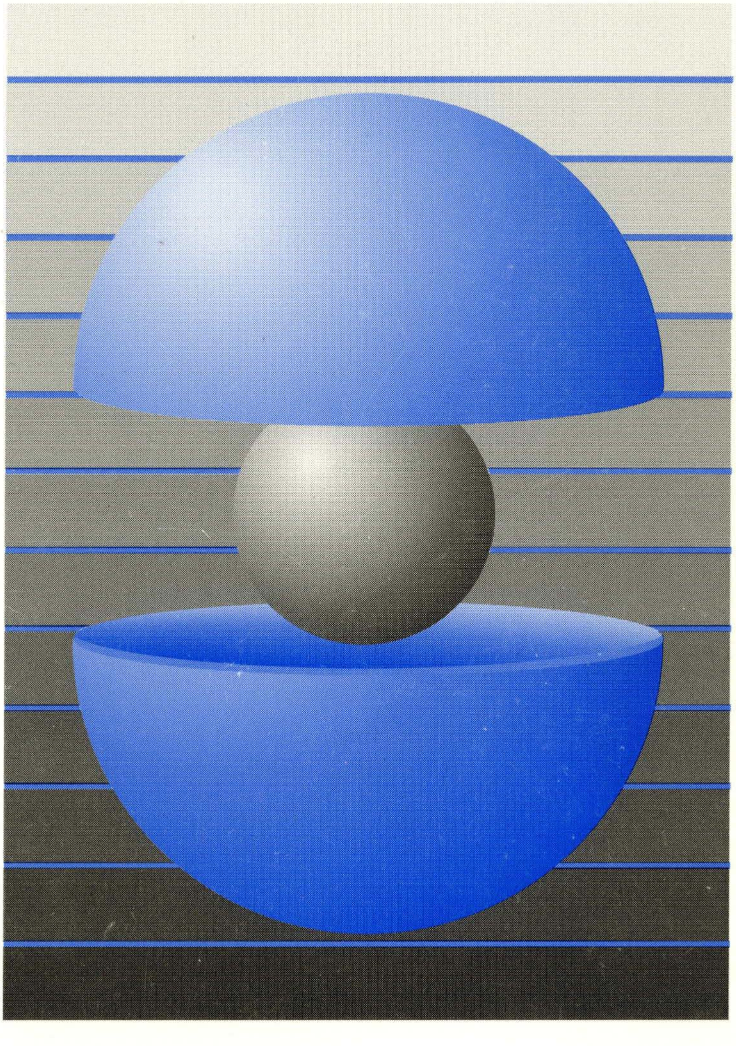
DEC OSF/1

Writing Device Drivers for the SCSI/CAM Architecture Interfaces



Part Number: AA-PS3GB-TE

DEC OSF/1

Writing Device Drivers for the SCSI/CAM Architecture Interfaces

Order Number: AA-PS3GB-TE

February 1994

Product Version: DEC OSF/1 Version 2.0 or higher

This manual contains information on how to write device drivers for the SCSI/CAM Architecture interfaces.

digital equipment corporation
Maynard, Massachusetts

Contents

2 CAM User Agent Modules

3 S/CA Common Modules

4 S/CA Generic Modules

5 CAM Data Structures

6 SCSI/CAM Configuration Driver Modules

7 CAM XPT I/O Support Routines

8 CAM SIM Modules

9 S/CA Error Handling

10 S/CA Debugging Facilities

11 Programmer-Defined SCSI/CAM Device Drivers

12 SCSI/CAM Special I/O Interface

A Header Files Used by Device Drivers

B SCSI/CAM Utility Program

C SCSI/CAM Routines

Examples

Figures

Tables

About This Manual

This manual contains information needed by systems programmers who write device drivers for the SCSI/CAM Architecture interfaces.

Audience

This manual is intended for systems programmers who:

- Develop programs in the C language using standard library routines
- Know one or more UNIX shells, other than `csh`
- Understand basic DEC OSF/1 components such as the kernel, shells, processes, configuration, autoconfiguration, and so forth
- Understand how to use the DEC OSF/1 programming tools, compilers, and debuggers
- Develop programs in an environment that includes dynamic memory allocation, linked list data structures, multitasking and symmetric multiprocessing (SMP)
- Understand the hardware device for which the driver is being written

Organization

This manual is organized as follows:

Chapter 1	SCSI/CAM Software Architecture Presents an overview of the DEC OSF/1 SCSI/CAM Architecture (S/CA).
Chapter 2	CAM User Agent Modules Describes the User Agent routines provided by Digital for SCSI/CAM peripheral device driver writers.
Chapter 3	S/CA Common Modules Describes the common data structures, routines, and macros provided by Digital for SCSI/CAM peripheral device driver writers.
Chapter 4	S/CA Generic Modules Describes the generic routines provided by Digital for SCSI/CAM peripheral device driver writers.

Chapter 5	CAM Data Structures Describes members of the CAM data structures used by SCSI device drivers.
Chapter 6	SCSI/CAM Configuration Driver Modules Describes the CAM Configuration driver data structures and routines that call the initialization routines in all the CAM subsystem modules.
Chapter 7	CAM XPT I/O Support Routines Discusses the Transport (XPT) layer routines used with SCSI device drivers.
Chapter 8	CAM SIM Modules Discusses the data structures and routines used with the SCSI Interface Module (SIM) layers that interface with the CAM subsystem.
Chapter 9	S/CA Error Handling Discusses the macro, data structures, and routines supplied by Digital for error handling in SCSI/CAM device drivers.
Chapter 10	S/CA Debugging Facilities Describes the debugging routines supplied by Digital for SCSI/CAM peripheral device driver writers.
Chapter 11	Programmer-Defined SCSI/CAM Device Drivers Describes and provides examples of how programmers can define SCSI/CAM device drivers.
Chapter 12	SCSI/CAM Special I/O Interface Describes and provides examples of the SCSI/CAM special I/O interface supplied by Digital to process special SCSI I/O commands.
Appendix A	Header Files Used by SCSI/CAM Device Drivers Summarizes the header files used by SCSI/CAM device drivers.
Appendix B	The SCSI/CAM Utility (SCU) Describes the SCSI/CAM Utility (SCU) used for maintenance and diagnostics of SCSI peripheral devices and the CAM subsystem.
Appendix C	SCSI/CAM Routines in Reference Page Format Provides more detailed descriptions of the S/CA routines in reference page format.
Appendix D	Sample Generic CAM Peripheral Driver Contains the header file and source file for a sample generic CAM peripheral driver.

Related Documentation

The printed version of the DEC OSF/1 documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General Users	G	Teal
System Administrators	S	Red
Network Administrators	N	Yellow
Programmers	P	Blue
Reference Page Users	R	Black

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the books in the DEC OSF/1 documentation set.

Readers of this guide are assumed to be familiar with the following documents:

- American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module*, working draft, X3T9.2/90-186

  Terms used throughout this guide, such as CAM Control Block (CCB), are defined in the American National Standard document. Copies can be purchased from Global Engineering, 2805 McGaw St, Irvine, CA 92714. (Telephone: 800-854-7179)

- American National Standard for Information Systems, *Small Computer Systems Interface - 2* (SCSI - 2), X3.131-199X

The following documents contain information that pertains to writing device drivers:

- *Writing Device Drivers, Volume 1: Tutorial*

  This manual provides information for systems engineers who write device drivers for hardware that runs the DEC OSF/1 operating system. Systems engineers can find information on driver concepts, device driver interfaces, kernel interfaces used by device drivers, kernel data structures, configuration of device drivers, and header files related to device drivers.

- *Writing Device Drivers, Volume 2: Reference*

  This manual contains descriptions of the header files, kernel support interfaces, `ioctl` commands, global variables, data structures, device driver interfaces, and bus configuration interfaces associated with device drivers. The descriptions are formatted similar to the DEC OSF/1 reference pages.

- *System Administration*

  This manual describes how to configure, use, and maintain the DEC OSF/1 operating system. It includes information on general day-to-day activities and tasks, changing your system configuration, and locating and eliminating sources of trouble.

  This manual is for the system administrators responsible for managing the operating system. It assumes a knowledge of operating system concepts, commands, and configurations.

- *Kernel Debugging*

  This manual provides information on debugging a kernel and analyzing a crash dump of a DEC OSF/1 operating system. The manual provides an overview of kernel debugging and crash dump analysis and describes the tools used to perform these tasks. The manual includes examples with commentary that show how to analyze a running kernel or crash dump. The manual also describes how to write a `kdbx` utility extension and how to use the various utilities for exercising disk, tape, memory, and communications devices.

  This manual is for system administrators responsible for managing the operating system and for systems programmers writing applications and device drivers for the operating system.

Reader's Comments

Digital welcomes your comments on this or any other DEC OSF/1 manual. You can send your comments in the following ways:

- Internet electronic mail:
  `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32
- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of each printed DEC OSF/1 manual.

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

Conventions

This document uses the following conventions:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
`%` **`cat`**	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

SCSI/CAM Software Architecture 1

This chapter provides an overview of the DEC OSF/1 Small Computer System Interface (SCSI) Common Access Method (CAM) Architecture (S/CA), which is a reliable, maintainable, and high performance SCSI subsystem based on the industry-standard CAM architecture. Readers of this guide should be familiar with the following documents:

- American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module*, working draft, X3T9.2/90-186

Terms used in this guide, such as CAM Control Block (CCB), are defined in that document. Copies can be purchased from Global Engineering, 2805 McGaw St, Irvine, CA 92714. (Telephone: 800-854-7179)

- American National Standard for Information Systems, *Small Computer Systems Interface - 2* (SCSI - 2), X3.131-199X

Readers should also be familiar with the following two manuals that are part of the DEC OSF/1 documentation:

- *Writing Device Drivers, Volume 1: Tutorial*
- *Writing Device Drivers, Volume 2: Reference*

This chapter describes the following:

- CAM and DEC OSF/1 S/CA environment models
- User Agent driver
- SCSI/CAM peripheral device driver routines:
  - CAM common routines supplied by Digital
  - Generic routines supplied by Digital
  - SCSI disk device routines
  - SCSI tape device routines
  - SCSI CD-ROM/AUDIO device commands
  - SCSI/CAM Special I/O interface
- CAM Configuration driver

- CAM Transport layer
- SCSI Interface Module (SIM)

1.1 Overview

The CAM architecture defines a software model that is layered, providing hardware independence for SCSI device drivers and SCSI system software. In the CAM model, which is illustrated in Figure 1-1, a single SCSI/CAM peripheral driver controls SCSI devices of the same type, for example, direct access devices. This driver communicates with a device on the bus through a defined interface. Using this interface makes a SCSI/CAM peripheral device driver independent of the underlying SCSI Host Bus Adapter (HBA).

This hardware independence is achieved by using the Transport (XPT) and SCSI Interface Module (SIM) components of CAM. Because the XPT/SIM interface is defined and standardized, users and third parties can write SCSI/CAM peripheral device drivers for a variety of devices and use existing operating system support for SCSI. The drivers do not contain SCSI HBA dependencies; therefore, they can run on any hardware platform that has an XPT/SIM interface present.

Figure 1-1: CAM Environment Model

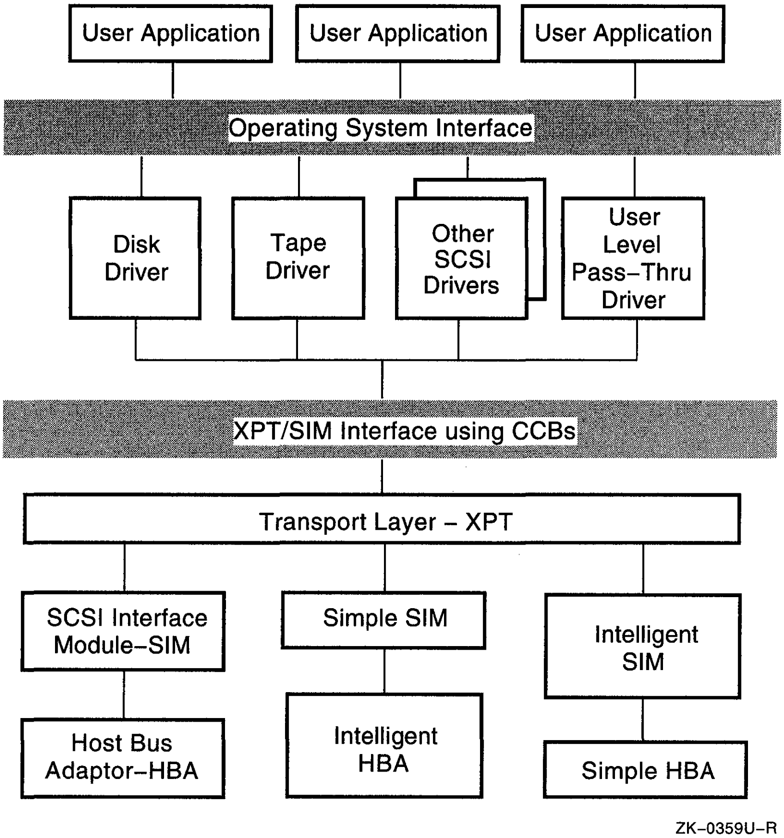
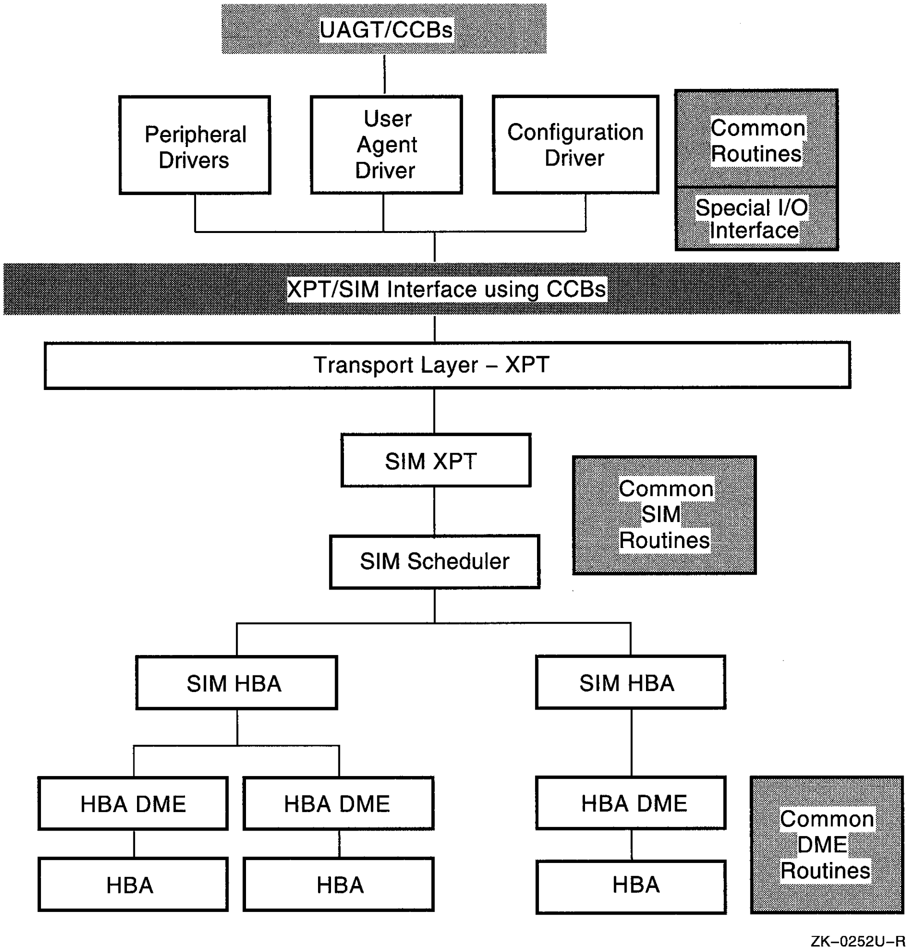


Figure 1-2 illustrates the DEC OSF/1 SCSI/CAM implementation of that model.

Figure 1-2: SCSI/CAM Architecture Implementation Model



1.2 CAM User Agent Device Driver

The User Agent driver routes user-process CAM Control Block (CCB) requests to the XPT for processing. The CCB contains all information required to fulfill the request. The user process calls the User Agent indirectly, using the `ioctl`(2) system call. A new User Agent CCB is allocated by a call to the XPT layer, and the user-process CCB information is copied into kernel space. The new CCB is filled in with the CCB values from the user process. If necessary, the user data areas are locked in memory. The CCB is then sent to the CAM subsystem for processing.

When the request has completed, the User Agent driver's completion routine is called. That routine performs all necessary cleanup operations and notifies the user process that the request is complete.

The User Agent allows multiple processes to issue CCBs, so there may be multiple processes sleeping on the User Agent. All CCBs are queued at the SIM layer.

1.3 SCSI/CAM Peripheral Device Drivers

SCSI/CAM peripheral device drivers convert operating system requests, such as user-process reads or writes, into CAM requests that the SCSI/CAM subsystem can process. Each type of SCSI/CAM peripheral driver is responsible for a specific class of SCSI device, such as SCSI tape devices. The SCSI/CAM peripheral driver handles error codes and conditions for its SCSI device class.

SCSI/CAM peripheral drivers convert input/output (I/O) requests into CAM Control Blocks (CCBs) that contain SCSI Command Descriptor Blocks (CDBs). CCBs are presented to the underlying transport layer, XPT, to initiate I/O requests. SCSI/CAM peripheral drivers implement SCSI device error recovery, for example, dynamic bad block replacement (DBBR). The SCSI device driver has no access to SCSI device control and status registers (CSRs) and receives no SCSI device interrupts.

The major/minor device-number pair, which is 32 bits wide, is used as an argument when creating the device special file associated with a specific SCSI device and is contained in the `buf` structure when accessing the device in raw or blocked mode. Figure 1-3 shows how the 32 bits are allocated.

Figure 1-3: Major/Minor Device-Number Pair

31 20	19 14	13 10	9 6	5 0
Major Index	Bus #	Target ID	LUN	Device Specific

ZK-0403U-R

This section provides overviews of the following:

- Common SCSI device driver modules
- Generic SCSI device driver modules
- SCSI disk device driver modules
- SCSI tape device driver modules

- SCSI CD-ROM/AUDIO device driver modules

Chapters 3, 4, and 11 describe the data structures and the routines associated with each module.

1.3.1 S/CA Common Device Driver Modules

The common SCSI device driver structures and routines can be shared among all the SCSI/CAM peripheral drivers written by device driver writers for DEC OSF/1. Using these common routines can speed the process of writing a SCSI device driver because they are routines that any SCSI device driver can use to perform operations.

1.3.2 S/CA Generic Device Driver Modules

Digital supplies predefined data structures and formats that SCSI device driver writers can use to write generic SCSI/CAM peripheral device drivers. These data structures and formats can be used in conjunction with the common routines.

1.3.3 CAM SCSI Disk Device Driver Modules

The SCSI/CAM peripheral disk driver supports removable (floppy) and nonremovable direct access SCSI disk devices and CD-ROM devices. The user interface consists of the major/minor device number pair and the `ioctl` commands supported by the SCSI disk device driver. The SCSI disk device driver also uses the common routines.

1.3.4 CAM SCSI Tape Device Driver Modules

The SCSI tape device structures and routines are exclusive to the SCSI/CAM peripheral tape driver. The user interface consists of the major/minor device number pair and the `ioctl` commands supported by the SCSI tape device driver. The SCSI tape device driver also uses the common routines.

1.3.5 CAM SCSI CD-ROM/AUDIO Device Driver Modules

The SCSI CD-ROM/AUDIO device commands, which are described in Chapter 11, use the SCSI CD-ROM/AUDIO device structures. The SCSI CD-ROM/AUDIO device driver also uses the common routines.

1.4 SCSI/CAM Special I/O Interface

The S/CA software includes an interface developed to process special SCSI I/O control commands used by the existing Digital SCSI subsystem and to aid in porting new or existing SCSI device drivers from other vendors to the

S/CA. With the SCSI/CAM special I/O interface, SCSI/CAM peripheral driver writers do not need detailed knowledge of either the system-specific or the CAM-specific structures and routines used to issue a SCSI command to the CAM I/O subsystem.

1.5 The SCSI/CAM Configuration Driver

The Configuration driver is responsible for configuring and initializing the CAM subsystem. This driver is also responsible for maintaining the `cam_edt[]` information structure.

When the system powers up, the Configuration driver initializes the local and global CAM subsystem data structures. The Configuration driver also calls the XPT and SIM initialization routines. When the subsystems are initialized, the Configuration driver performs a SCSI-bus scan by sending the SCSI Device Inquiry command. The `edt_dir[]` structure contains pointers to the EDT (Equipment Device Table) structure for each bus. The EDT contains the returned SCSI inquiry data for the SCSI/CAM peripheral drivers to access. The drivers, using the XPT_GDEV_TYPE and XPT_SDEV_TYPE get and set device information CCBs and can access the data contained in `cam_edt[]`.

1.6 CAM Transport Layer (XPT)

The CAM Transport layer, XPT, handles the CAM requests from the SCSI/CAM peripheral drivers and routes them to the appropriate SIM module. The XPT provides routines which are called by the SCSI/CAM peripheral driver to allocate and deallocate CAM control blocks (CCBs). In addition, the XPT provides routines that are used to initiate requests to the SIM and to issue asynchronous callbacks.

1.7 SCSI Interface Module Layers (SIM)

The SCSI Interface Module, SIM, has the most interaction with the SCSI bus protocol, timings, and other hardware-specific operations. Although this is a single component in the CAM model, it is divided into four logical sublayers in DEC OSF/1:

- SIM XPT – The SIM layer that interfaces to the XPT to initiate I/O on behalf of the SCSI/CAM peripheral drivers
- SIM SCHEDULER – The SIM layer that schedules requests to the SIM HBAs
- SIM HBA – The SIM layer that contains the HBA device-specific information

- SIM DME – A low level layer that contains the architecture-specific data-movement code

CAM User Agent Modules 2

This chapter describes the functions of the DEC OSF/1 User Agent SCSI device driver. It also describes the User Agent data structures and routines used by the User Agent SCSI device driver.

2.1 User Agent Introduction

The DEC OSF/1 User Agent SCSI device driver lets device driver writers write an application program to build a CAM Control Block (CCB) request. The User Agent driver lets the user-process request pass through to the XPT layer for processing. This gives user processes access to the SCSI/CAM subsystem and to all types of SCSI/CAM peripheral devices attached to the system.

This is a simple method for passing the CCB's SCSI request to the devices using the SIMs. The kernel does not have to be rebuilt if the device driver writer wants to change values within the CCBs.

The CCB contains all the information required to perform the request. The user process must first open the user agent driver using `/dev/cam` to obtain a file descriptor. The user process calls the User Agent SCSI device driver using the `ioctl` system call. See `ioctl`(2) for more information. The User Agent ioctl routine, `uagt_ioctl`, is called through the device switch table, which is indexed by the major device number of the User Agent driver specified in the file descriptor obtained from the `open` system call passed in the `ioctl` call. The `ioctl` commands supported by the User Agent SCSI device driver are: DEVIOCGET, which returns the SCSI device driver status; UAGT_CAM_IO, which sends the specified CCB to the XPT layer for processing; UAGT_CAM_SINGLE_SCAN, which causes the scan of a bus, target, and LUN; and UAGT_CAM_FULL_SCAN, which causes the scan of a bus.

A CCB is allocated in the kernel and the user process's CCB is copied to the kernel CCB. The User Agent SCSI device driver sleeps waiting for the request to complete; then, all necessary cleanup is performed, and the user process is notified of the completion of the request. If a signal is caught, an ABORT CCB is issued to try to terminate the outstanding CCB for the user process.

The User Agent SCSI device driver allows multiple processes access to the XPT layer; therefore, there may be multiple processes sleeping on the User

Agent. All CCBs passed through by the User Agent are queued at the SIM layer.

2.2 User Agent Error Handling

The User Agent SCSI device driver performs limited error checking on the CCB pointed to in the UAGT_CAM_CCB structure passed from the user process. The User Agent driver verifies that the `uagt_ccblen` is not greater than the maximum length for a CCB, checks that the XPT function code is valid, and checks that the Target ID and LUN specified are within the range allowed. The User Agent does not issue a REQUEST SENSE command in response to a CHECK CONDITION status. Autosensing is assumed to be enabled in the `cam_ch.cam_flags` field of the SCSI I/O CCB. The application program is responsible for issuing a RELEASE SIM QUEUE CCB.

The following error codes are returned by the User Agent `ioctl` function:

- EFAULT – An error occurred in copying to or from user space.
- EBUSY – Out of resources (the User Agent request table is full).
- EINVAL – An invalid target or LUN was passed to the User Agent driver, the CCB copied from the user process contained an invalid parameter, or an invalid `ioctl` command.

2.3 User Agent Data Structures

This section describes the data structures the User Agent uses.

2.3.1 The UAGT_CAM_CCB Data Structure

The User Agent SCSI device driver uses the UAGT_CAM_CCB data structure with the UAGT_CAM_IO ioctl command to communicate with the user processes requesting access to the SCSI/CAM subsystem.

The user process fills in the pointers in the UAGT_CAM_CCB data structure. The structure is copied into kernel space. The user process's CCB is copied into kernel space by the User Agent.

If necessary, the user data area and the sense data area are locked in memory. If any pointers in the UAGT_CAM_CCB structure are not needed with the requested CCB, the pointers must be set to NULL.

The CCB contains all the information necessary to execute the requested XPT function. The addresses in the CCB are used by the SIM and must be valid. The User Agent will not modify the corresponding pointers in the user's CCB.

The CCB definition is different for each of the following XPT functions supported by the User Agent SCSI device driver:

- XPT_NOOP – Execute nothing
- XPT_SCSI_IO – Execute the requested SCSI IO
- XPT_GDEV_TYPE – Get the device type information
- XPT_PATH_INQ – Path inquiry
- XPT_REL_SIMQ – Release the SIM queue that was frozen intentionally or by a previous error.
- XPT_SDEV_TYPE – Set the device type information
- XPT_ABORT – Abort the selected CCB
- XPT_RESET_BUS – Reset the SCSI bus
- XPT_RESET_DEV – Reset the SCSI device, BDR
- XPT_TERM_IO – Terminate the selected CCB

If a signal is generated by the user process, the User Agent creates an XPT_ABORT CCB to abort the outstanding I/O and then waits for the completion of the I/O and notifies the user process when the aborted CCB is returned to the User Agent.

The UAGT_CAM_CCB structure is defined as follows:

```
typedef struct uagt_cam_ccb
{
    CCB_HEADER *uagt_ccb;       /* pointer to the users CCB */
    u_long uagt_ccblen;         /* length of the users CCB */
    u_char *uagt_buffer;        /* pointer for the data buffer */
    u_long uagt_buflen;         /* length of user request */
    u_char *uagt_snsbuf;        /* pointer for the sense buffer */
    u_long uagt_snslen;         /* length of user's sense buffer */
    CDB_UN *uagt_cdb;           /* ptr for a CDB if not in CCB */
    u_long uagt_cdblen;         /* CDB length if appropriate */
    u_long uagt_flags;          /* See below */
} UAGT_CAM_CCB;
```

2.3.1.1 The uagt_ccb Member

The `uagt_ccb` member contains a pointer to the user process's CCB that will be copied into kernel space.

2.3.1.2 The uagt_ccblen Member

The `uagt_ccblen` member contains the length of the user process's CCB.

2.3.1.3 The uagt_buffer Member

The `uagt_buffer` member contains a pointer to the user process's data buffer. This member is used only by the User Agent.

2.3.1.4 The uagt_buflen Member

The `uagt_buflen` member contains the length of the user process's data buffer. This member is used only by the User Agent.

2.3.1.5 The uagt_snsbuf Member

The `uagt_snsbuf` member contains a pointer to the user process's autosense data buffer. This member is used only by the User Agent.

2.3.1.6 The uagt_snslen Member

The `uagt_snslen` member contains the length of the user process's autosense data buffer. This member is used only by the User Agent.

2.3.1.7 The uagt_cdb Member

If the user process's CCB contains a pointer to a CDB, then the `uagt_cdb` also contains a pointer to a Command Descriptor Block (CDB) that is to be locked in memory. This member and the `uagt_cdblen` member are used only by the User Agent driver. The CCB must also contain valid pointers and counts.

2.3.1.8 The uagt_cdblen Member

The `uagt_cdblen` contains the length of the Command Descriptor Block, if appropriate.

2.3.1.9 The uagt_flags Member

The `uagt_flags` contains the UAGT_NO_INT_SLEEP bit, which, if set, indicates that the User Agent should not sleep at an interruptible priority.

2.3.2 The UAGT_CAM_SCAN Data Structure

The User Agent SCSI device driver uses the UAGT_CAM_SCAN data structure to communicate with user level programs that need to have access to the CAM subsystem. The structure is copied into kernel space as part of the `ioctl` system call from user space for the UAGT_CAM_SINGLE_SCAN and UAGT_CAM_FULL_SCAN commands. The user program fills in the pointers in this structure and the User Agent SCSI device driver correctly fills in the corresponding pointers in the CCB.

The UAGT_CAM_SCAN structure is defined as follows:

```
typedef struct uagt_cam_scan  {
        u_char  ucs_bus;                  /* Bus id for scan */
        u_char  ucs_target;               /* Target id for scan */
        u_char  ucs_lun;                  /* LUN for scan */
} UAGT_CAM_SCAN;
```

2.4 User Agent Routines

This section describes the User Agent routines supplied by Digital. Table 2-1 lists the name of each routine and gives a summary description of its function. The sections that follow contain a more detailed description of each User Agent routine. Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix C.

Table 2-1: User Agent Routines

Routine	Summary Description
`uagt_open`	Handles the open of the User Agent driver
`uagt_close`	Handles the close of the User Agent driver
`uagt_ioctl`	Handles the `ioctl` system call for the User Agent driver

2.4.1 The uagt_open Routine

The `uagt_open` routine handles the open of the User Agent driver.

The character device special file name used for the open is `/dev/cam`.

2.4.2 The uagt_close Routine

The `uagt_close` routine handles the close of the User Agent driver. For the last close operation for the driver, if any queues are frozen, a RELEASE SIM QUEUE CCB is sent to the XPT layer for each frozen queue detected by the User Agent.

2.4.3 The uagt_ioctl Routine

The `uagt_ioctl` routine handles the `ioctl` system call for the User Agent driver. The `ioctl` commands supported are: DEVIOCGET, to obtain the User Agent driver's SCSI device status; UAGT_CAM_IO, the `ioctl` define for sending CCBs to the User Agent driver;

UAGT_CAM_SINGLE_SCAN, to scan a bus, target, and LUN; and UAGT_CAM_FULL_SCAN, to scan a bus.

For SCSI I/O CCB requests, the user data area is locked before passing the CCB to the XPT. The User Agent sleeps waiting for the I/O to complete and issues an ABORT CCB if a signal is caught while sleeping.

2.5 Sample User Agent Drivers

Two sample User Agent driver programs follow. The first sample program uses the User Agent driver to perform a SCSI INQUIRY command to a device on a selected nexus.

The second sample program is a scanner control program that sets up a scanner, reads scan line data from the device, and writes the data to a file, using the User Agent driver.

Both programs are included with the S/CA software and reside in the `/usr/examples` directory.

2.5.1 Sample User Agent Driver Inquiry Program

This section contains the User Agent sample inquiry application program, `caminq.c`, with annotations to the code. The user enters the string `inq` followed by the numbers identifying the bus, target, and LUN nexus to be checked for a valid device. If the device is valid, the INQUIRY data is displayed at the console. If the device is invalid, an error message appears.

2.5.1.1 The include Files and Definitions Section

This section describes the portion of the User Agent sample inquiry application program that lists the `include` files, local definitions, and data initialization for the program.

```
/* ------------------------------------------------------------- */
/* Include files needed for this program. */

#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <strings.h>
#include <ctype.h>
#include <io/common/iotypes.h>
#include <io/cam/cam.h>      /* CAM defines from the CAM document */
#include <io/cam/dec_cam.h>/* CAM defines for Digital CAM source files */
#include <io/cam/uagt.h>     /* CAM defines for the UAgt driver */
#include <io/cam/scsi_all.h> /* CAM defines for ALL SCSI devices */
```

```
/* ---------------------------------------------------------------- */
/* Local defines */
#define INQUIRY_LEN    36    /* general inquiry length */ [1]
/* ---------------------------------------------------------------- */
/* Initialized and uninitialized data. */
u_char buf[ INQUIRY_LEN ]; [2]
```

[1] This line defines a constant of 36 bytes for the length of the inquiry expected by the user from the SCSI device.

[2] This line declares a global character array, `buf`, with a size of 36 bytes as defined by the INQUIRY_LEN constant.

2.5.1.2 The Main Program Section

This section describes the `main` program portion of the User Agent sample inquiry application program.

```
/* ---------------------------------------------------------------- */
/* The main code path.  The CCB/CDB and UAGT_CAM_CCB are set up for
   an INQUIRY command to the Bus/Target/Lun selected by the command
   line arguments.  The returned INQUIRY data is displayed to the
   user if the status is valid.  If the returned status indicates
   an error, the error is reported instead of the INQUIRY data. */
main(argc, argv)
int argc;
char *argv[];
{
    extern void print_inq_data(); [1]
    extern void print_ccb_status();

    u_char id, targid, lun;     /* from the command line */
    int fd;                     /* unit number from the open */ [2]

    UAGT_CAM_CCB ua_ccb;        /* local uagt structure */ [3]
    CCB_SCSIIO ccb;             /* local CCB */ [4]
    ALL_INQ_CDB *inq;           /* pointer for the CDB */ [5]

    /* Make sure that all the arguments are there. */ [6]

    if (argc != 4) {
      printf("SCSI INQ bus target lun\n");
      exit();
    }

    /* Convert the nexus information from the command line. */ [7]

    id     = atoi(argv[1]);
    targid = atoi(argv[2]);
    lun    = atoi(argv[3]);
```

[1] These two forward references define routines that are used later in the program to print out the INQUIRY data or to print out the CAM status if there was an error.

[2] The file descriptor for the User Agent driver returned by the `open` system call, which executes in Section 2.5.1.3.

3 This line declares an uninitialized local data structure, `ua_ccb`, of the type UAGT_CAM_CCB, which is defined in the file `/usr/sys/include/io/cam/uagt.h`. This structure is copied from user space into kernel space as part of the `ioctl` system call. Section 2.5.1.7 describes this procedure.

4 This line declares an uninitialized local data structure, `ccb`, of the type CCB_SCSIIO , which is defined in the file `/usr/sys/include/io/cam/cam.h`. The members of this structure needed for the XPT_SCSI_IO request are filled in Section 2.5.1.4. The members of this structure needed for the INQUIRY command are filled in Section 2.5.1.5.

5 This line declares a pointer, `inq`, to a data structure, ALL_INQ_CDB, which is defined in the file `/usr/sys/include/io/cam/scsi_all.h`. This structure is filled in Section 2.5.1.5.

6 This section of code makes sure the user entered the correct number of arguments. The user should have entered the string `inq`, followed by three numeric characters representing the bus, target, and LUN to be checked for a valid status.

7 This section of code converts the numeric characters entered and assigns them, in order, to bus, target, and LUN.

2.5.1.3 The User Agent Open Section

This section describes the portion of the User Agent sample inquiry application program where the User Agent is opened.

```
/* Open the User Agent driver and report any errors. */
if ((fd = open("/dev/cam", O_RDWR, 0)) < 0 ) 1
{
  perror("Error on CAM UAgt Open:");
  exit(1);
}
```

1 The program attempts to open the User Agent device special file, `/dev/cam`, with the O_RDWR flag, which allows reading and writing. If the file descriptor returned by the `open` system call indicates that the open failed by returning a negative value, `< 0`, the program reports an error and exits. Otherwise, the program opens the device.

2.5.1.4 Filling in XPT_SCSI_IO Request CCB_HEADER Fields

This section describes the portion of the User Agent sample inquiry application program where the members of the CCB_HEADER needed for an

XPT_SCSI_IO request are filled in.

```
bzero((caddr_t)&ccb,sizeof(CCB_SCSIIO))
                    /* Clear the CCB structure */
/* Set up the CCB for an XPT_SCSI_IO request.  The INQUIRY command
will be sent to the device, instead of sending an XPT_GDEV_TYPE. */

/* Set up the CAM header for the XPT_SCSI_IO function. */

ccb.cam_ch.my_addr = (struct ccb_header *)&ccb; /* "Its" address */ 1
ccb.cam_ch.cam_ccb_len = sizeof(CCB_SCSIIO);  /* a SCSI I/O CCB */
ccb.cam_ch.cam_func_code = XPT_SCSI_IO;       /* the opcode */
ccb.cam_ch.cam_path_id = id;            /* selected bus */ 2
ccb.cam_ch.cam_target_id = targid;      /* selected target */
ccb.cam_ch.cam_target_lun = lun;              /* selected lun */

/* The needed CAM flags are : CAM_DIR_IN - The data will come from
the target, CAM_DIS_AUTOSENSE - Do not issue a REQUEST SENSE packet
if there is an error. */

ccb.cam_ch.cam_flags = CAM_DIR_IN | CAM_DIS_AUTOSENSE; 3
```

1 This section of code fills in some of the CCB_HEADER fields of the SCSI I/O CCB structure defined as `ccb`, for processing by the XPT layer. The structure was declared in Section 2.5.2.5.

2 These three lines assign the bus, target, and LUN to the corresponding fields in the CCB_HEADER structure.

3 This line sets the necessary CAM flags for the INQUIRY: CAM_DIR_IN, which specifies that the direction of the data is incoming; and CAM_DIS_AUTOSENSE, which disables the autosense feature. These flags are defined in `/usr/sys/include/io/cam/cam.h`.

2.5.1.5 Filling in INQUIRY Command CCB_HEADER Fields

This section describes the portion of the User Agent sample inquiry application program where the members of the CCB_HEADER needed for the INQUIRY command are filled in. This is the structure that is passed to the XPT layer by the User Agent driver.

```
    /* Set up the rest of the CCB for the INQUIRY command. */

ccb.cam_data_ptr = &buf[0];             /* where the data goes */ 1
ccb.cam_dxfer_len = INQUIRY_LEN;        /* how much data */
ccb.cam_timeout = CAM_TIME_DEFAULT;     /* use the default timeout */ 2
ccb.cam_cdb_len = sizeof( ALL_INQ_CDB );
                                  /* how many bytes for inquiry */3

/* Use a local pointer to access the particular fields in the INQUIRY
CDB. */

inq = (ALL_INQ_CDB *)&ccb.cam_cdb_io.cam_cdb_bytes[0]; 4

inq->opcode = ALL_INQ_OP;               /* inquiry command */ 5
inq->evpd = 0;                          /* no product data */
inq->lun = 0;                          /* not used in SCSI-2 */
inq->page = 0;                         /* no product pages */
inq->alloc_len = INQUIRY_LEN;           /* for the buffer space */
inq->control = 0;                   /* no control flags */
```

1 This line sets the `cam_data_ptr` member of the SCSI I/O CCB structure to the address of the first element in the `buf` array, which is defined as 36 bytes in Section 2.5.1.1.

2 This line specifies using the default timeout, which is the value assigned to the CAM_TIME_DEFAULT constant. This constant is set in the `/usr/sys/include/io/cam/cam.h` file to indicate that the SIM layer's default timeout is to be used. The current value of the SIM layer's default timeout is five seconds.

3 This line sets the length of the Command Descriptor Block in the CCB to the length of an inquiry CDB.. The inquiry CDB, ALL_INQ_CDB, which is defined in the `/usr/sys/include/io/cam/scsi_all.h` file, is six bytes.

4 This line assigns the `inq` pointer, which is type ALL_INQ_CDB, to the address of the `cam_cdb_bytes` member of the CDB_UN union. This union is defined in `/usr/sys/include/io/cam/cam.h` as the `cam_cdb_io` member of the SCSI I/O CCB structure.

5 These lines use the `inq` pointer to access the fields of the `cam_cdb_bytes` array within the `ccb` structure as though it is an ALL_INQ_CDB structure. The ALL_INQ_CDB structure is defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

2.5.1.6 Filling in the UAGT_CAM_CCB Fields

This section describes the portion of the User Agent sample inquiry application program where the members of the UAGT_CAM_CCB structure are filled in for the `ioctl` call. This is the structure that is passed to the User Agent driver.

```
bzero((caddr_t)&ua_ccb,sizeof(UAGI_CAM_CCB))
                    /* Clear the ua_ccb structure */

/* Set up the fields for the User Agent Ioctl call. */

ua_ccb.uagt_ccb = (CCB_HEADER *)&ccb;      /* where the CCB is */ 1
ua_ccb.uagt_ccblen = sizeof(CCB_SCSIIO);
                                  /* how many bytes to pull in */ 2
ua_ccb.uagt_buffer = &buf[0];          /* where the data goes */ 3
ua_ccb.uagt_buflen = INQUIRY_LEN;        /* how much data */ 4

ua_ccb.uagt_snsbuf = (u_char *)NULL;     /* no Autosense data */ 5
ua_ccb.uagt_snslen = 0;                  /* no Autosense data */
ua_ccb.uagt_cdb = (CDB_UN *)NULL;        /* CDB is in the CCB */ 6
ua_ccb.uagt_cdblen = 0;                  /* CDB is in the CCB */
```

1 This line initializes the `uagt_ccb` member of the `ua_ccb` structure with the address of the local CCB_HEADER structure, `ccb`.

2 This line sets the length of the `uagt_ccblen` member to the length of the SCSI I/O CCB structure that will be used for this call.

3 This line initializes the uagt_buffer member with the user space address of the array buf, which was allocated 36 bytes in Section 2.5.1.1.

4 This line initializes the uagt_buflen member with the value of the constant INQUIRY_LEN, which is the number of bytes of inquiry data that will be returned.

5 These two lines reflect that the autosense features are turned off in the CAM flags.

6 These two lines reflect that the Command Descriptor Block information is in the SCSI I/O CCB structure filled in Section 2.5.1.4.

2.5.1.7 Sending the CCB to the CAM Subsystem

This section describes the portion of the User Agent sample inquiry application program where the ccb is sent to the CAM subsystem.

```
/* Send the CCB to the CAM subsystem using the User Agent driver,
and report any errors. */
if( ioctl(fd, UAGT_CAM_IO, (caddr_t)&ua_ccb) < 0 ) 1
{
perror("Error on CAM UAGT Ioctl:");
close(fd);                          /* close the CAM file */ 2
      exit(1);
}
/* If the CCB completed successfully, then print out the INQUIRY
information; if not, report the error. */
if (ccb.cam_ch.cam_status != CAM_REQ_CMP)
{
      print_ccb_status( &(ccb.cam_ch) );
                                       /* report the error values */ 3
}
else
{
      print_inq_data( &buf[0] );         /* report the INQUIRY info */ 4
}
}
```

1 This line passes the local UAGT_CAM_CCB structure, ua_ccb, to the User Agent driver, using the ioctl system call. The arguments passed are the file descriptor returned by the open system call; the User Agent ioctl command, UAGT_CAM_IO, which is defined in the /usr/sys/include/io/cam/uagt.h file; and the contents of the ua_ccb structure. The User Agent driver copies in the SCSI I/O CCB and sends it to the XPT layer. When the I/O completes, the User Agent returns to the application program, returning status within the ua_ccb structure.

2 If the ioctl call fails, this code displays an error message, closes the device special file, /dev/cam, and exits.

3 If the CAM status is anything other than CAM_REQ_CMP, indicating the request completed with an error, then an error message is printed indicating the CAM status returned.

4 If the request completes, the `print_inq_data` routine is called to display the INQUIRY data.

2.5.1.8 Print INQUIRY Data Routine

This section of the User Agent sample inquiry application program converts the rest of the fields of inquiry data to a human-readable form and sends it to the user's screen.

```
/* Define the type and qualifier string arrays as globals to allow for
   compile-time initialization of the information. */
        caddr_t periph_type[] = {        /* Peripheral Device Type */
                "Direct-access",         /* 00h */
                "Sequential-access",     /* 01h */
                "Printer",               /* 02h */
                "Processor",             /* 03h */
                "Write-once",            /* 04h */
                "CD-ROM",                /* 05h */
                "Scanner",               /* 06h */
                "Optical memory",        /* 07h */
                "Medium changer",        /* 08h */
                "Communications",        /* 09h */
                "Graphics Arts"          /* 0Ah */
       };       /* Same as 0A */         /* 0Bh */
                /* Reserved */           /* 0Ch - 1Eh */
                /* Unknown */            /* 1Fh */
        caddr_t periph_qual[] = {        /* Peripheral Qualifier */
                "Device supported, is (may be) connected", /* 000b */
                "Device supported, is not connected",      /* 001b */
                "<Reserved qualifier>",                    /* 010b */
                "No device supported for this Lun"         /* 011b */
        };      /* Vendor specific */                      /* 1xxb */
/* ------------------------------------------------------------------- */
/* Local routine to print out the INQUIRY data to the user. */
void
print_inq_data( ip ) 1
    ALL_INQ_DATA *ip;
{
      char vendor_id[9]; 2
      char prod_id[17];
      char prod_rev_lvl[5];
        caddr_t periph_type_ptr, periph_qual_ptr;
        int  ptype;
      /* Make local copies of the ASCII text, so that it can be NULL
         terminated for the printf() routine. */
      strncpy(vendor_id, (caddr_t)ip->vid, 8); 3
      vendor_id[8] = '\0';
      strncpy(prod_id, (caddr_t)ip->pid, 16);
      prod_id[16] = '\0';
      strncpy(prod_rev_lvl, (caddr_t)ip->revlevel, 4);
      prod_rev_lvl[4] = '\0';
```

```
    /* Convert sparse device type and qualifier values into strings */
      ptype = ip->dtype; [4]
      periph_type_ptr = "Reserved";
      if (ptype == 0x1F) periph_type_ptr = "Unknown";
      if (ptype == 0x0B) ptype = 0x0A;
      if (ptype <= 0x0A) periph_type_ptr = periph_type[ptype];

      periph_qual_ptr = "<Vendor Specific qualifier>";
      if (ip->pqual <= 3) periph_qual_ptr = periph_qual[ip->pqual];

      printf("Periph Device Type   = 0x%X = %s Device\n", [5]
              ip->dtype, periph_type_ptr);
      printf("Periph Qualifier     = 0x%X = %s\n", ip->pqual,
              periph_qual_ptr);
      printf("Device Type Modifier = 0x%X\tRMB  = 0x%X = Medium %s\n",
             ip->dmodify, ip->rmb, (ip->rmb?"is removable":
              "is not removable"));
      printf("ANSI Version = 0x%X\t\tECMA Version = 0x%X\n",
              ip->ansi, ip->ecma);
      printf("ISO Version  = 0x%X\t\tAENC = 0x%X\tTrmIOP = 0x%X\n",
              ip->iso, ip->aenc, ip->trmiop);
      printf("Response Data Format   = 0x%X\tAddit Length = 0x%d\n",
              ip->rdf, ip->addlen);
      printf("SftRe  = 0x%XCmdQue = 0x%X\tLinked = 0x%X\tSync = 0x%X\n",
              ip->sftre, ip->cmdque, ip->linked, ip->sync);
      printf("Wbus16 = 0x%X\tWbus32 = 0x%X\tRelAdr = 0x%X\n",
              ip->wbus16, ip->wbus32, ip->reladdr);
      printf("Vendor Identification = %s\nProduct Identification = %s\n",
              vendor_id, prod_id );
      printf("Product Revision Level = %s\n\n",
              prod_rev_lvl);
      fflush(stdout); [6]
}
```

[1] This line declares the `print_inq_data` function that prints out the INQUIRY data for a valid nexus. The function's argument, `ip`, is a pointer to the ALL_INQ_DATA structure defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

[2] These three lines declare three character arrays to contain the Vendor ID, the Product ID, and the Product revision level to be displayed. Each array is declared with one extra byte to hold the NULL string terminator.

[3] This section copies the ALL_INQ_DATA member, `vid`, into the local array `vendor_id`; the ALL_INQ_DATA member, `pid`, into the local array `prod_id`; and the ALL_INQ_DATA member, `revlevel`, into the local array, `prod_rev_lvl`. The arrays are passed to the standard C library function, `strncpy`, which copies the data and then terminates each string copy with a NULL, so that it can be output to the `printf` function in the format desired.

[4] This section converts the device type and qualifier values into human-readable words. The conversions are performed on defined and undefined numeric combinations.

[5] This section decodes and displays the inquiry data as hexadecimal numbers and strings.

6 This line calls the standard C I/O function, `fflush`, to write out the data from the internal buffers.

2.5.1.9 Print CAM Status Routine

This section describes the portion of the User Agent sample inquiry application program that defines the routine to print out the CAM status for an invalid nexus.

```
/* ---------------------------------------------------------------- */
/* Local routines and data structure to report in text and Hex
form the returned CAM status. */

struct cam_statustable { 1
      u_char      cam_status;
      caddr_t     status_msg;
} cam_statustable[] = { 2
    {    CAM_REQ_INPROG,          "CCB request is in progress"        },
    {    CAM_REQ_CMP ,            "CCB request completed w/out error" },
    {    CAM_REQ_ABORTED,         "CCB request aborted by the host"   },
    {    CAM_UA_ABORT,            "Unable to Abort CCB request"       },
    {    CAM_REQ_CMP_ERR,         "CCB request completed with an err" },
    {    CAM_BUSY,                "CAM subsystem is busy"             },
    {    CAM_REQ_INVALID,         "CCB request is invalid"            },
    {    CAM_PATH_INVALID,        "Bus ID supplied is invalid"        },
    {    CAM_DEV_NOT_THERE,       "Device not installed/there"        },
    {    CAM_UA_TERMIO,           "Unable to Terminate I/O CCB req"    },
    {    CAM_SEL_TIMEOUT,         "Target selection timeout"          },
    {    CAM_CMD_TIMEOUT,         "Command timeout"             },
    {    CAM_MSG_REJECT_REC,      "Reject received"             },
    {    CAM_SCSI_BUS_RESET,      "Bus reset sent/received"           },
    {    CAM_UNCOR_PARITY,        "Parity error occurred"             },
    {    CAM_AUTOSENSE_FAIL,      "Request sense cmd fail"      },
    {    CAM_NO_HBA,              "No HBA detected Error"             },
    {    CAM_DATA_RUN_ERR,        "Overrun/underrun error"      },
    {    CAM_UNEXP_BUSFREE,       "BUS free"              },
    {    CAM_SEQUENCE_FAIL,       "Bus phase sequence failure"        },
    {    CAM_CCB_LEN_ERR,         "CCB length supplied is inadequate"  },
    {    CAM_PROVIDE_FAIL,        "To provide requ. capability"      },
    {    CAM_BDR_SENT,            "A SCSI BDR msg was sent to target"  },
    {    CAM_REQ_TERMIO,          "CCB request terminated by the host" },
    {    CAM_LUN_INVALID,         "LUN supplied is invalid"     },
    {    CAM_TID_INVALID,         "Target ID supplied is invalid"},
    {    CAM_FUNC_NOTAVAIL,       "Requested function is not available" },
    {    CAM_NO_NEXUS,            "Nexus is not established"          },
    {    CAM_IID_INVALID,         "The initiator ID is invalid"   },
    {    CAM_CDB_RECVD,           "The SCSI CDB has been received"      },
    {    CAM_SCSI_BUSY,           "SCSI bus busy"                       }
};
int cam_statusentrys = sizeof(cam_statustable) / \
                        sizeof(cam_statustable[0]); 3

char *
camstatus( cam_status ) 4
    register u_char cam_status;
{
      register struct cam_statustable *cst = cam_statustable; 5
      register entrys;

      for( entrys = 0; entrys < cam_statusentrys; cst++ ) { 6
```

```
            if( cst->cam_status == cam_status ) {
                  return( cst->status_msg );
            }
      }
      return( "Unknown CAM Status" );
}
void
print_ccb_status(cp) 7
CCB_HEADER *cp;
{
      printf( "cam_status = 0x%X\t (%s%s%s)\n", cp->cam_status,
      ((cp->cam_status & CAM_AUTOSNS_VALID) ? "AutoSns Valid-" : "" ),
      ((cp->cam_status & CAM_SIM_QFRZN) ? "SIM Q Frozen-" : "" ),
      camstatus( cp->cam_status & CAM_STATUS_MASK ));
      fflush(stdout); 8
}
```

1 This line defines an array of structures. It is declared as a global array to allow compile-time initialization. Each structure element of the array contains two members, `cam_status`, the CAM status code, and `status_msg`, a brief description of the meaning of the status code. The CAM status codes and messages are defined in the `/usr/sys/include/io/cam/cam.h` file.

