xxx_close 4-18
dev_xxx_close_dump 4-38
dev_xxx_configure 1-17, 4-13
dev_xxx_deconfigure 4-40
dev_xxx_def.h 3-1
dev_xxx_deregister_requester 4-60
dev_xxx_driver.c 3-4
dev_xxx_get_device_info 4-63
dev_xxx_global_data.c 3-1
dev_xxx_init 4-12
dev_xxx_issue_async_command 4-62
dev_xxx_issue_command 4-61
dev_xxx_issue_command_physical_mode 4-64
dev_xxx_maddmap 4-45
dev_xxx_mmap 4-46
dev_xxx_munmap 4-47
dev_xxx_name 4-43
dev_xxx_open 1-17, 4-16
dev_xxx_open_dump 4-33
dev_xxx_read_dump 4-37
dev_xxx_read_write 4-22
dev_xxx_register_requester 4-58
dev_xxx_select 4-25
dev_xxx_service_interrupt 4-20
dev_xxx_set_unit_options 4-59
dev_xxx_start_io 4-29
dev_xxx_write_dump 4-35
Device
  adding to list of disks 8-10
Device code
  format of 8-2
Device handle 3-13, 4-3, **4-14**, 8-21, **D-1**
Device information structure 3-5, 3-11, 3-12, **D-1**
Device Interrupt Table (DIT) 3-13, **D-1**

Device numbers 3-12
Device specification structure 3-14
DG/UX system call
  ioctl 3-6, 7-21
  open 1-16
  read 3-6, 7-1, 7-28
  readv 7-1, 7-28
  select 3-6
  write 3-6, 7-1, 7-28
  writev 7-1, 7-28
Driver Daemon **3-17**
  number of messages 8-31
  queuing a message to 8-29

E

Encoding
  error statuses 8-35
err 3-17
Error Daemon 3-17
Errors
  encoding 3-17, 8-35
  logging 3-17
  system error file 3-17
  user-level 3-17
ESDI **D-1**
Eventcounter 6-2, 8-41
  converting into clock value 6-8
  name 6-3
  reading 6-9, 6-14
  value 6-3
Events
  defining 6-3

F

fs_check_self_id 8-48
fs_submit_dev_request 8-9

G

Generic Daemon **3-17**

H

Host memory **D-1**

I

I/O interfaces
  block 1-18
  character 1-18
I/O request
  asynchronous 4-5
  information 4-4
I/O routines 1-12
Include files 3-1, 3-2
Interfaces
  close 3-5, 4-18
  close_dump 3-6, 4-38
  complete_io 3-8, 4-31
  configure 3-4, 4-13
  deconfigure 3-6, 4-40
  device_to_name 3-7
  init 3-4, 4-12
  ioctl 3-6
  name_to_device 3-7, 4-43
  open 3-5, 4-16
  open_dump 3-6, 4-33
  powerfail 3-7
  read_write 3-5, 4-22
  select 3-6, 4-25
  service_interrupt 3-8, 4-20
  start_io 3-5, 3-8, 4-29
  write_dump 3-6, 4-35
Interrupt handler 1-19, 3-17
Interrupts 1-19, 3-17
  disabling 6-51
  enabling 6-52
  handling 6-46
  in a multiprocessor system 1-12
io_add_to_buffer_vector_position 7-30
io_add_to_register_list 8-10
io_allocate_device_number 8-11
io_buffer_vector_control_type 4-8
io_buffer_vector_type 4-6
io_check_device_spec 8-15
io_deallocate_device_number 8-13
io_deregister_device_info 8-14
io_do_first_long_board_access 8-18
io_do_first_short_board_access 8-17
io_err_log_error 8-39
io_forget_device_spec 8-16
io_get_buffer_vector_byte_count 7-35
io_get_buffer_vector_io_info 7-31
io_get_buffer_vector_position 7-33
io_get_buffer_vector_residual 7-34