2 These lines initialize the CAM status array with the status values and their text equivalents.

3 This line declares an integer variable whose contents equal the size of the total CAM status array divided by the size of an individual array element. This integer is the number of the element in the array.

4 The next two lines define a function that returns a pointer to a text string with the `cam_status` field of the CCB_HEADER as an argument. The `cam_status` member is declared as a register variable so that its values are stored in a machine register for efficiency.

5 This line declares a register structure pointer to point to each element of the CAM status array and initializes it to point to the beginning of the CAM status array. A local register variable, `entrys`, will be used to traverse the CAM status array.

6 This section of code examines each element in the array, incrementing `cst` until a match between the status from the CCB and a status value in the array is found, in which case the address of the CAM status description string, `status_msg`, is returned. If all the elements are examined without a match, the "Unknown CAM Status" message address is returned.

7 The next two lines define a routine that uses a pointer to the CCB_HEADER structure of the INQUIRY CCB and calls the C library routine, `printf`, to print out the hexadecimal value and the appropriate description of the CAM status returned.

8 This line calls the standard C I/O function, `fflush`, to write out the data from the internal buffers.

2.5.1.10 Sample Output for a Valid Nexus

This section contains an example of the output of the User Agent sample inquiry application program when the user enters a valid nexus.

```
#inq 0 0 0

Periph Device Type = 0x0          Periph Qualifier = 0x0 1
Device Type Modifier = 0x0        RMB = 0x0
ANSI Version = 0x1                ECMA Version = 0x0
ISO Version = 0x0                 AENC = 0x0      TrmIOP = 0x0
Response Data Format = 0x1        Addit Length = 0x31
SftRe = 0x0      CmdQue = 0x0     Linked = 0x0    Sync = 0x1
Wbus16 = 0x0     Wbus32 = 0x0     RelAdr = 0x0
Vendor ID = DEC 2
Product ID = RZ56      (C) DEC 3
Product Rev Level = 0300 4
```

1 See the American National Standard for Information Systems, *Small Computer Systems Interface - 2* (SCSI - 2), X3.131-199X for a description of each of the fields of the inquiry data returned.

2 This line shows the value of the `vendor_id` variable declared in the `print_inq_data` routine in Section 2.5.1.8 as a local copy of the text string.

3 This line shows the value of the `prod_id` variable declared in the `print_inq_data` routine in Section 2.5.1.8 as a local copy of the text string.

4 This line shows the value of the `prod_rev_lvl` variable declared in the `print_inq_data` routine in Section 2.5.1.8 as a local copy of the text string.

2.5.1.11 Sample Output for an Invalid Nexus

This section contains an example of the output of the User Agent sample inquiry application program when the user enters an invalid nexus.

```
#inq 0 2 0

cam_status = 0x4A        (SIM Q Frozen-Target selection timeout) 1
```

1 This line shows that the contents of the `cam_status` member of the CCB_HEADER structure returned was CAM_SIM_QFRZN, which indicates a lack of response from the specified nexus. See the `cam_statustable` in Section 2.5.1.9.

2.5.1.12 Sample Shell Script

This section contains a sample C-shell script, `caminq.csh`, that compiles and executes the User Agent sample inquiry application program.

```
#cc -o caminq caminq.c
inq 0 6 0
inq 0 2 0
inq 0 5 0
inq 0 3 0
```

2.5.2 Sample User Agent Scanner Driver Program

This section contains the User Agent sample scanner program, `cscan.c`, with annotations to the code. It also contains the `cscan.h` file, which defines the WINDOW_PARAM_BLOCK structure used in the program.

The `cscan.c` program assumes that the environment variable SCAN-NEXUS has been set. The sample C-shell script that follows, `cscan.csh`, compiles the program and sets SCAN-NEXUS to bus 1, target 3, and LUN 0:

```
#cc -o cscan cscan.c
#setenv SCAN-NEXUS "1 3 0"
```

2.5.2.1 Scanner Program Header File

This section describes the header file, `cscan.h`, that contains definitions of structures for the program to use.

```
/* cscan.h   Header file for cscan.c (CAM Scanner driver)  28-Oct-1991 */

/* Scanner Window Parameter Block definition; all multi-byte quantities
   are defined as unsigned bytes due to the need to store the values in
   swapped order. */

typedef struct {
      u_char rsvd1[6];    /* Reserved bytes in Header: Must Be Zero */
      u_char WDBLen[2];   /* Number of Window Parameter bytes
                                                    following */ [1]
      u_char WID;         /* Window ID: Must Be Zero */
      u_char rsvd2;       /* Reserved bytes in Header: Must Be Zero */
      u_char XRes[2];     /* X-axis resolution: MUST be same as YRes */
      u_char YRes[2];     /* Y-axis resolution: MUST be same as XRes */
      u_char UpLeftX[4];  /* Upper left X positon of scan window */
      u_char UpLeftY[4];  /* Upper left Y positon of scan window */
      u_char Width[4];    /* Scan width (Y-axis length) */
      u_char Length[4];   /* Scan length (X-axis length) */
      u_char Bright;      /* Brightness: Must Be Zero */
      u_char Thresh;      /* Threshold: Must Be Zero */
      u_char Contrast;    /* Contrast: Must Be Zero */
      u_char ImgTyp;      /* Image type: 0 = bi-level mono; 2 = multi-level
                             mono; 3 = bi-level full color; 5 = multi-
                             level full color; others reserved */
      u_char PixBits;     /* Bits per pixel: 1 = bi-level; 4 = 16 shades;
                             8 = 256 shades; others reserved */
      u_char HalfTone[2];/* Halftone Pattern: Must Be Zero */
      u_char PadTyp:3;    /* Padding type for non-byte pixels: MUST BE 1 */
```

```
        u_char rsvd3:4;      /* Reserved bits: Must Be Zero */
        u_char RevImg:1;     /* 0 = normal image; 1 = reverse image */
        u_char BitOrder[2];/* Bit ordering: Must Be Zero */
        u_char CompTyp;      /* Compression type: Must Be Zero */
        u_char CompArg;      /* Compression argument: Must Be Zero */
        u_char rsvd4[6];     /* Reserved: Must Be Zero */
        u_char HdrSel;       /* Header select (return with data):
                                0 = no header;
                                1 = return header with data;
                                others reserved */
        u_char ColorSel;     /* Color select (selects color to use when doing a
                                mono-color scan): 0 = default to Green; 1 =
                                scan using Red; 2 = scan using Green; 3 =
                                scan using Blue; others reserved */
        u_char ImgCorr;      /* Image data correction method: 0 = default to
                                normal; 1 = soft image; 2 = enhance (low);
                                3 = enhance (high); others reserved */
        u_char ThreshR;      /* Threshold level, Red: 0 = default level */
        u_char ThreshG;      /* Threshold level, Green: 0 = default level */
        u_char ThreshB;      /* Threshold level, Blue: 0 = default level */
        u_char ShtTyp:1;     /* Sheet type: 0 = reflection;
                                1 = transparency */
        u_char rsvd5:3;      /* Reserved bits: Must Be Zero */
        u_char ShtDen:4;     /* Sheet density (transparency): 0 = normal; 1 =
                                light; 2 = dark; others reserved */
}WINDOW_PARAM_BLOCK;
```

[1] The length in bytes of a single scan window descriptor. The first 48 bytes are defined in the American National Standard for Information Systems, *Small Computer Systems Interface - 2* (SCSI - 2), X3.131-199X and the remaining bytes are vendor-specific. The specific structure members used may depend on the scanner device.

2.5.2.2 The include Files Section

This section, which is the beginning of the `cscan` program, describes the portion of the User Agent sample scanner program that lists the `include` files for the program.

```
/* ---------------------------------------------------------------- */
/* Include files needed for this program. */

#include <stdio.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/uio.h>
#include <strings.h>
#include <ctype.h>
#include <math.h>
#include <io/common/iotypes>
#include <io/cam/cam.h>      /* CAM defines from the CAM document */
#include <io/cam/dec_cam.h>  /* CAM defines for Digital CAM source files */
#include <io/cam/uagt.h>     /* CAM defines for the UAgt driver */
#include <io/cam/scsi_all.h> /* CAM defines for ALL SCSI devices */
#include "cscan.h"           /* Scanner structure definitions */
```

2.5.2.3 The CDB Setup Section

This section describes the portion of the User Agent sample scanner program that defines the CDBs for the program.

```
/* The Define Window Parameters CDB (10 bytes).  */

typedef struct {
        u_char  opcode;         /* 24 hex                       */
        u_char          : 5,     /* 5 bits reserved              */
                lun     : 3;    /* logical unit number          */
        u_char          : 8;    /* Reserved byte                */
        u_char          : 8;    /* Reserved byte                */
        u_char          : 8;    /* Reserved byte                */
        u_char          : 8;    /* Reserved byte                */
        u_char  param_len2;     /* MSB parameter list length */ [1]
        u_char  param_len1;     /*     parameter list length */
        u_char  param_len0;     /* LSB parameter list length */
        u_char  control;        /* The control byte             */
}SCAN_DEF_WIN_CDB;

/* The Define Window Parameters op code */

#define SCAN_DEF_WIN_OP         0x24

/* The Read (data or gamma table) CDB (10 bytes).  */

typedef struct {
        u_char  opcode;         /* 28 hex                         */
        u_char          : 5,    /* 5 bits reserved                */
                lun     : 3;    /* logical unit number            */
        u_char  tran_type;      /* transfer data type:             */
                                /*   0=data, 3=gamma              */ [2]
        u_char          : 8;    /* Reserved byte                  */
        u_char  tran_id1;       /* MSB transfer identification */ [3]
        u_char  tran_id2;       /* LSB trans id:                  */
                                /*   0 =data, 1/2/3= gamma        */
        u_char  param_len2;     /* MSB parameter list length      */
        u_char  param_len1;     /*     parameter list length      */
        u_char  param_len0;     /* LSB parameter list length      */
        u_char  control;        /* The control byte               */
}SCAN_READ_CDB;

/* The Read (data or gamma table) op code */

#define SCAN_READ_OP        0x28
```

[1] The parameter list length members specify the number of bytes sent during the DATAOUT phase. The parameters are usually mode parameters, diagnostic parameters, and log parameters that are sent to a target. If set to 0 (zero), no data is to be transferred.

[2] The types of data that are to be read. The choices are: image data scan lines or gamma correction table data.

[3] These two bytes are used with the transfer type byte to indicate that the data to be read is image scan lines, 0 (zero), or one of the following types of gamma correction table data: red, 1; green, 2; or blue, 3.

2.5.2.4 The Definitions Section

This section describes the portion of the User Agent sample scanner program that specifies the local definitions and initializes data.

```
/* ---------------------------------------------------------------- */
/* Local defines */
#define SENSE_LEN18    /* max sense length from scanner */ 1

/* ---------------------------------------------------------------- */
/* Initialized and uninitialized data. */

u_char sense[ SENSE_LEN ]; 2
```

1 This line defines a constant of 18 bytes for the length of the sense data from the scanner.

2 This line declares a character array, `sense`, with a size of 18 bytes as defined by the SENSE_LEN constant.

2.5.2.5 The Main Program Section

This section describes the `main` program portion of the User Agent sample scanner program.

```
/* -------------------------------------------------------------------- */
/* The main code path.  The CCB/CDB and UAGT_CAM_CCB are set up for the
   DEFINE WINDOW PARAMETERS and READ commands to the Bus/Target/LUN. */

UAGT_CAM_CCB ua_ccb_sim_rel;     /* uagt structure */ 1
                                        /* for the RELEASE SIMQUE CCB */
CCB_RELSIM ccb_sim_rel;          /* RELEASE SIMQUE CCB */ 2
UAGT_CAM_CCB ua_ccb_reset_dev;   /* uagt structure */ 3
                                        /* for the RESET DEVICE CCB */
CCB_RESETDEV ccb_reset_dev;      /* RESET DEVICE CCB */ 4

UAGT_CAM_CCB ua_ccb;             /* uagt structure */ 5
                                        /* for the SCSI I/O CCB */
CCB_SCSIIO ccb;                  /* SCSI I/O CCB */ 6

main(argc, argv,envp)
int argc;
char *argv[];
char *envp[];
{
/*--------------------------------------------------------------------- */
/* Local variables and structures */

    extern void clear_mem(); 7
    extern void swap_short_store();
    extern void swap_long_store();

    u_char id, targid, lun;        /* from envir variable SCAN-NEXUS */ 8
    char *cp;
    int nexus;

    int fd;                        /* unit number for the CAM open */ 9
    int od;                        /* unit number for the file open */ 10
    char FileHead[200];           /* buffer for file header info */
    int i, n;
    u_char *bp;                   /* general usage byte pointer */
    int retry_cnt;                /* error retry counter */
```

```
int reset_flag;                 /* flag to indicate reset tried */

double Xwid, Ylen;              /* scan area in inches */ [11]
u_short WXYRes;                 /* variables for window calculations */
u_long WWidth, WLength, WinPix, LineBytes, TotalBytes; [12]
u_char WHdrSel; [13]

SCAN_DEF_WIN_CDB *win;          /* pointer for window def CDB */ [14]
SCAN_READ_CDB *read;            /* pointer for read CDB */ [15]

WINDOW_PARAM_BLOCK Window;      /* parameter block, window def */ [16]

u_char ReadData[ 400*12*3 ];    /* Max bytes/line */ [17]
u_char *RDRp, *RDGp, *RDBp;     /* Red, Green, Blue pointers */
u_char WriteData[ 400*12*3 ];   /* Max bytes/line */ [18]
u_char *WDp;                    /* WriteData pointer */
```

[1] This line declares a global data structure, `ua_ccb_sim_rel`, to be used with the RELEASE SIM QUEUE CCB for the UAGT_CAM_IO ioctl command.

[2] This line declares a global data structure, `ccb_sim_rel`, of the type CCB_RELSIM, which is defined in the file `/usr/sys/include/io/cam/cam.h`.

[3] This line declares a global data structure, `ua_ccb_reset_dev`, to be used for the BUS DEVICE RESET CCB for the UAGT_CAM_IO ioctl command.

[4] This line declares a global data structure, `ccb_reset_dev`, of the type CCB_RESETDEV, which is defined in the file `/usr/sys/include/io/cam/cam.h`.

[5] This line declares a global data structure, `ua_ccb`, of the type UAGT_CAM_CCB, which is defined in the file `/usr/sys/include/io/cam/uagt.h`. This structure is copied from user space into kernel space as part of the `ioctl` system call for the UAGT_CAM_IO ioctl command.

[6] This line declares a global data structure, `ccb`, of the type CCB_SCSIIO, which is defined in the file `/usr/sys/include/io/cam/cam.h`.

[7] These forward references declare routines that are used later in the program. The routines are defined in Section 2.5.2.14.

[8] The bus, target, and LUN are specified in octal digits in the SCAN-NEXUS environment variable. The value for the LUN should be 0 (zero).

[9] The file descriptor for the User Agent driver returned by the `open` system call, which executes in Section 2.5.2.7.

[10] The file descriptor for the output file returned by the `open` system call, which executes in Section 2.5.2.7.

[11] Real values to contain the `X` and `Y` dimensions of the scan window.

12 Variables to hold calculated information about the scan window.

13 Variable to hold the flag bytes indicating whether window header is to be returned with the data. The value of the variable is stored in the `HdrSel` member of the WINDOW_PARAM_BLOCK structure is set to 1. The WINDOW_PARAM_BLOCK is defined in Section 2.5.2.1.

14 This line declares a pointer to the data structure SCAN_DEF_WIN_CDB, which is defined in Section 2.5.2.3.

15 This line declares a pointer to the data structure SCAN_READ_CDB, which is defined in Section 2.5.2.3.

16 This line declares an uninitialized local data structure, `Window`, of the type WINDOW_PARAM_BLOCK, which is defined in Section 2.5.2.1.

17 This line declares an array to contain a scan line of the maximum size that can be read, which is 14,400 bytes. This array is used to read a scan line from the scanner.

18 This line declares an array large enough to contain the maximum-size scan line, which is 14,400 bytes. This array is used to write the scan line, converted to 3-byte pixels, to the output file.

2.5.2.6 The Nexus Conversion Section

This section describes the portion of the User Agent sample scanner program where the nexus information contained in the SCAN-NEXUS environment variable is converted to the values for bus, target, and LUN.

```
/* Find the environment variable SCAN-NEXUS. If not found, return
   error message.  If found, convert the nexus information from the
   variable to bus, target ID and LUN values.  Return an error
   message if any of the values are not octal digits. */
    nexus = 0;                                  /* Reset valid data flag */
    for (i=0; envp[i] != NULL; i++)
    {
      cp = envp[i]; 1
      if (strncmp(cp,"SCAN-NEXUS=",11) == 0)
                                                /* Find environment variable */
      {
        nexus = -1;                               /* Set tentative flag */
        cp += 11;                                 /* Advance to data */
        if (*cp < '0' || *cp > '7') break; 2
        id = (u_char)(*cp++) - (u_char)('0');
        if (*cp++ != ' ') break;
        if (*cp < '0' || *cp > '7') break;
        targid = (u_char)(*cp++) - (u_char)('0');
        if (*cp++ != ' ') break;
        if (*cp < '0' || *cp > '7') break;
        lun = (u_char)(*cp) - (u_char)('0');
        nexus = 1;                          /* Set good data flag */
      }
    }
    if (nexus == -1) 3
```

```
{
 printf("Invalid SCAN-NEXUS; set to octal digits 'bus target lun'\n");
  exit(1);
}
if (nexus == 0) [4]
{
  printf("Set environment variable SCAN-NEXUS to 'bus target lun'
                          (octal\ digits)\n\n");
  exit(1);
}
printf("Scanner nexus set to: bus %d, target %d, LUN %d\n\n",id, \
                          targid, lun); [5]
```

[1] This section scans through all of the environment variables passed to the program by the system, looking for the variable SCAN-NEXUS.

[2] This section checks to make sure SCAN-NEXUS contains octal digits for bus, target, and LUN.

[3] This error message appears if the digits are not octal.

[4] This error message appears if SCAN-NEXUS is not set.

[5] This message displays the values for bus, target, and LUN.

2.5.2.7 The Parameter Assignment Section

This section describes the portion of the User Agent sample scanner program that assigns the parameters entered by the user on the command line to the appropriate variables and opens the necessary files.

```
/* Make sure that the correct number of arguments are present.
   If not, return an error message with usage information. */
    if (argc != 5) { [1]
      printf("Usage is: cscan XYres Xwid Ylen out_file\n");
      printf(" XYres is integer pix/inch; Xwid & Ylen are real \
                          inches\n\n");
      exit();
    }
/* Convert the parameter information from the command line. */
    WXYRes = atoi(argv[1]);        /* X & Y resolution */
    Xwid   = atof(argv[2]);        /* X width in inches */
    Ylen   = atof(argv[3]);        /* Y length in inches */
/* Verify that the X & Y resolution is one of the legal values */
    switch (WXYRes) [2]
    {
      case 25:
      case 150:
      case 200:
      case 300:
      case 400:
        break;
      default:
        printf("Illegal X & Y resolution; must be 25, 150, 200, \
                                300, 400\n");
      exit(1);
```

```
    }
/* Verify that the X width is positive and less than 11.69 inches */ 3
    if (Xwid < 0 || Xwid > 11.69)
    {
      printf("X width must be positive and less than 11.69 inches\n");
      exit(1);
    }
/* Verify that the Y length is positive and less than 17.00 inches */
    if (Ylen < 0 || Ylen > 17.00)
    {
      printf("Y length must be positive and less than 17.00 inches\n");
      exit(1);
    }
/* Open the output file ("truncating" it if it exists) and report */
/* any errors. */ 4
    if ((od = open(argv[4], O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0 )
    {
      perror("Error on Output File Open");
      exit(1);
    }
/* Open the User Agent driver and report any errors. */
    if ((fd = open("/dev/cam", O_RDWR, 0)) < 0 )
    {
      perror("Error on CAM UAgt Open");
      exit(1);
    }
```

1. The user enters the X and Y scan resolutions in pixels per inch, the width (X) and length (Y) of the scan area in inches, and the name of the output file on the command line.
2. This section checks for the legal scan resolutions the user can enter.
3. These two sections check that the user entered legal values for X and Y.
4. These two sections open the User Agent driver and the output file.

2.5.2.8 The Data Structure Setup Section

This section describes the portion of the User Agent sample scanner program that sets up the data structures for the XPT_REL_SIMQ and XPT_RESET_DEV commands.

```
/* - Begin static setups of SIMQ Release and Device Reset structures - */
/* Set up the CCB for an XPT_REL_SIMQ request. */
/* Set up the CAM header for the XPT_REL_SIMQ function. */
    ccb_sim_rel.cam_ch.my_addr = (struct ccb_header *)&ccb_sim_rel;
                                                   /* "Its" address */ 1
    ccb_sim_rel.cam_ch.cam_ccb_len = sizeof(CCB_RELSIM);
                                                   /* a SIMQ release  */
    ccb_sim_rel.cam_ch.cam_func_code = XPT_REL_SIMQ; /* the opcode */
    ccb_sim_rel.cam_ch.cam_path_id   = id;           /* selected bus */
    ccb_sim_rel.cam_ch.cam_target_id = targid;       /* selected target */
```

```
    ccb_sim_rel.cam_ch.cam_target_lun = lun;         /* selected lun */
/* The needed CAM flags are: CAM_DIR_NONE - No data will be transferred. */
    ccb_sim_rel.cam_ch.cam_flags = CAM_DIR_NONE;
/* Set up the fields for the User Agent Ioctl call. */
    ua_ccb_sim_rel.uagt_ccb = (CCB_HEADER *)&ccb_sim_rel;
                                                   /* where the CCB is */ [2]
    ua_ccb_sim_rel.uagt_ccblen = sizeof(CCB_RELSIM); /* bytes in CCB */
    ua_ccb_sim_rel.uagt_buffer = (u_char *)NULL;     /* no data */
    ua_ccb_sim_rel.uagt_buflen = 0;                  /* no data */
    ua_ccb_sim_rel.uagt_snsbuf = (u_char *)NULL;  /* no Autosense data */
    ua_ccb_sim_rel.uagt_snslen = 0;               /* no Autosense data */
    ua_ccb_sim_rel.uagt_cdb    = (CDB_UN *)NULL;  /* CDB is in the CCB */
    ua_ccb_sim_rel.uagt_cdblen = 0;               /* CDB is in the CCB */
/* Set up the CCB for an XPT_RESET_DEV request. */
/* Set up the CAM header for the XPT_RESET_DEV function. */
    ccb_reset_dev.cam_ch.my_addr = (struct ccb_header *)&ccb_reset_dev;
                                                      /* "Its" address */ [3]
    ccb_reset_dev.cam_ch.cam_ccb_len = sizeof(CCB_RESETDEV);
                                                      /* a SCSI I/O CCB */
    ccb_reset_dev.cam_ch.cam_func_code = XPT_RESET_DEV;  /* the opcode */
    ccb_reset_dev.cam_ch.cam_path_id   = id;          /* selected bus */
    ccb_reset_dev.cam_ch.cam_target_id = targid;      /* selected target */
    ccb_reset_dev.cam_ch.cam_target_lun = lun;        /* selected lun */
/* The needed CAM flags are: CAM_DIR_NONE - No data will be transferred. */
    ccb_reset_dev.cam_ch.cam_flags = CAM_DIR_NONE;
/* Set up the fields for the User Agent Ioctl call. */
    ua_ccb_reset_dev.uagt_ccb = (CCB_HEADER *)&ccb_reset_dev;
                                                   /* where the CCB is */ [4]
    ua_ccb_reset_dev.uagt_ccblen = sizeof(CCB_RESETDEV);
                                                      /* bytes in CCB */
    ua_ccb_reset_dev.uagt_buffer = (u_char *)NULL;
                                                      /* no data */
    ua_ccb_reset_dev.uagt_buflen = 0;                 /* no data */
    ua_ccb_reset_dev.uagt_snsbuf = (u_char *)NULL;
                                                      /* no Autosense data */
    ua_ccb_reset_dev.uagt_snslen = 0;                 /* no Autosense data */
    ua_ccb_reset_dev.uagt_cdb    = (CDB_UN *)NULL;
                                                      /* CDB is in the CCB */
    ua_ccb_reset_dev.uagt_cdblen = 0;                 /* CDB is in the CCB */
/* -- End of static setups of SIMQ Release and Device Reset structures -- */
```

[1] This section of code fills in some of the CCB_HEADER fields of the CCB_RELSIM structure defined as `ccb_sim_rel`, for the XPT_REL_SIMQ command. The structure was declared in Section 2.5.1.2.

[2] This section of code fills in the UAGT_CAM_CCB structure defined as `ua_ccb_sim_rel`, for the UAGT_CAM_IO ioctl command. The structure was declared in Section 2.5.1.2.

[3] This section of code fills in some of the CCB_HEADER fields of the CCB_RESETDEV structure defined as `ccb_reset_dev`, for the XPT_RESET_DEV command. The structure was declared in Section 2.5.1.2.

4 This section of code fills in the UAGT_CAM_IO structure defined as `ua_ccb_reset_dev`, for the UAGT_CAM_IO ioctl command. The structure was declared in Section 2.5.1.2.

2.5.2.9 The Window Parameters Setup Section

This section describes the portion of the User Agent sample inquiry application program that fills in the scan window parameters and sends a SCSI SET WINDOW PARAMETERS command to the scanner.

```
/* Fill in window parameters for scanner and send DEFINE WINDOW */
/* PARAMETERS command to the scanner.  Note that the X&Y resolution */
/* and the X width and Y length are specified on the command line. */

    WWidth = Xwid*(double)WXYRes;    /* X width inches to pixels */ 1
    WLength = Ylen*(double)WXYRes;   /* Y length inches to lines */
    WHdrSel = 0;                     /* Don't return header */
#ifdef NO_HEADER_FOR_NOW
    WHdrSel = 1;                     /* Return header w. data */
#endif

    WinPix = WWidth*WLength;         /* Pixels in window */ 2
    LineBytes = WWidth*3;            /* Full color, 8-bit pixels */
    TotalBytes = WHdrSel*256 + WinPix*3;  /* Full color, 8-bit pixels */

    printf("Window parameters:\n"); 3
    printf(" Width = %6d pixels/line, Length = %6d lines;
                                         Total = %10d pixels\n",
      WWidth, WLength, WinPix);
    printf(" Bytes/line = %6d; Total bytes/image = %10d\n", LineBytes,
      TotalBytes);

/* Fill in window parameters for scanner and
                                 send DEFINE WINDOW PARAMETERS */
/* command to the scanner. */

    clear_mem(&Window, sizeof(Window));     /* Clear whole DWP block */ 4
    swap_short_store(&Window.WDBLen[0], 0x2F); /* REQUIRED length */ 5
    swap_short_store(&Window.XRes[0], WXYRes);
                                         /* X and Y MUST BE THE SAME */
    swap_short_store(&Window.YRes[0], WXYRes);
                                         /* X and Y MUST BE THE SAME */
    /* Upper Left X & Y left at zero */
    swap_long_store(&Window.Width[0], WWidth);
    swap_long_store(&Window.Length[0], WLength);
    Window.ImgTyp = 5;                /* Multi-level full color */ 6
    Window.PixBits = 8;               /* 8-bit pixels */ 7
    Window.PadTyp = 1;                /* REQUIRED value */ 8
    Window.RevImg = 1;                /* Reverse == 0,0,0 = black */ 9
    Window.HdrSel = WHdrSel;          /* Set return header control */ 10
    /* All other values left at zero */

/* Display current contents of bytes in window parameter block */ 11

    printf("Window Parameter block (in hex):\n");
      for( i=0, bp=(u_char *)&Window; i < sizeof(Window); i++, bp++) {
          printf("%.2x ", *bp);
          if (i == 7) printf("\n");
          if (i == 8+21) printf("\n");
      }
    printf("\n\n" );
```

[1] This section converts the X and Y values entered from the command line in inches into pixels. The value of `WXYRes` is an `int`; however, the values of `Xwid` and `Ylen` are floating point values. To perform the calculations to determine the values of `WWidth`, the number of pixels per line, and `WLength`, the number of scan lines, the value of `WXYRes` must be converted to a `real` number. For example, if the value entered for `X` were 4.5 and the resolution selected were 300, `WWidth` would equal 1,350 pixels per line. If the value entered for `Y` were 3.5, the result would be 1,050 scan lines.

[2] This section of the program calculates the number of bytes in the scan window based on the total number of pixels. For example, the calculation using the previous figures would yield 1,417,500 pixels as the value of `WinPix`. To calculate the number of bytes per line, `WWidth` is multiplied by 3, which is the number of bytes per pixel. The total number of bytes in the scan window, using the figures in the example, would be 4,252,500 bytes.

[3] These lines display the results of the calculations.

[4] This line calls the `clear_mem` function to set the local WINDOW_PARAM_BLOCK structure, `Window`, to 0's (zeroes) in preparation for storing the byte values in swapped order. The WINDOW_PARAM_BLOCK structure was defined in Section 2.5.2.1. The `clear_mem` function is defined in Section 2.5.2.14.

[5] This section of code calls the functions that put the bytes of short and long integer values into big-endian storage. The functions are defined in Section 2.5.2.14.

[6] This line sets the image type for the scanner. The setting of 5 means multilevel, full color.

[7] This line sets the number of bits per pixel. The setting of 8 means 256 shades.

[8] This line sets the padding type for nonbyte pixels. The setting of 1 means pad with 0 (zero).

[9] This line sets the reverse image. The setting of 1 means white pixels are indicated by 1 (one) and black pixels are indicated by 0 (zero).

[10] This line sets the selection for returning a header with the data. The setting of `WHdrSel` was set to 0 (do not include the header).

[11] This section displays the contents of the bytes in the window parameter block.

2.5.2.10 CCB Setup for the DEFINE WINDOW Command

This section describes the portion of the User Agent sample scanner program where the fields of the CCB_HEADER needed for an XPT_SCSI_IO request are filled in.

```
/* Set up the CCB for an XPT_SCSI_IO request.  The DEFINE WINDOW
   PARAMETERS command will be sent to the device.  */

/* Set up the CAM header for the XPT_SCSI_IO function. */

    ccb.cam_ch.my_addr = (struct ccb_header *)&ccb;
                                              /* "Its" address */ 1
    ccb.cam_ch.cam_ccb_len = sizeof(CCB_SCSIIO);   /* a SCSI I/O CCB */
    ccb.cam_ch.cam_func_code = XPT_SCSI_IO;        /* the opcode */
    ccb.cam_ch.cam_path_id   = id;              /* selected bus */
    ccb.cam_ch.cam_target_id = targid;        /* selected target */
    ccb.cam_ch.cam_target_lun = lun;               /* selected lun */

/* The needed CAM flags are: CAM_DIR_OUT -
                                 The data will go to the target. */

    ccb.cam_ch.cam_flags = CAM_DIR_OUT;

/* Set up the rest of the CCB for the DEFINE WINDOW PARAMETERS
     command. */

    ccb.cam_data_ptr = (u_char *)&Window;
                                   /* where the parameters are */ 2
    ccb.cam_dxfer_len = sizeof(Window); /* how much data */ 3
    ccb.cam_timeout = CAM_TIME_DEFAULT; /* use the default timeout */ 4
    ccb.cam_cdb_len = sizeof(SCAN_DEF_WIN_CDB);
                                   /* how many bytes for cdb */ 5
    ccb.cam_sense_ptr = &sense[0];      /* Autosense data area */
    ccb.cam_sense_len = SENSE_LEN;      /* Autosense data length */

/* Use a local pointer to access the fields in the DEFINE WINDOW
     PARAMETERS CDB. */

    win = (SCAN_DEF_WIN_CDB *)&ccb.cam_cdb_io.cam_cdb_bytes[0]; 6

    clear_mem(win,sizeof(SCAN_DEF_WIN_CDB));
                                   /* clear all bits in CDB */ 7
    win->opcode = SCAN_DEF_WIN_OP;      /* define window command */ 8
    win->lun = lun;                     /* lun on target */
    win->param_len0 = sizeof(Window);   /* for the buffer space */
    win->param_len1 = 0;
    win->param_len2 = 0;
    win->control    = 0;                /* no control flags */

/* Set up the fields for the User Agent Ioctl call. */ 9

    ua_ccb.uagt_ccb = (CCB_HEADER *)&ccb;
                                  /* where the CCB is */ 10
    ua_ccb.uagt_ccblen = sizeof(CCB_SCSIIO);
                                  /* how many bytes to gather */ 11
    ua_ccb.uagt_buffer = (u_char *)&Window;
                                  /* where the parameters are */ 12
    ua_ccb.uagt_buflen = sizeof(Window);
                                  /* how much data */ 13

    ua_ccb.uagt_snsbuf = &sense[0];    /* Autosense data area */ 14
    ua_ccb.uagt_snslen = SENSE_LEN;    /* Autosense data length */
    ua_ccb.uagt_cdb    = (CDB_UN *)NULL;
                                  /* CDB is in the CCB */ 15
    ua_ccb.uagt_cdblen = 0;           /* CDB is in the CCB */
```

1 This section of code fills in some of the CCB_HEADER fields of the SCSI I/O CCB structure defined as `ccb`, for processing by the XPT layer. The structure was declared in Section 2.5.1.2.

2 This line assigns the `cam_data_ptr` member of the local CCB_SCSIIO data structure, `ccb`, to the address of the `Window` parameter block. The `Window` parameter block structure was filled in Section 2.5.2.9.

3 This line sets the data transfer length to the length of the `Window` structure.

4 This line specifies using the default timeout, which is the value assigned to the CAM_TIME_DEFAULT constant. This constant is set in the `/usr/sys/include/io/cam/cam.h` file to indicate that the SIM layer's default timeout is to be used. The current value of the SIM layer's default timeout is five seconds.

5 This line sets the length of the `cam_cdblen` member to the length of the SCAN_DEF_WIN_CDB structure.

6 This line assigns the `win` pointer, which is type SCAN_DEF_WIN_CDB, to the address of the `cam_cdb_bytes` member of the CDB_UN union. This union is defined in `/usr/sys/include/io/cam/cam.h` as the `cam_cdb_io` member of the SCSI I/O CCB structure.

7 This line calls the `clear_mem` function to clear the local SCAN_DEF_WIN_CDB structure in preparation for storing the values needed for the DEFINE WINDOW operation. The SCAN_DEF_WIN_CDB structure is defined in Section 2.5.2.3. The `clear_mem` function is defined in Section 2.5.2.14.

8 These lines use the `win` pointer to access the bytes of the `cam_cdb_bytes` array as though it is a SCAN_DEF_WIN_CDB structure. The SCAN_DEF_WIN_CDB structure is defined in Section 2.5.2.3.

9 This section of the code assigns the program address of the CCB into the CCB pointer member and the program address of the `Window` parameter block into the data pointer member of the `ua_ccb` structure of type UAGT_CAM_CCB, as defined in the `/usr/sys/include/io/cam/uagt.h` file. This structure is copied from user space into kernel space as part of the `ioctl` system call that is executed in Section 2.5.2.11. This structure was declared in Section 2.5.2.3.

10 This line initializes the `uagt_ccb` member of the `ua_ccb` structure with the address of the local CCB_SCSIIO structure, `ccb`.

11 This line sets the length of the uagt_ccblen member to the length of the SCSI I/O CCB structure that will be used for this call.

12 This line initializes the uagt_buffer member with the user space address of the Window parameter block.

13 This line initializes the uagt_buflen member with the number of bytes in the Window parameter block.

14 These two lines reflect that the autosense features are turned on in the CAM flags.

15 These two lines reflect that the Command Descriptor Block information is in the SCSI I/O CCB structure filled in Section 2.5.1.2.

2.5.2.11 The Error Checking Section

This section describes the portion of the User Agent sample scanner program that attempts to set the window parameters and recover from possible scanner errors.

```
/* Send the CCB to the CAM subsystem using the User Agent driver.
   If an error occurs, report it and attempt corrective action. */

    retry_cnt = 10;                     /* initialize retry counter */
    reset_flag = 0;                     /* initialize reset flag */

retry_SWP:
    printf("Attempt to Set Window Parameters\n");
    if( ioctl(fd, UAGT_CAM_IO, (caddr_t)&ua_ccb) < 0 ) 1
    {
      perror("Error on CAM UAgt Ioctl to Define Window Parameters");
      close(fd);                        /* close the CAM file */
      exit(1);
    }
/* If the CCB did not complete successfully then report the error. */

    if ((ccb.cam_ch.cam_status & CAM_STATUS_MASK) != CAM_REQ_CMP)
    {
      print_ccb_status("CAM UAgt Define Window Ioctl",
        &(ccb.cam_ch) );                /* report the error values */
      printf("  cam_scsi_status = 0x%.2X\n", ccb.cam_scsi_status); 2

/* 1st check if the SIM Queue is frozen.  If it is, release it. */

      if (ccb.cam_ch.cam_status & CAM_SIM_QFRZN) {
          printf("Attempt to release SIM Queue\n");
          if( ioctl(fd, UAGT_CAM_IO, (caddr_t)&ua_ccb_sim_rel) < 0 ) { 3
              perror("Error on CAM UAgt Release Sim Queue Ioctl");
              close(fd);                /* close the CAM file */
              exit(1);
          }

      /* If the Release Sim Q CCB did not complete successfully then
         report the error and exit. */

          print_ccb_status("CAM UAgt Release SIM Queue Ioctl",
            &(ccb_sim_rel.cam_ch) );    /* report the error values */

        if (ccb_sim_rel.cam_ch.cam_status != CAM_REQ_CMP) {
          print_ccb_status("CAM UAgt Release SIM Queue Ioctl",
            &(ccb_sim_rel.cam_ch) );    /* report the error values */ 4
```

```
            close(fd);                  /* close the CAM file */
            exit(1);
        }
    }
/* Next, if we haven't done one yet, attempt a device reset to clear any
   device error. */
    if (reset_flag++ == 0)
    {
        printf("Attempt to Reset the scanner\n");
        if( ioctl(fd, UAGT_CAM_IO, (caddr_t)&ua_ccb_reset_dev) < 0 ) { 5
            perror("Error on CAM UAgt Device Reset Ioctl");
            close(fd);                  /* close the CAM file */
            exit(1);
        }
      /* If the Reset Device CCB did not complete successfully then
         report the error and exit. */
            print_ccb_status("CAM UAgt Device Reset Ioctl",
             &(ccb_reset_dev.cam_ch) );
                                        /* report the error values */
          if (ccb_reset_dev.cam_ch.cam_status != CAM_REQ_CMP) { 6
            print_ccb_status("CAM UAgt Device Reset Ioctl",
             &(ccb_reset_dev.cam_ch) );
                                        /* report the error values */
              close(fd);              /* close the CAM file */
              exit(1);
          }
      /* Wait the 28 seconds that the scanner takes to come back to life
         after a reset; no use to do anything else. */
          printf("Scanner was reset.
                          Wait 28 Seconds for it to recover...\n");
          sleep(28);
    }
/* Last, count if all retries are used up.  If not, try the SWP again.
    If so, give up and exit. */
      printf("Retry counter value = %d\n",retry_cnt);
      if (retry_cnt-- > 0) goto retry_SWP;
      close(fd);                        /* close the CAM file */
      exit(1);
    }
    else
    {
/* Output status information on success for debugging. */
    print_ccb_status("CAM UAgt SET WINDOW PARAMETERS Ioctl",
        &(ccb.cam_ch) );                /* report the error values */
    printf("  cam_scsi_status = 0x%.2X\n", ccb.cam_scsi_status);
    printf("\nWindow parameter set up successful\n");
    }
/* Output header information (magic number, informational comment,
    X and Y dimensions and maximum pixel values) to the data file
    and display it for the user. */
   sprintf(FileHead,"P6\n\# X&Y resolution = %d dpi, %d pixels/line, \
                                      %d lines", 7
      WXYRes,WWidth,WLength);
   sprintf(strchr(FileHead,NULL),"\n%d  %d  255\n",WWidth,WLength);
   write(od,FileHead,strlen(FileHead));
```

```
printf("File header data --\n%s\n",FileHead);
```

1 This section of code attempts to set the window parameters. This line passes the local UAGT_CAM_CCB structure, `ua_ccb`, to the User Agent driver, using the `ioctl` system call. The arguments passed are the file descriptor returned by the `open` system call; the User Agent `ioctl` command, UAGT_CAM_IO, which is defined in the `/usr/sys/include/io/cam/uagt.h` file; and the contents of the `ua_ccb` structure. The User Agent driver copies in the SCSI I/O CCB and sends it to the XPT layer. When the I/O completes, the User Agent returns to the application program, returning status within the `ua_ccb` structure.

2 If the CAM status is anything other than CAM_REQ_CMP, indicating the request completed, an error message is printed indicating the CAM status returned.

3 This section of code attempts to clear the SIM queue if it is frozen. This line passes the local UAGT_CAM_CCB structure, `ua_ccb_sim_rel`, to the User Agent driver, using the `ioctl` system call. The arguments passed are the file descriptor returned by the `open` system call; the User Agent `ioctl` command, UAGT_CAM_IO, which is defined in the `/usr/sys/include/io/cam/uagt.h` file; and the contents of the `ua_ccb_sim_rel` structure. The User Agent driver copies in the SCSI I/O CCB and sends it to the XPT layer. When the operation completes, the User Agent returns to the application program, returning status within the `ua_ccb` structure.

4 If the CAM status is anything other than CAM_REQ_CMP, indicating the request completed, an error message is printed indicating the CAM status returned. An error message is displayed and the program exits.

5 This section of code attempts a device reset. This line passes the local UAGT_CAM_CCB structure, `ua_ccb_reset_dev`, to the User Agent driver, using the `ioctl` system call. The arguments passed are: the file descriptor returned by the `open` system call; the User Agent `ioctl` command, UAGT_CAM_IO, which is defined in the `/usr/sys/include/io/cam/uagt.h` file; and the contents of the `ua_ccb_reset_dev` structure. The User Agent driver copies in the SCSI I/O CCB and sends it to the XPT layer. When the operation completes, the User Agent returns to the application program, returning status within the `ua_ccb` structure.

6 If the CAM status is anything other than CAM_REQ_CMP, indicating the request completed, an error message is printed indicating the CAM status returned. An error message is displayed and the program exits.

7 If the scan window parameters were set up successfully, a portable pixmap P6 file is created. This section displays the X and Y resolutions

in dots per inch, pixels per line, and number of lines, taking the values that were generated from the code in Section 2.5.2.9.

2.5.2.12 CCB Setup for the READ Command

This section describes the portion of the User Agent sample scanner application program that sets up the CCBs for a READ command.

```
/* Set up the CCB for an XPT_SCSI_IO request.  The READ (data) command
     will be sent to the device.  */

/* Set up the CAM header for the XPT_SCSI_IO function. */

    ccb.cam_ch.my_addr = (struct ccb_header *)&ccb;
                                                    /* "Its" address */ 1
    ccb.cam_ch.cam_ccb_len = sizeof(CCB_SCSIIO); /* a SCSI I/O CCB */
    ccb.cam_ch.cam_func_code = XPT_SCSI_IO;      /* the opcode */
    ccb.cam_ch.cam_path_id   = id;               /* selected bus */
    ccb.cam_ch.cam_target_id = targid;           /* selected target */
    ccb.cam_ch.cam_target_lun = lun;             /* selected lun */

/* The needed CAM flags are: CAM_DIR_IN - The data will come from
     the target. */

    ccb.cam_ch.cam_flags = CAM_DIR_IN;

/* Set up the rest of the CCB for the READ command. */

    ccb.cam_data_ptr = (u_char *)ReadData;    /* where the data goes */ 2
    ccb.cam_dxfer_len = LineBytes;      /* how much data */
    ccb.cam_timeout = 100;              /* use timeout of 100Sec */
    ccb.cam_cdb_len = sizeof( SCAN_READ_CDB );
                                        /* how many bytes for read */ 3
    ccb.cam_sense_ptr = &sense[0];      /* Autosense data area */
    ccb.cam_sense_len = SENSE_LEN;      /* Autosense data length */

/* Use a local pointer to access the fields in the DEFINE WINDOW
   PARAMETERS CDB. */

    read = (SCAN_READ_CDB *)&ccb.cam_cdb_io.cam_cdb_bytes[0]; 4

    clear_mem(read,sizeof(SCAN_READ_CDB));  /* clear all bits in CDB */ 5
    read->opcode = SCAN_READ_OP;        /* define window command */
    read->lun = lun;                    /* lun on target */
    read->param_len0 = LineBytes&255;         /* for the buffer space */
    read->param_len1 = (LineBytes>>8)&255;
    read->param_len2 = (LineBytes>>16)&255;
    read->control    = 0;               /* no control flags */

/* Set up the fields for the User Agent Ioctl call. */

    ua_ccb.uagt_ccb = (CCB_HEADER *)&ccb;     /* where the CCB is */ 6
    ua_ccb.uagt_ccblen = sizeof(CCB_SCSIIO);
                                      /* how many bytes to pull in */ 7
    ua_ccb.uagt_buffer = ReadData;     /* where the data goes */ 8
    ua_ccb.uagt_buflen = LineBytes;          /* how much data */ 9

    ua_ccb.uagt_snsbuf = &sense[0];          /* Autosense data area */ 10
    ua_ccb.uagt_snslen = SENSE_LEN;          /* Autosense data length */
    ua_ccb.uagt_cdb    = (CDB_UN *)NULL;     /* CDB is in the CCB */ 11
    ua_ccb.uagt_cdblen = 0;                  /* CDB is in the CCB */

    n = TotalBytes + strlen(FileHead);
    printf("Total bytes in file  =    %12d.\n", n);

    printf("\nRead data from scanner and write to file\n");
```

1 This section of code fills in some of the CCB_HEADER fields of the SCSI I/O CCB structure defined as `ccb`, for processing by the XPT layer. The structure was declared in Section 2.5.1.2.

2 This line sets the `cam_data_ptr` to the address of the `ReadData` array defined in Section 2.5.1.2.

3 This line sets the data transfer length to the length of the SCAN_READ_CDB structure.

4 This line sets the `read` pointer, which is type SCAN_READ_CDB, to the address of the `cam_cdb_len` member of the CDB_UN union. This union is defined in `/usr/sys/include/io/cam/cam.h` as the `cam_cdb_io` member of the SCSI I/O CCB structure.

5 This line calls the `clear_mem` function to clear the local SCAN_READ_CDB structure, `read`, in preparation for storing the values needed for the READ operation. The SCAN_READ_CDB structure was defined in Section 2.5.2.3. The `clear_mem` function is defined in Section 2.5.2.14.

6 These lines use the `read` pointer to access the bytes of the `cam_cdb_bytes` array as though they are in a SCAN_DEF_WIN_CDB structure. The SCAN_READ_CDB structure is defined in Section 2.5.2.3.

7 This line sets the length of the `uagt_ccblen` member to the length of the SCSI I/O CCB structure that will be used for this call.

8 This line sets the `uagt_buffer` member of the `ua_ccb` structure.

9 This line sets the size of the data buffer to the number of bytes contained in the buffer pointed to by the `cam_data_ptr` member of the `ccb` structure.

10 These two lines reflect that the autosense features are turned on in the CAM flags.

11 These two lines reflect that the Command Descriptor Block information is in the SCSI I/O CCB structure filled in Section 2.5.1.2.

2.5.2.13 The Read and Write Loop Section

This section describes the portion of the program where the data is read, reformatted, and placed in the output buffer.

```
/* ****************** Beginning of read/write loop ***************** */
  for (i=0; i<WLength; i++) {
    printf(" Read scanner line number %8d\r",i);
    fflush(stdout); [1]
/* Send the CCB to the CAM subsystem via the User Agent driver,
   and report any errors. */
    if( ioctl(fd, UAGT_CAM_IO, (caddr_t)&ua_ccb) < 0 ) [2]
    {
      perror("\nError on CAM UAgt Ioctl to Read data line");
      close(fd);                    /* close the CAM file */
      exit(1);
    }
/* If the CCB completed successfully then print out the data read,
   if not report the error. */
    if (ccb.cam_ch.cam_status != CAM_REQ_CMP)
    {
      printf("\n");
      print_ccb_status("CAM UAgt Read data line Ioctl",
        &(ccb.cam_ch) );                /* report the error values */
      printf("  cam_scsi_status = 0x%.2X\n", ccb.cam_scsi_status);
      close(fd);  /* close the CAM file */
      exit(1);
    }
    else
    {
#ifdef CUT_FOR_NOW
      printf(" Data line read successfully\n");
#endif
/* Re-format the data from blocks of R, G and B data to tuples
   of (R,G,B) data for the data file.  Set up pointers to the
   beginning of each of the blocks of the Red, the Green and the
   Blue data bytes and another pointer to the output buffer.
   Then loop, collecting one each of Red, Green and Blue,
   putting each into the output data buffer. */ [3]
      RDRp = ReadData;         /* Red bytes are first */
      RDGp = RDRp + WWidth;    /* Green bytes are next */
      RDBp = RDGp + WWidth;    /* Blue bytes are last */
      WDp  = WriteData;
      for (n = 0 ; n < WWidth; n++)
      {
        *WDp++ = *RDRp++;
        *WDp++ = *RDGp++;
        *WDp++ = *RDBp++;
      }
/* Now write the re-formatted data to the output file. */
      write(od,WriteData,LineBytes);  /* write data to file */
    }
} /* ****************** End of read/write loop ***************** */
      printf("\nSuccessful read and write to file\n");
      close(fd);  /* close the CAM file */
      close(od);  /* close the output file */
}
```

[1] This line calls the standard C I/O function, `fflush`, to force the scan line number to the user's display.

2 This section of code attempts to read a scan line. This line passes the local UAGT_CAM_CCB structure, `ua_ccb`, to the User Agent driver, using the `ioctl` system call. The arguments passed are the file descriptor returned by the `open` system call; the User Agent `ioctl` command, UAGT_CAM_IO, which is defined in the `/usr/sys/include/io/cam/uagt.h` file; and the contents of the `ua_ccb` structure. The User Agent driver copies in the SCSI I/O CCB and sends it to the XPT layer. When the I/O completes, the User Agent returns to the application program, returning status within the `ua_ccb` structure.

3 The scan line read in contains all the red bytes, then all the green bytes, then all the blue bytes, in sequence. This section of code reformats the bytes into pixels for the output file by placing a red byte, then a green byte, then a blue byte together on the output file scan line.

2.5.2.14 The Local Function Definition Section

This section describes the portion of the User Agent sample scanner program that defines functions used within the program.

```
/* Local routines and data structure to report in text and Hex form the
returned CAM status. */
struct cam_statustable { 1
        u_char  cam_status;
        caddr_t status_msg;
} cam_statustable[] = {
    {   CAM_REQ_INPROG,         "CCB request is in progress"          },
    {   CAM_REQ_CMP ,           "CCB request completed w/out error" },
    {   CAM_REQ_ABORTED,        "CCB request aborted by the host"   },
    {   CAM_UA_ABORT,           "Unable to Abort CCB request"       },
    {   CAM_REQ_CMP_ERR,        "CCB request completed with an err" },
    {   CAM_BUSY,               "CAM subsystem is busy"              },
    {   CAM_REQ_INVALID,        "CCB request is invalid"             },
    {   CAM_PATH_INVALID,       "Bus ID supplied is invalid"         },
    {   CAM_DEV_NOT_THERE,      "Device not installed/there"         },
    {   CAM_UA_TERMIO,          "Unable to Terminate I/O CCB req"    },
    {   CAM_SEL_TIMEOUT,        "Target selection timeout"           },
    {   CAM_CMD_TIMEOUT,        "Command timeout"              },
    {   CAM_MSG_REJECT_REC,     "Reject received"              },
    {   CAM_SCSI_BUS_RESET,     "Bus reset sent/received"            },
    {   CAM_UNCOR_PARITY,       "Parity error occurred"              },
    {   CAM_AUTOSENSE_FAIL,     "Request sense cmd fail"       },
    {   CAM_NO_HBA,             "No HBA detected Error"              },
    {   CAM_DATA_RUN_ERR,       "Overrun/underrun error"       },
    {   CAM_UNEXP_BUSFREE,      "BUS free"              },
    {   CAM_SEQUENCE_FAIL,      "Bus phase sequence failure"         },
    {   CAM_CCB_LEN_ERR,        "CCB length supplied is inadequate"  },
    {   CAM_PROVIDE_FAIL,       "To provide requ. capability"       },
    {   CAM_BDR_SENT,           "A SCSI BDR msg was sent to target"  },
    {   CAM_REQ_TERMIO,         "CCB request terminated by the host" },
    {   CAM_LUN_INVALID,        "LUN supplied is invalid"      },
    {   CAM_TID_INVALID,        "Target ID supplied is invalid"},
    {   CAM_FUNC_NOTAVAIL,      "Requested function is not available" },
    {   CAM_NO_NEXUS,           "Nexus is not established"           },
```

```
    {   CAM_IID_INVALID,        "The initiator ID is invalid"   },
    {   CAM_CDB_RECVD,          "The SCSI CDB has been received"      },
    {   CAM_SCSI_BUSY,          "SCSI bus busy"                       }
};
int cam_statusentrys = sizeof(cam_statustable) /
sizeof(cam_statustable[0]);
char * camstatus( cam_status )
register u_char cam_status;
{
        register struct cam_statustable *cst = cam_statustable;
        register entrys;
        for( entrys = 0; entrys < cam_statusentrys; cst++ ) {
                if( cst->cam_status == cam_status ) {
                        return( cst->status_msg );
                }
        }
        return( "Unknown CAM Status" );
}
void print_ccb_status(id_string,cp) 2
char *id_string;
CCB_HEADER *cp;
{
        register i;
        printf("Status from %s0,id_string);
        printf(" cam_status = 0x%.2X    (%s%s%s)0, cp->cam_status,
        ((cp->cam_status & CAM_AUTOSNS_VALID) ? "AutoSns Valid-" : "" ),
        ((cp->cam_status & CAM_SIM_QFRZN) ? "SIM Q Frozen-" : "" ),
        camstatus( cp->cam_status & CAM_STATUS_MASK ));
        if (cp->cam_status & CAM_AUTOSNS_VALID) {
                printf("AutoSense Data (in hex):0);
                for( i=0; i < SENSE_LEN; i++)
                        printf("%.2X ", sense[i]);
                printf("0 );
        }
        fflush(stdout);
}
void clear_mem(bp,n)      /* Clear n bytes of memory beginning at bp */ 3
u_char *bp;
int   n;
{
      register i;
      register u_char *ptr;
      for(i=0, ptr=bp; i<n; i++, ptr++) *ptr = 0;
}
void swap_short_store(bp,val)
                          /* Store short into byte-reversed storage */ 4
u_char *bp;
u_short val;
{
      u_short temp;
      register u_char *ptr;
      ptr = bp;                      /* Copy pointer */
      *(bp++) = (u_char)(val>>8);    /* Store high byte first */
      *bp     = (u_char)val;         /* Then store low byte */
}
void swap_long_store(bp,val)
                          /* Store long into byte-reversed storage */ 5
u_char *bp;
```

```
u_long val;
{
      *(bp++) = (u_char)(val>>24); /* Store high byte first */
      *(bp++) = (u_char)(val>>16);
      *(bp++) = (u_char)(val>>8);
      *bp     = (u_char)val;       /* Store low byte last */
}
```

1 This function is described in Section 2.5.1.9.

2 This function prints out the CCB status.

3 This function clears out all the bits in an area of memory, such as a structure or an array, to be sure all are set to 0 (zero) and that there is no extraneous data before executing a SCSI/CAM command.

4 This function puts the bytes of a short (16-bit) integer value into big-endian storage to conform with SCSI byte ordering.

5 This function puts the bytes of a long (32-bit) integer value into byte-reversed storage to conform with SCSI byte ordering.

S/CA Common Modules 3

This chapter describes the common data structures, macros, and routines provided by Digital for SCSI/CAM peripheral device driver writers. These data structures, macros, and routines are used by the generic SCSI/CAM peripheral device driver routines described in Chapter 4.

Using the common and generic routines helps ensure that your SCSI/CAM peripheral device drivers are consistent with the SCSI/CAM Architecture. See Chapter 11 if you plan to define your own SCSI/CAM peripheral device drivers. See Chapter 12 for information about the SCSI/CAM special I/O interface to process special SCSI I/O commands.

3.1 Common SCSI Device Driver Data Structures

This section describes the following SCSI/CAM peripheral common data structures:

- PDRV_UNIT_ELEM, the Peripheral Device Unit Table
- PDRV_DEVICE, the Peripheral Device Structure
- DEV_DESC, the Device Descriptor Structure
- MODESEL_TBL, the Mode Select Table Structure
- DENSITY_TBL, the Density Table Structure
- PDRV_WS, the SCSI/CAM Peripheral Device Driver Working Set Structure

The descriptions provide information only for those members of a data structure that a SCSI/CAM device driver writer needs to understand.

3.1.1 Peripheral Device Unit Table

The Peripheral Device Unit Table is an array of SCSI/CAM peripheral device unit elements. The size of the array is the maximum number of possible devices, which is determined by the maximum number of SCSI controllers allowed for the system.

The structure is allocated statically and is defined as follows:

```
typedef struct pdrv_unit_elem {
   PDRV_DEVICE *pu_device;
                    /* Pointer to peripheral device structure */
   u_short pu_opens;     /* Total number of opens against unit */
   u_short pu_config;     /* Indicates whether the device type */
                          /* configured at this address */
   u_char  pu_type;  /* Device type - byte 0 from inquiry data */
 PDRV_UNIT_ELEM;
```

The pu_device field is filled in with a pointer to a CAM-allocated peripheral SCSI device (PDRV_DEVICE) structure when the first call to the ccmn_open_unit routine is issued for a SCSI device that exists.

3.1.2 Peripheral Device Structure

A SCSI/CAM peripheral device structure, PDRV_DEVICE, is allocated for each SCSI device that exists in the system. This structure contains the queue header structure for the SCSI/CAM peripheral device driver CCB request queue. It also contains the Inquiry data obtained from a GET DEVICE TYPE CCB. Table 3-1 lists the members of the PDRV_DEVICE structure that a SCSI/CAM peripheral device driver writer using the common routines provided by Digital must use. Chapter 11 shows the complete structure for those driver writers who are not using the common routines.

Table 3-1: Members of the PDRV_DEVICE Structure

Member Name	Data Type	Description
pd_dev	dev_t	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device. Passed to the common open routine.
pd_bus	u_char	SCSI target's bus controller number.
pd_target	u_char	SCSI target's ID number.
pd_lun	u_char	SCSI target's logical unit number.
pd_flags	u_long	May be used to indicate the state of a SCSI device driver.
pd_state	u_char	May be used for recovery.
pd_abort_cnt	u_char	May be used for recovery.
pd_dev_inq[INQLEN]	u_char	Inquiry data obtained from issuing a GET DEVICE TYPE CCB.

Table 3-1: (continued)

Member Name	Data Type	Description
*pd_dev_desc	DEV_DESC	Pointer to the SCSI device descriptor.
pd_specific	caddr_t	Pointer to device-specific information.
pd_spec_size	u_long	Size of device-specific information structure.
*(pd_recov_hand)()	void	Recovery handler.
pd_lk_device	lock_t	SMP lock for the device.

The `pd_specific` field is filled in with a pointer to an allocated structure that contains device-specific information.

3.1.2.1 The pd_dev Member

The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

3.1.2.2 The pd_spec_size Member

The size, in bytes, of the device-specific information structure passed from the SCSI device driver to the common open routine.

3.1.3 Device Descriptor Structure

There is a `read-only` SCSI device descriptor structure, DEV_DESC, defined for each device supported by Digital. A user may supply a new DEV_DESC structure by adding it to `/usr/sys/data/cam_data.c` and relinking the kernel. The DEV_DESC structure follows:

```
typedef struct dev_desc {
        u_char  dd_pv_name[IDSTRING_SIZE];
                          /* Product ID and vendor string from */
                          /* Inquiry data */
        u_char  dd_length;
                          /* Length of dd_pv_name string */
        u_char  dd_dev_name[DEV_NAME_SIZE];
                          /* Device name string - see defines */
                          /* in devio.h */
        U32    dd_device_type;
                          /* Bits 0 - 23 contain the device */
                          /* class, bits 24-31 contain the */
                          /* SCSI device type */
        struct  pt_info *dd_def_partition;
```

```
                        /* Default partition sizes - disks */
        U32     dd_block_size;
                        /* Block/sector size  */
        U32     dd_max_record;
                        /* Maximum transfer size in bytes */
                        /* allowed for the device */
        DENSITY_TBL *dd_density_tbl;
                        /* Pointer to density table - tapes */
        MODESEL_TBL *dd_modesel_tbl;
                        /* Mode select table pointer - used */
                        /* on open and recovery */
        U32     dd_flags;
                        /* Option flags (bbr, etc) */
        U32     dd_scsi_optcmds;
                        /* Optional commands supported */
        U32     dd_ready_time;
                    /* Time in seconds for powerup dev ready */
        u_short dd_que_depth;
                        /* Device queue depth for devices */
                        /* which support command queueing */
        u_char  dd_valid;
                        /* Indicates which data length */
                        /* fields are valid */
        u_char  dd_inq_len;
                        /* Inquiry data length for device */
        u_char  dd_req_sense_len;
                        /* Request sense data length for */
                        /* this device */
}DEV_DESC;
```

3.1.4 Mode Select Table Structure

The Mode Select Table Structure is read and sent to the SCSI device when the first call to the SCSI/CAM peripheral open routine is issued on a SCSI device. There can be a maximum of eight entries in the Mode Select Table Structure. Chapter 11 contains a description of each structure member. The definition for the Mode Select Table Structure, MODESEL_TBL, follows:

```
typedef struct modesel_tbl {
        struct ms_entry{
           u_char  ms_page;  /* Page number */
           u_char  *ms_data; /* Pointer to Mode Select data */
           u_char  ms_data_len; /* Mode Select data length */
           u_char  ms_ent_sp_pf;
                             /* Save Page and Page format bits */
                             /* BIT 0   1=Save Page, */
                             /*         0=Don't Save Page */
                             /* BIT 1  1=SCSI-2, 0=SCSI-1 */
     }ms_entry[MAX_OPEN_SELS];
}MODESEL_TBL;
```

3.1.5 Density Table Structure

The Density Table Structure allows for the definition of eight densities for each type of SCSI tape device unit. Chapter 11 contains a description of each structure member. The definition for the Density Table Structure, DENSITY_TBL, follows:

```
typedef struct density_tbl {
        struct density{
           u_char    den_flags;     /* VALID, ONE_FM etc */
           u_char    den_density_code;
           u_char    den_compress_code;
                              /* Compression code if supported */
           u_char    den_speed_setting;
                                   /* for this density */
           u_char    den_buffered_setting;
                                   /* Buffer control setting */
           u_long    den_blocking; /* 0 variable etc. */
        }density[MAX_TAPE_DENSITY];
}DENSITY_TBL;
```

3.1.5.1 The den_blocking Member

The `den_blocking` member contains the blocking factor for this SCSI tape device. A NULL (0) setting specifies that the blocking factor is variable. A positive value represents the number of bytes in a block, for example, 512 or 1024.

3.1.6 SCSI/CAM Peripheral Device Driver Working Set Structure

The SCSI I/O CCB contains `cam_pdrv_ptr`, a pointer to the SCSI/CAM peripheral device driver working set area for the CCB. This structure is also allocated by the XPT when the `xpt_ccb_alloc` routine is called to allocate a CCB. The PDRV_WS structure follows:

```
typedef   struct      pdrv_ws   {
        struct pdrv_ws  *pws_flink;
                              /* Linkage of working set CCBs */
        struct pdrv_ws  *pws_blink;
                              /* that we have queued */
        CCB_SCSIIO      *pws_ccb;
                              /* Pointer to this CCB. */
        u_long          pws_flags;
                              /* Generic to driver */
        u_long          pws_retry_cnt;
                              /* Retry count for this request */
        u_char          *pws_pdrv;
                              /* Pointer to peripheral device */
                              /* structure */
        u_char          pws_sense_buf[DEC_AUTO_SENSE_SIZE];
} PDRV_WS;
```

3.1.6.1 The pws_flink Member

The `pws_flink` member of the `pdrv_ws` structure is a pointer to the forward link of the working set CCBs that have been queued.

3.1.6.2 The pws_blink Member

The `pws_blink` member of the `pdrv_ws` structure is a pointer to the backward link of the working set CCBs that have been queued.

3.1.6.3 The pws_ccb Member

The `pws_ccb` member is a pointer to this CCB. The CCB header is filled in by the common routines.

3.2 Common SCSI Device Driver Macros

The SCSI/CAM peripheral device driver common macros are supplied by Digital for SCSI device driver writers to use. These macros are defined in the `/usr/sys/include/io/cam/pdrv.h` file. There are two categories of macros:

- Macros to obtain identification information about each SCSI device
- Locking macros

Table 3-2 lists each identification macro name, its call syntax, and a brief description of its purpose.

Table 3-2: Common Identification Macros

Name	Syntax	Description
DEV_BUS_ID	DEV_BUS_ID(dev)	Returns the bus ID of the device that is identified in the major/minor device number pair
DEV_TARGET	DEV_TARGET(dev)	Returns the target ID of the device that is identified in the major/minor device number pair
DEV_LUN	DEV_LUN(dev)	Returns the target LUN of the device that is identified in the major/minor device number pair
GET_PDRV_UNIT_ELEM	GET_PDRV_UNIT_ELEM(dev)	

Table 3-2: (continued)

Name	Syntax	Description
		Returns the Peripheral Device Unit Table entry for the device that is identified in the major/minor device number pair
GET_PDRV_PTR	GET_PDRV_PTR(dev)	Returns the pointer to the Peripheral Device Structure for the device that is identified in the major/minor device number pair

Table 3-3 lists each locking macro name, its call syntax, and a brief description of its purpose.

Note

Symmetric Multiprocessing (SMP) is not enabled in this release.

Table 3-3: Common Lock Macros

Name	Syntax	Description
PDRV_INIT_LOCK	PDRV_INIT_LOCK(pd)	Initializes the Peripheral Device Structure lock
PDRV_IPLSMP_LOCK	PDRV_IPLSMP_LOCK(pd, lk_type, saveipl)	Raises the IPL and locks the Peripheral Device Structure
PDRV_IPLSMP_UNLOCK	PDRV_IPLSMP_UNLOCK(pd, saveipl)	Unlocks the Peripheral Device Structure and lowers the IPL
PDRV_SMP_LOCK	PDRV_SMP_LOCK(pd)	Locks the Peripheral Device Structure
PDRV_SMP_SLEEPUNLOCK	PDRV_SMP_SLEEPUNLOCK(chan, pri, pd)	Unlocks the Peripheral Device Structure

3.3 Common SCSI Device Driver Routines

The SCSI/CAM peripheral common device driver routines can be allocated into categories as follows:

- Initialization, open, and close routines, which handle the initialization of SCSI/CAM peripheral device drivers and the common open and close of the drivers. The following routines are in this category:
  - `ccmn_init`
  - `ccmn_open_unit`
  - `ccmn_close_unit`
- CCB queue manipulation routines, which manage placing and removing CCBs from the appropriate queues as well as aborting and terminating I/O for SCSI I/O CCBs on the queue's active list. The following routines are in this category:
  - `ccmn_send_ccb`
  - `ccmn_rem_ccb`
  - `ccmn_abort_que`
  - `ccmn_term_que`
- CCB allocation, build, and deallocation routines, which allocate CCBs, fill in the common portion of the CCB_HEADER, as well as create and send specific types of CCB requests to the XPT. The following routines are in this category:
  - `ccmn_get_ccb`
  - `ccmn_rel_ccb`
  - `ccmn_io_ccb_bld`
  - `ccmn_gdev_ccb_bld`
  - `ccmn_sdev_ccb_bld`
  - `ccmn_sasy_ccb_bld`
  - `ccmn_rsq_ccb_bld`
  - `ccmn_pinq_ccb_bld`
  - `ccmn_abort_ccb_bld`
  - `ccmn_term_ccb_bld`
  - `ccmn_bdr_ccb_bld`
  - `ccmn_br_ccb_bld`
- Common routines to build and send SCSI I/O commands, which are called during the open or recovery sequence of a device. The calling

routine must sleep while the command completes, if necessary. The following routines are in this category:

  - ccmn_tur
  - ccmn_start_unit
  - ccmn_mode_select

- CCB status routine, which assigns CAM status values to a few general classifications. The following routine is in this category:
  - ccmn_ccb_status
- Buf structure pool allocation and deallocation routines, which allocate and deallocate `buf` structures from the buffer pool. The following routines are in this category:
  - ccmn_get_bp
  - ccmn_rel_bp
- Data buffer pool allocation and deallocation routines, which allocate and deallocate data buffer areas from the pool. The following routines are in this category:
  - ccmn_get_dbuf
  - ccmn_rel_dbuf
- Routines used specifically for loadable device drivers. The following routines are in this category:
  - ccmn_check_idle
  - ccmn_find_ctlr
  - ccmn_attach_device
- Routines to perform miscellaneous operations. The following routines are in this category:
  - ccmn_ccbwait
  - ccmn_SysSpecialCmd
  - ccmn_DoSpecialCmd
  - ccmn_errlog

Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

3.3.1 Common I/O Routines

This section describes the common SCSI/CAM peripheral device driver initialization and I/O routines. Table 3-4 lists the name of each routine and gives a summary description of its function. The sections that follow contain a more detailed description of each routine.

Table 3-4: Common I/O Routines

Routine	Summary Description
ccmn_init	Initializes the XPT and the unit table lock structure
ccmn_open_unit	Handles the common open for all SCSI/CAM peripheral device drivers
ccmn_close_unit	Handles the common close for all SCSI/CAM peripheral device drivers

3.3.1.1 The ccmn_init Routine

The ccmn_init routine initializes the XPT and the unit table lock structure. The first time the ccmn_init routine is called, it calls the xpt_init routine to request the XPT to initialize the CAM subsystem.

3.3.1.2 The ccmn_open_unit Routine

The ccmn_open_unit routine handles the common open for all SCSI/CAM peripheral device drivers. It must be called for each open before any SCSI device-specific open code is executed.

On the first call to the ccmn_open_unit routine for a device, the ccmn_gdev_ccb_bld routine is called to issue a GET DEVICE TYPE CCB to obtain the Inquiry data. The ccmn_open_unit routine allocates the Peripheral Device Structure, PDRV_DEVICE, and a device-specific structure, either TAPE_SPECIFIC or DISK_SPECIFIC, based on the device size argument passed. The routine also searches the cam_devdesc_tab to obtain a pointer to the Device Descriptor Structure for the SCSI device and increments the open count. The statically allocated pdrv_unit_table structure contains a pointer to the PDRV_DEVICE structure. The PDRV_DEVICE structure contains pointers to the DEV_DESC structure and to the device-specific structure.

3.3.1.3 The ccmn_close_unit Routine

The ccmn_close_unit routine handles the common close for all SCSI/CAM peripheral device drivers. It sets the open count to zero.

3.3.2 Common Queue Manipulation Routines

This section describes the common SCSI/CAM peripheral device driver queue manipulation routines. Table 3-5 lists the name of each routine and gives a summary description of its function. The sections that follow contain a more detailed description of each routine.

Table 3-5: Common Queue Manipulation Routines

Routine	Summary Description
`ccmn_send_ccb`	Sends CCBs to the XPT layer by calling the `xpt_action` routine
`ccmn_send_ccb_wait`	Sends SCSI I/O CCBs to the XPT layer by calling the `xpt_action` routine and then sleeps while waiting for the CCB to complete. This function assumes that the callback completion function for the SCSI I/O CCB will issue the wakeup.
`ccmn_rem_ccb`	Removes a SCSI I/O CCB request from the SCSI/CAM peripheral driver active queue and starts a tagged request if a tagged CCB is pending.
`ccmn_abort_que`	Sends an ABORT CCB request for each SCSI I/O CCB on the active queue.
`ccmn_term_que`	Sends a TERMINATE I/O CCB request for each SCSI I/O CCB on the active queue.

3.3.2.1 The ccmn_send_ccb Routine

The `ccmn_send_ccb` routine sends CCBs to the XPT layer by calling the `xpt_action` routine. This routine must be called with the Peripheral Device Structure locked.

For SCSI I/O CCBs that are not retries, the request is placed on the active queue. If the CCB is a tagged request and the tag queue size for the device has been reached, the request is placed on the tagged pending queue so that the request can be sent to the XPT at a later time. A high-water mark of half the queue depth for the SCSI device is used for tagged requests so that other initiators on the SCSI bus will not be blocked from using the device. (The queue depth is defined in the device descriptor entry for the device.)

3.3.2.2 The ccmn_send_ccb_wait Routine

The `ccmn_send_ccb_wait` routine sends SCSI I/O CCBs to the XPT layer by calling `xpt_action`. The routine then calls `sleep` to wait for the CCB to complete. This routine must be called with the peripheral device structure locked. The `ccmn_send_ccb_wait` routine requires the callback completion function to issue a wakeup on the address of the CCB. If the sleep priority is greater than PZERO, the `ccmn_send_ccb_wait` routine sleeps at an interruptible priority in order to catch signals.

For SCSI I/O CCBs that are not retries, the request is placed on the active queue. If the CCB is a tagged request and the tag queue size for the device has been reached, the request is placed on the tagged pending queue so that the request can be sent to the XPT at a later time. A high-water mark of half the queue depth for the SCSI device is used for tagged requests so that other initiators on the SCSI bus will not be blocked from using the device. (The queue depth is defined in the device descriptor entry for the device.)

3.3.2.3 The ccmn_rem_ccb Routine

The `ccmn_rem_ccb` routine removes a SCSI I/O CCB request from the SCSI/CAM peripheral driver active queue and starts a tagged request if a tagged CCB is pending. If a tagged CCB is pending, the `ccmn_rem_ccb` routine places the request on the active queue and calls the `xpt_action` routine to start the tagged request.

3.3.2.4 The ccmn_abort_que Routine

The `ccmn_abort_que` routine sends an ABORT CCB request for each SCSI I/O CCB on the active queue. This routine must be called with the Peripheral Device Structure locked.

The `ccmn_abort_que` routine calls the `ccmn_abort_ccb_bld` routine to create an ABORT CCB for the first active CCB on the active queue and send it to the XPT. It calls the `ccmn_send_ccb` routine to send the ABORT CCB for each of the other CCBs on the active queue that are marked as active to the XPT. The `ccmn_abort_que` routine then calls the `ccmn_rel_ccb` routine to return the ABORT CCB to the XPT.

3.3.2.5 The ccmn_term_que Routine

The `ccmn_term_que` routine sends a TERMINATE I/O CCB request for each SCSI I/O CCB on the active queue. This routine must be called with the Peripheral Device Structure locked.

The `ccmn_term_que` routine calls the `ccmn_term_ccb_bld` routine to create a TERMINATE I/O CCB for the first active CCB on the active queue and send it to the XPT. It calls the `ccmn_send_ccb` routine to send the

TERMINATE I/O CCB for each of the other CCBs on the active queue that are marked as active to the XPT. The ccmn_term_que routine then calls the ccmn_rel_ccb routine to return the TERMINATE I/O CCB to the XPT.

3.3.3 Common CCB Management Routines

This section describes the common SCSI/CAM peripheral device driver CCB allocation, build, and deallocation routines. Table 3-6 lists the name of each routine and gives a summary description of its function. The sections that follow contain a more detailed description of each routine.

Table 3-6: Common CCB Management Routines

Routine	Summary Description
ccmn_get_ccb	Allocates a CCB and fills in the common portion of the CCB header
ccmn_rel_ccb	Releases a CCB and returns the sense data buffer for SCSI I/O CCBs, if allocated
ccmn_io_ccb_bld	Allocates a SCSI I/O CCB and fills it in
ccmn_gdev_ccb_bld	Creates a GET DEVICE TYPE CCB and sends it to the XPT
ccmn_sdev_ccb_bld	Creates a SET DEVICE TYPE CCB and sends it to the XPT
ccmn_sasy_ccb_bld	Creates a SET ASYNCHRONOUS CALLBACK CCB and sends it to the XPT
ccmn_rsq_ccb_bld	Creates a RELEASE SIM QUEUE CCB and sends it to the XPT
ccmn_pinq_ccb_bld	Creates a PATH INQUIRY CCB and sends it to the XPT
ccmn_abort_ccb_bld	Creates an ABORT CCB and sends it to the XPT
ccmn_term_ccb_bld	Creates a TERMINATE I/O CCB and sends it to the XPT
ccmn_bdr_ccb_bld	Creates a BUS DEVICE RESET CCB and sends it to the XPT
ccmn_br_ccb_bld	Creates a BUS RESET CCB and sends it to the XPT

3.3.3.1 The ccmn_get_ccb Routine

The ccmn_get_ccb routine allocates a CCB and fills in the common portion of the CCB header. The routine calls the xpt_ccb_alloc routine to allocate a CCB structure. The ccmn_get_ccb routine fills in the common portion of the CCB header and returns a pointer to that CCB_HEADER.

3.3.3.2 The ccmn_rel_ccb Routine

The `ccmn_rel_ccb` routine releases a CCB and returns the sense data buffer for SCSI I/O CCBs, if allocated. The routine calls the `xpt_ccb_free` routine to release a CCB structure. For SCSI I/O CCBs, if the sense data length is greater than the default sense data length, the `ccmn_rel_ccb` routine calls the `ccmn_rel_dbuf` routine to return the sense data buffer to the data buffer pool.

3.3.3.3 The ccmn_io_ccb_bld Routine

The `ccmn_io_ccb_bld` routine allocates a SCSI I/O CCB and fills it in. The routine calls the `ccmn_get_ccb` routine to obtain a CCB structure with the header portion filled in. The `ccmn_io_ccb_bld` routine fills in the SCSI I/O-specific fields from the parameters passed and checks the length of the sense data to see if it exceeds the length of the reserved sense buffer. If it does, a sense buffer is allocated using the `ccmn_get_dbuf` routine.

3.3.3.4 The ccmn_gdev_ccb_bld Routine

The `ccmn_gdev_ccb_bld` routine creates a GET DEVICE TYPE CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The `ccmn_gdev_ccb_bld` routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.5 The ccmn_sdev_ccb_bld Routine

The `ccmn_sdev_ccb_bld` routine creates a SET DEVICE TYPE CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the device type field of the CCB and calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.6 The ccmn_sasy_ccb_bld Routine

The `ccmn_sasy_ccb_bld` routine creates a SET ASYNCHRONOUS CALLBACK CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the asynchronous fields of the

CCB and calls the ccmn_send_ccb routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.7 The ccmn_rsq_ccb_bld Routine

The ccmn_rsq_ccb_bld routine creates a RELEASE SIM QUEUE CCB and sends it to the XPT. The routine calls the ccmn_get_ccb routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the ccmn_send_ccb routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.8 The ccmn_pinq_ccb_bld Routine

The ccmn_pinq_ccb_bld routine creates a PATH INQUIRY CCB and sends it to the XPT. The routine calls the ccmn_get_ccb routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the ccmn_send_ccb routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.9 The ccmn_abort_ccb_bld Routine

The ccmn_abort_ccb_bld routine creates an ABORT CCB and sends it to the XPT. The routine calls the ccmn_get_ccb routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the address of the CCB to be aborted and calls the ccmn_send_ccb routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.10 The ccmn_term_ccb_bld Routine

The ccmn_term_ccb_bld routine creates a TERMINATE I/O CCB and sends it to the XPT. The routine calls the ccmn_get_ccb routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the CCB to be terminated and calls the ccmn_send_ccb routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.11 The ccmn_bdr_ccb_bld Routine

The `ccmn_bdr_ccb_bld` routine creates a BUS DEVICE RESET CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.3.12 The ccmn_br_ccb_bld Routine

The `ccmn_br_ccb_bld` routine creates a BUS RESET CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

3.3.4 Common SCSI I/O Command Building Routines

This section describes the common SCSI/CAM peripheral device driver SCSI I/O command build and send routines. Table 3-7 lists the name of the routine and gives a summary description of its function. The sections that follow contain a more detailed description of each routine.

Table 3-7: Common SCSI I/O Command Building Routines

Routine	Summary Description
`ccmn_tur`	Creates a SCSI I/O CCB for the TEST UNIT READY command and sends it to the XPT for processing and sleeps waiting for its completion.
`ccmn_start_unit`	Creates a SCSI I/O CCB for the START UNIT command and sends it to the XPT for processing and sleeps waiting for its completion.
`ccmn_mode_select`	Creates a SCSI I/O CCB for the MODE SELECT command and sends it to the XPT for processing and sleeps waiting for its completion.

3.3.4.1 The ccmn_tur Routine

The `ccmn_tur` routine creates a SCSI I/O CCB for the TEST UNIT READY command, sends it to the XPT for processing, and waits for it to complete.

The `ccmn_tur` routine calls the `ccmn_io_ccb_bld` routine to obtain a SCSI I/O CCB structure. The `ccmn_tur` routine calls the `ccmn_send_ccb_wait` routine to send the SCSI I/O CCB to the XPT and wait for it to complete.

3.3.4.2 The ccmn_start_unit Routine

The `ccmn_start_unit` routine creates a SCSI I/O CCB for the START UNIT command, sends it to the XPT for processing, and waits for it to complete.

The `ccmn_start_unit` routine calls the `ccmn_io_ccb_bld` routine to obtain a SCSI I/O CCB structure. The `ccmn_start_unit` routine calls the `ccmn_send_ccb_wait` routine to send the SCSI I/O CCB to the XPT and wait for it to complete.

3.3.4.3 The ccmn_mode_select Routine

The `ccmn_mode_select` routine creates a SCSI I/O CCB for the MODE SELECT command, sends it to the XPT for processing and waits for it to complete.

The routine calls the `ccmn_io_ccb_bld` routine to obtain a SCSI I/O CCB structure. It uses the *ms_index"* parameter to index into the Mode Select Table pointed to by the `dd_modesel_tbl` member of the Device Descriptor Structure for the SCSI device. The `ccmn_mode_select` routine calls the `ccmn_send_ccb_wait` routine to send the SCSI I/O CCB to the XPT and wait for it to complete.

3.3.5 Common CCB Status Routine

This section describes the common SCSI/CAM peripheral device driver CCB status routine. The `ccmn_ccb_status` routine assigns individual CAM status values to generic categories. The following table shows the returned category for each CAM status value:

CAM Status	Assigned Category
CAM_REQ_INPROG	CAT_INPROG
CAM_REQ_CMP	CAT_CMP
CAM_REQ_ABORTED	CAT_ABORT
CAM_UA_ABORT	CAT_ABORT
CAM_REQ_CMP_ERR	CAT_CMP_ERR

CAM Status	Assigned Category
CAM_BUSY	CAT_BUSY
CAM_REQ_INVALID	CAT_CCB_ERR
CAM_PATH_INVALID	CAT_NO_DEVICE
CAM_DEV_NOT_THERE	CAT_NO_DEVICE
CAM_UA_TERMIO	CAT_ABORT
CAM_SEL_TIMEOUT	CAT_DEVICE_ERR
CAM_CMD_TIMEOUT	CAT_DEVICE_ERR
CAM_MSG_REJECT_REC	CAT_DEVICE_ERR
CAM_SCSI_BUS_RESET	CAT_RESET
CAM_UNCOR_PARITY	CAT_DEVICE_ERR
CAM_AUTOSENSE_FAIL	CAT_BAD_AUTO
CAM_NO_HBA	CAT_NO_DEVICE
CAM_DATA_RUN_ERR	CAT_DEVICE_ERR
CAM_UNEXP_BUSFREE	CAT_DEVICE_ERR
CAM_SEQUENCE_FAIL	CAT_DEVICE_ERR
CAM_CCB_LEN_ERR	CAT_CCB_ERR
CAM_PROVIDE_FAIL	CAT_CCB_ERR
CAM_BDR_SENT	CAT_RESET
CAM_REQ_TERMIO	CAT_ABORT
CAM_LUN_INVALID	CAT_NO_DEVICE
CAM_TID_INVALID	CAT_NO_DEVICE
CAM_FUNC_NOTAVAIL	CAT_CCB_ERR
CAM_NO_NEXUS	CAT_NO_DEVICE
CAM_IID_INVALID	CAT_NO_DEVICE
CAM_SCSI_BUSY	CAT_SCSI_BUSY
Other	CAT_UNKNOWN

3.3.6 Common Buf Structure Pool Management Routines

This section describes the common SCSI/CAM peripheral device driver `buf` structure pool allocation and deallocation routines.

3.3.6.1 The ccmn_get_bp Routine

The `ccmn_get_bp` routine allocates a `buf` structure. This function must not be called at interrupt context. The function may sleep waiting for resources.

3.3.6.2 The ccmn_rel_bp Routine

The `ccmn_rel_bp` routine deallocates a `buf` structure.

3.3.7 Common Data Buffer Pool Management Routines

This section describes the common SCSI/CAM peripheral device driver data buffer pool allocation and deallocation routines.

3.3.7.1 The ccmn_get_dbuf Routine

The `ccmn_get_dbuf` routine allocates a data buffer area of the size specified by calling the kernel memory allocation routines .

3.3.7.2 The ccmn_rel_dbuf Routine

The `ccmn_rel_dbuf` routine deallocates a data buffer.

3.3.8 Common Routines for Loadable Drivers

This section describes the common SCSI/CAM peripheral device driver routines specific to loadable device drivers. Table 3-8 provides a summary description of the routines specific to loadable drivers.

Table 3-8: Common Routines for Loadable Drivers

Routine	Summary Description
`ccmn_check_idle`	Checks that there are no opens against a device
`ccmn_find_ctlr`	Finds the controller structure that corresponds to the SCSI controller that the device must be attached to
`ccmn_attach_device`	Creates and attaches a device structure to the controller structure that corresponds to the SCSI controller

3.3.8.1 The ccmn_check_idle Routine

The `ccmn_check_idle` routine checks that there are no opens against a device. This routine calls the `ccmn_rel_dbuf` routine to deallocate all structures pertaining to the device whose driver is being unloaded.

The `ccmn_check_idle` routine scans the Peripheral Device Unit Table looking for devices that match the block device major number and the character device major number in the PDRV_DEVICE structure members, `pd_bmajor` and `pd_cmajor`. If no opens exist for the devices that are to be unloaded, it rescans the Peripheral Device Unit Table and deallocates all structures relating to the devices whose driver is being unloaded. The `ccmn_check_idle` routine must be called with the Peripheral Device Unit

Table locked.

3.3.8.2 The ccmn_find_ctlr Routine

The `ccmn_find_ctlr` routine finds the controller structure that corresponds to the SCSI controller that the device must be attached to. This routine must be called with the Peripheral Device Unit Table locked.

3.3.8.3 The ccmn_attach_device Routine

The `ccmn_attach_device` routine creates and attaches a device structure to the controller structure that corresponds to the SCSI controller. The routine finds the controller structure for a device, fills in the device structure, and attaches the device structure to the controller structure.

3.3.9 Miscellaneous Common Routines

This section describes the common SCSI/CAM peripheral device driver routines that perform miscellaneous operations. Table 3-9 lists the name of each routine and gives a summary description of its function.

Table 3-9: Miscellaneous Common Routines

Routine	Summary Description
`ccmn_DoSpecialCmd`	Provides a simplified interface to the special command routine.
`ccmn_SysSpecialCmd`	Lets a system request issue SCSI I/O commands to the SCSI/CAM special I/O interface.
`ccmn_errlog`	Reports error conditions for the SCSI/CAM peripheral device driver.

3.3.9.1 The ccmn_DoSpecialCmd Routine

The `ccmn_DoSpecialCmd` routine provides a simplified interface to the special command routine. The routine prepares for and issues special commands.

3.3.9.2 The ccmn_SysSpecialCmd Routine

The ccmn_SysSpecialCmd routine lets a system request issue SCSI I/O commands to the SCSI/CAM special I/O interface. This permits existing SCSI commands to be issued from within kernel code.

3.3.9.3 The ccmn_errlog Routine

The ccmn_errlog routine reports error conditions for the SCSI/CAM peripheral device driver. The routine is passed a pointer to the name of the function in which the error was detected. The routine builds informational strings based on the error condition.

S/CA Generic Modules 4

This chapter describes the generic data structures and routines provided by Digital for SCSI/CAM peripheral device driver writers. The generic data structures and routines can be used as templates for SCSI/CAM peripheral device drivers to interface with the CAM subsystem to perform standard I/O operations. See Chapter 12 for a description of the SCSI/CAM special I/O interface, which processes special I/O control commands that are not issued to the device through the standard driver entry points.

The generic routines use the common SCSI/CAM peripheral device driver routines described in Chapter 3. Using the common and generic routines helps ensure that SCSI/CAM peripheral device drivers are consistent with the SCSI/CAM Architecture. See Chapter 11 if you plan to define your own SCSI/CAM peripheral device drivers. See Appendix D for the source to the generic driver.

4.1 Prerequisites for Using the CAM Generic Routines

The generic device driver routines use the common routines and data structures supplied by Digital. See Chapter 3 for information about how to use the common data structures and routines.

The following routines must be called with the Peripheral Device Structure locked:

- `ccmn_send_ccb`
- `ccmn_send_ccb_wait`
- `ccmn_abort_que`
- `ccmn_term_que`

4.1.1 Ioctl Commands

The writer of a generic SCSI/CAM peripheral device driver has two options for implementing `ioctl` commands within the driver:

- Use the `ioctl` commands that are already defined in `/usr/sys/include/sys/ioctl.h` and implement those that are appropriate for the type of device.

- Create new `ioctl` definitions by modifying the `/usr/sys/include/sys/ioctl.h` file to reflect the new `ioctl` definitions and to implement the new `ioctl` commands within the driver. See the *Writing Device Drivers, Volume 1: Tutorial* and *Writing Device Drivers, Volume 2: Reference* for more information.

It is possible that conflicts with future releases of the operating system may result when new `ioctl` commands are implemented.

See Chapter 12 for information about the SCSI/CAM special I/O interface to handle SCSI special I/O commands.

4.1.2 Error Handling

The writer of the device driver is responsible for all error handling within the driver and for notifying the user process of the error.

4.1.3 Kernel Interface

The kernel entry points for any device driver are defined for both character and block devices in the structures `cdevsw` and `bdevsw` defined in the `/usr/sys/include/sys/conf.h` file. The kernel entry points are implemented in the `cdevsw` and `bdevsw` switch tables in the `/usr/sys/io/common/conf.c` file. If the device driver does not implement a specific kernel entry point, then the corresponding entries in the `cdevsw` and `bdevsw` switch tables must be null.

4.2 Data Structures Used by Generic Routines

This section describes the generic data structures programmers adapt when they write their own SCSI/CAM peripheral device drivers. The following data structures are described:

- CGEN_SPECIFIC, the Generic-Specific Structure
- CGEN_ACTION, the Generic Action Structure

4.2.1 The Generic-Specific Structure

A SCSI/CAM peripheral device structure, CGEN_SPECIFIC, is defined for the device controlled by the driver. The CGEN_SPECIFIC structure is defined as follows:

```
typedef generic_specific struct {
        u_long  gen_flags;       /* flags - EOM, write locked */
        u_long  gen_state_flags;/* STATE - UNIT_ATTEN, RESET etc. */
        u_long  gen_resid;       /* Last operation residual count */
}CGEN_SPECIFIC;
```

4.2.1.1 The gen_flags Member

The `gen_flags` member is used to indicate certain conditions of the SCSI unit. The possible flags are:

Flag Name	Description
CGEN_EOM	The unit is positioned at the end of media.
CGEN_OFFLINE	The device is returning DEVICE NOT READY in response to a command. The media is either not loaded or is being loaded.
CGEN_WRT_PROT	The unit is either write protected or is opened read only.
CGEN_SOFTERR	A soft error has been reported by the SCSI unit.
CGEN_HARDERR	A hard error has been reported by the SCSI unit. It can be reported either through an `ioctl` or by marking the `buf` structure as EIO.

4.2.1.2 The gen_state_flags Member

The `gen_state_flags` member is used to indicate certain states of the driver and of the SCSI unit. The possible flags are:

Flag Name	Description
CGEN_NOT_READY_STATE	The unit was opened with the FNDELAY flag and the unit had a failure during the open, but was seen.
CGEN_UNIT_ATTEN_STATE	A check condition occurred and the sense key was UNIT ATTENTION. This usually indicates that a media change has occurred, but it could indicate power up or reset. Either way, current settings are lost.
CGEN_RESET_STATE	Indicates notification of a reset condition on the device or bus.
CGEN_RESET_PENDING_STATE	A reset is pending.

Flag Name	Description
CGEN_OPENED_STATE	The unit is opened.

4.2.1.3 The gen_resid Member

The `gen_resid` member contains the residual byte count from the last operation.

4.2.2 The Generic Action Structure

The SCSI/CAM peripheral device structure, CGEN_ACTION, is passed to the generic driver's action routines to be filled in according to the success or failure of the command. The CGEN_ACTION structure definition is:

```
typedef struct generic_action {
        CCB_SCSIIO      *act_ccb;
                              /* The CCB that is returned to caller */
        long            act_ret_error;
                              /* Error code if any */
        u_long          act_fatal;
                              /* Is this considered fatal? */
        u_long          act_ccb_status;
                              /* The CCB status code */
        u_long          act_scsi_status;
                              /* The SCSI error code */
        u_long          act_chkcond_error;
                              /* The check condition error */
}CGEN_ACTION;
```

4.2.2.1 The act_ccb Member

The `act_ccb` member is a pointer to the SCSI I/O CCB returned to the calling routine.

4.2.2.2 The act_ret_error Member

The `act_ret_error` contains the error code, if any, returned from the operation.

4.2.2.3 The act_fatal Member

The act_fatal indicates whether an error returned was fatal. The possible flags are:

Flag Name	Description
ACT_FAILED	The action has failed.
ACT_RESOURCE	Memory availability problem.
ACT_PARAMETER	An invalid parameter was passed.
ACT_RETRY_EXCEDED	The maximum retry count for the operation has been exceeded.

4.2.2.4 The act_ccb_status Member

The act_ccb_status member indicates the CAM generic category code for the CCB that was returned from the ccmn_ccb_status routine.

4.2.2.5 The act_scsi_status Member

The act_scsi_status member indicates the SCSI status code if the CCB completed with an error status. The SCSI status codes are defined in the /usr/sys/include/io/cam/scsi_status.h file.

4.2.2.6 The act_chkcond_error Member

The act_chkcond_error member contains the check condition code returned from the cgen_ccb_chkcond routine, if the cam_scsi_status member of the SCSI I/O CCB is equal to SCSI_STAT_CHECK_CONDITION. The Check Condition codes are defined in the generic.h file shown in Appendix D.

4.3 Generic I/O Routines

The generic routines described in this section handle open, close, read, write, and other I/O requests from user processes. Table 4-1 lists the name of each routine and gives a short description of its function. The sections that follow contain a more detailed description of each routine. Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix C.

Table 4-1: Generic I/O Routines

Routine	Summary Description
cgen_open	Called by the kernel when a user process requests an open of the device.
cgen_close	Closes the device.
cgen_read	Handles synchronous read requests for user processes through the raw interface.
cgen_write	Handles synchronous write requests for user processes through the raw interface.
cgen_strategy	Handles all I/O requests for user processes through the block inerface and the raw interface via a call to physio.
cgen_ioctl	Handles user process requests for specific actions other than read, write, open, or close for generic devices.

4.3.1 The cgen_open Routine

The cgen_open routine is called by the kernel when a user process requests an open of the device. The cgen_open routine calls the ccmn_open_unit routine, which manages the SMP_LOCKS and, if passed the exclusive use flag for SCSI devices, makes sure that no other process has opened the device. If the ccmn_open_unit routine returns success, the necessary data structures are allocated.

The cgen_open routine calls the ccmn_sasy_ccb_bld routine to register for asynchronous event notification for the device. The cgen_open routine then enters a for loop based on the power-up time specified in the Device Descriptor Structure for the device. Within the loop, calls are made to the cgen_ready routine, which calls the ccmn_tur routine to issue a TEST UNIT READY command to the device.

The cgen_open routine calls the ccmn_rel_ccb routine to release the CCB. The cgen_open routine checks certain state flags for the device to decide whether to send the initial SCSI mode select pages to the device. Depending on the setting of the state flags CGEN_UNIT_ATTEN_STATE and CGEN_RESET_STATE, the cgen_open routine calls the cgen_open_sel routine for each mode select page to be sent to the device. The cgen_open_sel routine fills out the Generic Action Structure based on the completion status of the CCB for each mode select page it sends.

4.3.2 The cgen_close Routine

The cgen_close routine closes the device. The routine checks any device flags that are defined to see if action is required, such as rewind on close or release the unit. The cgen_close closes the device by calling the ccmn_close_unit routine.

4.3.3 The cgen_read Routine

The cgen_read routine handles synchronous read requests for user processes. It passes the user process requests to the cgen_strategy routine. The cgen_read routine calls the ccmn_get_bp routine to allocate a buf structure for the user process read request. When the I/O is complete, the cgen_read routine calls the ccmn_rel_bp routine to deallocate the buf structure.

4.3.4 The cgen_write Routine

The cgen_write routine handles synchronous write requests for user processes. The routine passes the user process requests to the cgen_strategy routine. The cgen_write routine calls the ccmn_get_bp routine to allocate a buf structure for the user process write request. When the I/O is complete, the cgen_write routine calls the ccmn_rel_bp routine to deallocate the buf structure.

4.3.5 The cgen_strategy Routine

The cgen_strategy routine handles all I/O requests for user processes. It performs specific checks, depending on whether the request is synchronous or asynchronous and on the SCSI device type. The cgen_strategy routine calls the ccmn_io_ccb_bld routine to obtain an initialized SCSI I/O CCB and build either a read or a write command based on the information contained in the buf structure. The cgen_strategy routine then calls the ccmn_send_ccb to place the CCB on the active queue and send it to the XPT layer.

4.3.6 The cgen_ioctl Routine

The cgen_ioctl routine handles user process requests for specific actions other than read, write, open, or close for SCSI tape devices. The routine currently issues a DEVIOCGET ioctl command for the device, which fills out the devget structure passed in, and then calls the cgen_mode_sns routine which issues a SCSI_MODE_SENSE to the device to determine the device's state. The routine then calls the ccmn_rel_ccb routine to release the CCB. When the call to cgen_mode_sns completes, the cgen_ioctl routine fills out the rest of the devget structure based on information contained in the mode sense data.

4.4 Generic Internal Routines

The generic routines described in this section are examples that show one method of handling errors, events, and conditions. SCSI/CAM peripheral device driver writers must implement routines for handling errors, events, and conditions that are compatible with the design and the functionality of the specific device. Table 4-2 lists the name of each routine and gives a short description of its function. Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

Table 4-2: Generic Internal Routines

Routine	Summary Description
`cgen_ccb_chkcond`	Decodes the autosense data for a device driver
`cgen_done`	The entry point for all nonread and nonwrite I/O callbacks
`cgen_iodone`	The entry point for all read and write I/O callbacks
`cgen_async`	Handles notification of asynchronous events
`cgen_minphys`	Compares the `b_bcount` with the maximum transfer limit for the device
`cgen_slave`	Called at system boot to initialize the lower levels
`cgen_attach`	Called for each bus, target, and LUN after the `cgen_slave` routine returns SUCCESS

4.4.1 The cgen_ccb_chkcond Routine

The `cgen_ccb_chkcond` routine decodes the autosense data for a device driver and returns the appropriate status to the calling routine. The routine is called when a SCSI I/O CCB is returned with a CAM status of CAM_REQ_CMP_ERR (request completed with error) and a SCSI status of SCSI_STAT_CHECK_CONDITION. The routine also sets the appropriate flags in the Generic-Specific Structure.

4.4.2 The cgen_done Routine

The `cgen_done` routine is the entry point for all nonread and nonwrite I/O callbacks. The generic device driver uses two callback entry points, one for all nonuser I/O requests and one for all user I/O requests. The SCSI/CAM peripheral device driver writer can declare multiple callback routines for each type of command and can fill the CCB with the address of the appropriate callback routine.

This is a generic routine for all nonread and nonwrite SCSI I/O CCBs. The SCSI I/O CCB should not contain a pointer to a `buf` structure in the `cam_req_map` member of the structure. If it does, then a wake-up call is issued on the address of the CCB and the error is reported. If the SCSI I/O CCB does not contain a pointer to a `buf` structure in the `cam_req_map` member, then a wake-up call is issued on the address of the CCB and the CCB is removed from the active queues. No CCB completion status is checked because that is the responsibility of the routine that created the CCB and is waiting for completion status. When this routine is entered, context is on the interrupt stack and the driver cannot sleep waiting for an event.

4.4.3 The cgen_iodone Routine

The `cgen_iodone` routine is the entry point for all read and write I/O callbacks. This is a generic routine for all read and write SCSI I/O CCBs. The SCSI I/O CCB should contain a pointer to a `buf` structure in the `cam_req_map` member of the structure. If it does not, then a wake-up call is issued on the address of the CCB and the error is reported. If the SCSI I/O CCB does contain a pointer to a `buf` structure in the `cam_req_map` member, as it should, then the completion status is decoded. Depending on the CCB's completion status, the correct fields within the `buf` structure are filled out.

The device's active queues may need to be aborted because of errors or because the device is a sequential access device and the transaction was an asynchronous request.

The CCB is removed from the active queues by a call to the `ccmn_rem_ccb` routine and is released back to the free CCB pool by a call to the `ccmn_rel_ccb` routine. When the `cgen_iodone` routine is entered, context is on the interrupt stack and the driver cannot sleep waiting for an event.

4.4.4 The cgen_async Routine

The `cgen_async` routine handles notification of asynchronous events. The routine is called when an Asynchronous Event Notification(AEN), Bus Device Reset (BDR), or Bus Reset (BR) occurs. The routine sets the CGEN_RESET_STATE flag and clears the CGEN_RESET_PEND_STATE flag for BDRs and bus resets. The routine sets the CGEN_UNIT_ATTEN_STATE flag for AENs.

4.4.5 The cgen_minphys Routine

The `cgen_minphys` routine compares the `b_bcount` with the maximum transfer limit for the device. The routine compares the `b_bcount` field in the `buf` structure with the maximum transfer limit for the device in the Device Descriptor Structure. The count is adjusted if it is greater than the limit.

4.4.6 The cgen_slave Routine

The `cgen_slave` routine is called at system boot to initialize the lower levels. The routine also checks the bounds for the unit number to ensure it is within the allowed range and sets the device-configured bit for the device at the specified bus, target, and LUN.

4.4.7 The cgen_attach Routine

The `cgen_attach` routine is called for each bus, target, and LUN after the `cgen_slave` routine returns SUCCESS. The routine calls the `ccmn_open_unit` routine, passing the bus, target, and LUN information.

The `cgen_attach` routine calls the `ccmn_close_unit` routine to close the device. If a device of the specified type is found, the device identification string is printed.

4.5 Generic Command Support Routines

The generic routines described in this section are SCSI/CAM command support routines. Table 4-3 lists the name of each routine and gives a short description of its function. The sections that follow contain a more detailed description of each routine. Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

Table 4-3: Generic Command Support Routines

Routine	Summary Description
`cgen_ready`	Issues a TEST UNIT READY command to the unit defined
`cgen_open_sel`	Issues a SCSI_MODE_SELECT command to the SCSI device

Table 4-3: (continued)

Routine	Summary Description
cgen_mode_sns	Issues a SCSI_MODE_SENSE command to the unit defined

4.5.1 The cgen_ready Routine

The cgen_ready routine issues a TEST UNIT READY command to the unit defined. The routine calls the ccmn_tur routine to issue the TEST UNIT READY command and sleeps waiting for command status.

4.5.2 The cgen_open_sel Routine

The cgen_open_sel routine issues a SCSI_MODE_SELECT command to the SCSI device. The mode select data sent to the device is based on the data contained in the Mode Select Table Structure for the device, if one is defined. The CGEN_ACTION structure is filled in for the calling routine based on the completion status of the CCB.

The cgen_open_sel routine calls the ccmn_mode_select routine to create a SCSI I/O CCB and send it to the XPT for processing.

4.5.3 The cgen_mode_sns Routine

The cgen_mode_sns routine issues a SCSI_MODE_SENSE command to the unit defined. The CGEN_ACTION structure is filled in for the calling routine based on the completion status of the CCB.

CAM Data Structures 5

Data structures are the mechanism used to pass information between peripheral device drivers and the CAM subsystem. This chapter describes the CAM data structures used by peripheral device drivers. They are defined in the file `/usr/sys/include/io/cam/cam.h`. This chapter discusses the following:

- CAM Control Block (CCB)
- Input/Output (I/O) data structures
- Control CCB structures
- Configuration data structures

Other chapters reference these structures. You can read this chapter now to become familiar with the structures, or you can refer to it when you encounter references to the structures in other chapters.