io_get_device_info 8-19
io_hex_str_to_int 8-49
io_init_buffer_vector 7-36
io_init_one_entry_buffer_vector 7-37
io_map_device_number 8-21
io_mask_interrupt_variety 6-48
io_nodevice_madd.nap 8-75
io_nodevice_mmap 8-73
io_nodevice_munmap 8-74
io_nodevice_read_dump 8-70
io_parse_device_spec 8-23
io_perform_reset 8-25
io_queue_message_to_driver_demon 8-29
io_queue_message_to_generic_demon 8-32
io_read_from_buffer_vector 7-38
io_register_device_info 8-26
io_reset_buffer_vector_position 7-39
io_select_cancel 8-43
io_select_init 8-44
io_select_register 8-45
io_select_satisfy 8-46
io_set_buffer_vector_residual 7-40
io_specify_max_demon_messages 8-31
io_specify_max_generic_demon_messages 8-34
io_unmask_interrupt_variety 6-49
io_write_to_buffer_vector 7-41
ioctl 7-21

J

Job processor (JP) 1-11, **D-1**
Jumper **D-1**

K

Kernel address space **D-1**
Kernel completion routine 4-30
Kernel I/O completion routine 4-31

L

lm_initialize_sequenced_lock 6-29
lm_initialize_unsequenced_lock 6-30
lm_obtain_sequenced_lock 6-31
lm_obtain_sequenced_lock_no_wait 6-32
lm_obtain_unsequenced_lock 6-33
lm_release_sequenced_lock 6-34
lm_release_unsequenced_lock 6-35
lm_sequenced_lock_type 6-27
lm_unsequenced_lock_type 6-27
Local memory **D-1**
Locks 6-26
  initializing 6-29, 6-30
  releasing 6-35
  sequenced 6-26
  spin 6-26
  unsequenced 6-26
Logical addresses **D-2**

M

Major number 1-17, 2-3, 4-3, 8-21
Master file 1-13, **2-1**
  alias section 2-1, 2-5
  device section 2-1, 2-2
  keyword section 2-3
  keywords section 2-1
Medium Term Scheduler (MTS) **D-2**
Memory
  allocating 7-2
  global kernel 1-18
  per-process kernel 1-18
  releasing 7-2
  unwired 7-2, 7-6, 7-15, 7-17, 7-19
  wired 7-2, 7-7, 7-16, 7-18, 7-20
Minor number 1-17, 4-3, 4-14, 8-21
  assigning 8-11
misc_format_line 8-50
misc_obtain_spin_lock 6-36
misc_release_spin_lock 6-37
misc_spin_lock_type 6-28
Modes
  changing 8-3
Multiprocessors 1-12

N

Nodes **1-16**
  major number 1-17
  minor number 1-17

O

open 1-16
Operation record packet 3-14

P

Page **D-2**
Page faults 1-19
Page frame **D-2**
Page Table Entry (PTE) **D-2**
Panic 8-53
Physical addresses **D-2**
pm_get_my_pgrp 6-19
pm_get_my_pid 6-18
pm_is_interrupted 6-20
pm_is_super_user 8-52
pm_is_terminated 6-22
pm_send_signal_by_index 6-23
pm_send_signal_by_process_id 6-25
pm_signal_by_process_group 6-24
Pointers vectors
  driver-supplied 4-3