5.1 CAM Control Blocks

The CAM Control Block (CCB) data structures let the device driver writer specify the action to be performed by the XPT and SIM. The CCBs are allocated by calling the `xpt_ccb_alloc` routine. Table 5-1 contains the name of each CCB data structure and a brief description.

Table 5-1: CAM Control Blocks

CCB Name	Description
CCB_SCSIIO	Requests SCSI I/O
CCB_GETDEV	Gets device type
CCB_PATHINQ	Sends a path inquiry
CCB_RELSIM	Releases SIM queue
CCB_SETASYNC	Sets asynchronous callback
CCB_SETDEV	Sets device type
CCB_ABORT	Aborts XPT request
CCB_RESETBUS	Resets SCSI bus

Table 5-1: (continued)

CCB_RESETDEV	Resets SCSI device
CCB_TERMIO	Terminates I/O process request

All CCBs contain a CCB_HEADER structure. Peripheral device driver writers need to understand the CCB_HEADER data structure, which is discussed in the section that follows.

5.1.1 The CCB_HEADER Structure

SCSI/CAM peripheral device driver writers allocate a CCB structure by calling the xpt_ccb_alloc routine. The CCB_HEADER structure is common to all CCBs and is the first structure filled in. It contains the following members:

```
typedef struct ccb_header
{
    struct ccb_header *my_addr; /* The address of this CCB */
    u_short cam_ccb_len;        /* Length of the entire CCB */
    u_char cam_func_code;       /* XPT function code */
    u_char cam_status;          /* Returned CAM subsystem */
                                /* status */
    u_char cam_path_id;         /* Path ID for the request */
    u_char cam_target_id;       /* Target device ID */
    u_char cam_target_lun;      /* Target LUN number */
    u_long cam_flags;           /* Flags for operation of */
                                /* the subsystem */
} CCB_HEADER;
```

5.1.1.1 The my_addr and cam_ccb_len Members

The my_addr member is set to a pointer to the virtual address of the starting address of the CAM Control Block (CCB). It is automatically filled in by the xpt_ccb_alloc routine.

The cam_ccb_len member is set to the length in bytes of this specific CCB type. This field is filled in by the ccmn_get_ccb routine. The length includes the my_addr and cam_ccb_len members.

5.1.1.2 The cam_func_code Member

The cam_func_code member lets device-driver writers specify the CCB type XPT/SIM functions. Device-driver writers can set this member to one of the function codes listed in Table 5-2. They are defined in the file /usr/sys/include/io/cam/cam.h.

Table 5-2: CAM Function Codes

Function Code	Meaning
XPT_NOOP	Do not execute anything in the XPT/SIM.
XPT_SCSI_IO	Execute the requested SCSI I/O. Specify the details of the SCSI I/O by setting the appropriate members of the CCB_SCSIIO structure.
XPT_GDEV_TYPE	Get the device type information. Obtain this information by referencing the CCB_GETDEV structure.
XPT_PATH_INQ	Get the path inquiry information. Obtain this information by referencing the CCB_PATHINQ structure.
XPT_REL_SIMQ	Release the SIM queue that is frozen.
XPT_ASYNC_CB	Set the asynchronous callback parameters. Obtain asynchronous callback information from the CCB_SETASYNC structure.
XPT_SDEV_TYPE	Set the device type information. Obtain the device type information from the CCB_SETDEV structure.
XPT_ABORT	Abort the specified CCB. Specify the abort to the CCB by setting the appropriate member of the CCB_ABORT structure.
XPT_RESET_BUS	Reset the SCSI bus.
XPT_RESET_DEV	Reset the SCSI device.
XPT_TERM_IO	Terminate the I/O process. Specify the CCB process to terminate by setting the appropriate member of the CCB_TERMIO structure.

5.1.1.3 The cam_status Member

The `cam_status` member is the action or event that occurred during this CAM Control Block (CCB) request. The `cam_status` member is set by the XPT/SIM after the specified function completes. A `CAM_REQ_INPROG` status indicates that either the function is still executing or is still in the queue. The XPT/SIM can set this member to one of the CAM status codes listed in Table 5-3. They are defined in the file `/usr/sys/include/io/cam/cam.h`.

Table 5-3: CAM Status Codes

CAM Status Code	Meaning
CAM_REQ_INPROG	A CCB request is in progress.
CAM_REQ_CMP	A CCB request completed without errors.
CAM_REQ_ABORTED	A CCB request was aborted by the host processor.
CAM_REQ_UA_ABORT	The SIM was not able to abort the specified CCB.
CAM_REQ_CMP_ERR	The specified CCB request completed with an error.
CAM_BUSY	The CAM subsystem is busy. The CCB returns to the caller; the request must be resubmitted.
CAM_REQ_INVALID	The specified CCB request is not valid.
CAM_PATH_INVALID	The path ID specified in the cam_path_id member of the CCB_HEADER structure is not valid.
CAM_DEV_NOT_THERE	The specified SCSI device is not installed at this location.
CAM_UA_TERMIO	The CAM subsystem was unable to terminate the specified CCB I/O request.
CAM_SEL_TIMEOUT	A target-selection timeout occurred.
CAM_CMD_TIMEOUT	A command timeout occurred.
CAM_MSG_REJECT_REC	A message rejection was received by the SIM.
CAM_SCSI_BUS_RESET	The SCSI bus-reset was issued by the SIM or was seen on the bus by the SIM.
CAM_UNCOR_PARITY	An uncorrectable parity error occurred.
CAM_AUTOSENSE_FAIL	The autosense request-sense command failed.
CAM_NO_HBA	No HBA was detected.
CAM_DATA_RUN_ERR	A data overflow or underflow error occurred.
CAM_UNEXP_BUSFREE	An unexpected bus free was detected.
CAM_SEQUENCE_FAIL	A target bus phase-sequence failure occurred.
CAM_CCB_LEN_ERR	The CCB length specified in the cam_ccb_len member of the CCB_HEADER structure is incorrect.
CAM_PROVIDE_FAIL	The requested capability could not be provided.
CAM_BDR_SENT	A SCSI BDR message was sent to the target.
CAM_REQ_TERMIO	The CCB request was terminated by the host.

Table 5-3: (continued)

CAM Status Code	Meaning
CAM_LUN_INVALID	The LUN supplied is invalid.
CAM_TID_INVALID	The target ID suplied is invalid.
CAM_FUNC_NOTAVAIL	The requested function is not available.
CAM_NO_NEXUS	A nexus has not been established.
CAM_IID_INVALID	The initiator ID is invalid.
CAM_CDB_RECVD	The SCSI CDB has been received.
CAM_SCSI_BUSY	The SCSI bus is busy.
CAM_SIM_QFRZN	The SIM queue is frozen.
CAM_AUTOSNS_VALID	Autosense data is valid for the target.
CAM_STATUS_MASK	The mask bits are only for the status.

5.2 I/O Data Structure

Peripheral device drivers make SCSI device action requests through the following data structures:

- The CCB_SCSIIO structure
- The CDB_UN structure

5.2.1 The CCB_SCSIIO Structure

A peripheral driver indicates to the XPT/SIM that it wants to make a SCSI device action request by setting the `cam_func_code` member of the `CCB_HEADER` structure to the constant `XPT_SCSI_IO`. The peripheral-driver writer then uses the `CCB_SCSIIO` structure to specify the requests.

The `CCB_SCSIIO` structure contains the following members:

```
typedef struct
{
    CCB_HEADER cam_ch;          /* Header information fields */
    u_char *cam_pdrv_ptr;       /* Ptr to the Peripheral driver */
                                /* working set */
    CCB_HEADER *cam_next_ccb;   /* Ptr to the next CCB for action */
    u_char *cam_req_map;        /* Ptr for mapping info on the Req. */
    void (*cam_cbfcnp)();       /* Callback on completion function */
    u_char *cam_data_ptr;       /* Pointer to the data buf/SG list */
    u_long cam_dxfer_len;       /* Data xfer length */
    u_char *cam_sense_ptr;      /* Pointer to the sense data buffer */
    u_char cam_sense_len;       /* Num of bytes in the Autosense buf */
    u_char cam_cdb_len;         /* Number of bytes for the CDB */
```

```
    u_short cam_sglist_cnt;     /* Num of scatter/gather list entries */
    u_long cam_sort;            /* Value used by the SIM to sort on  */
    cam_scsi_status             /* Returned SCSI device status */
    cam_sense_resid             /* Autosense residual length: */
                                        /* two's complement */
    cam_osd_rsvdi[2]            /* OSD reserved field for alignment */
    long cam_resid;             /* Transfer residual length: */
                                        /* two's complement */
    CDB_UN cam_cdb_io;          /* Union for CDB bytes/pointer */
    u_long cam_timeout;         /* Timeout value */
    u_char *cam_msg_ptr;        /* Pointer to the message buffer */
    u_short cam_msgb_len;       /* Num of bytes in the message buf */
    u_short cam_vu_flags;       /* Vendor unique flags */
    u_char cam_tag_action;      /* What to do for tag queuing */
    u_char cam_iorsvd0[3];      /* Reserved field, for alignment */
    u_char cam_sim_priv[ SIM_PRIV ];    /* SIM private data area */
} CCB_SCSIIO;
```

5.2.2 The CDB_UN Structure

The CDB_UN structure contains:

```
typedef union
{
    u_char *cam_cdb_ptr;                /* Pointer to the CDB bytes */
                                                   /* to send */
    u_char cam_cdb_bytes[ IOCDBLEN ];   /* Area for the inline CDB *?
                                                   /* to send */
} CDB_UN;
```

5.3 Control CCB Structures

The control CCB structures allow the driver writer to specify such tasks as resetting the SCSI bus, terminating an I/O process request, and so forth. This section discusses the following control structures:

- CCB_RELSIM
- CCB_SETASYNC
- CCB_ABORT
- CCB_RESETBUS
- CCB_RESETDEV
- CCB_TERMIO

These structures are discussed in the sections that follow.

5.3.1 The CCB_RELSIM Structure

Device-driver writers use the `CCB_RELSIM` structure to release the SIM's internal CCB queue. The CCB_RELSIM structure contains:

```
typedef struct
{
    CCB_HEADER cam_ch;           /* Header information fields */
} CCB_RELSIM;
```

5.3.2 The CCB_SETASYNC Structure

SCSI/CAM peripheral device driver writers use the CCB_SETASYNC structure to set the asynchronous callback for notification of the following events when they occur:

- Unsolicited SCSI BUS DEVICE RESET (BDR)
- Unsolicited RESELECTION
- SCSI AEN (asynchronous event notification enabled)
- Sent BDR to target
- SIM module loaded
- SIM module unloaded
- New devices found

The CCB_SETASYNC structure is defined as follows:

```
typedef struct
{
    CCB_HEADER cam_ch;         /* Header information fields */
    u_long cam_async_flags;    /* Event enables for Callback response */
    void (*cam_async_func)();  /* Async Callback function address */
    u_char *pdrv_buf;          /* Buffer set aside by the */
                               /* peripheral driver */
    u_char pdrv_buf_len;       /* The size of the buffer */
} CCB_SETASYNC;
```

5.3.3 The CCB_ABORT Structure

Device-driver writers use the `CCB_ABORT` structure to abort a CCB that is on the SIM queue. The `CCB_ABORT` structure contains:

```
typedef struct
{
    CCB_HEADER cam_ch;            /* Header information fields */
    CCB_HEADER *cam_abort_ch;     /* Pointer to the CCB to abort */
} CCB_ABORT;
```

5.3.4 The CCB_RESETBUS Structure

Device-driver writers use the `CCB_RESETBUS` structure to reset the SCSI bus. The CCB_RESETBUS structure is defined as follows:

```
typedef struct
{
    CCB_HEADER cam_ch;       /* Header information fields */
} CCB_RESETBUS;
```

5.3.5 The CCB_RESETDEV Structure

Device-driver writers use the CCB_RESETDEV structure to reset a single SCSI device. The CCB_RESETDEV structure is defined as follows:

```
typedef struct
{
    CCB_HEADER cam_ch;       /* Header information fields */
} CCB_RESETDEV;
```

5.3.6 The CCB_TERMIO Structure

Device-driver writers use the CCB_TERMIO structure to terminate an I/O process request. The CCB_TERMIO structure is defined as follows:

```
typedef struct
{
    CCB_HEADER cam_ch;               /* Header information fields */
    CCB_HEADER *cam_termio_ch;       /* Pointer to the CCB to terminate */
} CCB_TERMIO;
```

5.4 Configuration CCB Structures

The configuration CCB structures let the driver writer obtain information such as the device type, version number for the SIM/HBA, and vendor IDS. The following configuration CCBs are described in this section:

- The CCB_GETDEV structure
- The CDB_SETDEV structure
- The CDB_PATHINQ structure

These structures are discussed in the following sections.

5.4.1 The CCB_GETDEV Structure

Device-driver writers use the CCB_GETDEV structure to obtain a device type and inquiry information. The CCB_GETDEV structure is defined as follows:

```
typedef struct
{
    CCB_HEADER cam_ch;          /* Header information fields */
    u_char cam_pd_type;         /* Peripheral device type from the TLUN */
    char *cam_inq_data;         /* Ptr to the inquiry data space */
} CCB_GETDEV;
```

5.4.2 The CCB_SETDEV Structure

Device-driver writers use the CCB_SETDEV structure to set the device type. The CCB_SETDEV structure is defined as follows:

```
typedef struct
{
    CCB_HEADER cam_ch;         /* Header information fields */
    u_char cam_dev_type;       /* Value for the dev type field in EDT */
} CCB_SETDEV;
```

5.4.3 The CCB_PATHINQ Structure

Device-driver writers use the CCB_PATHINQ structure to obtain SIM information such as supported features and version numbers. The CCB_PATHINQ structure is defined as follows:

```
typedef struct
{
   CCB_HEADER cam_ch;                 /* Header information fields */
   u_char cam_version_num;            /* Version number for the SIM/HBA */
   u_char cam_hba_inquiry;            /* Mimic of INQ byte 7 for the HBA */
   u_char cam_target_sprt;            /* Flags for target mode support */
   u_char cam_hba_misc;               /* Misc HBA feature flags */
   u_char cam_vuhba_flags[ VUHBA ];   /* Vendor unique capabilities */
   u_long cam_sim_priv;               /* Size of SIM private data area */
   u_long cam_async_flags;            /* Event cap. for Async Callback */
   u_char cam_hpath_id;               /* Highest path ID in subsystem */
   u_char cam_initiator_id;           /* ID of the HBA on the SCSI bus */
   char cam_sim_vid[ SIM_ID ];        /* Vendor ID of the SIM */
   char cam_hba_vid[ HBA_ID ];        /* Vendor ID of the HBA */
   u_char *cam_osd_usage;             /* Ptr for the OSD specific area */
} CCB_PATHINQ;
```

SCSI/CAM Configuration Driver Modules 6

This chapter describes the data structures and routines used by the Configuration driver to interface with the CAM subsystem. It also describes the `/usr/sys/include/io/cam/cam_config.c` file, which contains SCSI/CAM peripheral device driver configuration information. SCSI/CAM peripheral device driver writers add to this file external declarations and entries to the SCSI/CAM peripheral driver configuration table for their peripheral device drivers.

6.1 Configuration Driver Introduction

The Configuration driver dynamically initializes the XPT and SIM layers of the CAM subsystem, at run time. This enables support for a generic kernel that is configured for all processors and all CAM subsystem software, for example, all HBA drivers. After initialization is complete, the Configuration driver scans the SCSI bus and stores INQUIRY information about each SCSI device detected.

Once the CAM subsystem is initialized and the scanning information stored, the SCSI/CAM peripheral device drivers can use the subsystem. They can determine what devices have been detected and allocate memory appropriately. They can also request resources from the XPT layer using the XPT_GDEV_TYPE and XPT_SDEV_TYPE get and set device information CCBs.

The Configuration driver module logically exists in the SCSI/CAM peripheral device driver layer above the XPT.

6.2 Configuration Driver XPT Interface

The Configuration driver is responsible for supporting the following XPT commands:

- GET DEVICE TYPE CCB
- SET DEVICE TYPE CCB
- SET ASYNCHRONOUS CALLBACK CCB

The Configuration driver also supports the configuration and bus scanning for loaded SIM modules.

6.3 Configuration Driver Data Structures

This section describes the following Configuration driver data structures:

- CCFG_CTRL – The Configuration driver control structure
- EDT – The CAM equipment device table
- CAM_PERIPHERAL_DRIVER – The SCSI/CAM peripheral driver configuration structure

6.3.1 The Configuration Driver Control Structure

The Configuration driver control structure, CCFG_CTRL, contains flags used by the Configuration driver for the scanning process. It also sets aside an area to contain the data returned from the INQUIRY CCBs during the initial scanning process. The structure is defined as follows:

```
typedef struct ccfg_ctrl
{
    u_long ccfg_flags;          /* controlling flags */
    ALL_INQ_DATA inq_buf;       /* scratch area for the INQUIRY data */
    struct lock_t c_lk_ctrl;    /* for locking on the control struct */
} CCFG_CTRL;
```

6.3.1.1 The ccfg_flags Member

The `ccfg_flags` member contains the flags used by the Configuration driver to control operations. The possible settings are as follows:

- EDT_INSCAN – Which signals that an EDT scan is in progress
- INQ_INPROG – Which indicates that an INQUIRY CCB is in progress

6.3.1.2 The inq_buf Member

The `inq_buf` member sets aside a working or temporary area to hold the returned data described in the standard INQUIRY structure, ALL_INQ_DATA, which is defined in the file `/usr/sys/include/io/cam/scsi_all.h`.

6.3.2 The CAM Equipment Device Table

The Configuration driver works with the XPT to allocate, initialize, and maintain the CAM equipment device table structure, EDT. An EDT structure is allocated for each SCSI bus. The structure is an 8x8-element array that contains device inquiry information, asynchronous callback flags, and a signal flag if a device was found, based on the number of targets and the number of LUNs on the SCSI bus. The structure is defined as follows:

```
typedef struct edt
{
    CAM_EDT_ENTRY edt[ NDPS ][ NLPT ];     /* A layer for targets/LUNs */
    u_long edt_flags;                      /* Flags for EDT access */
    u_long edt_scan_count;                 /* # of XPT ASYNC CB readers */
    struct lock_t c_lk_edt                 /* For locking per bus */
} EDT;
```

6.3.2.1 The edt Member

The `edt` member is a structure of the type CAM_EDT_ENTRY, which is defined in the `/usr/sys/include/io/cam/cam.h` file. Each CAM_EDT_ENTRY structure is an entry in the CAM equipment device table containing the SCSI ID and LUN for each device on the SCSI bus. The array dimensions are the number of devices per SCSI bus (NDPS) and the number of LUNs per target (NLPT). The structure and constants are defined in the `/usr/sys/include/io/cam/dec_cam.h` file.

6.3.2.2 The edt_scan_count Member

The `edt_scan_count` member contains the number of processes reading the EDT structure.

6.3.2.3 The edt_flags Member

The `edt_flags` member sets the flags for controlling access to the CAM equipment device table.

6.3.3 The SCSI/CAM Peripheral Driver Configuration Structure

CAM_PERIPHERAL_DRIVER, the SCSI/CAM peripheral driver configuration structure, contains the name of the device and defines the routines that are accessed as part of the system configuration process. The structure is defined as follows:

```
typedef struct cam_peripheral_driver
    {
    char        *cpd_name;
    int         (*cpd_slave)();
    int         (*cpd_attach)();
    int         (*cpd_unload)();
    } CAM_PERIPHERAL_DRIVER;
```

6.3.3.1 The cpd_name Member

The `cpd_name` member is a pointer to the device name contained in the `dev_name` member of the kernel data structure, `device`. See the *Writing Device Drivers, Volume 1: Tutorial* and *Writing Device Drivers, Volume 2:*

Reference for more information.

6.3.3.2 The cpd_slave Member

The `cpd_slave` member is a function pointer to the SCSI/CAM peripheral device driver slave routine, which finds the device attached to the SCSI bus controller.

6.3.3.3 The cpd_attach Member

The `cpd_attach` member is a function pointer to the SCSI/CAM peripheral device driver attach routine, which attaches the device to the controller and initializes the driver fields for the device.

6.3.3.4 The cpd_unload Member

Not implemented.

6.4 The cam_config.c File

The Configuration driver file, `/usr/sys/io/cam/cam_config.c`, contains SCSI/CAM peripheral device driver configuration information. SCSI/CAM peripheral device driver writers edit the file, as the superuser, to add `extern` declarations for their hardware devices and to add entries for the device driver to the SCSI/CAM peripheral driver configuration table.

The section of the file where the `extern` declarations are added looks like the following:

```
extern int crzslave(), crzattach();      /* Disk Driver */
extern int ctzslave(), ctzattach();      /* Tape Driver */
extern int cczslave(), cczattach();      /* CD-ROM Driver */

/* VENDOR: Add the extern declarations for your hardware following this
    comment line. */
```

A sample declaration for third-party SCSI/CAM peripheral device driver might be as follows:

```
extern int toastslave(), toastattach();   /* Non-tape or -disk Driver */
```

The section of the file where the SCSI/CAM peripheral driver configuration table entries are added looks like the following:

```
/*
 * CAM Peripheral Driver Configuration Table.
 */
struct cam_peripheral_driver cam_peripheral_drivers[] = {
        { "crz", crzslave, crzattach },
        { "ctz", ctzslave, ctzattach },
        { "ccz", cczslave, cczattach }

/* VENDOR: Add your hardware entries following this comment line. */
};
```

When you add your entry, be sure to place a comma (,) after the last member in the structure supplied by Digital. A sample entry for third-party hardware might be as follows:

```
        { "ccz", cczslave, cczattach },
/* VENDOR: Add your hardware entries following this comment line. */
      { "wheat", toastslave, toastattach}  /* Non-tape or -disk Driver */
};
```

6.5 Configuration Driver Entry Point Routines

The following Configuration driver routines are entry point routines that are accessible to the XPT and SIM modules as part of the Configuration driver interface. Table 6-1 lists the name of each routine and gives a short description of its function. The sections that follow contain a more detailed description of each routine. Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

Table 6-1: Configuration Driver Entry Point Routines

Routine	Summary Description
`ccfg_slave`	Calls a SCSI/CAM peripheral driver's slave routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found
`ccfg_attach`	Calls a SCSI/CAM peripheral driver's attach routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found
`ccfg_action`	Calls the internal routines that handle any CCB that accesses the CAM equipment device table structure
`ccfg_edtscan`	Issues SCSI INQUIRY commands to all possible SCSI targets and LUNs attached to a bus or a particular `bus/target/lun`.

6.5.1 The ccfg_slave Routine

The `ccfg_slave` routine calls a SCSI/CAM peripheral driver's slave routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found. The routine is called during autoconfiguration. The `ccfg_slave` routine locates the configured driver in the SCSI/CAM peripheral driver configuration table. If the driver is located successfully, the SCSI/CAM peripheral driver's slave routine is called with a pointer to the unit information structure for the device from the kernel device structure and the virtual address of its control and status register (CSR). The SCSI/CAM peripheral driver's slave routine performs its own slave initialization.

6.5.2 The ccfg_attach Routine

The ccfg_attach routine calls a SCSI/CAM peripheral driver's attach routine after a match on the cpd_name member of the CAM_PERIPHERAL_DRIVER structure is found. The routine is called during autoconfiguration. The ccfg_attach routine locates the configured driver in the SCSI/CAM peripheral driver configuration table. If the driver is located successfully, the SCSI/CAM peripheral driver's attach routine is called with a pointer to the unit information structure for the device from the kernel device structure. The SCSI/CAM peripheral driver's attach routine performs its own attach initialization.

6.5.3 The ccfg_action Routine

The ccfg_action routine calls the internal routines that handle any CCB that accesses the CAM equipment device table structure. The CAM function codes supported are XPT_GDEV_TYPE, XPT_SASYNC_CB, and XPT_SDEV_TYPE.

6.5.4 The ccfg_edtscan Routine

The ccfg_edtscan routine issues SCSI INQUIRY commands to all possible SCSI targets and LUNs attached to a bus or a particular bus/target/lun. The routine uses the CAM subsystem in the normal manner by sending SCSI I/O CCBs to the SIMs. The INQUIRY data returned is stored in the EDT structures and the cam_tlun_found flag is set. This routine can be called by the SCSI/CAM peripheral device drivers to reissue a full, partial, or single bus scan command.

CAM XPT I/O Support Routines 7

This chapter contains descriptions of the Transport (XPT) layer routines used by SCSI/CAM device driver writers. Table 7-1 contains a list of the routines with a short description of each. Following the table is a description of each routine. Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

Table 7-1: XPT I/O Support Routines

Routine	Summary Description
`xpt_action`	Calls the appropriate XPT/SIM routine
`xpt_ccb_alloc`	Allocates a CAM Control Block (CCB)
`xpt_ccb_free`	Frees a previously allocated CCB
`xpt_init`	Validates the initialized state of the CAM subsystem

7.1 The xpt_action Routine

The `xpt_action` routine calls the appropriate XPT/SIM routine. The routine routes the specified CCB to the appropriate SIM module or to the Configuration driver, depending on the CCB type and on the path ID specified in the CCB. Vendor-unique CCBs are also supported. Those CCBs are passed to the appropriate SIM module according to the path ID specified in the CCB.

7.2 The xpt_ccb_alloc Routine

The `xpt_ccb_alloc` routine allocates a CAM Control Block (CCB) for use by a SCSI/CAM peripheral device driver. The `xpt_ccb_alloc` routine returns a pointer to a preallocated data buffer large enough to contain any CCB structure. The peripheral device driver uses this structure for its XPT/SIM requests. The routine also ensures that the SIM private data space and peripheral device driver pointer, `cam_pdrv_ptr`, are set up.

7.3 The xpt_ccb_free Routine

The `xpt_ccb_free` routine frees a previously allocated CCB. The routine returns a CCB, previously allocated by a peripheral device driver, to the CCB pool.

7.4 The xpt_init Routine

The `xpt_init` routine validates the initialized state of the CAM subsystem. The routine initializes all global and internal variables used by the CAM subsystem through a call to the Configuration driver. Peripheral device drivers must call this routine either during or prior to their own initialization. The `xpt_init` routine simply returns to the calling SCSI/CAM peripheral device driver if the CAM subsystem was previously initialized.

CAM SIM Modules 8

This chapter describes how the SIM layers handle asynchronous callbacks. It also describes the following SIM routines:

- `sim_action`
- `sim_init`

Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

8.1 SIM Asynchronous Callback Handling

This section describes how the SIM layers handle asynchronous callbacks from the XPT to SCSI/CAM peripheral device drivers when an event such as a SCSI Bus Device Reset (BDR) or an Asynchronous Event Notification (AEN) occurs.

Each SCSI/CAM peripheral device driver registers an asynchronous callback function for each active SCSI device during driver initialization. The SCSI/CAM peripheral device drivers use the `ccmn_sasy_ccb_bld` routine to create a SET ASYNCHRONOUS CALLBACK CCB and send it to the XPT.

The `async_flags` field of the CCB are set to `1` for those events of which the SCSI/CAM peripheral device driver wants to be notified using the asynchronous callback function. The possible `async_flags` settings are:

Flag Name	Description
`AC_FOUND_DEVICES`	A new device was found during a rescan.
`AC_SIM_DEREGISTER`	A previously loaded SIM driver has deregistered.
`AC_SIM_REGISTER`	A loaded SIM driver has registered.
`AC_SENT_BDR`	A bus device reset (BDR) message was sent to the target.
`AC_SCSI_AEN`	A SCSI Asynchronous Event Notification has been received.
`AC_UNSOL_RESEL`	An unsolicited reselection of the system by a device on the bus has occurred.

Flag Name	Description
AC_BUS_RESET	A SCSI bus RESET occurred.

These define statements are in /usr/sys/include/io/cam/cam.h.

8.2 SIM Routines Used by Device Driver Writers

This section describes the SIM routines device driver writers need to understand.

8.2.1 The sim_action Routine

The sim_action routine initiates an I/O request from a SCSI/CAM peripheral device driver. The routine is used by the XPT for immediate as well as for queued operations. for the SCSI I/O CCB, when the operation completes, the SIM calls back directly to the peripheral driver using the CCB callback address, if callbacks are enabled and the operation is not to be carried out immediately.

The SIM determines whether an operation is to be carried out immediately or to be queued according to the function code of the CCB structure. All queued operations, such as ''Execute SCSI I/O'' (reads or writes), are placed by the SIM on a nexus-specific queue and return with a CAM status of CAM_INPROG.

Some immediate operations, as described in the American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module",* working draft, X3T9.2/90-186, may not be executed immediately. However, all CCBs to be carried out immediately return to the XPT layer immediately. For example, the ABORT CCB command does not always complete synchronously with its call; however, the CCB_ABORT is returned to the XPT immediately. An XPT_RESET_BUS CCB returns to the XPT following the reset of the bus.

8.2.2 The sim_init Routine

The sim_init routine initializes the SIM. The SIM clears all its queues and releases all allocated resources in response to this call. This routine is called using the function address contained in the CAM_SIM_ENTRY structure. This routine can be called at any time; the SIM layer must ensure that data integrity is maintained.

8.3 Digital-Specific Features of the SIM Layers

This section describes Digital-specific features of the SIM layers of the CAM subsystem.

8.3.1 SCSI I/O CCB Priorities

In the Digital implementation of the SCSI/CAM architecture, the SIM layer of the CAM subsystem can give priority to certain SCSI I/O CCBs based on the value of the `cam_vu_flags` member of the CCB_SCSIIO. The following priorities are defined in the `/usr/sys/include/io/cam/dec_cam.h` file and can be set by a SCSI/CAM peripheral device driver in the `cam_vu_flags` member of the CCB_SCSIIO:

Flag Name	Description
DEC_CAM_HIGH_PRIOR	This CCB is assigned high priority by the SIM.
DEC_CAM_MED_PRIOR	This CCB is assigned medium priority by the SIM.
DEC_CAM_LOW_PRIOR	This CCB is assigned low priority by the SIM.
DEC_CAM_ZERO_PRIOR	This CCB is not assigned a priority by the SIM.

The Digital SCSI/CAM peripheral disk device driver uses this feature in its `cdisk_strategy` function for reads and writes that do not have the B_ASYNC bit set in the `b_flags` member of the `buf` structure associated with the read or write request.

This feature can be used in conjunction with the Digital SCSI/CAM SCSI I/O CCB reordering feature.

You can disable this feature by setting the `sim_allow_io_priority_sorting` variable in the `/usr/sys/data/cam_data.c` file to 0 (zero).

The following example shows how the SIM performs priority sorting:

1. The SIM queue for the bus, target, and LUN of `0, 0, 0` contains the following SCSI I/O CCB:

   ```
   CCB_SCSIIO #1 - Priority of DEC_CAM_LOW_PRIOR
   ```

2. The SIM receives a second SCSI I/O CCB, CCB_SCSIIO #2, with a priority level of DEC_CAM_HIGH_PRIOR, which is given priority over

CCB_SCSIIO #1. The SIM queue now appears as follows:

```
CCB_SCSIIO #2 - Priority of DEC_CAM_HIGH_PRIOR
CCB_SCSIIO #1 - Priority of DEC_CAM_LOW_PRIOR
```

3. The SIM receives a third SCSI I/O CCB, CCB_SCSIIO #3, with a priority level of DEC_CAM_HIGH_PRIOR. CCB_SCSIIO #3 is given priority over CCB_SCSIIO #1, but is placed after CCB_SCSIIO #2:

```
CCB_SCSIIO #2 - Priority of DEC_CAM_HIGH_PRIOR
CCB_SCSIIO #3 - Priority of DEC_CAM_HIGH_PRIOR
CCB_SCSIIO #1 - Priority of DEC_CAM_LOW_PRIOR
```

8.3.2 SCSI I/O CCB Reordering

In the Digital implementation of the SCSI/CAM architecture, the SIM layer of the CAM subsystem can reorder SCSI I/O CCBs based on a value provided by the SCSI/CAM peripheral device driver. SCSI I/O CCB reordering obtains maximum performance from the device by minimizing the head movement of the device.

The SCSI disk device driver uses SCSI I/O CCB reordering for devices that have the SZ_REORDER flag set in the `dd_flags` member of the device descriptor entry in the `cam_devdesc_tab` device array contained in `/usr/sys/data/cam_data.c.` file. The following SCSI I/O CCBs can be reordered:

- Character and block device reads
- Block device writes

SCSI I/O CCB reordering performed in the SIM layer does not affect any SCSI/CAM peripheral device drivers that do not use it.

The `cam_sort` member has been added to the CCB_SCSIIO structure. This member replaces the `cam_osd_rsvd0` member specified in American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module",* working draft, X3T9.2/90-186.

The SCSI disk driver specifies that a SCSI I/O CCB can be reordered by assigning a value to the `cam_sort` member. Typically, this value is the logical block number (LBN) specified by the Command Descriptor Block for the SCSI I/O CCB. If the `cam_sort` member has a value of 0 (zero), the SCSI I/O CCB is not reordered and no SCSI I/O CCBs are placed before it in the SIM queue for that device. The CAM flag, CAM_SIM_QHEAD, takes priority over the `cam_sort` member. A CCB with the CAM_SIM_QHEAD flag set is always placed at the head of the SIM queue for that device.

You can disable this feature by setting the `sim_allow_io_sorting` variable in the `/usr/sys/data/cam_data.c` file to 0 (zero).

In a busy system, some SCSI I/O CCBs may have to wait if reordering is allowing many other SCSI I/O CCBs to be handled first. The SIM has been configured so that it does not allow a SCSI I/O CCB to wait for more than two seconds. If a SCSI I/O CCB has reached this maximum wait limit, no other SCSI I/O CCBs can be inserted before it. You can change the two-second limit by assigning the desired value to the `sim_sort_age_time` variable in the `/usr/sys/data/cam_data.c` file.

The following example shows how the SIM performs SCSI I/O CCB reordering:

1. Assume that the last `cam_sort` value processed by the SIM was LBN 20.
2. The SIM receives SCSI I/O CCBs with the following `cam_sort` values in the following order:

   ```
   50 23 1 7 28 15 19 60
   ```

3. Sorting produces the following result:

   ```
   23 28 50 60 1 4 7 15 19
   ```

S/CA Error Handling 9

This chapter describes the error-logging macros, data structures, and routines provided by Digital for SCSI/CAM peripheral device driver writers.

9.1 CAM Error Handling Macro

Digital supplies an error-logging macro, CAM_ERROR, with the S/CA software. SCSI device driver writers can activate the macro by defining the constant CAMERRLOG. Errors are reported using the same error-logging interface to each of the modules within the CAM subsystem.

The macro is defined in the `/usr/sys/include/io/cam/cam_errlog.h` file as follows:.

```
static void (*local_errorlog)();
```

The CAM_ERROR() macro presents a consistent error-logging interface to the modules within the CAM subsystem. Using the macro lets all the routines within each module that need to report and log error information use the same macro call and arguments. Using this macro also keeps each reported error string with the code within the module that originally reported the error.

Individual modules contain their own module-specific error-logging routines. Each source file contains a declaration of the pointer to the local error-logging routine as follows:

```
static void (*local_errorlog)();
```

The macro calls the local error-logging routine through the local pointer. The pointer is loaded with the local error-handler address, either within the initialization code for that module or as part of the initialized data. The following example shows the address of the `sx_errorlog` function being loaded to the local error-logging variable, `local_errlog`:

```
extern void sx_errorlog();
static void (*local_errlog)() = sx_errorlog;
```

SCSI/CAM peripheral common modules can declare the local pointer to contain the error handler from another SCSI/CAM peripheral common module.

9.2 CAM Error Logging Structures

This section describes the following CAM error-logging data structures:

- CAM_ERR_ENTRY, the Error Entry Structure
- CAM_ERR_HDR, the Error Header Structure

The structures are defined in the `/usr/sys/include/io/cam/cam_logger.h` file.

9.2.1 The Error Entry Structure

The Error Entry Structure, CAM_ERR_ENTRY, describes an entry in the error log packet. There can be multiple entries in an error log packet. The structure is defined as follows:

```
typedef struct cam_err_entry {
    u_long ent_type;        /* String, TAPE_SPECIFIC, CCB, etc */
    u_long ent_size;        /* Size of the data (CCB, TAPE_SPEC)*/
    u_long ent_total_size;  /* To preserve alignment (uerf) */
    u_long ent_vers;        /* Version number of type  */
    u_char *ent_data;       /* Pointer to whatever string, etc */
    u_long ent_pri;         /* FULL or Brief uerf output */
}CAM_ERR_ENTRY;
```

9.2.1.1 The ent_type Member

The `ent_type` member contains the type of data in the entry, which can be a string, a structure, or a CCB. Numerous types of strings are defined in the `/usr/sys/include/io/cam/cam_logger.h` file. CCBs are assigned to one of the XPT function codes listed in the `/usr/sys/include/io/cam/cam.h` file.

9.2.1.2 The ent_size Member

The `ent_size` member contains the size, in bytes, of the data in the entry.

9.2.1.3 The ent_total_size Member

The `ent_total_size` member preserves long-word alignment for compatibility with the `uerf` error-reporting utility. The `cam_logger` routine fills in this member. See the *System Administration* for more information about the `uerf` utility.

9.2.1.4 The ent_vers Member

The `ent_vers` member is the version number of the contents of the `ent_type` member. See the `#define PDRV_DEVICE_VERS` line in the `/usr/sys/include/io/cam/pdrv.h` file for an example of associating a version number with a structure.

9.2.1.5 The ent_data Member

The ent_data member contains a pointer to the contents of the ent_type member.

9.2.1.6 The ent_pri Member

The ent_pri member contains the output from the uerf utility, which can be in brief or full report format. See the *System Administration* for information about the uerf utility.

9.2.2 The Error Header Structure

The Error Header Structure, CAM_ERR_HDR, contains all the data needed by the uerf utility to determine that the packet is a CAM error log packet. See the *System Administration* for information about the uerf utility. The structure is defined as follows:

```
typedef struct cam_err_hdr {
    u_short hdr_type;        /* Packet type - CAM_ERR_PKT */
    u_long  hdr_size;        /* Filled in by cam_logger */
    u_char  hdr_class;       /* Sub system class Tape, disk,
                                         * sii_dme , etc..
                                         */
    u_long  hdr_subsystem;   /*
                                         * Mostly for controller type
                                         * But the current errlogger uses
                                         * disk tape etc if no controller
                                         * is known.. So what we will do
                                         * is dup the disk and tape types
                                         * in the lower number 0 - 1f and
                                         * the controllers asc sii 5380
                                         * etc can use the uppers.
                                         */
    u_long  hdr_entries;     /* Number of error entries in list*/
    CAM_ERR_ENTRY *hdr_list;       /* Pointer to list of error entries*/
    u_long  hdr_pri;         /* Error logger priority.  */
}CAM_ERR_HDR;
```

9.2.2.1 The hdr_type Member

The hdr_type member contains the error-packet type, which must be CAM_ERR_PKT.

9.2.2.2 The hdr_size Member

The hdr_size member is filled in by the cam_logger routine.

9.2.2.3 The hdr_class Member

The hdr_class member identifies the CAM module that detected the error and assigns it to one of the Defined Device Types listed in the /usr/sys/include/io/cam/scsi_all.h file. The device classes are defined in the /usr/sys/include/io/cam/cam_logger.h file.

9.2.2.4 The hdr_subsystem Member

The hdr_subsystem member identifies the CAM subsystem controller that detected the error and assigns it to one of the Defined Device Types listed in the /usr/sys/include/io/cam/scsi_all.h file. The device classes are defined in the /usr/sys/include/io/cam/cam_logger.h file.

9.2.2.5 The hdr_entries Member

The hdr_entries member contains the number of entries in the header list.

9.2.2.6 The hdr_list Member

The hdr_list member contains a pointer to a list of error entries.

9.2.2.7 The hdr_pri Member

The hdr_pri member identifies the priority of the error and assigns it to one of the priorities listed in the /usr/sys/include/io/cam/errlog.h file.

9.3 Event Reporting

This section contains information about event reporting.

9.3.1 The uerf Utility

To see all the CAM error reports when you use the uerf utility, use the -o full option. For example:

```
uerf -o full | more
```

9.4 The cam_logger Routine

The cam_logger routine allocates a system error log buffer and fills in a uerf error log packet. The routine fills in the bus, target, and LUN information from the Error Header Structure passed to it and copies the Error Header Structure and the Error Entry Structures and data to the error log buffer.

S/CA Debugging Facilities 10

This chapter describes the debugging macros and routines provided by Digital for SCSI/CAM peripheral device driver writers.

10.1 CAM Debugging Variables

There are two levels of debugging within the CAM modules: debugging independent of a bus, target, or LUN, and debugging that tracks a specific bus, target, or LUN. S/CA debugging is turned on by defining the program constant CAMDEBUG in the `/usr/sys/include/io/cam/cam_debug.h` file and recompiling the source files.

This section describes the variables that contain the information for each level of debugging the CAM subsystem. The variables are:

- `camdbg_flag` – Which turns on specific `cprintf` calls within the kernel, depending on its setting, to capture information independent of a particular SCSI ID.
- `camdbg_id` – Which contains the specific bus, target, and LUN information for tracking.

The macros, PRINTD and CALLD, use the variables for tracking target-specific messages and for allowing specific subsets of the DEBUG statements to be printed. The macros are defined in the `/usr/sys/include/io/cam/cam_debug.h` file.

10.1.1 The camdbg_flag Variable

The most significant bit, bit 31, of the `camdbg_flag` variable is the bit that indicates whether the target information is valid. If set, it indicates that the `camdbg_id` variable contains valid bus, target, and LUN information for the device to be tracked. Bits 30 to 0 define the debug flag setting. The possible settings, in ascending hexadecimal order, with a brief description of each, follow:

Flag Name	Description
CAMD_INOUT	Routine entry and exit
CAMD_FLOW	Code flow through the modules
CAMD_PHASE	SCSI phase values
CAMD_SM	State machine settings
CAMD_ERRORS	Error handling
CAMD_CMD_EXP	Expansion of commands and responses
CAMD_IO_MAPPING	Data Movement Engine I/O mapping for user space
CAMD_DMA_FLOW	Data Movement Engine flow
CAMD_DISCONNECT	Signal disconnect handling
CAMD_TAGS	Tag queuing code
CAMD_POOL	XPT tracking in the DEC CAM packet pool
CAMD_AUTOS	Autosense handling
CAMD_CCBALLOC	CCB allocation and free flow
CAMD_MSGOUT	Messages going out
CAMD_MSGIN	Messages coming in
CAMD_STATUS	SCSI status bytes
CAMD_CONFIG	CAM configuration paths
CAMD_SCHED	SIM scheduler points
CAMD_SIMQ	SIM queue manipulation
CAMD_TAPE	SCSI/CAM peripheral tape flow
CAMD_COMMON	SCSI/CAM peripheral common flow
CAMD_DISK	SCSI/CAM peripheral disk flow
CAMD_DISK_REC	SCSI/CAM peripheral disk recovery flow
CAMD_DBBR	SCSI/CAM peripheral disk Dynamic Bad Block Recovery flow
CAMD_CDROM	SCSI/CAM peripheral CDROM functions
CAMD_INTERRUPT	SIM trace Interrupts
TVALID	The bus, target, and LUN bits are valid in the camdbg_id variable

10.1.2 The camdbg_id Variable

The `camdbg_id` variable contains the bus, target, and LUN (B/T/L) information for a specific target to track for debugging information. In the current implementation, the bits are divided into three parts, with the remainder reserved. The bits are allocated as follows: bits 31 to 16, Reserved; bits 15 to 8, Bus number; bits 7 to 4, Target number; and bits 3 to 0, LUN number. Multiples of four bits are used to assign hexadecimal values into the `camdbg_id` variable.

10.2 CAM Debugging Macros

The PRINTD and CALLD macros track target-specific messages and allow specific subsets of the debugging statements to be printed.

This PRINTD macro, which prints debugging information if CAMDEBUG is defined, follows.

```
/*
 * Conditionally Print Debug Information.
 */
#if defined(CAMDEBUG) && !defined(lint)
#   define PRINTD(B, T, L, F, X)
  { \ 1
    /* NOSTRICT */                              \
    if( camdbg_flag & (int)F ) \ 2
    { \
      if( ((camdbg_flag & TVALID) == 0) || \ 3
          (((camdbg_flag & TVALID) != 0) && \ 4
          ((((camdbg_id & BMASK) >> BSHIFT) == B) || (B == NOBTL)) && \ 5
          ((((camdbg_id & TMASK) >> TSHIFT) == T) || (T == NOBTL)) && \
          ((((camdbg_id & LMASK) >> LSHIFT) == L) || (L == NOBTL))) ) \
      { \
      /* VARARGS */ \
      (void)(*cdbg_printf) X ; \
      } \
    } \
  }
#endif
```

1. The B, T, and L arguments are for target-specific tracking. The F argument is a flag for tracking specific subsets of the `printf` statements. The F flag argument is compared with the `camdbg_flag` variable to determine if the user wants to see the message. The X argument must be a complete `printf` argument set enclosed within parentheses ( ) to allow the preprocessor to include it in the final `printf` statement.

2. This statement checks to see if any of the flags for the PRINTD macro are turned on. It does not look for an exact match so that the same PRINTD macro can be used for different settings of the flags in `camdbg_flag`.

3 This section of code checks for any target information available for tracing a target. The first condition checks to see if the target valid bit is not set. If it is not, the OR condition is met and the call to the `printf` utility is made.

4 If the TVALID bit is set, the bus, target, and LUN fields in the `camdbg_id` variable must be compared to the B, T, and L arguments. If TVALID is true and bus equals B, target equals T, and LUN equals L, then also print.

5 This construct checks the B, T, and L fields. For example, the following statement checks the B field:

```
((((camdbg_id & BMASK) >> BSHIFT) == B) || (B == NOBTL))
```

The statement masks out the other fields and shifts the bus value down to allow comparison with the B argument. The arguments can also have a ‘‘wildcard’’ value, NOBTL. When the wildcard value is used, the B or T or L comparison is always true.

The CALLD macro uses the same `if` statement constructs to conditionally call a debugging function using the following `define` statement:

```
#   define CALLD(B, T, L, F, X)
```

The `X` is a call to a CAM debugging routine described in the following section.

10.3 CAM Debugging Routines

The SCSI/CAM peripheral device debugging routines can be allocated into categories as follows:

- Routines that generate reports on CAM functions and status in either a brief form listing the name as it is defined in the applicable header file, or in the form of a sentence. The following routines are in this category:
  - `cdbg_CamFunction`
  - `cdbg_CamStatus`
  - `cdbg_ScsiStatus`
  - `cdbg_SystemStatus`
- Routines that dump the contents of CCBs, SCSI/CAM Peripheral Device Driver Working Set Structures, and other SCSI/CAM commands for examination. The following routines are in this category:
  - `cdbg_DumpCCBHeader`
  - `cdbg_DumpCCBHeaderFlags`

- cdbg_DumpSCSIIO
- cdbg_DumpPDRVws
- cdbg_DumpABORT
- cdbg_DumpTERMIO
- cdbg_DumpBuffer
- cdbg_GetDeviceName
- cdbg_DumpInquiryData

Descriptions of the routines with syntax information, in DEC OSF/1 reference page format, are included in alphabetical order in Appendix D.

10.3.1 CAM Debugging Status Routines

This section describes the SCSI/CAM peripheral device debugging routines that report status. Table 10-1 lists the name of each routine and gives a summary description of its function. The sections that follow contain a more detailed description of each routine.

Table 10-1: CAM Debugging Status Routines

Routine	Summary Description
cdbg_CamFunction	Reports CAM XPT function codes
cdbg_CamStatus	Decodes CAM CCB status codes
cdbg_ScsiStatus	Reports SCSI status codes
cdbg_SystemStatus	Reports system error codes

10.3.1.1 The cdbg_CamFunction Routine

The `cdbg_CamFunction` routine reports CAM XPT function codes. Program constants are defined to allow either the function code name only or a brief explanation to be printed. The XPT function codes are defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.1.2 The cdbg_CamStatus Routine

The `cdbg_CamStatus` routine decodes CAM CCB status codes. Program constants are defined to allow either the status code name only or a brief explanation to be printed. The CAM status codes are defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.1.3 The cdbg_ScsiStatus Routine

The `cdbg_ScsiStatus` routine reports SCSI status codes. Program constants are defined to allow either the status code name only or a brief explanation to be printed. The SCSI status codes are defined in the `/usr/sys/include/io/cam/scsi_status.h` file.

10.3.1.4 The cdbg_SystemStatus Routine

The cdbg_SystemStatus routine reports system error codes. The system error codes are defined in the `/usr/sys/include/sys/errno.h` file.

10.3.2 CAM Dump Routines

This section describes the SCSI/CAM peripheral device debugging routines that dump contents for examination. Table 10-2 lists the name of each routine and gives a summary description of its function. The sections that follow contain a more detailed description of each routine.

Table 10-2: CAM Dump Routines

Routine	Summary Description
`cdbg_DumpCCBHeader`	Dumps the contents of a CAM Control Block (CCB) header structure
`cdbg_DumpCCBHeaderFlags`	Dumps the contents of the `cam_flags` member of a CAM Control Block (CCB) header structure
`cdbg_DumpSCSIIO`	Dumps the contents of a SCSI I/O CCB
`cdbg_DumpPDRVws`	Dumps the contents of a SCSI/CAM Peripheral Device Driver Working Set Structure
`cdbg_DumpABORT`	Dumps the contents of an ABORT CCB
`cdbg_DumpTERMIO`	Dumps the contents of a TERMINATE I/O CCB
`cdbg_DumpBuffer`	Dumps the contents of a data buffer in hexadecimal bytes
`cdbg_GetDeviceName`	Returns a pointer to a character string describing the `dtype` member of an ALL_INQ_DATA structure
`cdbg_DumpInquiryData`	Dumps the contents of an ALL_INQ_DATA structure

10.3.2.1 The cdbg_DumpCCBHeader Routine

The `cdbg_DumpCCBHeader` routine dumps the contents of a CAM Control Block (CCB) header structure. The CAM Control Block (CCB) header structure is defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.2.2 The cdbg_DumpCCBHeaderFlags Routine

The `cdbg_DumpCCBHeaderFlags` routine dumps the contents of the `cam_flags` member of a CAM Control Block (CCB) header structure. The CAM Control Block (CCB) header structure is defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.2.3 The cdbg_DumpSCSIIO Routine

The `cdbg_DumpSCSIIO` routine dumps the contents of a SCSI I/O CCB. The SCSI I/O CCB is defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.2.4 The cdbg_DumpPDRVws Routine

The `cdbg_DumpPDRVws` routine dumps the contents of a SCSI/CAM Peripheral Device Driver Working Set Structure. The SCSI/CAM Peripheral Device Driver Working Set Structure is defined in the `/usr/sys/include/io/cam/pdrv.h` file.

10.3.2.5 The cdbg_DumpABORT Routine

The `cdbg_DumpABORT` routine dumps the contents of an ABORT CCB. The ABORT CCB is defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.2.6 The cdbg_DumpTERMIO Routine

The `cdbg_DumpTERMIO` routine dumps the contents of a TERMINATE I/O CCB. The TERMINATE I/O CCB is defined in the `/usr/sys/include/io/cam/cam.h` file.

10.3.2.7 The cdbg_DumpBuffer Routine

The `cdbg_DumpBuffer` routine dumps the contents of a data buffer in hexadecimal bytes. The calling routine must display a header line. The format of the dump is 16 bytes per line.

10.3.2.8 The cdbg_GetDeviceName Routine

The `cdbg_GetDeviceName` routine returns a pointer to a character string describing the `dtype` member of an ALL_INQ_DATA structure. The ALL_INQ_DATA structure is defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

10.3.2.9 The cdbg_DumpInquiryData Routine

The `cdbg_DumpInquiryData` routine dumps the contents of an ALL_INQ_DATA structure. The ALL_INQ_DATA structure is defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

Programmer-Defined SCSI/CAM Device Drivers 11

This chapter describes how programmers can write their own device drivers for SCSI/CAM peripheral devices using a combination of common data structures and routines provided by Digital and programmer-defined routines and data structures. This chapter describes only the programmer-defined data structures and routines. See Chapter 3 for a description of the common data structures and routines.

The chapter also describes how to add a programmer-defined device driver to the S/CA system.

11.1 Programmer-Defined SCSI/CAM Data Structures

This section describes the SCSI/CAM peripheral data structures programmers must use if they write their own device drivers. The following data structures are described:

- PDRV_UNIT_ELEM – The Peripheral Device Unit Table
- PDRV_DEVICE – The Peripheral Device Structure
- DEV_DESC – The Device Descriptor Structure
- DENSITY_TBL – The Density Table Structure
- MODESEL_TBL – The Mode Select Table Structure

11.1.1 Programmer-Defined Peripheral Device Unit Table

The Peripheral Device Unit Table is an array of SCSI/CAM peripheral device unit elements. The size of the array is the maximum number of possible devices, which is determined by the maximum number of SCSI controllers allowed for the system. The structure is allocated statically and is defined as follows:

```
typedef struct pdrv_unit_elem {
        PDRV_DEVICE *pu_device;
                                /* Pointer to peripheral device structure */
        u_short pu_opens;       /* Total number of opens against unit */
        u_short pu_config;
                                /* Indicates whether the device type */
                                /* configured at this address */
        u_char  pu_type;        /* Device type - byte 0 from inquiry data */
}    PDRV_UNIT_ELEM;
```

11.1.1.1 The pu_device Member

The `pu_device` field is filled in with a pointer to a CAM-allocated peripheral SCSI device (PDRV_DEVICE) structure when the first call to the `ccmn_open_unit` routine is issued for a SCSI device that exists.

11.1.1.2 The pu_opens Member

The total number of opens against the unit.

11.1.1.3 The pu_config Member

Indicates whether a device of the specified type is configured at this bus/target/LUN.

11.1.1.4 The pu_type Member

The device type from byte 0 (zero) of the Inquiry data.

11.1.2 Programmer-Defined Peripheral Device Structure

A SCSI/CAM peripheral device structure, PDRV_DEVICE, is allocated for each SCSI device that exists in the system. The PDRV_DEVICE structure is defined as follows:

```
typedef struct pdrv_device {
        PD_LIST pd_active_list;
                         /* Forward active pointer of CCBs */
                         /* which have been sent to the XPT */
        U32      pd_active_ccb;
                         /* Number of active CCBs on queue */
        U32      pd_que_depth;
                         /* Tagged queue depth - indicates the */
                         /* maximum number of commands the unit */
                         /* can store internally */
        PD_LIST pd_pend_list;
                         /* Forward active pointer of pending CCBs */
                         /* which have not been sent to the XPT due */
                         /* to a full queue for tagged requests */
        U32      pd_pend_ccb;
                         /* Number of pending CCBs */
        dev_t   pd_dev;  /* CAM major/minor number */
        u_char  pd_bus;  /* SCSI controller number */
        u_char  pd_target;
                         /* SCSI target id */
        u_char  pd_lun;  /* SCSI target lun */
        u_char  pd_unit; /* Unit number */
        U32     pd_log_unit;
                         /* Logical Unit number */
        U32      pd_soft_err;
                         /* Number of soft errors */
```

```
        U32      pd_hard_err;
                         /* Number of hard errors */
        u_short pd_soft_err_limit;
                         /* Max no. of soft errors to report */
        u_short pd_hard_err_limit;
                         /* Max no. of hard errors to report */
        U32      pd_flags;
                         /* Specific to peripheral drivers */
        u_char  pd_state;
                         /* Specific to peripheral drivers - can */
                         /* be used for recovery */
        u_char  pd_abort_cnt;
                         /* Specific to peripheral drivers - can */
                         /* be used for recovery */
        U32      pd_cam_flags;
                         /* Used to hold the default settings */
                         /* for the cam_flags field in CCBs */
        u_char  pd_tag_action;
                         /* Used to hold the default settings for */
                         /* the cam_tag_action field of the SCSI */
                         /* I/O CCB */
        u_char  pd_dev_inq[INQLEN];
                         /* Inquiry data obtained from GET */
                         /* DEVICE TYPE CCB */
        U32      pd_ms_index;
                         /* Contains the current index into the */
                         /* Mode Select Table when sending Mode */
                         /* Select data on first open */
        DEV_DESC  *pd_dev_desc;
                         /* Pointer to our device descriptor */
        caddr_t pd_specific;
                         /* Pointer to device specific info */
        u_short pd_spec_size;
                         /* Size of device specific info */
        caddr_t pd_sense_ptr;
                         /* Pointer to the last sense data */
                         /* bytes retrieved from device */
        u_short pd_sense_len;
                         /* Length of last sense data */
        void    (*pd_recov_hand)();
                         /* Specific to peripheral drivers - can */
                         /* be used to point to the recovery */
                         /* handler for the device */
        U32      pd_read_count;
                         /* Number of reads to device */
        U32      pd_write_count;
                         /* Number of writes to device */
        U32      pd_read_bytes;
                         /* Number of bytes read from device */
        U32      pd_write_bytes;
                         /* Number of bytes written to device */
        dev_t    pd_bmajor;
                         /* Block major number for loadables */
        dev_t    pd_cmajor;
                         /* Char major number for loadables */
        BOP_LOCK_STRUCT pd_lk_device;
                         /* SMP lock for the device */
} PDRV_DEVICE
```

The structure members and their descriptions follow:

Structure Member	Description
pd_active_list	A pointer to the first CCB on the active queue.
pd_active_ccb	The number of CCBs on the active queue.
pd_que_depth	The depth of the tagged queue, which is the maximum number of commands that the peripheral driver will send to the SCSI device.
pd_pend_list	A pointer to the first CCB on the pending queue.
pd_pend_ccb	The number of CCBs on the pending queue.
pd_dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.
pd_bus	SCSI target's bus controller number.
pd_target	SCSI target's ID number.
pd_lun	SCSI target's logical unit number.
pd_unit	SCSI device's unit number.
pd_log_unit	Logical Unit Number
pd_soft_err	Number of soft errors reported by each SCSI unit.
pd_hard_err	Number of hard errors reported by each SCSI unit.
pd_soft_err_limit	Maximum number of soft errors that can be reported by each SCSI unit.
pd_hard_err_limit	Maximum number of hard errors that can be reported by each SCSI unit.
pd_flags and pd_state	These are specific to SCSI/CAM peripheral device drivers. They can be used for recovery.
pd_abort_cnt	This is specific to SCSI/CAM peripheral device drivers. It can be used for recovery.
pd_cam_flags	This contains the default settings for the cam_flags field in the CAM Control Block (CCB) header structure. The flags are defined in the /usr/sys/include/io/cam/cam.h file.
pd_tag_action	This contains the default settings for the HBA/SIM queue actions field, cam_tag_action, in the SCSI I/O CCB structure. The queue actions are defined in the /usr/sys/include/io/cam/cam.h file.
pd_dev_inq	This is inquiry data.
pd_ms_index	The current index into the Mode Select Table that is pointed to in the Device Descriptor Structure.
pd_dev_desc	A pointer to the DEV_DESC structure for the SCSI device.

Structure Member	Description
pd_specific	A pointer to a device-specific structure filled in by the ccmn_open_unit routine.
pd_spec_size	The size of the device-specific information.
pd_sense_ptr	A pointer to the last sense data bytes retrieved from the device.
pd_sense_len	The length, in bytes, of the last sense data retrieved from the device.
pd_recov_hand	This is specific to SCSI/CAM peripheral device drivers. It can be used to point to the recovery handler for the device.
pd_read_count	Number of read operations from device. Used for performance statistics.
pd_write_count	Number of write operations to device. Used for performance statistics.
pd_read_bytes	Total number of bytes read from device. Used for performance statistics.
pd_write_bytes	Total number of bytes written to device. Used for performance statistics.
pd_bmajor	Block device major number for loadable drivers.
pd_cmajor	Character device major number for loadable drivers.
pd_lk_device	The lock structure.

11.1.3 Programmer-Defined Device Descriptor Structure

A Device Descriptor Structure entry, DEV_DESC, must be added to the cam_devdesc_tab for each programmer-defined SCSI device that exists in the system. The file /usr/sys/data/cam_data.c contains examples of entries supplied by Digital. The DEV_DESC structure is defined as follows:

```
typedef struct dev_desc {
        u_char  dd_pv_name[IDSTRING_SIZE];
                                /* Product ID and vendor string from */
                                /* Inquiry data */
        u_char  dd_length;      /* Length of dd_pv_name string */
        u_char  dd_dev_name[DEV_NAME_SIZE];
                                /* Device name string - see defines */
                                /* in devio.h */
        U32      dd_device_type; /* Bits 0 - 23 contain the device */
                                /* class, bits 24-31 contain the */
                                /* SCSI device type */
        struct  pt_info *dd_def_partition;
                                /* Default partition sizes - disks */
        U32      dd_block_size;  /* Block/sector size  */
        U32      dd_max_record;  /* Maximun transfer size in bytes */
```

```
                                /* allowed for the device */
          DENSITY_TBL *dd_density_tbl;
                                /* Pointer to density table - tapes */
          MODESEL_TBL *dd_modesel_tbl;
                                /* Mode select table pointer - used */
                                /* on open and recovery */
          U32       dd_flags;    /* Option flags (bbr, etc) */
          U32       dd_scsi_optcmds;/* Optional commands supported */
          U32       dd_ready_time;
                          /* Time in seconds for powerup dev ready */
          u_short dd_que_depth;  /* Device queue depth for devices */
                                /* which support command queueing */
          u_char  dd_valid;      /* Indicates which data length */
                                /* fields are valid */
          u_char  dd_inq_len;    /* Inquiry data length for device */
          u_char  dd_req_sense_len;
                                /* Request sense data length for */
                                /* this device */
}DEV_DESC;
```

The product ID and vendor returned string identifying the drive obtained from the Inquiry data. The product ID makes up the first eight characters of the string. The IDSTRING_SIZE constant is defined in the `/usr/sys/include/io/cam/pdrv.h` file. This specifies the length of the `dd_pv_name` string. The match is made on the total string returned by the unit.

11.1.3.1 The dd_dev_name Member

The DEC OSF/1 device name string, which is defined in the `/usr/sys/include/io/common/devio.h` file. A generic name of DEV_RZxx should be used for non-Digital disk devices. The following generic names are provided for tapes: DEV_TZQIC, for 1/4-inch cartridge tape units; DEV_TZ9TK for 9-track tape units; DEV_TZ8MM, for 8-millimeter tape units; DEV_TZRDAT, for RDAT tape units; DEV_TZ3480, for IBM 3480-compatible tape units; and DEV_TZxx, for tape units that do not fit into any of the predefined generic categories.

11.1.3.2 The dd_device_type Member

Bits 24-31 contain the SCSI device class, for example, ALL_DTYPE_DIRECT, which is defined in the `/usr/sys/include/io/cam/scsi_all.h` file. The bits 0-23 contain the device subclass, for example, SZ_HARD_DISK, which is defined in the `/usr/sys/include/io/cam/pdrv.h` file.

11.1.3.3 The dd_def_partition Member

A pointer to the default partition sizes for disks, which are defined in the `/usr/sys/data/cam_data.c` file. Tape devices should define this as `sz_null_sizes`. Disk devices may use `sz_rzxx_sizes`, which assumes that the disk has at least 48 Mbytes. The `sz_rzxx_sizes` should not be modified. If you want to create your own partition table, make an entry for your device in the device descriptor table in the `/usr/sys/data/cam_data.c` file.

11.1.3.4 The dd_block_size Member

The block or sector size of the unit, in bytes, for disks and CDROMs. You can obtain the correct number of bytes from the documentation for your device.

11.1.3.5 The dd_max_record Member

The maximum number of bytes that can be transferred in one request for raw I/O. Errors result if your system does not have enough physical memory or if the unit cannot handle the size of transfer specified.

11.1.3.6 The dd_density_tbl Member

A pointer to the Density Table Structure entry for a tape device.

11.1.3.7 The dd_modesel_tbl Member

A pointer to the Mode Select Table Structure entry for the devices. The Mode Select Table Structure is read and sent to the SCSI device when the first open call is issued and during recovery. This field is optional and should be used only for advanced SCSI device customization.

11.1.3.8 The dd_flags Member

The option flags, which can be SZ_NOSYNC, indicating that the device cannot handle synchronous transfers; SZ_BBR, indicating that the device allows bad block recovery; SZ_NO_DISC, indicating that the device cannot handle disconnects; and SZ_NO_TAG, indicating tagged queueing is not allowed. SZ_NO_TAG overrides inquiry data. The flags are defined in the `/usr/sys/include/io/cam/pdrv.h` file.

11.1.3.9 The dd_scsi_optcmds Member

The optional SCSI commands that are supported, as defined in the `/usr/sys/include/io/cam/pdrv.h` file. The possible commands are NO_OPT_CMDS; SZ_RW10, which enables reading and writing 10-byte CDBs; SZ_PREV_ALLOW, which prevents or allows media removal; and SZ_EXT_RESRV, which enables reserving or releasing file extents.

11.1.3.10 The dd_ready_time Member

The maximum time, in seconds, allowed for the device to power up. For disks, this represents power up and spin up time. For tapes, it represents power up, load, and rewind to Beginning of Tape.

11.1.3.11 The dd_que_depth Member

The maximum number of queued requests for devices that support queueing. Refer to the documentation for your device to determine if your device supports tag queuing and, if so, the depth of the queue.

11.1.3.12 The dd_valid Member

This indicates which data length fields are valid. The data length bits, DD_REQSNS_VAL and DD_INQ_VAL, are defined in the `/usr/sys/include/io/cam/pdrv.h` file.

11.1.3.13 The dd_inq_len Member

The inquiry data length for the device. This field must be used in conjunction with the DD_INQ_VAL flag.

11.1.3.14 The dd_req_sense_len Member

The request Sense data length for the device. This field must be used in conjunction with the DD_REQSNS_VAL flag.

11.1.4 Programmer-Defined Density Table Structure

The Density Table Structure allows for the definition of eight densities for each type of SCSI tape device unit. A density is defined using the lower three bits of the unit's minor number. Refer to the SCSI tape device unit documentation for the density code, compression code, and blocking factor for each density.

The `/usr/sys/data/cam_data.c` file contains Density Table Structure entries for all devices known to Digital. Programmers can add entries for other SCSI tape devices at the end of the Digital entries. The definition for the Density Table Structure, DENSITY_TBL, follows:

```
typedef struct density_tbl {
        struct density{
           u_char    den_flags;         /* VALID, ONE_FM etc */
           u_char    den_density_code;
           u_char    den_compress_code;
                                    /* Compression code if supported */
           u_char    den_speed_setting; /* for this density */
           u_char    den_buffered_setting;
                                          /* Buffer control setting */
```

```
            u_long    den_blocking;       /* 0 variable etc. */
        }density[MAX_TAPE_DENSITY];
}DENSITY_TBL;
```

11.1.4.1 The den_flags Member

The `den_flags` specified indicate which fields in the DENSITY_TBL structure are valid for this density. The flags are: DENS_VALID, to indicate whether the structure is valid; ONE_FM, to write one file mark on closing for QIC tape units; DENS_SPEED_VALID, to indicate the speed setting is valid for multispeed tapes; DENS_BUF_VALID, to run in buffered mode; and DENS_COMPRESS_VALID, to indicate compression code, if supported.

11.1.4.2 The den_density_code Member

The `den_density_code` member contains the SCSI density code for this density.

11.1.4.3 The den_compress_code Member

The `den_compress_code` member contains the SCSI compression code for this density, if the unit supports compression.

11.1.4.4 The den_speed_setting Member

The `den_speed_setting` member contains the speed setting for this density. Some units support variable speed for certain densities.

11.1.4.5 The den_buffered_setting Member

The `den_buffered_setting` member contains the buffer control setting for this density.

11.1.4.6 The den_blocking Member

The `den_blocking` member contains the blocking factor for this SCSI tape device. A NULL (0) setting specifies that the blocking factor is variable. A positive value represents the number of bytes in a block, for example, 512 or 1024.

11.1.4.7 Sample Density Table Structure Entry

This section contains a portion of a Density Table Structure entry for the TZK10 SCSI tape device, which supports both fixed and variable length records:

```
DENSITY_TBL
tzk10_dens = {
{ Minor 00

Flags
DENS_VALID | DENS_BUF_VALID |ONE_FM ,

Density code   Compression code        Speed setting
SEQ_8000R_BPI,         NULL,                   NULL,

Buffered setting       Blocking
1,                     512
},
   •
   •
   •
{ Minor 06

Flags
DENS_VALID | DENS_BUF_VALID |ONE_FM ,

Density code   Compression code        Speed setting
SEQ_QIC320,            NULL,                   NULL,

Buffered setting       Blocking
1,                     1024
},
{ Minor 07

Flags
DENS_VALID | DENS_BUF_VALID |ONE_FM ,

Density code   Compression code        Speed setting
SEQ_QIC320,            NULL,                   NULL,

Buffered setting       Blocking
1,                     NULL
}
}; end of tzk10_dens
```

11.1.5 Programmer-Defined Mode Select Table Structure

The Mode Select Table Structure is read and sent to the SCSI device when the first call to the SCSI/CAM peripheral open routine is issued on a SCSI device. There can be a maximum of eight entries in the Mode Select Table Structure. The definition for the Mode Select Table Structure, MODESEL_TBL, follows:

```
typedef struct modesel_tbl {
        struct ms_entry{
           u_char  ms_page;      /* Page number */
           u_char  *ms_data;     /* Pointer to Mode Select data */
           u_char  ms_data_len;  /* Mode Select data length */
           u_char  ms_ent_sp_pf;/* Save Page and Page format bits */
                                 /* BIT 0    1=Save Page, */
                                 /*          0=Don't Save Page */
```

```
                                      /* BIT 1  1=SCSI-2, 0=SCSI-1 */
      }ms_entry[MAX_OPEN_SELS];
}MODESEL_TBL;
```

11.1.5.1 The ms_page Member

The `ms_page` member contains the SCSI page number for the device type. For example, the page number would be 0x10 for the device configuration page for a SCSI tape device.

11.1.5.2 The ms_data Member

The `ms_data` member contains a pointer to the mode select data for the device. Set up the page data and place the address of the page structure in this field. A sample page definition for page 0x10 for the TZK10 follows:

```
SEQ_MODE_DATA6
tzk10_page10 = {

{ Parameter header

mode_len        medium type     speed
NULL,           NULL,           NULL,

Buf_mode        wp              blk_desc_len
0x01,           NULL,           sizeof(SEQ_MODE_DESC)
},
{ Mode descriptor

Density num_blks2       num_blks1
NULL,           NULL,           NULL,

num_blks0       reserved        blk_len2
NULL,                           NULL,

blk_len1        blk_len0
NULL,           NULL
},
{
Page data for page 0x2

PAGE header
byte0   byte1
0x10,       0x0e,

byte2    byte3    byte4    byte5    byte6
0x00,    0x00,    40,      40,      NULL,

byte7    byte8    byte9    byte10   byte11
NULL,    0xe0,    NULL,    0x38,    NULL,

byte12   byte13   byte14   byte15
NULL,    NULL,    NULL,    NULL
}
};
```

11.1.5.3 The ms_data_len Member

The `ms_data_len` member contains length of a page, which is the number of bytes to be sent to the device.

11.1.5.4 The ms_ent_sp_pf Member

The `ms_ent_sp_pf` member contains flags for the MODE SELECT CDB that the device driver formats.

11.1.5.5 Sample Mode Select Table Structure Entry

This section contains a sample portion of a Mode Select Table Structure entry for the TZK10 SCSI tape device:

```
MODESEL_TBL
tzk10_mod = {
{ MODE PAGE ENTRY 1

Page number              The data pointer
0x02,                    (u_char *)&tzk10_page2,

Data len                 SCSI2??
28,                   0x2
},
    .
    .
    .
{ MODE PAGE ENTRY 8

Page number              The data pointer
NULL,                    (u_char *)NULL,

Data len                 SCSI2??
NULL,                    NULL
},
};
```

11.2 Sample SCSI/CAM Device-Specific Data Structures

This section provides samples of the SCSI/CAM peripheral data structures programmers must define if they write their own device drivers. The following data structures are described:

- TAPE_SPECIFIC – The Tape-Specific Structure
- DISK_SPECIFIC – The Disk- and CDROM-Specific Structure

11.2.1 Programmer-Defined Tape-Specific Structure

SCSI/CAM peripheral device driver writers can create their own tape-specific data structures. Here is a sample TAPE_SPECIFIC structure for a SCSI tape device, as defined in the `/usr/sys/include/io/cam/cam_tape.h` file:

```
typedef struct {
        u_long  ts_flags;       /* Tape flags - BOM,EOT */
        u_long  ts_state_flags; /* STATE - UNIT_ATTEN, RESET etc. */
        u_long  ts_resid;       /* Last operation residual count */
        u_long  ts_block_size;  /* See below for a complete desc. */
        u_long  ts_density;     /* What density are we running at */
        u_long  ts_records;
                              /* How many records in since last tpmark */
        u_long  ts_num_filemarks; /* number of file marks into tape */
}TAPE_SPECIFIC;
```

11.2.1.1 The ts_flags Member

Flags used to indicate tape condition. The possible flags are:

Flag Name	Description
CTAPE_BOM	The tape is positioned at the beginning.
CTAPE_EOM	The unit is positioned at the end of media.
CTAPE_OFFLINE	The device is returning DEVICE NOT READY in response to a command. The media is either not loaded or is being loaded.
CTAPE_WRT_PROT	The unit is either write protected or is opened read only.
CTAPE_BLANK	The tape is blank.
CTAPE_WRITTEN	The tape has been written during this procedure.
CTAPE_CSE	Clear serious exception.
CTAPE_SOFTERR	A soft error has been reported by the SCSI unit.
CTAPE_HARDERR	A hard error has been reported by the SCSI unit. It can be reported either through an `ioctl` or by marking the `buf` structure as EIO.
CTAPE_DONE	The tape procedure is finished.
CTAPE_RETRY	Indicates a retry can be attempted.
CTAPE_ERASED	The tape has been erased.
CTAPE_TPMARK	A tape mark has been detected during a read operation. This cannot occur during a write operation.
CTAPE_SHRTREC	The size of the record retrieved is less than the size requested. Reported using an ioctl.
CTAPE_RDOPP	Reading in the reverse direction. This is not implemented.
CTAPE_REWINDING	The tape is rewinding.

Flag Name	Description
CTAPE_TPMARK_PENDING	The tape mark is to be reported on the next I/O operation.

11.2.1.2 The ts_state_flags Member

Flags used to indicate tape state. The possible flags include:

Flag Name	Description
CTAPE_NOT_READY_STATE	The unit was opened with the FNDELAY flag. The unit was detected, but the open failed.
CTAPE_UNIT_ATTEN_STATE	A check condition occurred and the sense key was UNIT ATTENTION. This usually indicates that the media was changed. Current tape position is lost.
CTAPE_RESET_STATE	Indicates a reset condition on the device or on the bus.
CTAPE_RESET_PENDING_STATE	A reset is pending.
CTAPE_OPENED_STATE	The unit is opened.
CTAPE_DISEOT_STATE	No notification of end of media is required.
CTAPE_ABORT_TPPEND_STATE	Indicates that a tape mark was detected for a fixed block operation with nonbuffered I/O. The queue is aborted.
CTAPE_AUTO_DENSITY_VALID_STATE	Directs the open routine to call the ctz_auto_density routine when a unit attention is noticed, because tape density has been determined and all reads are to occur at that density.
CTAPE_ORPHAN_CMD_STATE	This flag is set when a command is orphaned. The process does not wait for completion, such as a rewind operation.
CTAPE_POSITION_LOST_STATE	Tape position is lost due to command failure.

11.2.1.3 The ts_resid Member

Residual count from the last tape command.

11.2.1.4 The ts_block_size Member

Used to distinguish between blocks and bytes for fixed-block tapes. Commands for devices like 9-track tape, which have variable length records, assume bytes.

11.2.1.5 The ts_density Member

The current density at which the SCSI tape device is operating.

11.2.1.6 The ts_records Member

The number of records read since the last tape mark.

11.2.1.7 The ts_num_filemarks Member

The number of file marks encountered since starting to read the tape.

11.2.2 Programmer-Defined Disk- and CDROM-Specific Structure

SCSI/CAM peripheral device driver writers can create their own disk- and CDROM-specific data structures. A sample DISK_SPECIFIC structure for a SCSI disk device, as defined in the `/usr/sys/include/io/cam/cam_disk.h` file, follows:

```
typedef struct disk_specific  {
    struct buf *ds_bufhd;           /* Anchor for requests which come */
                                    /* into strategy that cannot be */
                                    /* started due to error recovery */
                                    /* in progress. */
    int         ds_dkn;             /* Used for system statistics */
    U32         ds_bbr_state;       /* Used indicate the current */
                                    /* BBR state if active */
    U32         ds_bbr_retry;       /* BBR retries for reassignment */
    u_char      *ds_bbr_buf;        /* Points to read/write and */
                                    /* reassign data buffer */
    CCB_SCSIIO *ds_bbr_rwccb;       /* R/W ccb used for BBR */
    CCB_SCSIIO *ds_bbr_reasccb;     /* Reassign ccb used for BBR */
    CCB_SCSIIO *ds_bbr_origccb;     /* Ccb which encountered bad block */
    CCB_SCSIIO *ds_tur_ccb;         /* SCSI I/O CCB for tur cmd */
                                    /* during recovery */
    CCB_SCSIIO *ds_start_ccb;       /* SCSI I/O CCB for start unit */
                                    /* cmd during recovery */
    CCB_SCSIIO *ds_mdsel_ccb;       /* SCSI I/O CCB for mode select */
                                    /* cmd during recovery */
    CCB_SCSIIO *ds_rdcp_ccb;        /* SCSI I/O CCB for read capacity */
                                    /* cmd during recovery */
    CCB_SCSIIO *ds_read_ccb;        /* SCSI I/O CCB for Read cmd */
                                    /* during recovery */
    CCB_SCSIIO *ds_prev_ccb;        /* SCSI I/O CCB for Prevent */
```

```
                                        /* Media Removal cmd during recovery */
    U32         ds_block_size;          /* This units block size */
    U32         ds_tot_size;            /* Total disk size in blocks */
    U32         ds_media_changes;       /* Number of times media was */
                                        /* changed - removables */
    struct pt   ds_pt;                  /* Partition structure */
    U32         ds_openpart;            /* Bit mask of open parts */
    U32         ds_bopenpart;           /* No of block opens */
    U32         ds_copenpart;           /* No of char opens */
    U32         ds_wlabel;              /* Write enable label */
    struct disklabel ds_label;          /* Disk label on device */
}DISK_SPECIFIC;
```

The structure members and their descriptions follow:

Structure Member	Description
`ds_bufhd`	Pointer to a buffer header structure to contain requests that come to the driver but cannot be started due to error recovery in progress. The requests are issued when error recovery is complete.
`ds_dkn`	Used for system statistics.
`ds_bbr_state`	Used to indicate the current state if bad block recovery (BBR) is active.
`ds_bbr_retry`	Number of retries to attempt for reassignment of bad blocks.
`ds_bbr_buf`	Pointer to the read/write and the reassign data buffers.
`ds_bbr_rwccb`	Pointer for the SCSI I/O CCB for the Read/Write command used for recovery.
`ds_bbr_reasccb`	Pointer for the SCSI I/O CCB for the Reassign command used for recovery.
`ds_bbr_origccb`	A CCB that encountered a bad block.
`ds_tur_ccb`	Pointer for the SCSI I/O CCB for the TEST UNIT READY command used for recovery.
`ds_start_ccb`	Pointer for the SCSI I/O CCB for the START UNIT command used for recovery.
`ds_mdsel_ccb`	Pointer for the SCSI I/O CCB for the MODE SELECT command used for recovery.
`ds_rdcp_ccb`	Pointer for the SCSI I/O CCB for the Read Capacity command used for recovery.
`ds_read_ccb`	Pointer for the SCSI I/O CCB for the Read command used for recovery.
`ds_prev_ccb`	Pointer for the SCSI I/O CCB for the Prevent Removal command during recovery.

Structure Member	Description
ds_block_size	This SCSI disk device's block size in bytes.
ds_tot_size	Total SCSI disk device size in blocks.
ds_media_changes	For removable media, the number of times the media was changed.
ds_pt	Structure defining the current disk partition layout.
ds_openpart	Bit mask of open partitions.
ds_bopenpart	Number of block opens.
ds_copenpart	Number of character opens.
ds_wlabel	The write-enable label.
ds_label	Disk label on device.

11.2.3 SCSI/CAM CDROM/AUDIO I/O Control Commands

This section describes the standard and vendor-unique I/O control commands to use for SCSI CDROM/AUDIO devices. The commands are defined in the `/usr/sys/include/io/cam/cdrom.h` file. See Chapter 13 of American National Standard for Information Systems, *Small Computer Systems Interface - 2* (SCSI - 2), X3.131-199X for general information about the CDROM device model. Table 11-1 lists the name of each command and describes its function.

Table 11-1: SCSI/CAM CDROM/AUDIO I/O Control Commands

Command	Description
Standard Commands	
CDROM_PAUSE_PLAY	Pauses audio operation
CDROM_RESUME_PLAY	Resumes audio operation
CDROM_PLAY_AUDIO	Plays audio in Logical Block Address (LBA) format
CDROM_PLAY_AUDIO_MSF	Plays audio in Minute-/Second-/Frame-units (MSF) format
CDROM_PLAY_AUDIO_TI	Plays audio track or index
CDROM_PLAY_AUDIO_TR	Plays audio track relative
CDROM_TOC_HEADER	Reads Table of Contents (TOC) header
CDROM_TOC_ENTRYS	Reads Table of Contents (TOC) entries
CDROM_EJECT_CADDY	Ejects the CDROM caddy
CDROM_READ_SUBCHANNEL	Reads subchannel data
CDROM_READ_HEADER	Reads track header
Vendor-Unique Commands	
CDROM_PLAY_VAUDIO	Plays audio LBA format
CDROM_PLAY_MSF	Plays audio MSF format
CDROM_PLAY_TRACK	Plays audio track
CDROM_PLAYBACK_CONTROL	Controls playback
CDROM_PLAYBACK_STATUS	Checks playback status
CDROM_SET_ADDRESS_FORMAT	Sets address format

11.2.3.1 Structures Used by SCSI/CAM CDROM/AUDIO I/O Control Commands

Some of the SCSI CDROM/AUDIO device I/O control commands use data structures. This section describes those data structures. The structures are defined in the `/usr/sys/include/io/cam/cam_disk.h` file. Table 11-2 lists the name of each structure and the commands that use it.

Table 11-2: Structures Used by SCSI/CAM CDROM/AUDIO I/O Control Commands

Structure	Command
cd_address	All
cd_play_audio	CDROM_PLAY_AUDIO CDROM_PLAY_VAUDIO
cd_play_audio_msf	CDROM_PLAY_AUDIO_MSF CDROM_PLAY_MSF

Table 11-2: (continued)

Structure	Command
cd_play_audio_ti	CDROM_PLAY_AUDIO_TI
cd_play_track	CDROM_PLAY_AUDIO_TR CDROM_PLAY_TRACK
cd_toc_header	CDROM_TOC_HEADER
cd_toc	CDROM_TOC_ENTRYS
cd_toc_entry	CDROM_TOC_ENTRYS
cd_sub_channel	CDROM_READ_SUBCHANNEL
cd_subc_position	CDROM_READ_SUBCHANNEL
cd_subc_media_catalog	CDROM_READ_SUBCHANNEL
cd_subc_isrc_data	CDROM_READ_SUBCHANNEL
cd_subc_header	CDROM_READ_SUBCHANNEL
cd_subc_channel_data	CDROM_READ_SUBCHANNEL
cd_subc_information	CDROM_READ_SUBCHANNEL
cd_read_header	CDROM_READ_HEADER
cd_read_header_data	CDROM_READ_HEADER
cd_playback	CDROM_PLAYBACK_CONTROL CDROM_PLAYBACK_STATUS

11.2.3.1.1 Structure Used by All SCSI/CAM CDROM/AUDIO I/O Control Commands – This section describes the `cd_address` union that defines the SCSI CDROM/AUDIO device Track Address structure and that all the SCSI CDROM/AUDIO device I/O control commands use. The SCSI CDROM/AUDIO device returns track addresses in either LBA or MSF format.

```
union cd_address {
      struct {
            u_char          : 8;
            u_char      m_units;
            u_char      s_units;
            u_char      f_units;
      } msf;                  /* Minutes/Seconds/Frame format    */
          struct {
            u_char      addr3;
            u_char      addr2;
            u_char      addr1;
            u_char      addr0;
      } lba;                  /* Logical Block Address format    */
};
```

```
/*
 * CD-ROM Address Format Definitions.
 */
#define CDROM_LBA_FORMAT        0
                       /* Logical Block Address format */
#define CDROM_MSF_FORMAT        1
                       /* Minute Second Frame format */
```

The structure members and their descriptions follow:

Structure Member	Description
m_units	The minute-units binary number of the MSF format for CDROM media
s_units	The second-units binary number of the MSF format for CDROM media
f_units	The frame-units binary number of the MSF format for CDROM media
addr3	The fourth logical block address of LBA format for disk media
addr2	The third logical block address of LBA format for disk media
addr1	The second logical block address of LBA format for disk media
addr0	The first logical block address of LBA format for disk media

11.2.3.1.2 Structure Used by the CDROM_PLAY_AUDIO and CDROM_PLAY_VAUDIO Commands – This section describes the structure that is used by the CDROM_PLAY_AUDIO and CDROM_PLAY_VAUDIO commands. The structure is defined as follows:

```
struct cd_play_audio {
        u_long  pa_lba;      /* Logical block address. */
        u_long  pa_length;   /* Transfer length in blocks. */
};
```

The structure members and their descriptions follow:

Structure Member	Description
pa_lba	The LBA where the audio playback operation is to begin.
pa_length	The number of contiguous logical blocks to be played.

11.2.3.1.3 Structure Used by the CDROM_PLAY_AUDIO_MSF and CDROM_PLAY_MSF Commands – This section describes the structure that is used by the CDROM_PLAY_AUDIO_MSF and CDROM_PLAY_MSF commands. The structure is defined as follows:

```
struct cd_play_audio_msf {
        u_char msf_starting_M_unit;     /* Starting M-unit */
        u_char msf_starting_S_unit;     /* Starting S-unit */
        u_char msf_starting_F_unit;     /* Starting F-unit */
        u_char msf_ending_M_unit;       /* Ending M-unit */
        u_char msf_ending_S_unit;       /* Ending S-unit */
        u_char msf_ending_F_unit;       /* Ending F-unit */
};
```

The structure members and their descriptions follow:

Structure Member	Description
msf_starting_M_unit	The minute-unit field of the absolute MSF address at which the audio play operation is to begin.
msf_starting_S_unit	The second-unit field of the absolute MSF address at which the audio play operation is to begin.
msf_starting_F_unit	The frame-unit field of the absolute MSF address at which the audio play operation is to begin.
msf_ending_M_unit	The minute-unit field of the absolute MSF address at which the audio play operation is to end.
msf_ending_S_unit	The second-unit field of the absolute MSF address at which the audio play operation is to end.
msf_ending_F_unit	The frame-unit field of the absolute MSF address at which the audio play operation is to end.

11.2.3.1.4 Structure Used by the CDROM_PLAY_AUDIO_TI Command – This section describes the structure that is used by the CDROM_PLAY_AUDIO_TI command. The structure is defined as follows:

```
/*
 * Define Minimum and Maximum Values for Track & Index.
 */
#define CDROM_MIN_TRACK         1       /* Minimum track number */
#define CDROM_MAX_TRACK         99      /* Maximum track number */
#define CDROM_MIN_INDEX         1       /* Minimum index value */
#define CDROM_MAX_INDEX         99      /* Maximum index value */

struct cd_play_audio_ti {
        u_char  ti_starting_track;      /* Starting track number */
        u_char  ti_starting_index;      /* Starting index value */
        u_char  ti_ending_track;        /* Ending track number */
        u_char  ti_ending_index;        /* Ending index value */
};
```

The structure members and their descriptions follow:

Structure Member	Description
ti_starting_track	The track number at which the audio play operation starts.
ti_starting_index	The index number within the track at which the audio play operation starts.
ti_ending_track	The track number at which the audio play operation ends.
ti_ending_index	The index number within the track at which the audio play operation ends.

11.2.3.1.5 Structure Used by the CDROM_PLAY_AUDIO_TR Command

– This section describes the structure that is used by the CDROM_PLAY_AUDIO_TR command. The structure is defined as follows:

```
struct cd_play_audio_tr {
        u_long  tr_lba;                  /* Track relative LBA */
        u_char  tr_starting_track;       /* Starting track number */
        u_short tr_xfer_length;          /* Transfer length */
};
```

The structure members and their descriptions follow:

Structure Member	Description
tr_lba	The logical block address relative to the track being played. A negative value indicates a start location within the audio pause area at the beginning of the track.
tr_starting_track	Track number at which play is to start.
tr_xfer_length	The number of contiguous logical blocks to be output as audio data.

11.2.3.1.6 Structure Used by the CDROM_TOC_HEADER Command –

This section describes the structure that is used by the

CDROM_TOC_HEADER command. The structure is defined as follows:

```
struct cd_toc_header {
        u_char  th_data_len1;            /* TOC data length MSB */
        u_char  th_data_len0;            /* TOC data length LSB */
        u_char  th_starting_track;       /* Starting track number */
        u_char  th_ending_track;         /* Ending track number */
};
```

The structure members and their descriptions follow:

Structure Member	Description
`th_data_len1`	The total number of bytes in the table of contents for MSF format.
`th_data_len0`	The total number of bytes in the table of contents for LBA format.
`th_starting_track`	Starting track number for which data is to be returned. If the value is 0 (zero), data is to be returned starting with the first track on the medium.
`th_ending_track`	The track number at which the audio play operation ends.

11.2.3.1.7 Structures Used by the CDROM_TOC_ENTRYS Command –

This section describes the structures that are used by the CDROM_TOC_ENTRYS command. The structures are defined as follows:

```
struct cd_toc {
        u_char  toc_address_format;      /* Address format to return */
        u_char  toc_starting_track;      /* Starting track number */
        u_short toc_alloc_length;        /* Allocation length */
        caddr_t toc_buffer;              /* Pointer to TOC buffer */
};
```

The structure members and their descriptions follow:

Structure Member	Description
`toc_address_format`	The address format, LBA or MSF.
`toc_starting_track`	The track number at which the audio play operation starts.
`toc_alloc_length`	The allocation length of the table of contents buffer in bytes
`toc_buffer`	A pointer to the TOC buffer.

```
struct cd_toc_entry {
        u_char                      : 8;    /* Reserved */
        u_char  te_control          : 4;    /* Control field (attributes) */
        u_char  te_addr_type        : 4;    /* Address type information */
        u_char  te_track_number;            /* The track number */
        u_char                      : 8;    /* Reserved */
        union cd_address te_absaddr;        /* Absolute CD-ROM Address */
};
```

The structure members and their descriptions follow:

Structure Member	Description
`te_control`	The control field containing attributes. The possible settings follow:

Bit No.	Set to 0 (Zero)	Set to 1
0	Audio without preemphasis	Audio with preemphasis
1	Digital copy prohibited	Digital copy permitted
2	Audio track	Data track
3	Two-channel audio	Four-channel audio

Structure Member	Description
`te_addr_type`	Address-type information, MSF or LBA
`te_track_number`	The current track number that is being played.
`te_absaddr`	The absolute address of the audio track, MSF or LBA format.

11.2.3.1.8 Structures Used by the CDROM_READ_SUBCHANNEL Command

Command – The CDROM_READ_SUBCHANNEL command requests subchannel data and the state of audio play operations from the target device. This section describes the structure that is used by the CDROM_READ_SUBCHANNEL command. The structure is defined as follows:

```
/*
 * CD-ROM Sub-Channel Q Address Field Definitions.
 */
#define CDROM_NO_INFO_SUPPLIED  0x0    /* Information not supplied */
#define CDROM_CURRENT_POS_DATA  0x1    /* Encodes current position data */
#define CDROM_MEDIA_CATALOG_NUM 0x2    /* Encodes media catalog number */
#define CDROM_ENCODES_ISRC      0x3    /* Encodes ISRC */
                                       /* ISRC=International-Standard- */
                                       /*               Recording-Code */
/* Codes 0x4 through 0x7 are Reserved */
```

```
/*
 * CD-ROM Data Track Definitions
 */
#define CDROM_AUDIO_PREMPH      0x01
                                    /* 0/1 = Without/With Pre-emphasis */
#define CDROM_COPY_PERMITTED    0x02  /* 0/1 = Copy Prohibited/Allowed */
#define CDROM_DATA_TRACK        0x04  /* 0 = Audio, 1 = Data track */
#define CDROM_FOUR_CHAN_AUDIO   0x10  /* 0 = 2 Channel, 1 = 4 Channel */

/*
 * Sub-Channel Data Format Codes
 */
#define CDROM_SUBQ_DATA         0x00  /* Sub-Channel data information */
#define CDROM_CURRENT_POSITION  0x01  /* Current position information */
#define CDROM_MEDIA_CATALOG     0x02  /* Media catalog number */
#define CDROM_ISRC              0x03  /* ISRC information */
                                      /* ISRC=International-Standard- */
                                      /*              Recording-Code */
/* Codes 0x4 through 0xEF are Reserved */
/* Codes 0xF0 through 0xFF are Vendor Specific */

/*
 * Audio Status Definitions returned by Read Sub-Channel Data Command
 */
#define AS_AUDIO_INVALID        0x00  /* Audio status not supported */
#define AS_PLAY_IN_PROGRESS     0x11  /* Audio play operation in prog */
#define AS_PLAY_PAUSED          0x12  /* Audio play operation paused */
#define AS_PLAY_COMPLETED       0x13  /* Audio play completed */
#define AS_PLAY_COMPLETED       0x13  /* Audio play completed */
#define AS_PLAY_ERROR           0x14  /* Audio play stopped by error */
#define AS_NO_STATUS            0x15  /* No current audio status */

struct cd_sub_channel {
        u_char  sch_address_format;   /* Address format to return */
        u_char  sch_data_format;      /* Sub-channel data format code */
        u_char  sch_track_number;     /* Track number */
        u_short sch_alloc_length;     /* Allocation length */
        caddr_t sch_buffer;           /* Pointer to SUBCHAN buffer */
};
```

The structure members and their descriptions follow:

Structure Member	Description
sch_address_format	The address format, LBA or MSF.
sch_data_format	The type of subchannel data to be returned.
sch_track_number	The track from which ISRC data is read.
sch_alloc_length	The allocation length of the table of contents buffer in bytes
sch_buffer	A pointer to the SUBCHAN buffer defined by the sch_data_format member.

```
struct cd_subc_position {
        u_char  scp_data_format;          /* Data Format code */
        u_char  scp_control     : 4;      /* Control field (attributes) */
        u_char  scp_addr_type   : 4;      /* Address type information */
        u_char  scp_track_number;         /* The track number */
        u_char  scp_index_number;         /* The index number */
        union cd_address scp_absaddr;     /* Absolute CD-ROM Address */
        union cd_address scp_reladdr;     /* Relative CD-ROM Address */
};

#define scp_absmsf scp_absaddr.msf
#define scp_abslba scp_absaddr.lba
#define scp_relmsf scp_reladdr.msf
#define scp_rellba scp_reladdr.lba
```

The structure members and their descriptions follow:

Structure Member	Description
`scp_data_format`	Data format code.
`scp_control`	The control field containing attributes. The possible settings follow:

Bit No.	Set to 0 (Zero)	Set to 1
0	Audio without preemphasis	Audio with preemphasis
1	Digital copy prohibited	Digital copy permitted
2	Audio track	Data track
3	Two-channel audio	Four-channel audio

Structure Member	Description
`scp_addr_type`	Address-type information, MSF or LBA format. The address format, LBA or MSF.
`scp_track_number`	The current track number that is being played.
`scp_index_number`	The index number within an audio track.
`scp_absaddr`	The absolute address of the audio track, MSF or LBA format.
`scp_reladdr`	The CDROM address relative to the track being played.

```
struct cd_subc_media_catalog {
        u_char  smc_data_format;        /* Data Format code */
        u_char                  : 8;    /* Reserved */
        u_char                  : 8;    /* Reserved */
        u_char                  : 8;    /* Reserved */
        u_char                  : 7,    /* Reserved */
                smc_mc_valid    : 1;    /* Media catalog valid 1 = True */
        u_char  smc_mc_number[15];      /* Media catalog number ASCII */
};
```

The structure members and their descriptions follow:

Structure Member	Description
smc_data_format	Data format code.
smc_mc_valid	Media catalog number is valid.
smc_mc_number	Media catalog number.

```
struct cd_subc_isrc_data {
        u_char  sid_data_format;        /* Data Format code */
        u_char                  : 8;    /* Reserved */
        u_char  sid_track_number;       /* The track number */
        u_char                  : 8;    /* Reserved */
        u_char                  : 7,    /* Reserved */
                sid_tc_valid    : 1;    /* Track code valid, 1 = True */
        u_char  sid_tc_number[15];      /* International-Standard- */
                                        /*   Recording-Code (ASCII) */
};
```

The structure members and their descriptions follow:

Structure Member	Description
sid_data_format	Data format code.
sid_track_number	The current track number at which ISRC is located.
sid_tc_valid	The track code is valid.
sid_tc_number[15]	The track code number.

```
struct cd_subc_header {
        u_char                  : 8;    /* Reserved */
        u_char  sh_audio_status;        /* Audio status */
        u_char  sh_data_len1;           /* Sub-Channel Data length MSB */
        u_char  sh_data_len0;           /* Sub-Channel Data length LSB */
};
```

The structure members and their descriptions follow:

Structure Member	Description
`sh_audio_status`	The audio status code.
`sh_data_len1`	The subchannel data length for MSF format.
`sh_data_len0`	The subchannel data length for LBA format.

```
struct cd_subc_channel_data {
        struct cd_subc_header scd_header;
        struct cd_subc_position scd_position_data;
        struct cd_subc_media_catalog scd_media_catalog;
        struct cd_subc_isrc_data scd_isrc_data;
};
```

The structure members and their descriptions follow:

Structure Member	Description
`scd_header`	The subchannel data header, which is four bytes.
`scd_position_data`	CDROM current-position data information.
`scd_media_catalog`	The Media Catalog Number data information.
`scd_isrc_data`	Track International-Standard-Recording-Code (ISRC) data information.

```
struct cd_subc_information {
        struct cd_subc_header sci_header;
        union {
                struct cd_subc_channel_data sci_channel_data;
                struct cd_subc_position sci_position_data;
                struct cd_subc_media_catalog sci_media_catalog;
                struct cd_subc_isrc_data sci_isrc_data;
        } sci_data;
};

#define sci_scd         sci_data.sci_channel_data
#define sci_scp         sci_data.sci_position_data
#define sci_smc         sci_data.sci_media_catalog
#define sci_sid         sci_data.sci_isrc_data

#define CDROM_DATA_MODE_ZERO    0       /* All bytes zero */
#define CDROM_DATA_MODE_ONE     1       /* Data mode one format */
#define CDROM_DATA_MODE_TWO     2       /* Data mode two format */
/* Modes 0x03-0xFF are reserved. */
```

This structure is used to allocate data space. The structure members and their descriptions follow:

Structure Member	Description
sci_channel_data	Space for channel data.
sci_position_data	Space for current position data.
sci_media_catalog	Space for Media Catalog data.
sci_isrc_data	Space for ISRC data.

11.2.3.1.9 Structures Used by the CDROM_READ_HEADER Command –

This section describes the structures that are used by the CDROM_READ_HEADER command. The structures are defined as follows:

```
struct cd_read_header {
        u_char  rh_address_format;      /* Address format to return */
        u_long  rh_lba;                 /* Logical block address */
        u_short rh_alloc_length;        /* Allocation length */
        caddr_t rh_buffer;              /* Pointer to header buffer */
};
```

The structure members and their descriptions follow:

Structure Member	Description
rh_address_format	The address format, LBA or MSF.
rh_lba	The logical block address for LBA format.
rh_alloc_length	The allocation length of the header buffer.
rh_buffer	A pointer to the header buffer.

```
struct cd_read_header_data {
        u_char  rhd_data_mode;          /* CD-ROM data mode */
        u_char                  : 8;    /* Reserved */
        u_char                  : 8;    /* Reserved */
        u_char                  : 8;    /* Reserved */
        union cd_address rhd_absaddr;   /* Absolute CD-ROM address */
};

#define rhd_msf rhd_absaddr.msf
#define rhd_lba rhd_absaddr.lba
```

The structure members and their descriptions follow:

Structure Member	Description
rhd_data_mode	The CDROM data mode type.
rhd_absaddr	The absolute address of the audio track, MSF or LBA format.

11.2.3.1.10 Structure Used by the CDROM_PLAY_TRACK Command –

This section describes the structure that is used by the CDROM_PLAY_TRACK command. The structure is defined as follows:

```
struct cd_play_track {
        u_char  pt_starting_track;      /* Starting track number */
        u_char  pt_starting_index;      /* Starting index value */
        u_char  pt_number_indexes;      /* Number of indexes */
};
```

The structure members and their descriptions follow:

Structure Member	Description
pt_starting_track	The track number at which the audio play operation starts.
pt_starting_index	The index number within the track at which the audio play operation starts.
pt_number_indexes	The number of index values in the audio encoding on CDROM media.

11.2.3.1.11 Structure Used by the CDROM_PLAYBACK_CONTROL and CDROM_PLAYBACK_STATUS Commands –

This section describes the structures that are used by the CDROM_PLAYBACK_CONTROL and CDROM_PLAYBACK_STATUS commands. The structures are defined as follows:

```
/*
 * Definitions for Playback Control/Playback Status Output Selection
Codes */
#define CDROM_MIN_VOLUME        0x0     /* Minimum volume level */
#define CDROM_MAX_VOLUME        0xFF    /* Maximum volume level */
#define CDROM_PORT_MUTED        0x0     /* Output port is muted */
#define CDROM_CHANNEL_0         0x1     /* Channel 0 to output port */
#define CDROM_CHANNEL_1         0x2     /* Channel 1 to output port */
#define CDROM_CHANNEL_0_1       0x3     /* Channel 0 & 1 to output port */

struct cd_playback {
```

```
        u_short pb_alloc_length;        /* Allocation length */
        caddr_t pb_buffer;              /* Pointer to playback buffer */
};
```

The structure members and their descriptions follow:

Structure Member	Description
pb_alloc_length	Allocation length of the playback buffer.
pb_buffer	A pointer to the playback buffer.

11.2.3.1.12 Structure Used by the CDROM_PLAYBACK_CONTROL Command – This section describes the structure that is used by the CDROM_PLAYBACK_CONTROL command. The structure is defined as follows:

```
struct cd_playback_control {
        u_char  pc_reserved[10];         /* Reserved */
        u_char  pc_chan0_select : 4,     /* Channel 0 selection code */
                                : 4;     /* Reserved */
        u_char  pc_chan0_volume;         /* Channel 0 volume level */
        u_char  pc_chan1_select : 4,     /* Channel 1 selection code */
                                : 4;     /* Reserved */
        u_char  pc_chan1_volume;         /* Channel 1 volume level */
        u_char  pc_chan2_select : 4,     /* Channel 2 selection code */
                                : 4;     /* Reserved */
        u_char  pc_chan2_volume;         /* Channel 2 volume level */
        u_char  pc_chan3_select : 4,     /* Channel 3 selection code */
                                : 4;     /* Reserved */
        u_char  pc_chan3_volume;         /* Channel 3 volume level */
};
```

The structure members and their descriptions follow:

Structure Member	Description
pc_chan0_select	The selection code for Channel 0. The low four bits are reserved.
pc_chan0_volume	The volume level value for Channel 0.
pc_chan1_select	The selection code for Channel 1. The low four bits are reserved.
pc_chan1_volume	The volume level value for Channel 1.
pc_chan2_select	The selection code for Channel 2. The low four bits are reserved.
pc_chan2_volume	The volume level value for Channel 2.
pc_chan3_select	The selection code for Channel 3. The low four bits are reserved.
pc_chan3_volume	The volume level value for Channel 3.