R

read 7-1, 7-28
Rebuilding the system 2-6
Registering device information 4-14
Request information packet 3-14
Routines
  dev_scsi_adapter_configure 5-3
  dev_scsi_adapter_deregister_requester 5-9
  dev_scsi_adapter_device_to_name 5-4
  dev_scsi_adapter_get_device_info 5-12
  dev_scsi_adapter_issue_async_command 5-11
  dev_scsi_adapter_issue_command 5-10
  dev_scsi_adapter_issue_command_physical_mode 5-13
  dev_scsi_adapter_name_to_device 5-5
  dev_scsi_adapter_open_dump 5-6
  dev_scsi_adapter_register_requester 5-7
  dev_scsi_adapter_set_unit_options 5-8
  dev_xxx_deregister_requester 4-60
  dev_xxx_get_device_info 4-63
  dev_xxx_issue_async_command 4-62
Routines *(cont.)*
  dev_xxx_issue_command 4-61
  dev_xxx_issue_command_physical_mode 4-64
  dev_xxx_maddmap 4-45
  dev_xxx_mmap 4-46
  dev_xxx_munmap 4-47
  dev_xxx_read_dump 4-37
  dev_xxx_register_requester 4-58
  dev_xxx_set_unit_options 4-59
  fs_check_self_id 8-48
  fs_submit_dev_request 8-9
  io_add_to_buffer_vector_position 7-30
  io_add_to_register_list 8-10
  io_allocate_device_number 8-11
  io_check_device_spec 8-15
  io_deallocate_device_number 8-13
  io_deregister_device_info 8-14
  io_do_first_long_board_access 8-18
  io_do_first_short_board_access 8-17
  io_err_log_error 8-39
  io_forget_device_spec 8-16
  io_get_buffer_vector_byte_count 7-35
  io_get_buffer_vector_io_info 7-31
  io_get_buffer_vector_position 7-33
  io_get_buffer_vector_residual 7-34
  io_get_device_info 8-19
  io_init_buffer_vector 7-36
  io_init_one_entry_buffer_vector 7-37
  io_map_device_number 8-21
  io_mask_interrupt_variety 6-48
  io_nodevice_maddmap 8-75
  io_nodevice_mmap 8-73
  io_nodevice_munmap 8-74
  io_nodevice_read_dump 8-70
  io_queue_message_to_driver_demon 8-29
  io_queue_message_to_generic_demon 8-32
  io_read_from_buffer_vector 7-38
  io_register_device_info 8-26
  io_reset_buffer_vector_position 7-39
  io_select_cancel 8-43
  io_select_init 8-44
  io_select_register 8-45
  io_select_satisfy 8-46
  io_set_buffer_vector_residual 7-40
  io_specify_max_demon_messages 8-31

Routines *(cont.)*
io_specify_max_generic_demon_messages 8-34
io_unmask_interrupt_variety 6-49
io_write_to_buffer_vector 7-41
lm_initialize_sequenced_lock 6-29
lm_initialize_unsequenced_lock 6-30
lm_obtain_sequenced_lock 6-31
lm_obtain_sequenced_lock_no_wait 6-32
lm_obtain_unsequenced_lock 6-33
lm_release_sequenced_lock 6-34
lm_release_unsequenced_lock 6-35
lm_sequenced_lock_type 6-27
lm_unsequenced_lock_type 6-27
misc_format_line 8-50
misc_obtain_spin_lock 6-36
misc_release_spin_lock 6-37
misc_spin_lock_type 6-28
pm_get_my_pgrp 6-19
pm_get_my_pid 6-18
pm_is_interrupted 6-20
pm_is_super_user 8-52
pm_is_terminated 6-22
pm_send_signal_by_index 6-23
pm_send_signal_by_process_id 6-25
pm_signal_by_process_group 6-24
sc_check_access_and_read_string_from_user 7-22
sc_check_byte_access 7-24
sc_panic 8-53
sc_read_bytes_from_user 7-25
sc_write_bytes_to_user 7-26
sc_write_string_to_user 7-27
vm_get_unwired_memory 7-6
vm_get_wired_memory 7-7
vm_mark_mod_and_ref_and_unwire_memory 7-13
vm_mark_ref_and_unwire_memory 7-14
vm_perhaps_get_unwired_memory 7-15
vm_perhaps_get_wired_memory 7-16
vm_release_unwired_memory 7-17
vm_release_wired_memory 7-18
vm_unwire_memory 7-19
vm_wire_memory 7-20
vp_add_to_ec_value 6-4
vp_advance_ec 6-5
Routines *(cont.)*
vp_are_ec_values_equal 6-16
vp_are_interrupts_disabled 6-50
vp_await_ec 6-6
vp_cancel_timeout 6-42
vp_convert_clock_value_to_ec_value 6-7
vp_convert_ec_value_to_clock_value 6-8
vp_create_clock_event 6-44
vp_disable_interrupts 6-51
vp_enable_interrupts 6-52
vp_establish_timeout 6-41
vp_get_next_ec_value 6-9
vp_has_event_occurred 6-10
vp_increment_ec_value 6-11
vp_initialize_ec 6-12
vp_initialize_sequencer 6-13
vp_read_ec 6-14
vp_read_system_clock 6-45
vp_specify_max_timeouts 6-43
vp_ticket_sequencer 6-15
Routines vector 3-12