11.2.3.1.13 Structure Used by the CDROM_PLAYBACK_STATUS Command

Command – This section describes the structure that is used by the CDROM_PLAYBACK_STATUS command. The structure is defined as follows:

```
/*
 * Audio status return by Playback Status Command.
 */
#define PS_PLAY_IN_PROGRESS     0x00  /* Audio Play Oper In Progress */
#define PS_PLAY_PAUSED          0x01  /* Audio Pause Oper In Progress */
#define PS_MUTING_ON            0x02  /* Audio Muting On */
#define PS_PLAY_COMPLETED       0x03  /* Audio Play Oper Completed */
#define PS_PLAY_ERROR           0x04  /* Error Occurred During Play */
#define PS_PLAY_NOT_REQUESTED   0x05  /* Audio Play Oper Not Requested */

/*
 * Data structure returned by Playback Status Command.
 */
struct cd_playback_status {
        u_char                  : 8;  /* Reserved */
        u_char  ps_lbamsf       : 1,  /* Address format 0/1 = LBA/MSF */
                                : 7;  /* Reserved */
        u_char  ps_data_len1;         /* Audio data length MSB */
        u_char  ps_data_len0;         /* Audio data length LSB */
        u_char  ps_audio_status;      /* Audio status */
        u_char  ps_control      : 4,  /* Control field (attributes) */
                                : 4;  /* Reserved */
        union cd_address ps_absaddr;  /* Absolute CD-ROM address */
        u_char  ps_chan0_select : 4,  /* Channel 0 selection code */
                                : 4;  /* Reserved */
        u_char  ps_chan0_volume;      /* Channel 0 volume level */
        u_char  ps_chan1_select : 4,  /* Channel 1 selection code */
```

```
                                : 4;   /* Reserved */
        u_char   ps_chan1_volume;      /* Channel 1 volume level */
        u_char   ps_chan2_select : 4,  /* Channel 2 selection code */
                                : 4;   /* Reserved */
        u_char   ps_chan2_volume;      /* Channel 2 volume level */
        u_char   ps_chan3_select : 4,  /* Channel 3 selection code */
                                : 4;   /* Reserved */
        u_char   ps_chan3_volume;      /* Channel 3 volume level */
};
```

The structure members and their descriptions follow:

Structure Member	Description
`ps_lbamsf`	The address format: a 0 (zero) means LBA; a 1 means MSF.
`ps_data_len1`	The audio data length if the address format is MSF.
`ps_data_len0`	The audio data length if the address format is LBA.
`ps_audio_status`	The audio status
`ps_control`	The control field containing attributes. The possible settings follow:

Bit No.	Set to 0 (Zero)	Set to 1
0	Audio without preemphasis	Audio with preemphasis
1	Digital copy prohibited	Digital copy permitted
2	Audio track	Data track
3	Two-channel audio	Four-channel audio

The low four bits are reserved.

Structure Member	Description
`ps_absaddr`	The absolute address of the audio track, MSF or LBA format.
`ps_chan0_select`	The selection code for Channel 0. The low four bits are reserved.
`ps_chan0_volume`	The volume level setting for Channel 0.
`ps_chan0_select`	The selection code for Channel 0. The low four bits are reserved.
`ps_chan1_volume`	The volume level setting for Channel 1.
`ps_chan1_select`	The selection code for Channel 1. The low four bits are reserved.
`ps_chan2_volume`	The volume level setting for Channel 2.

`ps_chan2_select`	The selection code for Channel 2. The low four bits are reserved.
`ps_chan3_volume`	The volume level setting for Channel 3.

11.3 Adding a Programmer-Defined SCSI/CAM Device

The procedure for installing device drivers described in *Writing Device Drivers, Volume 1: Tutorial* applies to adding SCSI/CAM peripheral device drivers to your system. Follow that procedure after completing the entries to the SCSI/CAM-specific structures described in this chapter and in Chapter 3.

SCSI/CAM Special I/O Interface 12

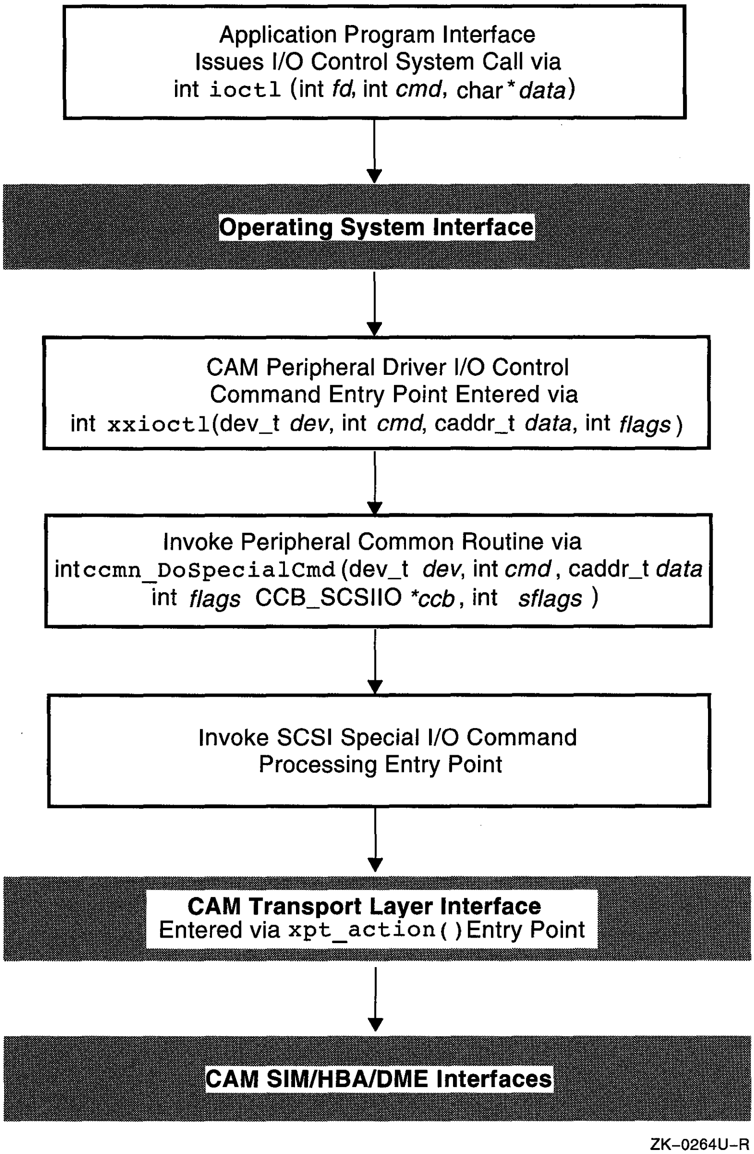
This chapter describes the SCSI/CAM special I/O interface. The S/CA software includes an interface developed to process special SCSI I/O control commands used by the existing Digital SCSI subsystem and to aid in porting new or existing SCSI device drivers from other vendors to the S/CA.

Application programs issue I/O control commands using the `ioctl` system call to send special SCSI I/O commands to a peripheral device. The term ''special'' refers to commands that are not usually issued to the device through the standard driver entry points. SCSI device drivers usually require the special I/O control commands in addition to the standard `read` and `write` system calls. With the SCSI/CAM special I/O interface, SCSI/CAM peripheral driver writers do not need detailed knowledge of either the system-specific or the CAM-specific structures and routines used to issue a SCSI command to the CAM I/O subsystem.

12.1 Application Program Access

Application programs access the SCSI/CAM special I/O interface by making requests to peripheral drivers using the `ioctl` system call. This system call is processed by system kernel support routines that invoke the device driver's I/O control command entry point in the character device switch table defined in the `/usr/sys/io/common/conf.c` file. The device driver's I/O control routine accesses the special I/O interface using either the supplied SCSI/CAM peripheral common routine, `ccmn_DoSpecialCmd`, or a driver-specific routine. Figure 12-1 shows the flow of application program requests through the operating system to the SCSI/CAM special I/O interface and the CAM I/O subsystem.

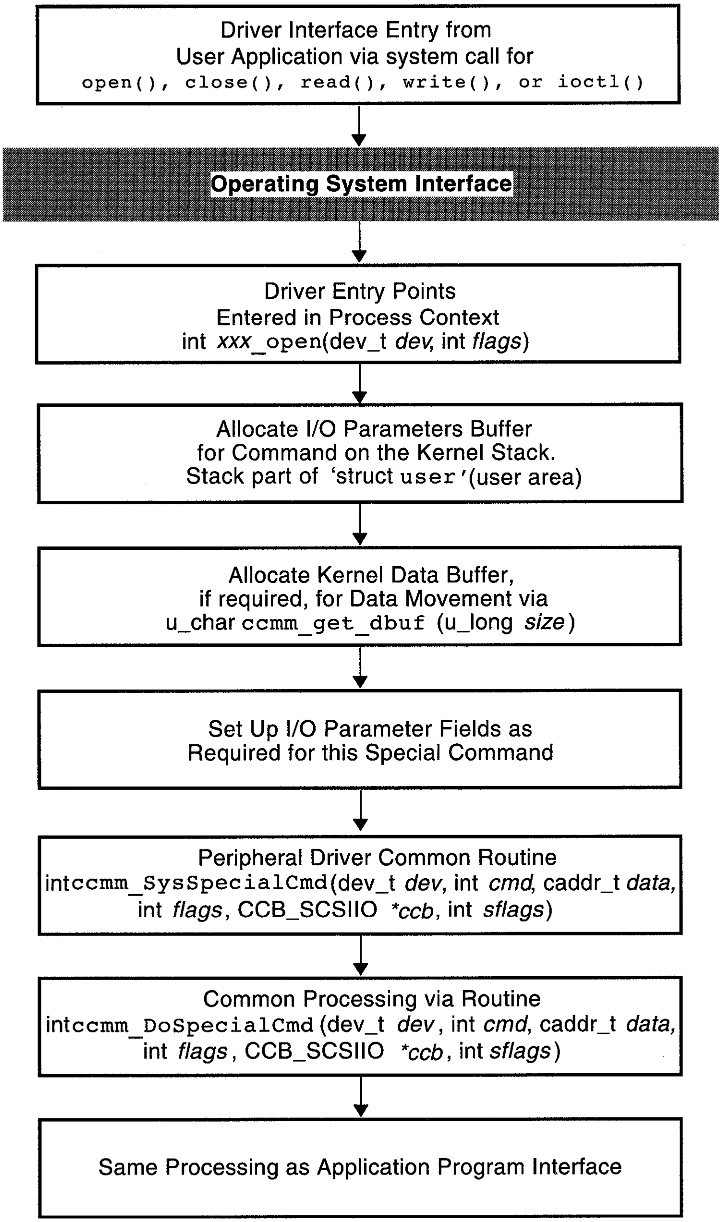
Figure 12-1: Application Program Flow Through SCSI/CAM Special I/O Interface



12.2 Device Driver Access

SCSI/CAM peripheral device drivers access the SCSI/CAM special I/O interface using either the supplied SCSI/CAM peripheral common routine, `ccmn_SysSpecialCmd`, or using a driver-specific routine. Figure 12-2 shows the flow of system requests from device drivers through the SCSI/CAM special I/O interface and the CAM I/O subsystem.

Figure 12-2: Device Driver Flow Through SCSI/CAM Special I/O Interface



ZK-0470U-R

12.3 SCSI/CAM Special Command Tables

The SCSI/CAM special I/O interface includes default command tables that provide backwards compatibility with existing SCSI I/O control commands. The following predefined SCSI/CAM Special Command Tables are included:

- `cam_GenericCmds`
- `cam_DirectCmds`
- `cam_AudioCmds`
- `cam_SequentialCmds`
- `cam_MtCmds`

The interface also allows commands to be added to the existing command tables and new command tables to be added. The SCSI/CAM special I/O interface includes routines that manipulate the tables so programmers can write device drivers to easily add and remove command tables.

The command table header structure, SPECIAL_HEADER, provides a bit mask of device types that can be used with a command table. The Special Command Header Structure is defined as follows:

```
/*
 * Special Command Header Structure:
 */

typedef struct special_header {
    struct special_header *sph_flink;   /* Forward link to next table */
    struct special_header *sph_blink;   /* Backward link to prev table */
    struct special_cmd *sph_cmd_table;  /* Pointer to command table */
    U32     sph_device_type;            /* The device types supported */
    U32     sph_table_flags;            /* Flags to control cmd lookup */
    caddr_t sph_table_name;             /* Name of this command table */
} SPECIAL_HEADER;
```

12.3.1 The sph_flink and sph_blink Members

These are table-linkage members that allow command tables to be dynamically added or removed from the list of tables searched by the SCSI/CAM special I/O interface when processing commands.

12.3.2 The sph_cmd_table Member

A pointer to the Special Command Entry Structure.

12.3.3 The sph_device_type Member

The device types supported by this SCSI/CAM Special Command Table.

12.3.4 The sph_table_flags Member

The SPH_SUB_COMMAND, which indicates that the command table contains subcommands.

12.3.5 The sph_table_name Member

The name of this SCSI/CAM Special Command Table.

12.4 SCSI/CAM Special Command Table Entries

Each SCSI/CAM Special Command Table contains multiple entries. Each entry provides enough information to process the command associated with that entry. The command tables can be dynamically added, but the entries within the command tables are not dynamic. Each command table's entries are statically defined so that individual entries cannot be appended to the table. The Special Command Entry Structure structure is defined as follows:

```
/*
 * Special Command Entry Structure:
 */
typedef struct special_cmd {
        u_int   spc_ioctl_cmd;        /* The I/O control command code */
        u_int   spc_sub_command;      /* The I/O control sub-command */
        u_char  spc_cmd_flags;        /* The special command flags */
        u_char  spc_cmd_code;         /* The special command code */
        u_short                 : 16; /* Unused ... align next field */
        U32     spc_device_type;      /* The device types supported */
        U32     spc_cmd_parameter;    /* Command parameter (if any) */
        U32     spc_cam_flags;        /* The CAM flags field for CCB */
        U32     spc_file_flags;       /* File control flags (fcntl) */
        int     spc_data_length;      /* Kernel data buffer length */
        int     spc_timeout;          /* Timeout for this command */
        int     (*spc_docmd)();       /* Function to do the command */
        int     (*spc_mkcdb)();       /* Function to make SCSI CDB */
        int     (*spc_setup)();       /* Setup parameters routine */
        caddr_t spc_cdbp;             /* Pointer to prototype CDB */
        caddr_t spc_cmdp;             /* Pointer to the command name */
} SPECIAL_CMD;
```

12.4.1 The spc_ioctl_cmd and spc_sub_command Members

These members contain the SCSI I/O control command code and subcommand used to locate the appropriate table entry. The subcommand is checked only if flags are set that indicate a subcommand exists.

12.4.2 The spc_cmd_flags Member

This member contains flags to control the action of the SCSI/CAM special I/O interface routines. The flag definitions are described in the following table:

Flag Name	Description
SPC_SUSER	Restricted to superuser.
SPC_COPYIN	User buffer to copy from.
SPC_COPYOUT	User buffer to copy to.
SPC_NOINTR	Do not allow sleep interrupts.
SPC_DATA_IN	Data direction is from device.
SPC_DATA_OUT	Data direction is to device.
SPC_DATA_NONE	No data movement for command.
SPC_SUB_COMMAND	Entry contains subcommand.
SPC_INOUT	Copy in and out.
SPC_DATA_INOUT	Copy data in and out.

12.4.3 The spc_command_code Member

This member contains the special SCSI opcode used to execute this command. This member is used during the creation of the CDB.

12.4.4 The spc_device_type Member

This member defines the specific device types with which this command is used. For example, direct-access and readonly direct-access devices share many of the same commands. Therefore, rather than duplicating command table entries, both device types can use the same command table. The values that are valid for this member are those defined in the Inquiry data device type member of the `inquiry_info` structure, which is defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

12.4.5 The spc_cmd_parameter Member

This member is used to define any special parameters used by the command. For example, the SCSI START CDB command, which is defined in the `/usr/sys/include/io/cam/scsi_direct.h` file, is used for stopping, starting, and ejecting a CDROM caddy. The parameter member can be defined as the subcommand code so a common routine can be used to create the CDB.

12.4.6 The spc_cam_flags Member

This member contains the CAM flags necessary for processing the command. The CAM flags are defined in the file `/usr/sys/include/io/cam/cam.h`.

12.4.7 The spc_file_flags Member

This member contains the file access bits required for accessing the command. For example, the command can be restricted to device files opened for read and write access. The file flags are defined in the file `/usr/sys/include/sys/file.h`.

12.4.8 The spc_data_length Member

This member describes the length of the buffer to hold additional kernel data that is required to process the command. Usually, this member is set to 0 (zero), since the data buffer lengths are normally decoded from the I/O command code or taken from a member in the I/O parameter buffer.

12.4.9 The spc_timeout Member

This member defines the default timeout for this command. This value is used for the SCSI I/O CCB timeout member, unless it is overridden by the timeout member in the Special I/O Argument Structure.

12.4.10 The spc_docmd Member

This member specifies the routine to invoke to execute the command. A routine is required by I/O commands that need special servicing. For example, if the I/O command does not return all the data read by the SCSI command, then a routine is needed to handle this special servicing.

12.4.11 The spc_mkcdb Member

This member specifies the routine that is invoked to create the CDB for the command. A routine is not necessary for simple commands, such as TEST UNIT READY. However, any command that requires additional members to be set up in the CDB prior to issuing the SCSI command must define this routine.

12.4.12 The spc_setup Member

This member is required by any command that has special setup requirements. For example, commands that pass a user buffer and length as part of the I/O parameters buffer structure must have a setup routine to copy these members to the Special I/O Argument Structure. This applies to all previously defined commands, but does not apply to commands implemented using the new SCSI_SPECIAL I/O control command code.

12.4.13 The spc_cdbp Member

This member is used by commands that can be implemented using a prototype CDB. A prototype CDB is a SCSI command that can be implemented using a statically defined SCSI CDB. Fields within the CDB do not change. Usually, simple SCSI commands, such as SCSI_START_UNIT, can be implemented with a prototype CDB so that the make CDB routine is not required.

12.4.14 The spc_cmdp Member

This member points to a string that describes the name of the command. This string is used during error reporting and during debugging.

12.4.15 Sample SCSI/CAM Special Command Table

The example that follows shows a sample SCSI/CAM Special Command Table with one entry defined:

```
#include "<cdrom.h"
#include "<mtio.h"
#include "<rzdisk.h"

#include <cam.h>
#include <cam_special.h>
#include <dec_cam.h>
#include <scsi_all.h>
#include <scsi_direct.h>
#include <scsi_rodirect.h>
#include <scsi_sequential.h>
#include <scsi_special.h>

extern int scmn_MakeFormatUnit(), scmn_SetupFormatUnit();

/*
 * Command Header for Direct-Access Command Table:
 */
struct special_header cam_DirectCmdsHdr = {
        (struct special_header *) 0,            /* sph_flink */
        (struct special_header *) 0,            /* sph_blink */
        cam_DirectCmds,                         /* sph_cmd_table */
        (BITMASK(ALL_DTYPE_DIRECT) |
                  BITMASK(ALL_DTYPE_RODIRECT)), /* sph_device_type */
        0,                                      /* sph_table_flags */
        "Direct Access Commands"                /* sph_table_name */
};

/*******************************************************************
 *                                                                 *
 *                Special Direct Access Command Table              *
 *                                                                 *
 *******************************************************************/
```

```
struct special_cmd cam_DirectCmds[] = {
    {   SCSI_FORMAT_UNIT,                          /* spc_ioctl_cmd */
        0,                                         /* spc_sub_command */
        (SPC_COPYIN | SPC_DATA_OUT),               /* spc_cmd_flags */
        DIR_FORMAT_OP,                             /* spc_cmd_code */
        BITMASK(ALL_DTYPE_DIRECT),                 /* spc_device_type */
        0,                                         /* spc_cmd_parameter */
        CAM_DIR_OUT,                               /* spc_cam_flags */
        FWRITE,                                    /* spc_file_flags */
        -1,                                        /* spc_data_length */
        (120 * ONE_MINUTE),                        /* spc_timeout */
        (int (*)()) 0,                             /* spc_docmd */
        scmn_MakeFormatUnit,                       /* spc_mkcdb */
        scmn_SetupFormatUnit,                      /* spc_setup */
        (caddr_t) 0,                               /* spc_cdbp */
        "format unit"                              /* spc_cmdp */
    },
        .
        .
        .
    { END_OF_CMD_TABLE }    /* End of cam_DirectCmds[] Table. */
};
/*
 * Define Special Commands Header & Table for Initialization Routine.
 */
struct special_header *cam_SpecialCmds = &cam_SpecialCmdsHdr;

struct special_header *cam_SpecialHdrs[] =
        { &cam_GenericCmdsHdr, &cam_DirectCmdsHdr, &cam_AudioCmdsHdr,
          &cam_SequentialCmdsHdr, &cam_MtCmdsHdr, 0 };
```

12.5 SCSI/CAM Special I/O Argument Structure

A Special I/O Argument Structure is passed to the SCSI/CAM special I/O interface to control processing of the I/O control command being executed. The structure members provide information to process a special command for different SCSI subsystems. Default settings and routines invoked by the SCSI/CAM special I/O interface can be overridden by the calling routine. Table 12-1 shows the members that are mandatory for the calling routine to set up, the members that are optional, and the members that are used or filled in by the SCSI/CAM special I/O interface.

Table 12-1: SCSI/CAM Special I/O Argument Structure

Member Name	Type	Description
`U32 sa_flags;`	M	Flags to control command
`dev_t sa_dev;`	M	Device major/minor number
`u_char sa_unit;`	U	Device logical unit number
`u_char sa_bus;`	M	SCSI host adapter bus number
`u_char sa_target;`	M	SCSI device target number

Table 12-1: (continued)

Member Name	Type	Description
u_char sa_lun;	M	SCSI logical unit number
u_int sa_ioctl_cmd;	M	The I/O control command
u_int sa_ioctl_scmd;	C	The subcommand, if any
caddr_t sa_ioctl_data;	C	The command data pointer
caddr_t sa_device_name;	M	Pointer to the device name
int sa_device_type;	M	The peripheral device type
int sa_iop_length;	I	Parameters' buffer length
caddr_t sa_iop_buffer;	I	Parameters' buffer address
int sa_file_flags;	M	The file control flags
u_char sa_sense_length;	O	Sense data buffer length
u_char sa_sense_resid;	I	Sense data residual count
caddr_t sa_sense_buffer;	O	Sense data buffer address
u_char sa_user_length;	I	User data buffer length
caddr_t sa_user_buffer;	I	User data buffer address
struct buf *sa_bp;	O	Kernel-only I/O request buffer
CCB_SCSIIO *sa_ccb;	O	CAM control block buffer
struct special_cmd *sa_spc;	I	Special command table entry
struct special_header *sa_sph;	O	Special command table header
U32 sa_cmd_parameter;	I	Command parameter, if any
int (*sa_error)();	O	The error report routine
int (*sa_start)();	O	The driver start routine
int sa_data_length;	I	Kernel data buffer length
caddr_t sa_data_buffer;	I	Kernel data buffer address
caddr_t sa_cdb_pointer;	I	Pointer to the CDB buffer
u_char sa_cdb_length;	I	Length of the CDB buffer
u_char sa_cmd_flags;	I	The special command flags
u_char sa_retry_count;	I	The current retry count
u_char sa_retry_limit;	O	Times to retry this command
int sa_timeout;	O	Timeout for this command
int sa_xfer_resid;	I	Transfer residual count
caddr_t sa_specific;	O	Driver-specific information

Legend: M = Mandatory. Must be set up by the caller.
C = Command-dependent. Depends on special command.
O = Optional. Optionally overrides defaults.
I = Interface. Used or filled in by SCSI/CAM special I/O interface.
U = Unused. Not used by SCSI/CAM special I/O interface.

Several of the members marked as mandatory in Table 12-1 are set up initially by the routine that allocates the Special I/O Argument Structure. The following members are initialized by the allocation routine: `sa_bus`; `sa_target`; `sa_lun`; `sa_unit` (same as target); `sa_retry_limit` (set to 30); and `sa_start` (set to .`xpt_action`)

Fields that are identified as optional in Table 12-1 can be defined by the caller to override some of the defaults used by the SCSI/CAM special I/O interface. The following table describes the defaults used by the SCSI/CAM special I/O interface:

Member Name	Default
`sa_sense_length`	Set to DEC_AUTO_SENSE_SIZE, which is defined in `/usr/sys/include/io/cam/dec_cam.h`.
`sa_sense_buffer`	Sense buffer in SCSI/CAM Peripheral Device Driver Working Set Structure.
`sa_bp`	Allocated as needed for data movement commands.
`sa_ccb`	Allocated by the CAM `xpt_ccb_alloc` routine.
`sa_error()`	Special interface error report routine.
`sa_start()`	Uses the CAM `xpt_action` routine.
`sa_timeout`	Uses the timeout value from the SCSI/CAM Special Command Table entry.
`sa_specific`	Is not set up or used by SCSI/CAM special I/O interface.

12.5.1 The sa_flags Member

This member is used to control the actions of the SCSI/CAM special I/O interface. The low order five bits of this member can be set by the calling routine. All other bits in this member are reserved. The table that follows shows the control flags that can be set by the calling routine:

Flag Name	Description
`SA_NO_ERROR_RECOVERY`	Do not perform error recovery.
`SA_NO_ERROR_LOGGING`	Do not log error messages.
`SA_NO_SLEEP_INTR`	Do not allow sleep interrupts.
`SA_NO_SIMQ_THAW`	Leave SIM queue frozen on errors.
`SA_NO_WAIT_FOR_IO`	Do not wait for I/O to complete.

12.5.2 The sa_dev Member

This member contains the device major/minor number pair passed into the device driver routines. It is used to fill in the `bp_dev` member of the system I/O request member.

12.5.3 The sa_unit, sa_bus, sa_target, and sa_lun Members

These members are used to address the SCSI device to which the command is being sent. The `sa_unit` member is not used, but has been included for device drivers that implement logical device mapping.

12.5.4 The sa_ioctl_cmd Member

This member contains the I/O control command to be processed. This command usually maps directly to a SCSI I/O Command, but that is not necessary. For example, the Digital-specific SCSI_GET_SENSE command returns the sense data from the last failing command. A REQUEST SENSE command is not issued to the device, because autosense is assumed to have been enabled on the failing command, and the sense data is part of the common Peripheral Device Structure.

12.5.5 The sa_ioctl_scmd Member

This member must be filled in for special commands implemented with a subcommand code. For example, magnetic tape I/O control commands have both an I/O control command code and a subroutine command code.

12.5.6 The sa_ioctl_data Member

An I/O parameters buffer is required if the I/O control command transfers data to and from the kernel. If the request came from an application program, this buffer is normally passed into the driver `ioctl` routine.

12.5.7 The sa_device_name Member

This member contains a pointer to the device name string that is used when reporting device errors.

12.5.8 The sa_device_type Member

This member contains the device type member from the Inquiry data. This member controls the SCSI/CAM Special Command Tables and the entries within each command table that are searched for the SCSI/CAM special I/O command being issued.

12.5.9 The sa_iop_length and sa_iop_buffer Members

These members are used internally by the SCSI/CAM special I/O interface when processing a command. If I/O would normally be performed directly to the I/O parameters buffer because no other buffer was set up, then a kernel buffer is allocated and set up in these members.

12.5.10 The sa_file_flags Member

This member contains the file flags passed into the device driver routines. The flags describe access control bits associated with the device. The file access flags are defined in the `/usr/sys/include/io/cam/fcntl.h` file.

12.5.11 The sa_sense_length and sa_sense_buffer Members

These members set up the sense buffer and expected sense data length that are used by autosense when device errors occur. If these members are not set up by the calling routine, then the SCSI/CAM special I/O interface uses the sense buffer allocated in the SCSI/CAM Peripheral Device Driver Working Set Structure that is pointed to by the SCSI I/O CCB.

12.5.12 The sa_user_length and sa_user_buffer Members

These members are set up by command setup routines to describe the user buffer and user data length required by a command. Requests from application programs that pass a user buffer and length in the I/O parameter buffers require a setup routine to copy this information into those members .

The SCSI/CAM special I/O interface checks access and locking on this address range and sets up the address and length in the SCSI I/O CCB for the command.

12.5.13 The sa_bp Member

This member contains a pointer to a system I/O request buffer for commands that perform data movement directly to user address space. A system buffer is not required if a kernel data buffer is used for I/O. If the calling routine does not pass a previously allocated request buffer in this member, and the SCSI/CAM special I/O interface determines that the I/O requires one based on the I/O buffer address, then a request buffer is allocated and deallocated automatically by the SCSI/CAM special I/O interface.

12.5.14 The sa_ccb Member

This member contains a pointer to the SCSI I/O CCB for a command. If the calling routine does not specify a SCSI I/O CCB in this member, then the SCSI/CAM special I/O interface automatically allocates and deallocates a SCSI I/O CCB for the command.

12.5.15 The special_cmd Member

This member is used internally by the SCSI/CAM special I/O interface to save the SPECIAL_CMD after a command is located.

12.5.16 The special_header Member

This member can be used by the calling routine to specify the SCSI/CAM Special Command Table to search for the special command. This lets device drivers restrict the SCSI/CAM Special Command Tables that are searched. If this member is not used, then all the SCSI/CAM Special Command Tables in the list are searched for an entry that matches the special command being processed.

12.5.17 The sa_cmd_parameter Member

This member is used to store the command parameter, if any, from the command entry associated with this special command. This member is used by special support routines when setting up members for a particular CDB.

12.5.18 The sa_error Member

This member contains the routine to be invoked when an error condition is detected. If not specified, a SCSI/CAM special I/O interface support routine handles the error condition. Otherwise, the routine is called as follows:

```
status = (*sap->sa_error)(ccb, sense);
```

This member can be specified for drivers requiring specialized error handling and for specific error logging. The SCSI/CAM special I/O interface's error logging uses the `mprintf` facility to report errors. Both sense key and CAM status members are logged.

12.5.19 The sa_start Member

This member contains the routine that starts processing the SCSI I/O CCB. If not specified, the CAM `xpt_action` routine is used. The routine is invoked as follows:

```
(void) ((sap->sa_start)(ccb);
```

12.5.20 The sa_data_length and sa_data_buffer Members

These members are used internally by the SCSI/CAM special I/O interface to store the address and length of an additional kernel buffer required for a command. These members are usually initialized by the resulting value of the Special Command Entry Structure member, `spc_data_length`, but can be used by SCSI/CAM special I/O command developers if needed.

12.5.21 The sa_cdb_pointer Member

This member is used internally by the SCSI/CAM special I/O interface to save a pointer to the CDB for this special command. This member may point to a prototype CDB; to a driver-allocated CDB buffer, if the CAM_CDB_POINTER flag is set in CCB header; or to the CDB buffer allocated within the SCSI I/O CCB. This member is set up with the CDB buffer address before the Special Command Header Structure make CDB routine is invoked as follows:

```
status = (*spc->spc_mkcdb)(sap, cdbp);
```

12.5.22 The sa_cdb_length Member

This member is used to specify the size in bytes of the CDB required by a SCSI command. If the Special Command Header Structure make CDB routine does not set up this member, then the SCSI Group Code is decoded to determine the length.

12.5.23 The sa_cmd_flags Member

This member is initialized from the Special Command Header Structure `spc_cmd_flags` member so SCSI/CAM special I/O command support routines have easy and quick access to the flags.

12.5.24 The sa_retry_count Member

This member contains the number of retrys that were required to successfully complete the request. It is filled in by the SCSI/CAM special I/O interface after processing the command.

12.5.25 The sa_retry_limit Member

This member contains the maximum number of times a command is retried. The only retries automatically handled by the SCSI/CAM special I/O interface are a sense key of Unit Attention, or a SCSI bus status of Bus Busy or Reservation Conflict. All other error conditions must be handled by the calling routine.

12.5.26 The sa_timeout Member

This member contains the timeout value, in seconds, to use with the command being processed. This member can be specified by the calling routine. If it is not specified, the timeout value is taken from the Special Command Entry Structure. This member is used to initialize the `cam_timeout` member of the SCSI I/O CCB before issuing the command.

12.5.27 The sa_xfer_resid Member

This member contains the residual byte count of data movement commands. This member is copied from the `cam_resid` member of the SCSI I/O CCB before returning to the caller.

12.5.28 The sa_specific Member

This member is not set up or used by the SCSI/CAM special I/O interface. It provides a mechanism for device driver code to pass driver-dependent information to SCSI/CAM special I/O command support routines. The SCSI/CAM peripheral driver common routine `ccmn_DoSpecialCmd` passes the pointer to the Peripheral Device Structure in this member.

12.5.29 Sample Function to Create a CDB

The following sample function illustrates how to use the SCSI/CAM special I/O interface to create a CDB for a SCSI FORMAT_UNIT command:

```
/*****************************************************************
 *                                                               *
 * scmn_MakeFormatUnit() - Make Format Unit Command Descriptor Block.*
 *                                                               *
 * Inputs:      sap = Special command argument block pointer.    *
 *              cdbp = Pointer to command descriptor block.      *
 *                                                               *
 * Return Value:                                                 *
 *              Returns 0 for SUCCESS, or error code on failures. *
 *                                                               *
 *****************************************************************/
int
scmn_MakeFormatUnit (sap, cdbp)
register struct special_args *sap; 1
register struct dir_format_cdb6 *cdbp; 2
{
        register struct special_cmd *spc = sap->sa_spc; 3
        register struct format_params *fp; 4

        fp = (struct format_params *) sap->sa_iop_buffer;
        cdbp->opcode = (u_char) spc->spc_cmd_code;
        if (fp->fp_defects == VENDOR_DEFECTS) { 5
            cdbp->fmt_data = 1;
            cdbp->cmp_list = 1;
        } else if (fp->fp_defects == KNOWN_DEFECTS) {
            cdbp->fmt_data = 1;
            cdbp->cmp_list = 0;
        } else if (fp->fp_defects == NO_DEFECTS) {
            cdbp->fmt_data = 0;
            cdbp->cmp_list = 0;
        }
        cdbp->defect_list_fmt = fp->fp_format; 6
        cdbp->vendor_specific = fp->fp_pattern;
        cdbp->interleave1 = 0;
        cdbp->interleave0 = fp->fp_interleave;
        return (SUCCESS);
}
```

1 This line declares a register structure pointer to a Special I/O Argument Structure that controls processing of the I/O command. The Special I/O Argument Structure is defined in the `/usr/sys/include/io/cam/cam_special.h` file.

2 This line declares a register structure pointer to a structure containing the format for a 6-byte CDB. The structure is defined in the `/usr/sys/include/io/cam/scsi_direct.h` file.

3 This line declares a register structure pointer to a Special I/O Control Commands Structure that saves the SPECIAL_CMD after it is located in the `sa_spc` member of the Special I/O Argument Structure. The Special I/O Control Commands Structure is defined in the `/usr/sys/include/io/cam/cam_special.h` file.

4 This line declares a register structure pointer to a structure containing the format parameters for a SCSI FORMAT UNIT command. The structure is defined in the `/usr/sys/include/io/cam/rzdisk.h` file.

5 This section tests the contents of the `fp_defects` member of the format parameters structure to determine the contents of the `fmt_data` and `cmp_list` members of the `dir_format_cdb6` structure.

6 This section assigns the contents of the `dir_format_cdb6` members to the equivalent members of the `format_params` structure.

12.5.30 Sample Function to Set Up Parameters

The following sample function illustrates how to use the SCSI/CAM special I/O interface to set up parameters for a SCSI FORMAT_UNIT command:

```
/*******************************************************************
 *                                                                 *
 * scmn_SetupFormatUnit() - Set up Format Unit Parameters.         *
 *                                                                 *
 * Inputs:      sap = Special command argument block pointer.      *
 *              data = The address of input/output arguments.      *
 *                                                                 *
 * Return Value:                                                   *
 *              Returns 0 for SUCCESS, or error code on failures.  *
 *                                                                 *
 *******************************************************************/
int
scmn_SetupFormatUnit (sap, data)
register struct special_args *sap; 1
caddr_t data;
{
        struct form2_defect_list_header defect_header; 2
        register struct form2_defect_list_header *ddh = &defect_header;
        register struct format_params *fp;  3

        fp = (struct format_params *) data;
        sap->sa_user_buffer = (caddr_t) fp->fp_addr; 4

        /*
         * For diskettes, there are no defect lists.
         */
        if ( ((sap->sa_user_length = fp->fp_length) == 0) &&
             (fp->fp_defects == NO_DEFECTS) ) {
            sap->sa_cmd_flags &= ~(SPC_INOUT | SPC_DATA_INOUT);
            return (SUCCESS);
        }

#ifdef KERNEL
        /*
         * Ensure the defect list address is valid (user address).
         */
        if ( ((sap->sa_flags & SA_SYSTEM_REQUEST) == 0) &&
                !CAM_IS_KUSEG(fp->fp_addr) ) {
            return (EINVAL);
        }
```

```
        /*
         * The format parameters structure is not set up with the length
         * of the defect lists as it should be.  Therefore, we must copy
         * in the defect list header then calculate the defect list length.
         */
        if (copyin ((caddr_t)fp->fp_addr, (caddr_t)ddh, sizeof(*ddh)) != 0) {
            return (EFAULT);
#else
    (void) bcopy ((caddr_t)fp->
    fp_addr, (caddr_t)ddh,
    sizeof(* ddh))
#endif
        }
        sap->sa_user_length = (int) (  (ddh->defect_len1 << 8) +
                                        ddh->defect_len0 + sizeof(*ddh) );

        return (SUCCESS);
}
```

1 This line declares a register structure pointer to a Special I/O Argument Structure that controls processing of the I/O command. The Special I/O Argument Structure is defined in the /usr/sys/include/io/cam/cam_special.h file.

2 This line declares a structure pointer to a structure containing the format defect list header for a SCSI FORMAT UNIT command. The structure is defined in the /usr/sys/include/io/cam/rzdisk.h file.

3 This line declares a register structure pointer to a structure containing the format parameters for a SCSI FORMAT UNIT command. The structure is defined in the /usr/sys/include/io/cam/rzdisk.h file.

4 This line assigns the user buffer data address to the defect list address.

12.6 SCSI/CAM Special I/O Control Command

A SCSI/CAM special I/O control command has been defined to provide a single standard method of implementing new SCSI/CAM special I/O commands. A subcommand member is used to determine the specific SCSI command being issued.

The SCSI/CAM special I/O control command structure can be used both in porting applications using existing SCSI I/O control commands and in implementing new SCSI commands. Applications can be modified to use this structure to gain control over subsystem processing. For example, the SCSI/CAM special I/O command flags can be set to control error recovery and error reporting; sense data can be returned automatically by specifying a sense buffer address and length; and the command timeout and retry limit can be specified.

A member in the Special I/O Control Commands Structure must be initialized to zero if a default value is desired. A nonzero member is used to override the default value.

The SCSI I/O control command and its associated structure and definitions are included in the file `/usr/sys/include/io/cam/scsi_special.h`. The `scsi_special` structure is defined as follows:

```
/*
 * Structure for Processing Special I/O Control Commands.
 */
struct scsi_special {
        U32     sp_flags;               /* The special command flags */
        dev_t   sp_dev;                 /* Device major/minor number */
        u_char  sp_unit;                /* Device logical unit number */
        u_char  sp_bus;                 /* SCSI host adapter bus number */
        u_char  sp_target;              /* SCSI device target number */
        u_char  sp_lun;                 /* SCSI logical unit number */
        u_int   sp_sub_command;         /* The subcommand */
        U32     sp_cmd_parameter;       /* Command parameter (if any) */
        int     sp_iop_length;          /* Parameters buffer length */
        caddr_t sp_iop_buffer;          /* Parameters buffer address */
        u_char  sp_sense_length;        /* Sense data buffer length */
        u_char  sp_sense_resid;         /* Sense data residual count */
        caddr_t sp_sense_buffer;        /* Sense data buffer address */
        int     sp_user_length;         /* User data buffer length */
        caddr_t sp_user_buffer;         /* User data buffer address */
        int     sp_timeout;             /* Timeout for this command */
        u_char  sp_retry_count;         /* Retrys performed on command */
        u_char  sp_retry_limit;         /* Times to retry this command */
        int     sp_xfer_resid;          /* Transfer residual count */
};
```

This structure is used with the following SCSI Special I/O Control Command:

```
#define SCSI_SPECIAL     _IOWR('p', 100, struct scsi_special)
```

12.6.1 The sp_flags Member

This member controls the actions of the SCSI/CAM special I/O interface. The low order three bits can be set by the calling routine. The other bits are reserved for use by SCSI/CAM peripheral drivers and the SCSI/CAM special I/O interface routines. The bits that can be set by the calling routine are described as follows:

Flag Name	Description
SA_NO_ERROR_RECOVERY	Do not perform error recovery.
SA_NO_ERROR_LOGGING	Do not log error messages.
SA_NO_SLEEP_INTR	Do not allow sleep interrupts.

12.6.2 The sp_dev, sp_unit, sp_bus, sp_target, and sp_lun Members

These members pass the device major/minor number pair and the device bus, target, LUN, and unit information to the SCSI/CAM special I/O interface when the I/O control command is not being issued to a SCSI/CAM peripheral device driver. These members provide the necessary hooks to allow software pseudodevice drivers, such as the User Agent driver, to send requests to the SCSI/CAM special I/O interface.

12.6.3 The sp_sub_command Member

This member contains the SCSI/CAM special I/O subcommand code of the SCSI command to execute. This member can also be defined as an I/O control command to support backwards compatibility with preexisting SCSI I/O control commands. The SCSI/CAM special I/O interface detects an I/O control command, as opposed to a subcommand code, and coerces the arguments into the appropriate format for processing by the support routines associated with that I/O control command. The predefined subcommand codes are listed in the file `/usr/sys/include/io/cam/scsi_special.h`.

12.6.4 The sp_cmd_parameter Member

This member contains the command parameter, if any, for the SCSI special I/O command being issued. This parameter is specific to the special command processing routines and is not used directly by the SCSI/CAM special I/O interface routines.

12.6.5 The sp_iop_length and sp_iop_buffer Members

These members contain the I/O parameters buffer and length for those commands that require additional parameters. These members are used by the special command processing routines to obtain and set up additional information prior to issuing the SCSI command. For example, the SCSI FORMAT_UNIT I/O control command passes a `format_params` structure that describes the format, length, pattern, and interleave information for the defect list. This information is used by the `scmn_MakeFormatUnit` support routine when creating the CDB for this command.

12.6.6 The sp_sense_length, sp_sense_resid, and sp_sense_buffer Members

These members contain the buffer, length, and residual byte count for the sense data that is returned when device errors occur. If these members are specified, then the last sense data is saved in the Peripheral Device Structure from which it can be obtained by the Digital-specific SCSI_GET_SENSE I/O control command.

12.6.7 The sp_user_length and sp_user_buffer Members

These members contain the user buffer and length for those commands that require them. The SCSI/CAM special I/O interface performs verification, locking, and unlocking of the user pages when processing the command.

12.6.8 The sp_timeout Member

This member can be specified to override the default timeout, in seconds, which is usually taken from the Special Command Entry Structure.

12.6.9 The sp_retry_count Member

This member contains the number of retrys that were required to successfully complete the request. It is filled in by the SCSI/CAM special I/O interface after processing the command.

12.6.10 The sp_retry_limit Member

This member contains the maximum number of times a command is retried. The only retries automatically handled by the SCSI/CAM special I/O interface are a sense key of Unit Attention, or a SCSI bus status of Bus Busy or Reservation Conflict. All other error conditions must be handled by the calling routine.

12.6.11 The sp_xfer_resid Member

This member is filled in with the transfer residual byte count when a command completes. The SCSI/CAM special I/O interface copies the `cam_resid` member of the SCSI I/O CCB to this member before completing the request.

12.6.12 Sample Function to Create an I/O Control Command

The following sample function illustrates how to use the SCSI/CAM special I/O interface to create an I/O control command:

```
/*************************************************************************
 *                                                                       *
 * DoIoctl()    Do An I/O Control Command.                               *
 *                                                                       *
 * Description:                                                          *
 *      This routine issues the specified I/O control command to the     *
 *      file descriptor associated with the CD-ROM device driver.        *
 *                                                                       *
```

```
 * Inputs:      cmd = The I/O control command.                          *
 *              argp = The command argument to pass.                    *
 *              msgp = The message to display on errors.                *
 *                                                                      *
 * Return Value:                                                        *
 *              Returns 0 / -1 = SUCCESS / FAILURE.                     *
 *                                                                      *
 ************************************************************************/
int
DoIoctl (cmd, argp, msgp)
int cmd;
caddr_t argp;
caddr_t msgp;
{
        int status;
#if defined(CAM)
        struct scsi_special special_cmd; [1]
        register struct scsi_special *sp = &special_cmd;
        register struct extended_sense *es; [2]

        es = (struct extended_sense *)SenseBufPtr;
        bzero ((char *) sp, sizeof(*sp));
        bzero ((char *) es, sizeof(*es));
        sp->sp_sub_command = cmd; [3]
        sp->sp_sense_length = sizeof(*es);
        sp->sp_sense_buffer = (caddr_t) es;
        sp->sp_iop_length = ((cmd & ~(_IOC_INOUT|_IOC_VOID)) >> 16);
        sp->sp_iop_buffer = argp;
        if ((status = ioctl (CdrFd, SCSI_SPECIAL, sp)) < 0) { [4]
            perror (msgp);
            if (es->snskey) {
                cdbg_DumpSenseData (es);
            }
        }
#else /* !defined(CAM) */
        if ((status = ioctl (CdrFd, cmd, argp)) < 0) {
                perror (msgp);
        }
#endif /* defined(CAM) */
        return (status);
}
```

[1] This line declares a structure to process a special I/O control command. The `scsi_special` structure is defined in the `/usr/sys/include/io/cam/scsi_special.h` file.

[2] This line declares a structure defining the extended sense format for a REQUEST SENSE command. The `extended_sense` structure is defined in the `/usr/sys/include/io/cam/rzdisk.h` file.

[3] This section assigns the program parameters to the `special_cmd` members.

[4] This is a standard I/O control call issued from application code. The SCSI_SPECIAL argument is defined in the `/usr/sys/include/io/cam/scsi_special.h` file.

12.7 Other Sample Code

This section contains other driver code samples that use the SCSI/CAM special I/O interface.

12.7.1 Sample Code to Open a Device

The following sample code illustrates how to use the SCSI/CAM special I/O interface to open a CDROM device from a device driver:

```
/***********************************************************************
 *                                                                     *
 * cdrom_open() - Driver Entry Point to Open CD-ROM Device.            *
 *                                                                     *
 * Inputs:      dev = The device major/minor number pair.              *
 *              flags = The file open flags (read/write/nodelay).      *
 *                                                                     *
 * Outputs:     Returns 0 for Success or error code on Failure.        *
 *                                                                     *
 ***********************************************************************/
cdrom_open (dev, flags)
dev_t dev;
int flags;
{
        register PDRV_DEVICE *pd; 1
        DIR_READ_CAP_DATA read_capacity; 2
        DIR_READ_CAP_DATA *capacity = &read_capacity;
                .
                .
                .
        pd = GET_PDRV_PTR(dev); 3
        status = cdrom_read_capacity (pd, capacity, flags);
                .
                .
                .
        return (status);
}
/***********************************************************************
 *                                                                     *
 * cdrom_read_capacity() - Obtain Disk Capacity Information.           *
 *                                                                     *
 * Inputs:      pd = Pointer to peripheral driver structure.           *
 *              capacity = Pointer to read capacity data buffer.       *
 *              flags = The file open flags.                           *
 *                                                                     *
 * Outputs:     Returns 0 for Success or error code on Failure.        *
 *                                                                     *
 ***********************************************************************/
int
cdrom_read_capacity (pd, capacity, flags)
PDRV_DEVICE *pd;
DIR_READ_CAP_DATA *capacity;
int flags;
{
     int status;

     PRINTD(DEV_BUS_ID(pd->pd_dev), DEV_TARGET(pd->pd_dev),
             DEV_LUN(pd->pd_dev), CAMD_CDROM, 4
```

```
            ("[%d/%d/%d] cdrom_read_capacity: ENTRY - pd = 0x%x, \
            capacity = 0x%x, flags = 0x%x0,
            DEV_BUS_ID(pd->pd_dev), DEV_TARGET(pd->pd_dev),
            DEV_LUN(pd->pd_dev), pd, capacity, flags));

    bzero ((char *)capacity, sizeof(*capacity));

    status = ccmn_SysSpecialCmd (pd->pd_dev, SCSI_READ_CAPACITY, 5
    (caddr_t) capacity, flags, (CCB_SCSIIO *) 0, SA_NO_ERROR_LOGGING);

    PRINTD(DEV_BUS_ID(pd->pd_dev), DEV_TARGET(pd->pd_dev),
            DEV_LUN(pd->pd_dev), CAMD_CDROM,
            ("[%d/%d/%d] cdrom_read_capacity: EXIT - status = %d (%s)0,
            DEV_BUS_ID(pd->pd_dev), DEV_TARGET(pd->pd_dev),
            DEV_LUN(pd->pd_dev), status, cdbg_SystemStatus(status))); 6

    return (status);
}
```

1 This line assigns a register to a Peripheral Device Structure pointer for the device to be opened. The Peripheral Device Structure is defined in the `/usr/sys/include/io/cam/pdrv.h` file.

2 This line declares a structure to contain the capacity data returned for the device. The DIR_READ_CAP_DATA structure is defined in the `/usr/sys/include/io/cam/scsi_direct.h` file.

3 This line calls the GET_PDRV_PTR macro to return a pointer to the Peripheral Device Structure for the device. The GET_PDRV_PTR macro is defined in the `/usr/sys/include/io/cam/pdrv.h` file.

4 This section uses the bus, target, and LUN information to be printed if the CAMD_CDROM flag is set. The CAMD_CDROM flag is defined in the `/usr/sys/include/io/cam/cam_debug.h` file.

5 This section calls the SCSI/CAM peripheral common routine `ccmn_SysSpecialCmd`, to issue the SCSI I/O command, passing the major/minor device number pair for the device and the SCSI_READ_CAPACITY `ioctl` command, which is defined in the `/usr/sys/include/io/cam/rzdisk.h` file. It sets the SA_NO_ERROR_LOGGING flag, which is defined in the `/usr/sys/include/io/cam/cam_special.h` file for device drivers, and in the `/usr/sys/include/io/cam/scsi_special.h` file for application programs.

6 This debug line calls the `cdbg_SystemStatus` routine, passing the status as an argument.

12.7.2 Sample Code to Create a Driver Entry Point

The following sample code illustrates how to use the SCSI/CAM special I/O interface to create a driver entry point for I/O control commands:

```
/************************************************************************
 *                                                                      *
 * cdrom_ioctl() - Driver Entry Point for I/O Control Commands.         *
 *                                                                      *
 * Inputs:      dev = The device major/minor number pair.               *
 *              cmd = The I/O control command code.                     *
 *              data = The I/O parameters data buffer.                  *
 *              flags = The file open flags (read/write/nodelay).       *
 *                                                                      *
 * Outputs:     Returns 0 for Success or error code on Failure.         *
 *                                                                      *
 ************************************************************************/
int
cdrom_ioctl (dev, cmd, data, flags)
dev_t dev;
register int cmd;
caddr_t data;
int flags;
{
        register PDRV_DEVICE *pd; [1]
        register DISK_SPECIFIC *cdisk;
        register DEV_DESC *dd;
        int status;

        pd    = GET_PDRV_PTR(dev); [2]
        dd    = pd->pd_dev_desc;
        cdisk = (DISK_SPECIFIC *)pd->pd_specific;

        switch (cmd) {
                .
                .
            /* Process Expected I/O Control Commands */
                .
                .
            default:
                /*
                 * Process Special I/O Control Commands.
                 */
                status = ccmn_DoSpecialCmd (dev, cmd, data, flags, [3]
                             (CCB_SCSIIO *) 0, 0);
                break;
        }
        return (status);
}
```

[1] This section reserves registers for pointers to a Peripheral Device Structure and a Device Descriptor Structure, both of which are defined in the `/usr/sys/include/io/cam/pdrv.h` file, and to a DISK_SPECIFIC structure, which is defined in the `/usr/sys/include/io/cam/cam_disk.h` file.

[2] This line calls the GET_PDRV_PTR macro to return a pointer to the Peripheral Device Structure for the device. The GET_PDRV_PTR macro

is defined in the /usr/sys/include/io/cam/pdrv.h

3 This section calls the SCSI/CAM peripheral common routine, ccmn_DoSpecialCmd, to issue the special I/O command.

Header Files Used by Device Drivers A

This appendix contains the following:

- A list of header files used by all device drivers
- A list of header files used by SCSI/CAM peripheral device drivers
- The contents of the `/usr/sys/include/io/cam/cam.h` file.