S

sc_check_access_and_read_string_from_user 7-22
sc_check_byte_access 7-24
SC_ENCODE_STATUS 8-38
sc_panic 8-53
sc_read_bytes_from_user 7-25
sc_write_bytes_to_user 7-26
sc_write_string_to_user 7-27
SCSI **D-2**
SCSI ID **1-15**
SCSI unit numbers 1-15
select 3-6
Select list 8-46
initializing 8-44
registering a select 8-45
removing processes from 8-43
Select manager 8-41
Signals 1-20, 6-17
handling 6-20
termination 6-22
Special files 1-16, 2-9
Status encoding 8-35
Statuses 3-17

Superuser permission 8-52
Synchronization 6-2
sysadm 2-6
Syslog.conf 3-17
Syslogd 3-17
System clock
  managing 6-38
  returning value of 6-45
System error file
  syslog.conf 3-17
System file 1-13, 2-5
  device selection section 2-5
  tunable parameters section 2-5

T

Timeout
  cancelling 6-42
  establishing 6-41

U

Unit numbers 1-15
Unwired memory **D-2**
  allocating 7-6, 7-15
  releasing 7-17
User address space **D-2**

V

Virtual processor (VP) 1-11, 6-2, 8-28, **D-3**
vm_get_physical_byte_address 7-5
vm_get_unwired_memory 7-6
vm_get_wired_memory 7-7
vm_map_physical_memory 7-8
vm_mark_mod_and_ref_and_unwire_memory 7-13
vm_mark_ref_and_unwire_memory 7-14
vm_perhaps_get_unwired_memory 7-15
vm_perhaps_get_wired_memory 7-16
vm_release_unwired_memory 7-17
vm_release_wired_memory 7-18
vm_unmap_physical_memory 7-11
vm_unwire_memory 7-19
vm_wire_memory 7-20
vp_add_to_ec_value 6-4
vp_advance_ec 6-5
vp_are_ec_values_equal 6-16
vp_are_interrupts_disabled 6-50
vp_await_ec 6-6
vp_cancel_timeout 6-42
vp_convert_clock_value_to_ec_value 6-7
vp_convert_ec_value_to_clock_value 6-8
vp_create_clock_event 6-44
vp_disable_interrupts 6-51
vp_enable_interrupts 6-52
vp_establish_timeout 6-41
vp_get_next_ec_value 6-9
vp_has_event_occurred 6-10
vp_increment_ec_value 6-11
vp_initialize_ec 6-12
vp_initialize_sequencer 6-13
vp_read_ec 6-14
vp_read_system_clock 6-45
vp_specify_max_timeouts 6-43
vp_ticket_sequencer 6-15

W

Wired memory **D-2**
  allocating 7-7, 7-16
  releasing 7-18
Wired page **D-2**
write 7-1, 7-28

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to:
      Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581-9973

   b) TELEPHONE – Call TIPS at (508) 870-1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1-4 Units	$5.00
5-10 Units	$8.00
11-40 Units	$10.00
41-200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1-$149.99	0%
$150-$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870-1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME _____	COMPANY NAME _____
ATTN: _____	ATTN: _____
ADDRESS _____	ADDRESS (NO PO BOXES) _____
CITY _____	CITY _____
STATE _____ ZIP _____	STATE _____ ZIP _____

Priority Code _____ (See label on back of catalog)

_____ _____ _____ _____ _____
Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) _____

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is _____. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

_____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Writing a Device Driver for the DG/UX™ System

093-701053-03

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580

093-701053-03