Table A-1 lists the header files used by all SCSI device drivers, with a short description of the contents of each. For convenience, the full path name for the file is given and the files are listed in alphabetical order. However, device driver code should be written to include header files by specifying the relative path name instead of the full path name. For example, `/usr/sys/include/sys/buf.h`, is the full path name for the file `buf.h` , but device driver code to include `buf.h` should be written as follows:

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

For a more complete list, refer to the *Writing Device Drivers, Volume 1: Tutorial.*

Table A-1: Header Files Used by Device Drivers

Header File	Contents
`/usr/sys/include/io/common/devio.h`	Defines common structures and definitions for device drivers and the `cmd` DEVIOCGET `ioctl`.
`/usr/sys/include/sys/buf.h`	Defines the `buf` structure used to pass I/O requests to the `strategy` routine of a block driver.

Table A-1: (continued)

Header File	Contents
`/usr/sys/include/sys/conf.h`	Defines the `bdevsw` (block device switch), `cdevsw` (character device switch), and `linesw` (tty control line switch) structures. This file is included in the source file `/usr/sys/io/common/conf.c`.
`/usr/sys/include/sys/errno.h`	Defines the error codes returned to a user process by a driver.
`/usr/sys/include/sys/fcntl.h`	Defines I/O mode flags supplied by user programs to `open` and `fcntl` system calls.
`/usr/sys/include/sys/ioctl.h`	Defines commands for `ioctl` interfaces in different drivers.
`/usr/sys/include/sys/kernel.h`	Defines global variables used by the kernel.
`/usr/sys/include/sys/map.h`	Defines structures associated with resource allocation maps.
`/usr/sys/include/sys/mbuf.h`	Defines macros to allocate memory resources.
`/usr/sys/include/sys/mtio.h`	Defines commands and structures for magnetic tape operations.
`/usr/sys/include/sys/param.h`	Defines constants and interfaces used by the DEC OSF/1 kernel.
`/usr/sys/include/sys/proc.h`	Defines the `proc` structure, which defines a user process.
`/usr/sys/include/sys/systm.h`	Defines generic kernel global variables.
`/usr/sys/include/sys/time.h`	Defines structures and symbolic names used by time-related routines and macros.

Table A-1: (continued)

Header File	Contents
/usr/sys/include/sys/tty.h	Defines parameters and structures associated with interactive terminals; also defines the clist structure. This file can be included by any device driver that uses the clist structure.
/usr/sys/include/sys/types.h	Defines system data types and major and minor device macros.
/usr/sys/include/sys/uio.h	Contains the definition of the uio structure, used by character device drivers that need to access the uio structure.
/usr/sys/include/sys/user.h	Defines the user structure that describes a user process.
/usr/sys/include/sys/vm.h	Contains a sequence of include statements that includes all of the virtual memory-related files. Including this file is a quick way of including all of the virtual memory-related files.
/usr/sys/include/sys/vmmac.h	Definitions for converting from bytes to pages or from pages to bytes.
/usr/sys/include/ufs/inode.h	Defines values associated with the generic file system.

Table A-2 lists the header files used by SCSI/CAM peripheral device drivers, along with a short description of the contents of each. For convenience, the full path name for the file is given and the files are listed in alphabetical order.

Table A-2: Header Files Used by SCSI/CAM Peripheral Drivers

Header File	Contents
/usr/sys/include/io/cam/cam.h	Definitions and data structures for the CAM subsystem interface.
/usr/sys/include/io/cam/cam_logger.h	Definitions and data structures for CAM subsystem error logging.
/usr/sys/include/io/cam/cam_special.h	Definitions for the SCSI/CAM special I/O interface.
/usr/sys/include/io/cam/dec_cam.h	Digital-specific definitions and data structures for the CAM routines.
/usr/sys/include/io/cam/pdrv.h	Definitions and data structures for the SCSI/CAM common routines.
/usr/sys/include/io/cam/scsi_special.h	Definitions and data structures for the SCSI/CAM special I/O control interface.
/usr/sys/include/io/cam/uagt.h	Definitions and data structures for the User Agent Device Driver (UAGT) that controls access to the CAM subsystem from a user process.
/usr/sys/include/io/cam/xpt.h	Definitions and data structures for the Transport Layer, XPT, in the CAM subsystem.
/usr/sys/include/io/cam/cam_config.h	SCSI/CAM peripheral device driver configuration definitions.
/usr/sys/include/io/cam/cam_debug.h	CAM debugging macros.
/usr/sys/include/io/cam/cam_disk.h	

Table A-2: (continued)

Header File	Contents
	Definitions and data structures for SCSI/CAM disk devices.
/usr/sys/include/io/cam/cam_errlog.h	CAM error logging macros.
/usr/sys/include/io/cam/cam_tape.h	Definitions and data structures for SCSI/CAM tape devices.
/usr/sys/include/io/cam/ccfg.h	Definitions and data structures for the Configuration driver module in the CAM subsystem.
/usr/sys/include/io/cam/rzdisk.h	Definitions and data structures for SCSI disks.
/usr/sys/include/io/cam/scsi_all.h	Definitions and data structures that apply to all SCSI device types according to Chapter 7 of the SCSI-2 specification.
/usr/sys/include/io/cam/scsi_cdbs.h	Definitions and data structures that apply to Command Descriptor Blocks.
/usr/sys/include/io/cam/scsi_direct.h	Definitions and data structures that apply to all SCSI direct-access devices according to Chapter 8 of the SCSI-2 specification.
/usr/sys/include/io/cam/scsi_opcodes.h	Definitions of operation codes according to Chapter 6 of the SCSI-2 specification.
/usr/sys/include/io/cam/scsi_phases.h	Definitions of SCSI bus phases according to Chapter 5 of the SCSI-2 specification.
/usr/sys/include/io/cam/scsi_rodirect.h	

Table A-2: (continued)

Header File	Contents
	Definitions and data structures that apply to read-only direct-access devices according to Chapter 13 of the SCSI 2 specification.
/usr/sys/include/io/cam/scsi_sequential.h	Definitions and data structures that apply to all SCSI sequential-access devices according to Chapter 9 of the SCSI-2 specification.
/usr/sys/include/io/cam/scsi_status.h	Definitions and data structures that apply to SCSI commands and status according to Chapter 6 of the SCSI 2 specification.

The contents of /usr/sys/include/io/cam/cam.h follow:

```
/* ---------------------------------------------------------------------- */

/* cam.h         Version 1.09                  Jul. 18, 1991 */

/*  This file contains the definitions and data structures for the CAM
    Subsystem interface.  The contents of this file should match what
    data structures and constants that are specified in the CAM document,
    X3T9.2/90-186 Rev 2.5 that is produced by the SCSI-2 committee.


/* ---------------------------------------------------------------------- */

/* Defines for the XPT function codes, Table 8-2 in the CAM spec. */

/* Common function commands, 0x00 - 0x0F */
#define XPT_NOOP 0x00  /* Execute Nothing */
#define XPT_SCSI_IO    0x01  /* Execute the requested SCSI IO */
#define XPT_GDEV_TYPE  0x02  /* Get the device type information */
#define XPT_PATH_INQ   0x03  /* Path Inquiry */
#define XPT_REL_SIMQ   0x04  /* Release the SIM queue that is frozen */
#define XPT_SASYNC_CB  0x05  /* Set Async callback parameters */
#define XPT_SDEV_TYPE  0x06  /* Set the device type information */

/* XPT SCSI control functions, 0x10 - 0x1F */
#define XPT_ABORT0x10  /* Abort the selected CCB */
#define XPT_RESET_BUS  0x11  /* Reset the SCSI bus */
#define XPT_RESET_DEV  0x12  /* Reset the SCSI device, BDR */
#define XPT_TERM_IO    0x13  /* Terminate the I/O process */
```

```
/* HBA engine commands, 0x20 - 0x2F */
#define XPT_ENG_INQ    0x20  /* HBA engine inquiry */
#define XPT_ENG_EXEC   0x21  /* HBA execute engine request */

/* Target mode commands, 0x30 - 0x3F */
#define XPT_EN_LUN     0x30  /* Enable LUN, Target mode support */
#define XPT_TARGET_IO  0x31  /* Execute the target IO request */

#define XPT_FUNC 0x7F  /* TEMPLATE */
#define XPT_VUNIQUE    0x80  /* All the rest are vendor unique commands */

/* ---------------------------------------------------------------------- */

/* General allocation length defines for the CCB structures. */

#define IOCDBLEN 12    /* Space for the CDB bytes/pointer */
#define VUHBA          14    /* Vendor Unique HBA length */
#define SIM_ID         16    /* ASCII string len for SIM ID */
#define HBA_ID         16    /* ASCII string len for HBA ID */
#define SIM_PRIV 50    /* Length of SIM private data area */

/* Structure definitions for the CAM control blocks, CCB's for the
subsystem. */

/* Common CCB header definition. */
typedef struct ccb_header
{
    struct ccb_header *my_addr;    /* The address of this CCB */
    u_short cam_ccb_len;     /* Length of the entire CCB */
    u_char cam_func_code;    /* XPT function code */
    u_char cam_status;       /* Returned CAM subsystem status */
    u_char cam_hrsvd0;       /* Reserved field, for alignment */
    u_char cam_path_id;          /* Path ID for the request */
    u_char cam_target_id;    /* Target device ID */
    u_char cam_target_lun;   /* Target LUN number */
    U32 cam_flags;           /* Flags for operation of the subsystem */
} CCB_HEADER;

/* Common SCSI functions. */

/* Union definition for the CDB space in the SCSI I/O request CCB */
typedef union cdb_un
{
    u_char *cam_cdb_ptr;           /* Pointer to the CDB bytes to send */
    u_char cam_cdb_bytes[ IOCDBLEN ];    /* Area for the CDB to send */
} CDB_UN;

/* Get device type CCB */
typedef struct ccb_getdev
{
    CCB_HEADER cam_ch;             /* Header information fields */
    char *cam_inq_data;            /* Ptr to the inquiry data space */
    u_char cam_pd_type;            /* Periph device type from the TLUN */
} CCB_GETDEV;

/* Path inquiry CCB */
typedef struct ccb_pathinq
{
    CCB_HEADER cam_ch;             /* Header information fields */
    u_char cam_version_num;        /* Version number for the SIM/HBA */
```

```
    u_char cam_hba_inquiry;         /* Mimic of INQ byte 7 for the HBA */
    u_char cam_target_sprt;         /* Flags for target mode support */
    u_char cam_hba_misc;            /* Misc HBA feature flags */
    u_short cam_hba_eng_cnt;        /* HBA engine count */
    u_char cam_vuhba_flags[ VUHBA ];      /* Vendor unique capabilities */
    U32 cam_sim_priv;         /* Size of SIM private data area */
    U32 cam_async_flags;            /* Event cap. for Async Callback */
    u_char cam_hpath_id;            /* Highest path ID in the subsystem */
    u_char cam_initiator_id;        /* ID of the HBA on the SCSI bus */
    u_char cam_prsvd0;              /* Reserved field, for alignment */
    u_char cam_prsvd1;              /* Reserved field, for alignment */
    char cam_sim_vid[ SIM_ID ];           /* Vendor ID of the SIM */
    char cam_hba_vid[ HBA_ID ];           /* Vendor ID of the HBA */
    u_char *cam_osd_usage;          /* Ptr for the OSD specific area */
} CCB_PATHINQ;

/* Release SIM Queue CCB */
typedef struct ccb_relsim
{
    CCB_HEADER cam_ch;              /* Header information fields */
} CCB_RELSIM;

/* SCSI I/O Request CCB */
typedef struct ccb_scsiio
{
    CCB_HEADER cam_ch;              /* Header information fields */
    u_char *cam_pdrv_ptr;           /* Ptr used by the Peripheral driver */
    CCB_HEADER *cam_next_ccb;       /* Ptr to the next CCB for action */
    u_char *cam_req_map;            /* Ptr for mapping info on the Req. */
    void (*cam_cbfcnp)();           /* Callback on completion function */
    u_char *cam_data_ptr;           /* Pointer to the data buf/SG list */
    U32 cam_dxfer_len;              /* Data xfer length */
    u_char *cam_sense_ptr;          /* Pointer to the sense data buffer */
    u_char cam_sense_len;           /* Num of bytes in the Autosense buf */
    u_char cam_cdb_len;                   /* Number of bytes for the CDB */
    u_short cam_sglist_cnt;         /* Num of scatter gather list entries */
    U32 cam_sort;                   /* Value used by SIM to sort on */
    u_char cam_scsi_status;         /* Returned scsi device status */
    u_char cam_sense_resid;         /* Autosense resid length: 2's comp */
    u_char cam_osd_rsvd1[2];        /* OSD Reserved field, for alignment */
    I32 cam_resid;                  /* Transfer residual length: 2's comp */
    CDB_UN cam_cdb_io;              /* Union for CDB bytes/pointer */
    U32 cam_timeout;                /* Timeout value */
    u_char *cam_msg_ptr;            /* Pointer to the message buffer */
    u_short cam_msgb_len;           /* Num of bytes in the message buf */
    u_short cam_vu_flags;           /* Vendor unique flags */
    u_char cam_tag_action;          /* What to do for tag queuing */
    u_char cam_iorsvd0[3];          /* Reserved field, for alignment */
    u_char cam_sim_priv[ SIM_PRIV ];      /* SIM private data area */
} CCB_SCSIIO;

/* Set Async Callback CCB */
typedef struct ccb_setasync
{
    CCB_HEADER cam_ch;              /* Header information fields */
    U32 cam_async_flags;            /* Event enables for Callback resp */
    void (*cam_async_func)();       /* Async Callback function address */
    u_char *pdrv_buf;               /* Buffer set aside by the Per. drv */
    u_char pdrv_buf_len;            /* The size of the buffer */
} CCB_SETASYNC;
```

```
/* Set device type CCB */
typedef struct ccb_setdev
{
    CCB_HEADER cam_ch;              /* Header information fields */
    u_char cam_dev_type;            /* Val for the dev type field in EDT */
} CCB_SETDEV;

/* SCSI Control Functions. */

/* Abort XPT Request CCB */
typedef struct ccb_abort
{
    CCB_HEADER cam_ch;              /* Header information fields */
    CCB_HEADER *cam_abort_ch;           /* Pointer to the CCB to abort */
} CCB_ABORT;

/* Reset SCSI Bus CCB */
typedef struct ccb_resetbus
{
    CCB_HEADER cam_ch;              /* Header information fields */
} CCB_RESETBUS;

/* Reset SCSI Device CCB */
typedef struct ccb_resetdev
{
    CCB_HEADER cam_ch;              /* Header information fields */
} CCB_RESETDEV;

/* Terminate I/O Process Request CCB */
typedef struct ccb_termio
{
    CCB_HEADER cam_ch;              /* Header information fields */
    CCB_HEADER *cam_termio_ch;      /* Pointer to the CCB to terminate */
} CCB_TERMIO;

/* Target mode structures. */

typedef struct ccb_en_lun
{
    CCB_HEADER cam_ch;              /* Header information fields */
    u_short cam_grp6_len;           /* Group 6 VU CDB length */
    u_short cam_grp7_len;           /* Group 7 VU CDB length */
    u_char *cam_ccb_listptr;        /* Pointer to the target CCB list */
    u_short cam_ccb_listcnt;        /* Count of Target CCBs in the list */
} CCB_EN_LUN;

/* HBA engine structures. */

typedef struct ccb_eng_inq
{
    CCB_HEADER cam_ch;              /* Header information fields */
    u_short cam_eng_num;            /* The number for this inquiry */
    u_char cam_eng_type;            /* Returned engine type */
    u_char cam_eng_algo;            /* Returned algorithm type */
    U32 cam_eng_memory;             /* Returned engine memory size */
} CCB_ENG_INQ;

typedef struct ccb_eng_exec  /* NOTE: must match SCSIIO size */
{
    CCB_HEADER cam_ch;                 /* Header information fields */
```

```
    u_char *cam_pdrv_ptr;           /* Ptr used by the Peripheral driver */
    U32 cam_engrsvd0;               /* Reserved field, for alignment */
    u_char *cam_req_map;            /* Ptr for mapping info on the Req. */
    void (*cam_cbfcnp)();           /* Callback on completion function */
    u_char *cam_data_ptr;           /* Pointer to the data buf/SG list */
    U32 cam_dxfer_len;              /* Data xfer length */
    u_char *cam_engdata_ptr;        /* Pointer to the engine buffer data */
    u_char cam_engrsvd1;            /* Reserved field, for alignment */
    u_char cam_engrsvd2;            /* Reserved field, for alignment */
    u_short cam_sglist_cnt;         /* Num of scatter gather list entries */
    U32 cam_dmax_len;               /* Destination data maximum length */
    U32 cam_dest_len;               /* Destination data length */
    I32 cam_src_resid;              /* Source residual length: 2's comp */
    u_char cam_engrsvd3[12];        /* Reserved field, for alignment */
    U32 cam_timeout;                /* Timeout value */
    U32 cam_engrsvd4;               /* Reserved field, for alignment */
    u_short cam_eng_num;            /* Engine number for this request */
    u_short cam_vu_flags;           /* Vendor unique flags */
    u_char cam_engrsvd5;            /* Reserved field, for alignment */
    u_char cam_engrsvd6[3];         /* Reserved field, for alignment */
    u_char cam_sim_priv[ SIM_PRIV ];/* SIM private data area */
} CCB_ENG_EXEC;

/* The CAM_SIM_ENTRY definition is used to define the entry points for
the SIMs contained in the SCSI CAM subsystem.  Each SIM file will
contain a declaration for it's entry.  The address for this entry will
be stored in the cam_conftbl[] array along will all the other SIM
entries. */

typedef struct cam_sim_entry
{
    I32 (*sim_init)();        /* Pointer to the SIM init routine */
    I32 (*sim_action)();      /* Pointer to the SIM CCB go routine */
} CAM_SIM_ENTRY;

/* ---------------------------------------------------------------------- */

/* Defines for the CAM status field in the CCB header. */

#define CAM_REQ_INPROG      0x00   /* CCB request is in progress */
#define CAM_REQ_CMP         0x01   /* CCB request completed w/out error */
#define CAM_REQ_ABORTED     0x02   /* CCB request aborted by the host */
#define CAM_UA_ABORT        0x03   /* Unable to Abort CCB request */
#define CAM_REQ_CMP_ERR     0x04   /* CCB request completed with an err */
#define CAM_BUSY            0x05   /* CAM subsystem is busy */
#define CAM_REQ_INVALID     0x06   /* CCB request is invalid */
#define CAM_PATH_INVALID    0x07   /* Path ID supplied is invalid */
#define CAM_DEV_NOT_THERE   0x08   /* SCSI device not installed/there */
#define CAM_UA_TERMIO       0x09   /* Unable to Terminate I/O CCB req */
#define CAM_SEL_TIMEOUT     0x0A   /* Target selection timeout */
#define CAM_CMD_TIMEOUT     0x0B   /* Command timeout */
#define CAM_MSG_REJECT_REC  0x0D   /* Message reject received */
#define CAM_SCSI_BUS_RESET  0x0E   /* SCSI bus reset sent/received */
#define CAM_UNCOR_PARITY    0x0F   /* Uncorrectable parity err occurred */
#define CAM_AUTOSENSE_FAIL  0x10   /* Autosense: Request sense cmd fail */
#define CAM_NO_HBA          0x11   /* No HBA detected Error */
#define CAM_DATA_RUN_ERR    0x12   /* Data overrun/underrun error */
#define CAM_UNEXP_BUSFREE   0x13   /* Unexpected BUS free */
#define CAM_SEQUENCE_FAIL   0x14   /* Target bus phase sequence failure */
#define CAM_CCB_LEN_ERR     0x15   /* CCB length supplied is inadequate */
```

```
#define CAM_PROVIDE_FAIL    0x16    /* Unable to provide requ. capability */
#define CAM_BDR_SENT        0x17    /* A SCSI BDR msg was sent to target */
#define CAM_REQ_TERMIO      0x18    /* CCB request terminated by the host */

#define CAM_LUN_INVALID     0x38    /* LUN supplied is invalid */
#define CAM_TID_INVALID     0x39    /* Target ID supplied is invalid */
#define CAM_FUNC_NOTAVAIL   0x3A    /* The requ. func is not available */
#define CAM_NO_NEXUS        0x3B    /* Nexus is not established */
#define CAM_IID_INVALID     0x3C    /* The initiator ID is invalid */
#define CAM_CDB_RECVD       0x3E    /* The SCSI CDB has been received */
#define CAM_SCSI_BUSY       0x3F    /* SCSI bus busy */

#define CAM_SIM_QFRZN       0x40    /* The SIM queue is frozen w/this err */
#define CAM_AUTOSNS_VALID   0x80    /* Autosense data valid for target */

#define CAM_STATUS_MASK     0x3F    /* Mask bits for just the status # */

/* ---------------------------------------------------------------------- */

/* Defines for the CAM flags field in the CCB header. */

#define CAM_DIR_RESV      0x00000000  /* Data direction (00: reserved) */
#define CAM_DIR_IN        0x00000040  /* Data direction (01: DATA IN) */
#define CAM_DIR_OUT       0x00000080  /* Data direction (10: DATA OUT) */
#define CAM_DIR_NONE      0x000000C0  /* Data direction (11: no data) */
#define CAM_DIS_AUTOSENSE 0x00000020  /* Disable autosense feature */
#define CAM_SCATTER_VALID 0x00000010  /* Scatter/gather list is valid */
#define CAM_DIS_CALLBACK  0x00000008  /* Disable callback feature */
#define CAM_CDB_LINKED    0x00000004  /* The CCB contains a linked CDB */
#define CAM_QUEUE_ENABLE  0x00000002  /* SIM queue actions are enabled */
#define CAM_CDB_POINTER   0x00000001  /* The CDB field contains a pointer */

#define CAM_DIS_DISCONNECT 0x00008000 /* Disable disconnect */
#define CAM_INITIATE_SYNC 0x00004000  /* Attempt Sync data xfer, and SDTR */
#define CAM_DIS_SYNC      0x00002000  /* Disable sync, go to async */
#define CAM_SIM_QHEAD     0x00001000  /* Place CCB at the head of SIM Q */
#define CAM_SIM_QFREEZE   0x00000800  /* Return the SIM Q to frozen state */
#define CAM_SIM_QFRZDIS   0x00000400  /* Disable the SIM Q frozen state */
#define CAM_ENG_SYNC      0x00000200  /* Flush resid bytes before cmplt */

#define CAM_ENG_SGLIST    0x00800000  /* The SG list is for the HBA engine */
#define CAM_CDB_PHYS      0x00400000  /* CDB pointer is physical */
#define CAM_DATA_PHYS     0x00200000  /* SG/Buffer data ptrs are physical */
#define CAM_SNS_BUF_PHYS  0x00100000  /* Autosense data ptr is physical */
#define CAM_MSG_BUF_PHYS  0x00080000  /* Message buffer ptr is physical */
#define CAM_NXT_CCB_PHYS  0x00040000  /* Next CCB pointer is physical */
#define CAM_CALLBCK_PHYS  0x00020000  /* Callback func ptr is physical */

#define CAM_DATAB_VALID   0x80000000  /* Data buffer valid */
#define CAM_STATUS_VALID  0x40000000  /* Status buffer valid */
#define CAM_MSGB_VALID    0x20000000  /* Message buffer valid */
#define CAM_TGT_PHASE_MOD 0x08000000  /* The SIM will run in phase mode */
#define CAM_TGT_CCB_AVAIL 0x04000000  /* Target CCB available */
#define CAM_DIS_AUTODISC  0x02000000  /* Disable autodisconnect */
#define CAM_DIS_AUTOSRP   0x01000000  /* Disable autosave/restore ptrs */

/* ------------------------------------------------------------------ */

/* Defines for the SIM/HBA queue actions.  These value are used in the
SCSI I/O CCB, for the queue action field. [These values should match
```

```
the defines from some other include file for the SCSI message phases.
We may not need these definitions here. ] */

#define CAM_SIMPLE_QTAG           0x20     /* Tag for a simple queue */
#define CAM_HEAD_QTAG        0x21     /* Tag for head of queue */
#define CAM_ORDERED_QTAG     0x22     /* Tag for ordered queue */

/* ----------------------------------------------------------------- */

/* Defines for the timeout field in the SCSI I/O CCB.  At this time a
value of 0xF-F indicates a infinite timeout.  A value of 0x0-0
indicates that the SIM's default timeout can take effect. */

#define CAM_TIME_DEFAULT    0x00000000   /* Use SIM default value */
#define CAM_TIME_INFINITY   0xFFFFFFFF   /* Infinite timeout for I/O */

/* ----------------------------------------------------------------- */

/* Defines for the Path Inquiry CCB fields. */

#define CAM_VERSION         0x25   /* Binary value for the current ver */

#define PI_MDP_ABLE         0x80   /* Supports MDP message */
#define PI_WIDE_3           0x40   /* Supports 32 bit wide SCSI */
#define PI_WIDE_1           0x20   /* Supports 16 bit wide SCSI */
#define PI_SDTR_ABLE        0x10   /* Supports SDTR message */
#define PI_LINKED_CDB       0x08   /* Supports linked CDBs */
#define PI_TAG_ABLE         0x02   /* Supports tag queue message */
#define PI_SOFT_RST         0x01   /* Supports soft reset */

#define PIT_PROCESSOR       0x80   /* Target mode processor mode */
#define PIT_PHASE           0x40   /* Target mode phase cog. mode */

#define PIM_SCANHILO        0x80   /* Bus scans from ID 7 to ID 0 */
#define PIM_NOREMOVE        0x40   /* Removable dev not included in scan */
#define PIM_NOINQUIRY       0x20   /* Inquiry data not kept by XPT */

/* ------------------------------------------------------------------ */

/* Defines for Asynchronous Callback CCB fields. */

#define AC_FOUND_DEVICES    0x80   /* During a rescan new device found */
#define AC_SIM_DEREGISTER   0x40   /* A loaded SIM has de-registered */
#define AC_SIM_REGISTER     0x20   /* A loaded SIM has registered */
#define AC_SENT_BDR         0x10   /* A BDR message was sent to target */
#define AC_SCSI_AEN         0x08   /* A SCSI AEN has been received */
#define AC_UNSOL_RESEL      0x02   /* A unsolicited reselection occurred */
#define AC_BUS_RESET        0x01   /* A SCSI bus RESET occurred */

/* ----------------------------------------------------------------- */

/* Typedef for a scatter/gather list element. */

typedef struct sg_elem
{
    u_char *cam_sg_address;  /* Scatter/Gather address */
    U32 cam_sg_count;        /* Scatter/Gather count */
} SG_ELEM;

/* ----------------------------------------------------------------- */
```

```
/* Defines for the HBA engine inquiry CCB fields. */

#define EIT_BUFFER          0x00   /* Engine type: Buffer memory */
#define EIT_LOSSLESS        0x01   /* Engine type: Lossless compression */
#define EIT_LOSSLY          0x02   /* Engine type: Lossly compression */
#define EIT_ENCRYPT         0x03   /* Engine type: Encryption */

#define EAD_VUNIQUE         0x00   /* Eng algorithm ID: vendor unique */
#define EAD_LZ1V1           0x00   /* Eng algorithm ID: LZ1 var. 1*/
#define EAD_LZ2V1           0x00   /* Eng algorithm ID: LZ2 var. 1*/
#define EAD_LZ2V2           0x00   /* Eng algorithm ID: LZ2 var. 2*/

/* ---------------------------------------------------------------- */
/* ---------------------------------------------------------------- */

/* UNIX OSD defines and data structures. */

#define INQLEN   36          /* Inquiry string length to store. */

#define CAM_SUCCESS    0     /* For signaling general success */
#define CAM_FAILURE    1     /* For signaling general failure */

#define CAM_FALSE 0     /* General purpose flag value */
#define CAM_TRUE 1      /* General purpose flag value */

#define XPT_CCB_INVALID     -1    /* for signaling a bad CCB to free */

/* General Union for Kernel Space allocation.  Contains all the possible
CCB structures.  This union should never be used for manipulating CCB's
its only use is for the allocation and deallocation of raw CCB space. */

typedef union ccb_size_union
{
    CCB_SCSIIO    csio;    /* Please keep this first, for debug/print */
    CCB_GETDEV    cgd;
    CCB_PATHINQ   cpi;
    CCB_RELSIM    crs;
    CCB_SETASYNC  csa;
    CCB_SETDEV    csd;
    CCB_ABORT     cab;
    CCB_RESETBUS  crb;
    CCB_RESETDEV  crd;
    CCB_TERMIO    ctio;
    CCB_EN_LUN    cel;
    CCB_ENG_INQ   cei;
    CCB_ENG_EXEC  cee;
} CCB_SIZE_UNION;

/* The typedef for the Async callback information.  This structure is
used to store the supplied info from the Set Async Callback CCB, in the
EDT table in a linked list structure. */

typedef struct async_info
{
  struct async_info *cam_async_next; /* pointer to the next structure */
  U32 cam_event_enable;             /* Event enables for Callback resp */
  void (*cam_async_func)();         /* Async Callback function address */
  U32 cam_async_blen;               /* Length of "information" buffer */
  u_char *cam_async_ptr;            /* Address for the "information */
} ASYNC_INFO;
```

```
/* The CAM EDT table contains the device information for all the
devices, SCSI ID and LUN, for all the SCSI busses in the system.  The
table contains a CAM_EDT_ENTRY structure for each device on the bus.
*/

typedef struct cam_edt_entry
{
    I32 cam_tlun_found;    /* Flag for the existence of the target/LUN */
    ASYNC_INFO *cam_ainfo; /* Async callback list info for this B/T/L */
    U32 cam_owner_tag;    /* Tag for the peripheral driver's ownership */
    char cam_inq_data[ INQLEN ];/* storage for the inquiry data */
} CAM_EDT_ENTRY;

/* ---------------------------------------------------------------- */

#endif /* _CAM_INCL_ */
```

SCSI/CAM Utility Program B

B.1 Introduction

The SCSI/CAM Utility Program, SCU, interfaces with the Common Access Method (CAM) I/O subsystem and the peripheral devices attached to Small Computer System Interface (SCSI) busses. This utility implements the SCSI commands necessary for normal maintenance and diagnostics of SCSI peripheral devices and the CAM I/O subsystem.

The format of a SCU command is as follows:

scu> [**-f** *device-name-path*] [*command* [*keyword* ...]]

If the `device-name-path` is not specified on the command line, the program checks the environment variable SCU_DEVICE to determine the device name. If SCU_DEVICE is not set, the `set nexus` command must be used to select the device and operation of some commands may be restricted. For example, if you do not specify a `device-name-path` and SCU_DEVICE is not set, you cannot format a disk because the `scu` utility cannot perform a mounted file system check. See Section B.2.4 for a description of the `set` command and its arguments.

If a command is not entered on the command line, the program prompts for commands until you terminate the program. In most cases, you can abbreviate commands to the lowest un-ambiguous number of characters.

This appendix contains an overview of the `scu` functions that device driver writers use. Detailed information is available through the online help for the `scu` utility. Once you are in the `scu` utility, issue the `help` command at the `scu>` prompt.

B.1.1 SCU Utility Conventions

The following conventions are used in describing `scu` utility syntax:

Convention	Meaning
keyword (alias)	Use a keyword or the specified alias.
address-format	Optionally accepts an address format.

Convention	Meaning
nexus-information	Optionally accepts nexus information.
test-parameters	Optionally accepts test parameters.
D: *value* or *string*	The value or string shown is the default.
R:*minimum–maximum*	Enter a value within the range specified.

The *address-format* parameter is optional. It is available for use with most CDROM Show Audio commands that specify the address format of information returned by the drive. The possible address formats are:

Format	Description
lba	Logical Block Address.
msf	Minute, Second, and Frame.

The syntax of a command using the *address-format* parameter follows:

scu> *command* [*address-format* **{ lba | msf }**]

The *nexus-information* parameter lets users specify values to override the bus, target, and LUN values normally taken from the selected SCSI device. The *nexus-information* keywords are:

Parameter	Description
bus (pid) R:0–3	SCSI bus number (path ID)
target (tid) R:0–7	SCSI target number (target ID)
lun R:0–7	SCSI Logical Unit Number (LUN)

The *test-parameter* variables are used to specify the physical limits of the media on which the command can operate. For example, these may be the starting and ending logical block numbers on a disk. The test parameters for a command use the following syntax:

scu> *command* [*media-limits*] [test-control]

The *media-limits* parameter, which controls the media tested, has the following syntax:

```
               { lba n }    { length n }
scu> command [ { starting n } ] [ { ending n } ] [ size n ]
               { limit   n }
               { records n }
```

The alias bs (block size) is accepted for the *size* keyword.

The *test-control* parameters control aspects of the test operation. The *test-control* parameters supported are listed below:

```
               { align Align-Offset }
               { compare { on | off }  }
scu> command [ { errors Error-Limit  } ]
               { passes Pass-Limit    }
               { pattern Data-Pattern  }
               { recovery { on | off } }
```

B.2 General SCU Commands

This section describes scu utility commands that are used for general purposes. The commands are:

- evaluate
- exit
- help
- scan
- set
- show
- source
- switch

B.2.1 The evaluate Command

The evaluate command evaulates the given expression and displays values in decimal, hexadecimal, blocks, kilobytes, megabytes, and gigabytes. The expression argument is the same as that described for test paramater values. The output depends on whether the verbose display flag is set. The format of the evaluate command is as follows:

scu> evaluate *expression*

The following example sets verbose mode for the first two evaluate commands and then turns off verbose mode for the last one.

```
scu> set verbose on
scu> evaluate 0xffff
Expression Values:

            Decimal: 65535
        Hexadecimal: 0xffff
    512 byte Blocks: 128.00
          Kilobytes: 64.00
          Megabytes: 0.06
          Gigabytes: 0.00

scu> evaluate 64k*512
Expression Values:

            Decimal: 33554432
        Hexadecimal: 0x2000000
    512 byte Blocks: 65536.00
          Kilobytes: 32768.00
          Megabytes: 32.00
          Gigabytes: 0.03

scu> set verbose off
scu> evaluate 0xffff
Dec: 65525 Hex: 0xffff Blks: 128.00 Kb: 64.00 Mb: 0.06 Gb: 0.00
```

B.2.2 The exit Command

The `exit` command is used to exit the program. You can use `quit` as an alias for `exit`. You can terminate the program in interactive mode by entering the end-of-file character (usually CTRL/D). The format of the `exit` command is as follows:

scu> exit

B.2.3 The help Command

The `help` command displays help information on topics. You can use a question mark (?) as an alias. If you issue the `help` command without specifying a topic, a list of all available topics is displayed. The format of the `help` command is as follows:

scu> help [*topic*]

B.2.4 The scan Command

The `scan` command scans either device media or the CAM Equipment Device Table (EDT). The format of the `scan` command is as follows:

scu> scan edt [[*nexus-information*] [*report-format*]]

scu> scan media [*test-parameters*]

The `edt` argument allows scanning of the SCSI bus which results in the CAM Equipment Device Table (EDT) being updated to reflect the devices found. If nexus information is omitted, the selected device is scanned.

The format of the command using the `edt` argument is as follows:

scu> scan edt [*nexus-information*]

Section B.1.1 contains a list of the valid test parameters.

The following examples use the `scan edt` command. The first example illustrates the command followed by the `show device` command to display the information resulting from the scan:

```
scu> scan edt
Scanning bus 1, target 6, lun 0, please be patient...

scu> show device
Inquiry Information:
                     SCSI Bus ID: 1
                  SCSI Target ID: 6
                 SCSI Target LUN: 0
          Peripheral Device Type: Direct Access
            Peripheral Qualifier: Peripheral Device Connected
           Device Type Qualifier: 0
                 Removable Media: No
                    ANSI Version: SCSI-1 Compliant
                    ECMA Version: 0
                     ISO Version: 0
            Response Data Format: CCS
               Additional Length: 31
           Vendor Identification: DEC
          Product Identification: RZ55     (C) DEC
         Firmware Revision Level: 0700

scu> scan edt bus 1
Scanning bus 1, target 6, lun 0, please be patient...
```

The `media` argument causes the device media to be scanned. This involves writing a data pattern to the media and then reading and verifying the data written. You must include test parameters that specify the media area to be scanned.

The format of the command using the `media` argument is as follows:

scu> scan media [*test-parameters*]

The following examples use the `scan media` command with different *test-parameters*:

```
scu> scan media
scu: No defaults, please specify test parameters for transfer...

scu> scan media length 100 recovery off
Scanning 100 blocks on /dev/rrzl0c (RX23) with pattern
                                                    0x39c39c39...

scu> scan media lba 200 limit 25k align '1p-1'
Scanning 50 blocks on /dev/rrzl0c (RX23) with pattern
                                                    0x39c39c39...

scu> scan media starting 0 bs 32k records 10
Scanning 640 blocks on /dev/rrzl0c (RX23) with pattern
                                                    0x39c39c39...
Scanning blocks [ 0 through 63 ]...
Scanning blocks [ 64 through 127 ]...
Scanning blocks [ 128 through 191 ]...
Scanning blocks [ 192 through 255 ]...
Scanning blocks [ 256 through 319 ]...
Scanning blocks [ 320 through 383 ]...
Scanning blocks [ 384 through 447 ]...
Scanning blocks [ 448 through 511 ]...
Scanning blocks [ 512 through 575 ]...
Scanning blocks [ 576 through 639 ]...
```

B.2.5 The set Command

The `set` command sets parameters for a device or sets environment parameters for the `scu` program. The format of the `set` command is as follows:

```
         { audio keywords ... }
         { cam debug hex-flags }
         { debug { on | off } }
         { default parameter }
scu> set { device device-type }
         { dump { on | off } }
         { dump-limit value }
         { log file-name-path }
         { nexus nexus-information }
         { pages [ mode-page [ pcf page-control-field ] }
         { pager paging-filter }
         { paging { on | off } }
         { recovery { on | off } }
         { verbose { on | off } }
         { watch { on | off } }
```

The audio keyword sets parameters for a CDROM audio device. The format of the command using the audio keyword is as follows:

```
                { address format { lba | msf }         }
scu> set audio  { volume [ channel-{ 0 | 1 } ] level n }
```

The address format parameter sets the default address format associated with CDROM audio commands. You must have write access to the device to issue this command because it modifies the device parameters. The possible address formats are:

Format	Description
lba	Logical Block Address
msf	Minute, Second, and Frame

The format of the command using the address format parameter is as follows:

```
scu> set audio address format { lba | msf }
```

The volume parameter sets the audio volume control levels. You can change either the right or left channel individually, or both channels at the same time.

The format of the command using the volume parameter is as follows:

```
scu> set audio volume [ channel-{ 0 | 1 } ] level n
```

You can use the aliases ch0 for channel-0 and ch1 for channel-1.

The cam argument lets you set parameters associated with the CAM subsystem. The format of the command using the cam argument is as follows:

```
scu>  set cam { debug debug-flags }
              { flags ccb-flags   }
```

The cam debug parameter lets you set the CAM debug flags that the user-level SCSI/CAM Special I/O interface functions use. The debug flags that you can specify are:

Debug Flag	Hex Value	Description
CAMD_INOUT	0x00000001	Routine entry and exit
CAMD_FLOW	0x00000002	Code flow through the modules
CAMD_ERRORS	0x00000010	Error handling

Debug Flag	Hex Value	Description
CAMD_CMD_EXP	0x00000020	Expansion of commands and responses

The format of the command using the `cam debug` parameter is as follows:

scu> set cam debug *hex-flags*

For example:

scu> **set cam debug 0xffff**

The `flags` parameter lets you specify CAM flags to be set in CCBs sent to the CAM subsystem. The flags that you can specify are:

CAM CCB Flag	Hex Value	Description
CAM_DIS_DISCONNECT	0x00008000	Disable disconnect.
CAM_INITIATE_SYNC	0x00004000	Attempt synchronous data transfer.
CAM_DIS_SYNC	0x00002000	Disable synchronous data transfer.
CAM_SIM_QHEAD	0x00001000	Place the CCB at the head of the SIM queue.
CAM_SIM_QFREEZE	0x00000800	Return the SIM queue to the frozen state.
CAM_SIM_QFRZDIS	0x00000400	Disable the SIM queue, that is, freeze on errors.
CAM_ENG_SYNC	0x00000200	Flush residual bytes before completion.

The default CAM CCB flag used by the `scu` program is CAM_SIM_QFRZDIS.

The format of the command using the `cam flags` parameter is as follows:

scu> set cam flags *hex-flags*

For example:

scu> **set cam flags 0x4000|0x400**

The `debug` argument enables or disables the program debugging flag. When the flag is enabled, the program displays additional debugging information during command processing. By default, debugging output is disabled.

The format of the command using the `flags` parameter is as follows:

```
scu> set debug { on | off }
```

The `default` argument lets you change certain program defaults.

The format of the command using the `default` argument is as follows:

scu> set default { savable | *test-parameters* }

The `savable` parameter lets you specify whether or not the mode page parameters are saved. By default, if the mode page is savable, the mode page parameters set by the `set page` or `change page` command are saved.

The format of the command using the `savable` parameter is as follows:

```
scu> set default savable { on | off }
```

The `test-parameters` parameter lets you set up the I/O test parameter defaults. The following `test-parameters` can be set:

Parameter	Type	Default	Description
`align`	`value`	`0`	Data buffer alignment offset.
`compare`	`flag`	`On`	Compare data during read operations. The possible values are: 1/0; on/off; or true/false.
`errors`	`value`	`10`	Error limit value.
`passes`	`value`	`1`	Number of passes to perform.
`pattern`	`value`	`0x39c39c39`	Data pattern to use (first pass).
`size (bs)`	`value`	`512`	Block size per I/O request.

The `device` argument issues a CAM Set Device Type CCB to change the device type in the EDT. If the `nexus-information` parameter is omitted, the command is issued to the selected device. This command is restricted to the superuser because the Set Device Type CCB overwrites existing device information in the CAM EDT.

The `set device` command can also be used to set up the device type or to override the existing device type.

The format of the command using the `device` argument is as follows:

scu> set device *device-type* [*nexus-information*]

The device-type argument specifies the device type to which the device is to be changed in the EDT. The valid SCSI device types are:

Device Type	CAM Definition	Value
direct-access		0
sequential-access	ALL_DTYPE_SEQUENTIAL	1
printer	ALL_DTYPE_PRINTER	2
processor	ALL_DTYPE_PROCESSOR	3
worm	ALL_DTYPE_WORM	4
rodirect	ALL_DTYPE_RODIRECT	5
scanner	ALL_DTYPE_SCANNER	6
optical	ALL_DTYPE_OPTICAL	7
changer	ALL_DTYPE_CHANGER	8
communication	ALL_DTYPE_COMM	9

The dump argument enables or disables the dump buffer flag. When the dump buffer flag is enabled, the program dumps the entire data buffer being operated on instead of the length returned from the CAM subsystem. By default, this flag is disabled.

If the dump buffer flag is enabled and the CAM debug flag, CAMD_CMD_EXP, is set during a data-in operation, the entire data buffer, up to the value of the dump-limit parameter, is dumped instead of the number of bytes indicated by the CCB fields.

If the dump buffer flag is enabled when performing diagnostic functions, the entire data buffer, up to the value of the dump-limit parameter, is dumped during data verification failures.

The format of the command using the dump argument is as follows:

```
scu> set dump { on | off }
```

The dump-limit argument limits the number of bytes dumped during debugging. This value is used in conjunction with the dump buffer control flag and it limits the number of data bytes displayed when buffer dumping is enabled or when the CAM debug flag, CAMD_CMD_EXP, is enabled during command execution. The default value is 512 bytes.

The format of the command using the dump-limit argument is as follows:

```
scu> set dump-limit value
```

The log argument opens a log file to capture text displayed by the program. When logging is active, text output is written both to the log file and to the terminal. Both standard output and standard error text is captured in the log file. The text displayed by the help command is not saved in the log file. This command is also used to close an existing log file by specifying a null file name string.

This command provides a simple mechanism to log an interactive session.

The format of the command using the `log` argument is as follows:

scu> set log *file-name-path*

The `paging` keyword controls paging when output is sent to a terminal device. By default, `paging` is enabled when standard output is a terminal device.

The format of the command using the `paging` keyword is as follows:

```
scu> set paging { on | off }
```

The `recovery` argument enables or disables the error-control parameters for the selected device. Ordinarily, the current parameters are used. The parameters are set from either the saved or the default error-control pages when the drive is powered on. The normal default is for error correction to be enabled. Disabling error correction is useful during device testing.

The format of the command using the `recovery` argument is as follows:

```
scu> set recovery { on | off }
```

The following conditions apply to the `set recovery` command:

- When error recovery is disabled, the previous error-control bits are saved and the disable correction (DCR), disable transfer on error (DTE), post recoverable error (PER), and transfer block on error (TB) bits are set, while all other bits are cleared.
- When error recovery is enabled, either the previously saved error-control bits or the error-control bits from the default error page are used.
- Only the current device's error mode-page parameters are affected when error recovery is enabled or disabled.

B.2.6 The show Command

The `show` command is used to display parameters for a device or the program. The *`parameter`* argument can be `audio` keywords, `capacity`, `defects`, `device`, `edt`, `nexus`, `pages`, or `path-inquiry`. The format for the `show` command is as follows:

scu> show *parameter*

B.2.7 The source Command

The `source` command allows you to source input from an external command file. If any errors occur during command parsing or execution, the command file is closed at that point. The format for the `source` command is as follows:

```
scu> source input-file
```

The default file name extension `.scu` is appended to the name of the input file if no extension is supplied. If the `scu` utility cannot find a file with the `.scu` extension, it attempts to locate the original input file.

B.2.8 The switch Command

The `switch` command accesses a new device or a previous device. If no device name is specified, the command acts as a toggle and simply switches to the previous device, if one exists. If a device is specified, it is validated and becomes the active device. The format of the `switch` command is as follows:

```
scu> switch [device-name]
```

B.3 Device and Bus Management Commands

This section describes `scu` utility commands that are used to manage SCSI devices and the CAM I/O subsystem. The commands are:

- `allow`
- `eject`
- `mt`
- `pause`
- `play`
- `prevent`
- `release`
- `reserve`
- `reset`
- `resume`
- `start`
- `stop`
- `tur`
- `verify`

B.3.1 The allow Command

The `allow` command allows media to be removed from the selected device. The format of the `allow` command is as follows:

```
scu> allow
```

B.3.2 The eject Command

The `eject` command is used with CD-ROMs to stop play and eject the caddy. The format of the `eject` command is as follows:

```
scu> eject
```

B.3.3 The mt Commands

The `mt` command issues one of the supported `mt` commands. Only those `mt` commands that do not require additional driver information have been implemented. Unless errors occur, the `mt` commands execute silently. Otherwise, the sense data, if any, returned from the failing command is displayed.

The format of the `mt` command is as follows:

`scu> mt` *command* [*count*]

For commands that accept a *`count`* parameter, if *`count`* is omitted, the default value is 1.

The `mt bsf` command is used to backward space *`count`* file marks. The format of the `mt bsf` command is as follows:

`scu> mt bsf [` *count*]

The `mt bsr` command is used to backward space *`count`* file records. The format of the `mt bsr` command is as follows:

`scu> mt bsr [` *count*]

The `mt erase` command is used to erase the tape. However, some tape drives reject this command unless the tape is positioned at beginning of media.

The syntax of the `mt erase` command is as follows:

```
scu> mt erase
```

The `mt fsf` command is used to forward space *count* file marks. The syntax of the `mt fsf` command is as follows:

```
scu> mt fsf [ count ]
```

The `mt fsr` command is used to forward space *count* file records. The syntax of the `mt fsr` command is as follows:

```
scu> mt fsr [ count ]
```

The `mt load` command is used to load a tape. This command is the same as the `mt online` command, except the immediate bit is enabled so that the command completes after the load is initiated.

The syntax of the `mt load` command is as follows:

```
scu> mt load
```

The `mt offline` command is used to take a tape offline, that is, to perform an unload operation. The syntax of the `mt offline` command is as follows:

```
scu> mt offline
```

You can use the alias `rewoffl` for the `mt offline` command.

The `mt online` command is used to bring a tape online, that is, to perform a load operation. The syntax of the `mt online` command is as follows:

```
scu> mt online
```

The `mt rewind` command rewinds a tape. The syntax of the `mt rewind` command is as follows:

```
scu> mt rewind
```

The `mt retension` command retensions a tape. Retension means moving the tape one complete pass between EOT and BOT. The syntax of the `mt retension` command is as follows:

```
scu> mt retension
```

The `mt seod` command spaces to end of data, that is, to the end of recorded media. The syntax of the `mt seod` command is as follows:

```
scu> mt seod
```

The `mt unload` command unloads a tape. This command is the same as the `mt offline` command, except the immediate bit is enabled so that the command completes after the unload operation is initiated.

The syntax of the `mt unload` command is as follows:

```
scu> mt unload
```

The `mt weof` command writes tape file marks. The syntax of the `mt weof` command is as follows:

```
scu> mt weof [ count ]
```

You can use the alias `eof` for the `mt weof` command.

B.3.4 The pause Command

The `pause` command is used to pause the playing of a CD-ROM audio disc. The format of the `pause` command is as follows:

```
scu> pause
```

B.3.5 The play Command

The `play` command is used to play audio tracks on a CD-ROM audio disc. If no keywords are specified, all audio tracks are played by default. You can specify a track number, a range of audio tracks, a logical block address, or a time address. The formats of the `play` command are as follows:

```
scu> play { [ starting n ] [ ending n ] }
          { [ track n ] }

scu> play audio { lba n }
                { length n }
                { lba n length n }

scu> play msf { starting keyword }
              { ending keyword }
              { starting keyword ending keyword }
```

The `starting` and `ending` keywords can be any conbination of the following:

- `minute-units` *n*
- `second-units` *n*
- `frame-units` *n*

B.3.6 The prevent Command

The `prevent` command prevents media removal from the selected device. The syntax of the `prevent` command is as follows:

`scu> prevent`

B.3.7 The release Command

The `release` command releases a reserved SCSI device or releases a frozen SIM queue after an error. The format of the `release` command is as follows:

`scu> release {` *device* | simqueue } [*nexus-information*]

The *`device`* argument specifies a reserved SCSI device to be released. The format of the command using the `device` argument is as follows:

`scu> release` *device* [*nexus-information*]

The extent release capability for direct access devices is not implemented.

The `simqueue` argument issues a Release SIMQ CCB to thaw a frozen SIM queue. Ordinarily, this command is not necessary because the SIM queue is automatically released after errors occur. If the nexus information is omitted, the SIM queue for the selected SCSI device is released.

The format of the command using the `simqueue` argument is as follows:

`scu> release simqueue [` *nexus-information*]

For example:

`scu>` **`release simqueue bus 1 target 6 lun 0`**

B.3.8 The reserve Command

The `reserve` command issues a SCSI Reserve command to the selected device. The entire logical unit is reserved for the exclusive use of the initiator. Extent reservation for direct access devices is not implemented. The format of the `reserve` command is as follows:

`scu> reserve` *device*

B.3.9 The reset Command

The `reset` command resets the SCSI bus or the selected SCSI device. The format of the `reset` command is as follows:

scu> reset { bus | device } [*nexus-information*]

The `bus` argument issues a CAM Bus Reset CCB. If the nexus information is omitted, the bus associated with the selected SCSI device is reset. The `reset bus` command is restricted to superuser (root) access because it can cause loss of data to some devices.

The format of the command using the `bus` argument is as follows:

scu> reset bus [*nexus-information*]

The `device` argument issues a CAM Bus Device Reset CCB. If the nexus information is omitted, the selected device is reset. The `reset device` command requires write access to the selected device because command can cause loss of data to some devices. If nexus information is specified, this command is restricted to the superuser.

The format of the command using the `device` argument is as follows:

scu> reset device [*nexus-information*]

B.3.10 The resume Command

The `resume` command causes a CD-ROM audio disc to resume play after it has been paused with the `pause` command. The format of the `resume` command is as follows:

```
scu> resume
```

B.3.11 The start Command

The `start` command issues a SCSI Start Unit command to the selected device. This action enables the selected device to allow media access operations. The format of the `start` command is as follows:

```
scu> start
```

B.3.12 The stop Command

The `stop` command issues a SCSI Stop Unit command to the selected device. This action disables the selected device from allowing media access operations.

The format of the `stop` command is as follows:

```
scu> stop
```

B.3.13 The tur Command

The `tur` command issues a Test Unit Ready command to determine the readiness of a device. If the command detects a failure, it automatically reports the sense data. The format of the `tur` command is as follows:

```
scu> tur
```

B.3.14 The verify Command

The `verify` command performs verify operations on the selected device. The format of the `verify` command is as follows:

scu> verify { media [*test-parameters*] }

The `media` argument verifies the data written on the device media. This activity involves reading and performing an ECC check of the data. If the test parameters are omitted, the entire device media is verified.

The format of the command using the `media` argument is as follows:

scu> verify media [*test-parameters*]

If the device does not support the `verify` command, the following error message appears:

```
scu> verify media starting 1000 length 1024
Verifying 1024 blocks on /dev/rrz10c (RX23),
                              please be patient...
Verifying blocks [ 1000 through 2023 ] ...
scu: Sense Key = 0x5 = ILLEGAL REQUEST -
                              Illegal request or CDB parameter,
     Sense Code/Qualifier = (0x20, 0) =
                              Invalid command operation code
```

When an error occurs, the sense key is examined. The expected sense keys are Recovered Error (0x01) or Medium Error (0x03). When these errors are detected, the following error message is displayed and verification continues with the block following the failing block:

```
scu: Verify error at logical block number 464392 (0x71608).
scu: Sense Key = 0x1 = RECOVERED ERROR -
                              Recovery action performed,
     Sense Code/Qualifier = (0x17, 0) = Recovered data with no
                              error correction applied
```

If any other sense key error occurs, the full sense data is displayed and the verification process is aborted.

The following conditions apply to the `verify` command:

- On failure, the failing logical block number (LBN) is reported and verification continues with the block following the failing block.
- By default, verification is performed using the current parameters in the Error Recovery mode page. Drive recovery can be disabled using the `set recovery off` command.

For example:

```
scu> verify media lba 464388
Verifying 1 blocks on /dev/rrz14c (RZ55), please be patient...
Verifying blocks [ 464388 through 464388 ] ...

scu> verify media starting 640000
Verifying 9040 blocks on /dev/rrz14c (RZ55), please be patient...
Verifying blocks [ 640000 through 649039 ] ...

scu> verify media starting 1000 length 250
Verifying 250 blocks on /dev/rrz14c (RZ55), please be patient...
Verifying blocks [ 1000 through 1249 ] ...

scu> verify media starting 1000 ending 2000
Verifying 1001 blocks on /dev/rrz14c (RZ55), please be patient...
Verifying blocks [ 1000 through 2000 ] ...
```

B.4 Device and Bus Maintenance Commands

This section describes `scu` utility commands that are used to maintain SCSI devices and the CAM I/O subsystem. The commands are:

- `change pages`
- `download`
- `format`
- `read`
- `reassign`
- `test`
- `write`

B.4.1 The change pages Command

The `change pages` command changes the mode pages for a device. The program prompts you with a list of the page fields that are marked as changeable. If you do not specify a mode page, all pages supported by the device are requested for changing. After you enter the new fields for each page, you use a mode select command to set the new page parameters.

The format for the `change pages` command is as follows:

`scu> change pages [` *mode-page* ... [`pcf` *page-control-field*]]

The *`mode-page`* argument describes the mode page to change. The mode pages are:

scu Keyword	Page Code	Description
`error-recovery`	0x01	Error recovery page
`disconnect`	0x02	Disconnect/reconnect page
`direct-access`	0x03	Direct access format page
`geometry`	0x04	Disk geometry page
`flexible`	0x05	Flexible disk page
`cache-control`	0x08	Cache control page
`cdrom`	0x0D	CD-ROM device page
`audio-control`	0x0E	Audio control page
`device-configuration`	0x10	Device configuration page
`medium-partition-1`	0x11	Medium partition page 1
`dec-specific`	0x25	Digital specific page
`readahead-control`	0x38	Read-ahead control page

The *`page-control-field`* argument specifies the type of mode pages to obtain from device. The page control fields that you can specify are as follows:

- `changeable`
- `current`
- `default`
- `saved`

The following example changes the error recover parameters:

```
scu> change pages error
Changing Error Recovery Parameters (Page 1 - current values):

Disable Transfer on Error (DTE) [R:0-1 D:0]:
Post Recoverable Error (PER) [R:0-1 D:1]:
Transfer Block (TB) [R:0-1 D:1]:
Retry Count [R:0-255 D:1]: 25
scu>
```

B.4.2 The download Command

The `download` command can be used with any device that supports the downloading of operating software through the Write Buffer command. The format for the `download` command is as follows:

scu> download *filename* [save]

The `save` keyword directs the device to save the new operating software in non-volatile memory if the `download` command completes successfully. With `save` specified, the downloaded code remains in effect after each power cycle and reset. If the `save` keyword is not specified, the downloaded software is placed in the control memory of the device. After a power cycle or reset, the device operation would revert to a vendor-specific condition.

B.4.3 The format Command

The `format` command formats both hard and flexible disk media. Since this command modifies the disk media, the full command name must be entered to be recognized. The format for the `format` command is as follows:

scu> format [density *density-type*] [defects *defect-list*]

The *`density-type`* parameter specifies the density type for flexible disk media. The *`defect-list`* parameter can be `all`, `primary`, or `none`. The default is to format with all known defects.

If you enter the `scu` utility using the default device `/dev/cam` and then set the device to format using the `set nexus` command, the code associated with checking for mounted file systems fails This failure avoids the possibility of accidentally formatting disks with mounted file systems.

B.4.4 The read Command

The `read` command performs read operations from the selected device. The command reads the device media and performs a data comparison of the data read. You must include test parameters that specify the media area to be read. Section B.1.1 contains a list of the valid test parameters.

The format of the `read` command is as follows:

scu> read { media [*test-parameters*] }

The examples that follow illustrate the use of the `read` command with several test-parameters:

```
scu> read media
scu: No defaults, please specify test parameters for transfer...

scu> read media lba 100
Reading 1 block on /dev/rrz10c (RX23) using pattern 0x39c39c39...

scu> read media lba 100 pattern 0x12345678
Reading 1 block on /dev/rrz10c (RX23) using pattern 0x12345678...
scu: Data compare error at byte position 0
scu: Data expected = 0x78, data found = 0x39

scu> read media ending 100 compare off bs 10k
Reading 101 blocks on /dev/rrz10c (RX23)...
Reading blocks [ 0 through 19 ]...
Reading blocks [ 20 through 39 ]...
Reading blocks [ 40 through 59 ]...
Reading blocks [ 60 through 79 ]...
Reading blocks [ 80 through 99 ]...
```

B.4.5 The reassign Command

The `reassign` command allows you to reassign a defective block on a disk device. Since this command modifies the disk media, the full command name must be entered to be recognized. The format of the `reassign` command is as follows:

scu> reassign lba *logical-block*

B.4.6 The test Command

The `test` command performs tests on a controller by issuing send and receive diagnostic commands or `write buffer` and `read buffer` commands for memory testing to the selected device. If you issue the `test` command with no arguments, the utility performs a self test, which is supported by most controllers. The format for the `test` command is as follows:

```
scu> test [ controller | drive | memory | selftest ]
```

B.4.7 The write Command

The `write` command writes to the selected device. The format of the `write` command is as follows:

scu> write { media [*test-parameters*] }

The `media` argument writes to the device media using various data patterns. The patterns default to 0x39c39c39 for the first pass, 0xc6dec6de for the second, and so on as shown in the last example. You must specify transfer parameters that specify the media area to be written.

The format of the command using the `media` argument is as follows:

scu> write media [*test-parameters*]

Section B.1.1 contains a list of the valid test parameters.

For example:

```
scu> write media
scu: No defaults, please specify test parameters for transfer...

scu> write media lba 100
Writing 1 block on /dev/rrz10c (RX23) with pattern 0x39c39c39...

scu> write media starting 100 ending 250
Writing 151 blocks on /dev/rrz10c (RX23) with pattern 0x39c39c39...

scu> write media starting 2800 limit 1m bs 10k
Writing 80 blocks on /dev/rrz10c (RX23) with pattern 0x39c39c39...
Writing blocks [ 2800 through 2819 ]...
Writing blocks [ 2820 through 2839 ]...
Writing blocks [ 2840 through 2859 ]...
Writing blocks [ 2860 through 2879 ]...

scu> write media lba 2879 passes 5
Writing 1 block on /dev/rrz10c (RX23) with pattern 0x39c39c39...
Writing 1 block on /dev/rrz10c (RX23) with pattern 0xc6dec6de...
Writing 1 block on /dev/rrz10c (RX23) with pattern 0x6db6db6d...
Writing 1 block on /dev/rrz10c (RX23) with pattern 0x00000000...
Writing 1 block on /dev/rrz10c (RX23) with pattern 0xffffffff...
```

SCSI/CAM Routines C

This appendix contains a description of each of the routines described in this guide, in reference page format. The routines are included in alphabetical order.

C.1 cam_logger

Name

cam_logger – Allocates a system error log buffer and fills in a `uerf` error log packet

Syntax

u_long cam_logger(*cam_err_hdr*, *bus*, *target*, *lun*)
CAM_ERR_HDR **cam_err_hdr*;
long *bus*;
long *target*;
long *lun*;

Arguments

cam_err_hdr	Pointer to the Error Header Structure.
bus	SCSI target's bus controller number.
target	SCSI target's ID number.
lun	SCSI target's logical unit number.

Description

The `cam_logger` routine allocates a system error log buffer and fills in a `uerf` error log packet. The routine fills in the bus, target, and LUN information from the Error Header Structure passed to it and copies the Error Header Structure and the Error Entry Structures and data to the error log buffer.

Return Value

None

C.2 ccfg_attach

Name

ccfg_attach – Calls a SCSI/CAM peripheral driver's attach routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found

Syntax

int ccfg_attach(*attach*)
struct device **attach*;

Arguments

attach Pointer to the device information contained in the `device` structure.

Description

The `ccfg_attach` routine calls a SCSI/CAM peripheral driver's attach routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found. The routine is called during autoconfiguration. The `ccfg_attach` routine locates the configured driver in the SCSI/CAM peripheral driver configuration table. If the driver is located successfully, the SCSI/CAM peripheral driver's attach routine is called with a pointer to the unit information structure for the device from the kernel `device` structure. The SCSI/CAM peripheral driver's attach routine performs its own attach initialization.

Return Value

0 = success
1 = failure
The return value is ignored by autoconfiguration code.

C.3 ccfg_edtscan

Name

ccfg_edtscan – Issues SCSI INQUIRY commands to all possible SCSI targets and LUNs attached to a bus or a particular `bus/target/lun`

Syntax

U32 ccfg_edtscan(*scan_type*, *bus*, *target*, *lun*)
long *scan_type*;
long *bus*;
long *target*;
long *lun*;

Arguments

scan_type	Types of scans are: EDT_FULLSCAN, which traverses the CAM_EDT_ENTRY structure and sends an INQUIRY command to each target and LUN; EDT_PARTSCAN, which sends an INQUIRY command only to targets and LUNs flagged as ''not found''; or EDT_SINGLESCAN, which sends an INQUIRY command to the selected bus, target, and LUN passed as arguments.
bus	SCSI target's bus controller number.
target	SCSI target's ID number.
lun	SCSI target's logical unit number.

Description

The `ccfg_edtscan` routine issues SCSI INQUIRY commands to all possible SCSI targets and LUNs attached to a bus or a particular `bus/target/lun` The routine uses the CAM subsystem in the normal manner by sending SCSI I/O CCBs to the SIMs. The INQUIRY data returned is stored in the EDT structures and the `cam_tlun_found` flag is set. This routine can be called by the SCSI/CAM peripheral device drivers to reissue a full, partial, or single bus scan command.

Return Value

CAM_SUCCESS
CAM_FAILURE

C.4 ccfg_slave

Name

ccfg_slave – Calls a SCSI/CAM peripheral driver's slave routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found

Syntax

int ccfg_slave(*attach*, *csr*)
struct device **attach*;
caddr_t *csr*;

Arguments

attach Pointer to the device information contained in the `device` structure.

csr The virtual address of the control and status register (CSR) address.

Description

The `ccfg_slave` routine calls a SCSI/CAM peripheral driver's slave routine after a match on the `cpd_name` member of the CAM_PERIPHERAL_DRIVER structure is found. The routine is called during autoconfiguration. The `ccfg_slave` routine locates the configured driver in the SCSI/CAM peripheral driver configuration table. If the driver is located successfully, the SCSI/CAM peripheral driver's slave routine is called with a pointer to the unit information structure for the device from the kernel `device` structure and the virtual address of its control and status register (CSR). The SCSI/CAM peripheral driver's slave routine performs its own slave initialization.

Return Value

0 = slave is alive
1 = slave is not alive

C.5 ccmn_DoSpecialCmd

Name

ccmn_DoSpecialCmd – Provides a simplified interface to the special command routine

Syntax

ccmn_DoSpecialCmd(*dev*, *cmd*, *data*, *flags*, *ccb*, *sflags*)
dev_t *dev*;
int *cmd*;
caddr_t *data*;
int *flags*;
CCB_SCSIIO **ccb*;
int *sflags*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cmd The ioctl command such as SCSI_FORMAT_UNIT. The ioctl commands are defined in `/usr/sys/include/io/cam/rzdisk.h`.

data The user data buffer.

flags Flags set when a file is open.

ccb Pointer to the SCSI I/O CCB structure or NULL.

sflags SCSI/CAM special I/O control flags. Setting this field is optional. The available bits are:

Flag Name	Description
`SA_NO_ERROR_RECOVERY`	Do not perform error recovery
`SA_NO_ERROR_LOGGING`	Do not log error messages
`SA_NO_SLEEP_INTR`	Do not allow sleep interrupts
`SA_NO_SIMQ_THAW`	Leave SIM queue frozen when there are errors
`SA_NO_WAIT_FOR_IO`	Do not wait for I/O to complete

Description

The `ccmn_DoSpecialCmd` routine provides a simplified interface to the special command routine. The routine prepares for and issues special SCSI ioctl commands.

Return Value

The `ccmn_DoSpecialCmd` routine returns a value of 0 (zero) upon successful completion. It returns the appropriate error code on failure.

C.6 ccmn_SysSpecialCmd

Name

ccmn_SysSpecialCmd – Lets a system request issue SCSI I/O commands to the SCSI/CAM special I/O interface

Syntax

ccmn_SysSpecialCmd(*dev*, *cmd*, *data*, *flags*, *ccb*, *sflags*)
dev_t *dev*;
int *cmd*;
caddr_t *data*;
int *flags*;
CCB_SCSIIO **ccb*;
int *sflags*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cmd The ioctl command. Refer to the commands defined in `/usr/sys/include/io/cam/rzdisk.h`.

data The kernel data buffer.

flags Flags set when a file is open.

ccb Pointer to the SCSI I/O CCB structure. This field is optional.

sflags SCSI/CAM special I/O control flags. The available flags are:

Flag Name	Description
SA_NO_ERROR_RECOVERY	Do not perform error recovery
SA_NO_ERROR_LOGGING	Do not log error messages
SA_NO_SLEEP_INTR	Do not allow sleep interrupts
SA_NO_SIMQ_THAW	Leave SIM queue frozen when there are errors
SA_NO_WAIT_FOR_IO	Do not wait for I/O to complete

Description

The ccmn_SysSpecialCmd routine lets a system request issue SCSI I/O commands to the SCSI/CAM special I/O interface. This permits existing SCSI commands to be issued from within kernel code.

Return Value

The ccmn_DoSpecialCmd routine returns a value of 0 (zero) upon successful completion. It returns the appropriate error code on failure.

C.7 ccmn_abort_ccb_bld

Name

ccmn_abort_ccb_bld – Creates an ABORT CCB and sends it to the XPT

Syntax

ccmn_abort_ccb_bld(*dev*, *cam_flags*, *abort_ccb*)
dev_t *dev*;
u_long *cam_flags*;
CCB_HEADER **abort_ccb*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue

Flag Name	Description
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

abort_ccb Pointer to the CAM Control Block (CCB) header structure to abort.

Description

The `ccmn_abort_ccb_bld` routine creates an ABORT CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the address of the CCB to be aborted and calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_ABORT pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.8 ccmn_abort_que

Name

ccmn_abort_que – Sends an ABORT CCB request for each SCSI I/O CCB on the active queue

Syntax

ccmn_abort_que(*pd*)
PDRV_DEVICE **pd*;

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

Description

The `ccmn_abort_que` routine sends an ABORT CCB request for each SCSI I/O CCB on the active queue. This routine must be called with the Peripheral Device Structure locked.

The `ccmn_abort_que` routine calls the `ccmn_abort_ccb_bld` routine to create an ABORT CCB for the first active CCB on the active queue and send it to the XPT. It calls the `ccmn_send_ccb` routine to send the ABORT CCB for each of the other CCBs on the active queue that are marked as active to the XPT. The `ccmn_abort_que` routine then calls the `ccmn_rel_ccb` routine to return the ABORT CCB to the XPT.

Return Value

None

See Also

`ccmn_abort_ccb_bld`, `ccmn_rel_ccb`, `ccmn_send_ccb`

C.9 ccmn_attach_device

Name

ccmn_attach_device – Creates and attaches a device structure to the controller structure that corresponds to the SCSI controller

Syntax

ccmn_attach_device(*dev*, *dev_type*, *dev_name*)

dev_t	*dev*;
caddr_t	*dev_type*;
caddr_t	*dev_name*;

Arguments

dev
: The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

dev_type
: Pointer to the device-type string, for example, "disk" or "tape".

dev_name
: Pointer to the device-name string as it appears in the `/dev` directory, for example, "rz0".

Description

The `ccmn_attach_device` routine creates and attaches a device structure to the controller structure that corresponds to the SCSI controller. The routine finds the controller structure for a device, fills in the device structure, and attaches the device structure to the controller structure.

Return Value

None

See Also

`ccmn_errlog`, `ccmn_find_ctlr`

C.10 ccmn_bdr_ccb_bld

Name

ccmn_bdr_ccb_bld – Creates a BUS DEVICE RESET CCB and sends it to the XPT

Syntax

ccmn_bdr_ccb_bld(*dev*, *cam_flags*)
dev_t *dev*;
u_long *cam_flags*;

Arguments

dev — The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags — The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue

Flag Name	Description
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

Description

The `ccmn_bdr_ccb_bld` routine creates a BUS DEVICE RESET CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_RESETDEV pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.11 ccmn_br_ccb_bld

Name

ccmn_br_ccb_bld – Creates a BUS RESET CCB and sends it to the XPT

Syntax

ccmn_br_ccb_bld(*dev*, *cam_flags*)
dev_t *dev*;
u_long *cam_flags*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state

Flag Name	Description
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

Description

The `ccmn_br_ccb_bld` routine creates a BUS RESET CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_RESETBUS pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.12 ccmn_ccb_status

Name

ccmn_ccb_status – Assigns individual CAM status values to generic categories

Syntax

ccmn_ccb_status(*ccb*)
CCB_HEADER **ccb*;

Arguments

ccb Pointer to the CAM Control Block (CCB) header structure whose status is to be categorized.

Description

The `ccmn_ccb_status` routine assigns individual CAM status values to generic categories. The following table shows the returned category for each CAM status value:

CAM Status	Assigned Category
CAM_REQ_INPROG	CAT_INPROG
CAM_REQ_CMP	CAT_CMP
CAM_REQ_ABORTED	CAT_ABORT
CAM_UA_ABORT	CAT_ABORT
CAM_REQ_CMP_ERR	CAT_CMP_ERR
CAM_BUSY	CAT_BUSY
CAM_REQ_INVALID	CAT_CCB_ERR
CAM_PATH_INVALID	CAT_NO_DEVICE
CAM_DEV_NOT_THERE	CAT_NO_DEVICE
CAM_UA_TERMIO	CAT_ABORT
CAM_SEL_TIMEOUT	CAT_DEVICE_ERR
CAM_CMD_TIMEOUT	CAT_DEVICE_ERR
CAM_MSG_REJECT_REC	CAT_DEVICE_ERR
CAM_SCSI_BUS_RESET	CAT_RESET
CAM_UNCOR_PARITY	CAT_DEVICE_ERR
CAM_AUTOSENSE_FAIL	CAT_BAD_AUTO
CAM_NO_HBA	CAT_NO_DEVICE
CAM_DATA_RUN_ERR	CAT_DEVICE_ERR
CAM_UNEXP_BUSFREE	CAT_DEVICE_ERR
CAM_SEQUENCE_FAIL	CAT_DEVICE_ERR
CAM_CCB_LEN_ERR	CAT_CCB_ERR

CAM Status	Assigned Category
CAM_PROVIDE_FAIL	CAT_CCB_ERR
CAM_BDR_SENT	CAT_RESET
CAM_REQ_TERMIO	CAT_ABORT
CAM_LUN_INVALID	CAT_NO_DEVICE
CAM_TID_INVALID	CAT_NO_DEVICE
CAM_FUNC_NOTAVAIL	CAT_CCB_ERR
CAM_NO_NEXUS	CAT_NO_DEVICE
CAM_IID_INVALID	CAT_NO_DEVICE
CAM_SCSI_BUSY	CAT_SCSI_BUSY
Other	CAT_UNKNOWN

Return Value

The following categories can be returned:

CAM Status	Assigned Category
CAT_INPROG	Request is in progress.
CAT_CMP	Request has completed without error.
CAT_CMP_ERR	Request has completed with error.
CAT_ABORT	Request either has been aborted or terminated, or it cannot be aborted or terminated.
CAT_BUSY	CAM is busy.
CAT_SCSI_BUSY	SCSI is busy.
CAT_NO_DEVICE	No device at address specified in request.
CAT_DEVICE_ERR	Bus or device problems.
CAT_BAD_AUTO	Invalid autosense data.
CAT_CCB_ERR	Invalid CCB.
CAT_RESET	Unit or bus has detected a reset condition.
CAT_UNKNOWN	Invalid CAM status.

C.13 ccmn_check_idle

Name

ccmn_check_idle – Checks that there are no opens against a device

Syntax

ccmn_check_idle(*start_unit*, *num_units*, *cmajor*, *bmajor*, *spec_size*)
U32 *start_unit*;
U32 *num_units*;
dev_t *cmajor*;
dev_t *bmajor*;
U32 *spec_size*;

Arguments

start_unit	The address (bus, target, and LUN) of the first unit to check for opens.
num_units	The number of units to check for opens.
cmajor	The character device major number.
bmajor	The block device major number.
spec_size	The size of the device-specific structure for the device.

Description

The `ccmn_check_idle` routine checks that there are no opens against a device. This routine calls the `ccmn_rel_dbuf` routine to deallocate all structures pertaining to the device whose driver is being unloaded.

The `ccmn_check_idle` routine scans the Peripheral Device Unit Table looking for devices that match the block device major number and the character device major number in the PDRV_DEVICE structure members, `pd_bmajor` and `pd_cmajor`. If no opens exist for the devices that are to be unloaded, it rescans the Peripheral Device Unit Table and deallocates all structures relating to the devices whose driver is being unloaded. The `ccmn_check_idle` routine must be called with the Peripheral Device Unit Table locked.

Return Value

None

See Also

`ccmn_rel_dbuf`

C.14 ccmn_close_unit

Name

ccmn_close_unit – Handles the common close for all SCSI/CAM peripheral device drivers

Syntax

ccmn_close_unit(*dev*)
dev_t *dev*;

Arguments

dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

Description

The `ccmn_close_unit` routine handles the common close for all SCSI/CAM peripheral device drivers. It sets the open count to zero.

Return Value

None

See Also

`ccmn_open_unit`

C.15 ccmn_errlog

Name

ccmn_errlog – Reports error conditions for the SCSI/CAM peripheral device driver

Syntax

ccmn_errlog(*func_str*, *opt_str*, *flags*, *ccb*, *dev*, *unused*)
u_char **func_str*;
u_char **opt_str*;
u_long *flags*;
CCB_HEADER **ccb*;
dev_t *dev*;
u_char **unused*;

Arguments

func_str	Pointer to function in which the error was detected.
opt_str	Pointer to optional logging string.
flags	Flags for peripheral drivers error types. The flags are: CAM_INFORMATIONAL; CAM_SOFTERR; CAM_HARDERR; CAM_SOFTWARE; and CAM_DUMP_ALL. They are defined in the `/usr/sys/include/io/cam/cam_logger.h` file.
ccb	Pointer to the CAM Control Block (CCB) header structure.
dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.
unused	Unused. It is needed to match the number of arguments expected by the CAM_ERROR macro, which is defined in the `/usr/sys/include/io/cam/cam_errlog.h` file

Description

The `ccmn_errlog` routine reports error conditions for the SCSI/CAM peripheral device driver. The routine is passed a pointer to the name of the function in which the error was detected. The routine builds informational strings based on the error condition.

Return Value

None

C.16 ccmn_find_ctlr

Name

ccmn_find_ctlr – Finds the controller structure that corresponds to the SCSI controller that the device must be attached to

Syntax

struct controller *
ccmn_find_ctlr(*dev*)
dev_t *dev*;

Arguments

dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

Description

The `ccmn_find_ctlr` routine finds the controller structure that corresponds to the SCSI controller that the device must be attached to. This routine must be called with the Peripheral Device Unit Table locked.

Return Value

Controller for the device or NULL if no controller is found.

C.17 ccmn_gdev_ccb_bld

Name

ccmn_gdev_ccb_bld – Creates a GET DEVICE TYPE CCB and sends it to the XPT

Syntax

ccmn_gdev_ccb_bld(*dev*, *cam_flags*, *inq_addr*)
dev_t *dev*;
u_long *cam_flags*;
u_char **inq_addr*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous

Flag Name	Description
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

inq_addr Pointer to the address for Inquiry data returned.

Description

The `ccmn_gdev_ccb_bld` routine creates a GET DEVICE TYPE CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The `ccmn_gdev_ccb_bld` routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_GETDEV pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.18 ccmn_get_bp

Name

ccmn_get_bp – Allocates a `buf` structure

Syntax

ccmn_get_bp()

Arguments

None

Description

The `ccmn_get_bp` routine allocates a `buf` structure. This function must not be called at interrupt context. The function may sleep waiting for resources.

Return Value

Pointer to buf structure. This pointer may be NULL.

C.19 ccmn_get_ccb

Name

ccmn_get_ccb – Allocates a CCB and fills in the common portion of the CCB header

Syntax

ccmn_get_ccb(*dev*, *func_code*, *cam_flags*, *ccb_len*)
dev_t *dev*;
u_char *func_code*;
u_long *cam_flags*;
u_short *ccb_len*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

func_code The XPT function code for the CCB. See American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module*, working draft, X3T9.2/90-186, Section 8.1.2, for a list of the function codes.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect

Flag Name	Description
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

ccb_len The length of the CCB.

Description

The `ccmn_get_ccb` routine allocates a CCB and fills in the common portion of the CCB header. The routine calls the `xpt_ccb_alloc` routine to allocate a CCB structure. The `ccmn_get_ccb` routine fills in the common portion of the CCB header and returns a pointer to that CCB_HEADER.

Return Value

Pointer to newly allocated CCB header.

See Also

`xpt_ccb_alloc`

C.20 ccmn_get_dbuf

Name

ccmn_get_dbuf – Allocates a data buffer area of the size specified by calling the kernel memory allocation routines

Syntax

ccmn_get_dbuf(*size*)
u_long *size*;

Arguments

size	Size of buffer in bytes.

Description

The `ccmn_get_dbuf` routine allocates a data buffer area of the size specified by calling the kernel memory allocation routines .

Return Value

Pointer to kernel data space. If this is NULL, no data buffers are available and no more can be allocated.

C.21 ccmn_init

Name

ccmn_init – Initializes the XPT and the unit table lock structure

Syntax

ccmn_init ()

Description

The `ccmn_init` routine initializes the XPT and the unit table lock structure. The first time the `ccmn_init` routine is called, it calls the `xpt_init` routine to request the XPT to initialize the CAM subsystem.

Return Value

None

See Also

`xpt_init`

C.22 ccmn_io_ccb_bld

Name

ccmn_io_ccb_bld – Allocates a SCSI I/O CCB and fills it in

Syntax

ccmn_io_ccb_bld(*dev*, *data_addr*, *data_len*, *sense_len*, *cam_flags*, \
comp_func, *tag_action*, *timeout*, *bp*)
dev_t *dev*;
u_char **data_addr*;
u_long *data_len*;
u_short *sense_len*;
u_long *cam_flags*;
void (**comp_func*) ();
u_char *tag_action*;
u_long *timeout*;
struct buf **bp*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

data_addr Pointer to the data buffer.

data_len Size of the data transfer.

sense_len Length of the sense data buffer to be returned on autosense, which is predefined as 64 bytes in the DEC_AUTO_SENSE_SIZE environment variable but can be larger.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
`CAM_DIR_RESV`	Data direction (00: reserved)
`CAM_DIR_IN`	Data direction (01: DATA IN)
`CAM_DIR_OUT`	Data direction (10: DATA OUT)
`CAM_DIR_NONE`	Data direction (11: no data)
`CAM_DIS_AUTOSENSE`	Disable autosense feature

Flag Name	Description
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

comp_func SCSI device driver I/O callback completion function. This pointer may be NULL if the CAM DISABLE CALLBACK bit is set in the CAM FLAGS field.

tag_action Type of action to perform for tagged requests:

`CAM_SIMPLE_QTAG`	Tag for simple queue
`CAM_HEAD_QTAG`	Tag for head of queue
`CAM_ORDERED_QTAG`	Tag for ordered queue

timeout Timeout for the request in seconds. A value of 0 (zero) indicates the default, which is five seconds.

bp A `buf` structure pointer, which is used for request mapping. This pointer may be NULL.

Description

The `ccmn_io_ccb_bld` routine allocates a SCSI I/O CCB and fills it in. The routine calls the `ccmn_get_ccb` routine to obtain a CCB structure with the header portion filled in. The `ccmn_io_ccb_bld` routine fills in the SCSI I/O-specific fields from the parameters passed and checks the length of the sense data to see if it exceeds the length of the reserved sense buffer. If it does, a sense buffer is allocated using the `ccmn_get_dbuf` routine.

Return Value

Pointer to a SCSI I/O CCB

See Also

`ccmn_get_ccb`, `ccmn_get_dbuf`

C.23 ccmn_mode_select

Name

ccmn_mode_select – Creates a SCSI I/O CCB for the MODE SELECT command, sends it to the XPT for processing, and sleeps waiting for it to complete.

Syntax

ccmn_mode_select(*pd*, *sense_len*, *cam_flags*, *comp_func*, *tag_action*, \
timeout, *ms_index*)
PDRV_DEVICE **pd*;
u_short *sense_len*;
u_long *cam_flags*;
void (**comp_func*) ();
u_char *tag_action*;
u_long *timeout*;
unsigned *ms_index*;

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

sense_len Length of the sense data buffer to be returned on autosense, which is predefined as 64 bytes in the DEC_AUTO_SENSE_SIZE environment variable but can be larger.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
`CAM_DIR_RESV`	Data direction (00: reserved)
`CAM_DIR_IN`	Data direction (01: DATA IN)
`CAM_DIR_OUT`	Data direction (10: DATA OUT)
`CAM_DIR_NONE`	Data direction (11: no data)
`CAM_DIS_AUTOSENSE`	Disable autosense feature
`CAM_SCATTER_VALID`	Scatter/gather list is valid
`CAM_DIS_CALLBACK`	Disable callback feature

Flag Name	Description
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

comp_func SCSI device driver I/O callback completion function. This pointer may be NULL if the CAM DISABLE CALLBACK bit is set in the CAM FLAGS field.

tag_action Type of action to perform for tagged requests:

`CAM_SIMPLE_QTAG`	Tag for simple queue
`CAM_HEAD_QTAG`	Tag for head of queue
`CAM_ORDERED_QTAG`	Tag for ordered queue

timeout Timeout for the request in seconds. A value of 0 (zero) indicates the default, which is five seconds.

ms_index An index into a page in the Mode Select Table that is pointed to in the Device Descriptor Structure.

Description

The `ccmn_mode_select` routine creates a SCSI I/O CCB for the MODE SELECT command and sends it to the XPT for processing. The routine calls the `ccmn_io_ccb_bld` routine to obtain a SCSI I/O CCB structure. It uses the *ms_index* parameter to index into the Mode Select Table pointed to by the `dd_modsel_tbl` member of the Device Descriptor Structure for the SCSI device. The `ccmn_mode_select` routine calls the `ccmn_send_ccb_wait` routine to send the SCSI I/O CCB to the XPT and wait for it to complete. The `ccmn_mode_select` routine sleeps at a non-interruptible priority. It requires the callback completion function to issue a wakeup call on the address of the CCB.

Return Value

CCB_SCSIIO pointer

See Also

`ccmn_io_ccb_bld`, `ccmn_send_ccb_wait`

C.24 ccmn_open_unit

Name

ccmn_open_unit – Handles the common open for all SCSI/CAM peripheral device drivers

Syntax

ccmn_open_unit(*dev*, *scsi_dev_type*, *flag*, *dev_size*)
dev_t *dev*;
u_long *scsi_dev_type*;
u_long *flag*;
u_long *dev_size*;

Arguments

dev — The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

scsi_dev_type — SCSI device type value from Inquiry data.

flag — Indicates whether or not the device is being opened for exclusive use. A setting of 1 means exclusive use; a setting of 0 (zero) means nonexclusive use.

dev_size — The device-specific structure size in bytes.

Description

The `ccmn_open_unit` routine handles the common open for all SCSI/CAM peripheral device drivers. It must be called for each open before any SCSI device-specific open code is executed.

On the first call to the `ccmn_open_unit` routine for a device, the `ccmn_gdev_ccb_bld` routine is called to issue a GET DEVICE TYPE CCB to obtain the Inquiry data. The `ccmn_open_unit` routine allocates the Peripheral Device Structure, PDRV_DEVICE, and a device-specific structure, either TAPE_SPECIFIC or DISK_SPECIFIC, based on the device size argument passed. The routine also searches the `cam_devdesc_tab` to obtain a pointer to the Device Descriptor Structure for the SCSI device and increments the open count. The statically allocated `pdrv_unit_table` structure contains a pointer to the PDRV_DEVICE structure. The PDRV_DEVICE structure contains pointers to the DEV_DESC structure and to the device-specific structure.

Return Value

The `ccmn_open_unit` routine returns a value of 0 (zero) upon successful completion.

Diagnostics

The `ccmn_open_unit` routine fails under the following conditions:

[EBUSY]	The device is already opened and the exclusive use bit is set.
[ENXIO]	The device does not exist or the *scsi_dev_type* parameter does not match the device type in the Inquiry data returned by GET DEVICE TYPE CCB. The *scsi_dev_type* was not configured.
[EFAULT]	The device requested would go beyond the size of the `pdrv_unit_table`.

See Also

`ccmn_close_unit`, `ccmn_gdev_ccb_bld`

C.25 ccmn_pinq_ccb_bld

Name

ccmn_pinq_ccb_bld – Creates a PATH INQUIRY CCB and sends it to the XPT

Syntax

ccmn_pinq_ccb_bld(*dev*, *cam_flags*)
dev_t *dev*;
u_long *cam_flags*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue

Flag Name	Description
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

Description

The `ccmn_pinq_ccb_bld` routine creates a PATH INQUIRY CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_PATHINQ pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.26 ccmn_rel_bp

Name

ccmn_rel_bp – Deallocates a `buf` structure

Syntax

ccmn_rel_bp(*bp*)
struct buf **bp*;

Arguments

bp A `buf` structure pointer, which is used for request mapping.

Description

The `ccmn_rel_bp` routine deallocates a `buf` structure.

Return Value

None

C.27 ccmn_rel_ccb

Name

ccmn_rel_ccb – Releases a CCB and returns the sense data buffer for SCSI I/O CCBs, if allocated

Syntax

ccmn_rel_ccb(*ccb*)
CCB_HEADER **ccb*;

Arguments

ccb Pointer to the CAM Control Block (CCB) header structure to be released.

Description

The `ccmn_rel_ccb` routine releases a CCB and returns the sense data buffer for SCSI I/O CCBs, if allocated. The routine calls the `xpt_ccb_free` routine to release a CCB structure. For SCSI I/O CCBs, if the sense data length is greater than the default sense data length, the `ccmn_rel_ccb` routine calls the `ccmn_rel_dbuf` routine to return the sense data buffer to the data buffer pool.

Return Value

None

See Also

`ccmn_rel_dbuf`, `xpt_ccb_free`

C.28 ccmn_rel_dbuf

Name

ccmn_rel_dbuf – Deallocates a data buffer

Syntax

ccmn_rel_dbuf(*addr*, *size*)
u_char **addr*;
U32 *size*;

Arguments

addr Pointer to the address of the data buffer to deallocate.

size Number of bytes to deallocate.

Description

The `ccmn_rel_dbuf` routine deallocates a data buffer.

Return Value

None

C.29 ccmn_rem_ccb

Name

ccmn_rem_ccb – Removes a SCSI I/O CCB request from the SCSI/CAM peripheral driver active queue and starts a tagged request if a tagged CCB is pending

Syntax

ccmn_rem_ccb(*pd*,*ccb*)
PDRV_DEVICE **pd*;
CCB_SCSIIO **ccb*;

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

ccb Pointer to the SCSI I/O CCB structure to remove from the active queue.

Description

The `ccmn_rem_ccb` routine removes a SCSI I/O CCB request from the SCSI/CAM peripheral driver active queue and starts a tagged request if a tagged CCB is pending. If a tagged CCB is pending, the `ccmn_rem_ccb` routine places the request on the active queue and calls the `xpt_action` routine to start the tagged request.

Return Value

None

See Also

`xpt_action`

C.30 ccmn_rsq_ccb_bld

Name

ccmn_rsq_ccb_bld – Creates a RELEASE SIM QUEUE CCB and sends it to the XPT

Syntax

ccmn_rsq_ccb_bld(*dev*, *cam_flags*)
dev_t *dev*;
u_long *cam_flags*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue

Flag Name	Description
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

Description

The `ccmn_rsq_ccb_bld` routine creates a RELEASE SIM QUEUE CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_RELSIM pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.31 ccmn_sasy_ccb_bld

Name

ccmn_sasy_ccb_bld – Creates a SET ASYNCHRONOUS CALLBACK CCB and sends it to the XPT

Syntax

ccmn_sasy_ccb_bld(*dev*, *cam_flags*, *async_flags*, *callb_func*, *buf*, *buflen*)
dev_t *dev*;
u_long *cam_flags*;
u_long *async_flags*;
void (**callb_func*) ();
u_char **buf*;
u_char *buflen*;

Arguments

dev — The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags — The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)

Flag Name	Description
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

async_flags Asynchronous Callback CCB flags for registering a callback routine for a specific bus, target, and LUN. The flags are defined in the `/usr/sys/include/io/cam/cam.h` file.

callb_func Asynchronous callback function.

buf SCSI/CAM peripheral buffer for asynchronous information.

buflen Allocated SCSI/CAM peripheral buffer length.

Description

The ccmn_sasy_ccb_bld routine creates a SET ASYNCHRONOUS CALLBACK CCB and sends it to the XPT. The routine calls the ccmn_get_ccb routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the asynchronous fields of the CCB and calls the ccmn_send_ccb routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_SETASYNC pointer

See Also

ccmn_get_ccb, ccmn_send_ccb

C.32 ccmn_sdev_ccb_bld

Name

ccmn_sdev_ccb_bld – Creates a SET DEVICE TYPE CCB and sends it to the XPT

Syntax

ccmn_sdev_ccb_bld(*dev*, *cam_flags*, *scsi_dev_type*)
dev_t *dev*;
u_long *cam_flags*;
u_char *scsi_dev_type*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous

Flag Name	Description
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

scsi_dev_type

SCSI device type value from Inquiry data.

Description

The `ccmn_sdev_ccb_bld` routine creates a SET DEVICE TYPE CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the device type field of the CCB and calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_SETDEV pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.33 ccmn_send_ccb

Name

ccmn_send_ccb – Sends CCBs to the XPT layer by calling the `xpt_action` routine

Syntax

ccmn_send_ccb(*pd*,*ccb*, *retry*)
PDRV_DEVICE **pd*;
CCB_HEADER **ccb*;
u_char *retry*

Arguments

pd	Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.
ccb	Pointer to the CAM Control Block (CCB) header structure to be sent to the `xpt_action` routine to handle the request.
retry	Indicates whether this request is a retry of a request that is already on the active queue. A 1 indicates RETRY, and a 0 (zero) indicates NOT_RETRY.

Description

The `ccmn_send_ccb` routine sends CCBs to the XPT layer by calling the `xpt_action` routine. This routine must be called with the Peripheral Device Structure locked.

For SCSI I/O CCBs that are not retries, the request is placed on the active queue. If the CCB is a tagged request and the tag queue size for the device has been reached, the request is placed on the tagged pending queue so that the request can be sent to the XPT at a later time. A high-water mark of half the queue depth for the SCSI device is used for tagged requests so that other initiators on the SCSI bus will not be blocked from using the device.

Return Value

Value returned from the `xpt_action` routine.

See Also

`xpt_action`

C.34 ccmn_send_ccb_wait

Name

ccmn_send_ccb_wait – Sends CCBs to the XPT layer by calling the `xpt_action` routine and sleeps while waiting for the CCB to complete.

Syntax

ccmn_send_ccb_wait(*pd*,*ccb*, *retry*, *sleep-pri*)
PDRV_DEVICE **pd*;
CCB_HEADER **ccb*;
u_char *retry*
int *sleep-pri*

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

ccb Pointer to the CAM Control Block (CCB) header structure to be sent to the `xpt_action` routine to handle the request.

retry Indicates whether this request is a retry of a request that is already on the active queue. A 1 indicates RETRY, and a 0 (zero) indicates NOT_RETRY.

sleep-pri Specifies the priority at which to sleep.

Description

The `ccmn_send_ccb_wait` routine sends CCBs to the XPT layer by calling the `xpt_action` routine. Then, it calls `sleep` to wait for the CCB to complete. The routine sleeps on the address of the CCB at the priority specified by *sleep-pri*. This routine requires the callback completion function for the SCSI I/O CCB to issue a wakeup call on the address of the CCB. The `ccmn_send_ccb_wait` routine should only be called to send SCSI I/O CCBs to the XPT layer. This routine must be called with the Peripheral Device Structure locked.

For SCSI I/O CCBs that are not retries, the request is placed on the active queue. If the CCB is a tagged request and the tag queue size for the device has been reached, the request is placed on the tagged pending queue so that the request can be sent to the XPT at a later time. A high-water mark of half the queue depth for the SCSI device is used for tagged requests so that other initiators on the SCSI bus will not be blocked from using the device.

Return Value

The following values can be returned:

Value	Description
EINTR	The sleep was interrupted by a signal. This status can only occur if the sleep-priority is interruptible.
0	The CCB has completed either because it received the return value from `xpt_action` or because a wakeup was issued by the callback completion function.

See Also

`xpt_action`

C.35 ccmn_start_unit

Name

ccmn_start_unit – Creates a SCSI I/O CCB for the START UNIT command, sends it to the XPT for processing, and sleeps waiting for it to complete

Syntax

ccmn_start_unit(*pd*, *sense_len*, *cam_flags*, *comp_func*, *tag_action*, *timeout*)
PDRV_DEVICE **pd*;
u_short *sense_len*;
u_long *cam_flags*;
void (**comp_func*) ();
u_char *tag_action*;
u_long *timeout*;

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

sense_len Length of the sense data buffer to be returned on autosense, which is predefined as 64 bytes in the DEC_AUTO_SENSE_SIZE environment variable but can be larger.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer

Flag Name	Description
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

comp_func SCSI device driver I/O callback completion function. This pointer may be NULL if the CAM DISABLE CALLBACK bit is set in the CAM FLAGS field.

tag_action Type of action to perform for tagged requests:

CAM_SIMPLE_QTAG	Tag for simple queue
CAM_HEAD_QTAG	Tag for head of queue
CAM_ORDERED_QTAG	Tag for ordered queue

timeout Timeout for the request in seconds. A value of 0 (zero) indicates the default, which is five seconds.

Description

The `ccmn_start_unit` routine creates a SCSI I/O CCB for the START UNIT command and sends it to the XPT for processing.

The `ccmn_start_unit` routine calls the `ccmn_io_ccb_bld` routine to obtain a SCSI I/O CCB structure. The `ccmn_start_unit` routine calls the `ccmn_send_ccb_wait` routine to send the SCSI I/O CCB to the XPT and wait for it to complete. The `ccmn_start_unit` routine sleeps at a non-interruptible priority. It requires the callback completion function to issue a wakeup call on the address of the CCB.

Return Value

CCB_SCSIIO pointer

See Also

`ccmn_io_ccb_bld`, `ccmn_send_ccb_wait`

C.36 ccmn_term_ccb_bld

Name

ccmn_term_ccb_bld – Creates a TERMINATE I/O CCB and sends it to the XPT

Syntax

ccmn_term_ccb_bld(*dev*, *cam_flags*, *term_ccb*)
dev_t *dev*;
u_long *cam_flags*;
CCB_HEADER **term_ccb*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
CAM_DIR_RESV	Data direction (00: reserved)
CAM_DIR_IN	Data direction (01: DATA IN)
CAM_DIR_OUT	Data direction (10: DATA OUT)
CAM_DIR_NONE	Data direction (11: no data)
CAM_DIS_AUTOSENSE	Disable autosense feature
CAM_SCATTER_VALID	Scatter/gather list is valid
CAM_DIS_CALLBACK	Disable callback feature
CAM_CDB_LINKED	CCB contains linked CDB
CAM_QUEUE_ENABLE	SIM queue actions are enabled
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous

Flag Name	Description
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

term_ccb Pointer to the CAM Control Block (CCB) header structure to terminate.

Description

The `ccmn_term_ccb_bld` routine creates a TERMINATE I/O CCB and sends it to the XPT. The routine calls the `ccmn_get_ccb` routine to allocate a CCB structure and fill in the common portion of the CCB header. The routine fills in the CCB to be terminated and calls the `ccmn_send_ccb` routine to send the CCB structure to the XPT. The request is carried out immediately, so it is not placed on the device driver's active queue.

Return Value

CCB_TERMIO pointer

See Also

`ccmn_get_ccb`, `ccmn_send_ccb`

C.37 ccmn_term_que

Name

ccmn_term_que – Sends a TERMINATE I/O CCB request for each SCSI I/O CCB on the active queue

Syntax

ccmn_term_que(*pd*)
PDRV_DEVICE **pd*;

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

Description

The `ccmn_term_que` routine sends a TERMINATE I/O CCB request for each SCSI I/O CCB on the active queue. This routine must be called with the Peripheral Device Structure locked.

The `ccmn_term_que` routine calls the `ccmn_term_ccb_bld` routine to create a TERMINATE I/O CCB for the first active CCB on the active queue and send it to the XPT. It calls the `ccmn_send_ccb` routine to send the TERMINATE I/O CCB for each of the other CCBs on the active queue that are marked as active to the XPT. The `ccmn_term_que` routine then calls the `ccmn_rel_ccb` routine to return the TERMINATE I/O CCB to the XPT.

Return Value

None

See Also

`ccmn_rel_ccb`, `ccmn_send_ccb`

C.38 ccmn_tur

Name

ccmn_tur – Creates a SCSI I/O CCB for the TEST UNIT READY command, sends it to the XPT for processing, and sleeps while waiting for it to complete.

Syntax

ccmn_tur(*pd*, *sense_len*, *cam_flags*, *comp_func*, *tag_action*, *timeout*)
PDRV_DEVICE **pd*;
u_short *sense_len*;
u_long *cam_flags*;
void (**comp_func*) ();
u_char *tag_action*;
u_long *timeout*;

Arguments

pd Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

sense_len Length of the sense data buffer to be returned on autosense, which is predefined as 64 bytes in the DEC_AUTO_SENSE_SIZE environment variable but can be larger.

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
`CAM_DIR_RESV`	Data direction (00: reserved)
`CAM_DIR_IN`	Data direction (01: DATA IN)
`CAM_DIR_OUT`	Data direction (10: DATA OUT)
`CAM_DIR_NONE`	Data direction (11: no data)
`CAM_DIS_AUTOSENSE`	Disable autosense feature
`CAM_SCATTER_VALID`	Scatter/gather list is valid
`CAM_DIS_CALLBACK`	Disable callback feature
`CAM_CDB_LINKED`	CCB contains linked CDB
`CAM_QUEUE_ENABLE`	SIM queue actions are enabled

Flag Name	Description
CAM_CDB_POINTER	CDB field contains pointer
CAM_DIS_DISCONNECT	Disable disconnect
CAM_INITIATE_SYNC	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
CAM_DIS_SYNC	Disable synchronous mode, go to asynchronous
CAM_SIM_QHEAD	Place CCB at head of SIM queue
CAM_SIM_QFREEZE	Return SIM queue to frozen state
CAM_SIM_QFRZDIS	Disable the SIM Q frozen state
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

comp_func SCSI device driver I/O callback completion function. This pointer may be NULL if the CAM DISABLE CALLBACK bit is set in the CAM FLAGS field.

tag_action Type of action to perform for tagged requests:

`CAM_SIMPLE_QTAG`	Tag for simple queue
`CAM_HEAD_QTAG`	Tag for head of queue
`CAM_ORDERED_QTAG`	Tag for ordered queue

timeout Timeout for the request in seconds. A value of 0 (zero) indicates the default, which is five seconds.

Description

The `ccmn_tur` routine creates a SCSI I/O CCB for the TEST UNIT READY command and sends it to the XPT for processing.

The `ccmn_tur` routine calls the `ccmn_io_ccb_bld` routine to obtain a SCSI I/O CCB structure. The `ccmn_tur` routine calls the `ccmn_send_ccb_wait` routine to send the SCSI I/O CCB to the XPT and waits for it to complete. The `ccmn_tur` routine sleeps at a non-interruptible priority. It requires the callback completion function to issue a wakeup call on the address of the CCB.

Return Value

CCB_SCSIIO pointer

See Also

`ccmn_io_ccb_bld`, `ccmn_send_ccb_wait`

C.39 cdbg_CamFunction

Name

cdbg_CamFunction – Reports CAM XPT function codes

Syntax

char * cdbg_CamFunction(*cam_function*, *report_format*)
register u_char *cam_function*;
int *report_format*;

Arguments

cam_function The entry from the CAM XPT Function Code Table.

report_format The format of the message text returned, which can be CDBG_BRIEF or CDBG_FULL.

Description

The `cdbg_CamFunction` routine reports CAM XPT function codes. Program constants are defined to allow either the function code name only or a brief explanation to be printed. The XPT function codes are defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

Returns a character pointer to a text string.

C.40 cdbg_CamStatus

Name

cdbg_CamStatus – Decodes CAM CCB status codes

Syntax

char * cdbg_CamStatus(*cam_status*, *report_format*)
register u_char *cam_status*;
int *report_format*;

Arguments

cam_status The information from the CAM SCSI I/O CCB.

report_format The format of the message text returned, which can be CDBG_BRIEF or CDBG_FULL.

Description

The `cdbg_CamStatus` routine decodes CAM CCB status codes. Program constants are defined to allow either the status code name only or a brief explanation to be printed. The CAM status codes are defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

Returns a character pointer to a text string.

C.41 cdbg_DumpABORT

Name

cdbg_DumpABORT – Dumps the contents of an ABORT CCB

Syntax

void cdbg_DumpABORT(*ccb*)
register CCB_ABORT **ccb*;

Arguments

ccb Pointer to the ABORT CCB.

Description

The `cdbg_DumpABORT` routine dumps the contents of an ABORT CCB. The ABORT CCB is defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

None

C.42 cdbg_DumpBuffer

Name

void cdbg_DumpBuffer – Dumps the contents of a data buffer in hexadecimal bytes

Syntax

void cdbg_DumpBuffer(*buffer*, *size*)
char **buffer*;
register int *size*;

Arguments

buffer	SCSI/CAM peripheral buffer pointer.
size	Size of buffer in bytes.

Description

The `cdbg_DumpBuffer` routine dumps the contents of a data buffer in hexadecimal bytes. The calling routine must display a header line. The format of the dump is 16 bytes per line.

Return Value

None

C.43 cdbg_DumpCCBHeader

Name

cdbg_DumpCCBHeader – Dumps the contents of a CAM Control Block (CCB) header structure

Syntax

void cdbg_DumpCCBHeader(*ccb*)
register CCB_HEADER **ccb*;

Arguments

ccb Pointer to the CAM Control Block (CCB) header structure.

Description

The `cdbg_DumpCCBHeader` routine dumps the contents of a CAM Control Block (CCB) header structure. The CAM Control Block (CCB) header structure is defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

None

C.44 cdbg_DumpCCBHeaderFlags

Name

cdbg_DumpCCBHeaderFlags – Dumps the contents of the `cam_flags` member of a CAM Control Block (CCB) header structure

Syntax

void cdbg_DumpCCBHeaderFlags(*cam_flags*)
register u_long *cam_flags*;

Arguments

cam_flags The *cam_flags* flag names and their bit definitions are listed in the table that follows:

Flag Name	Description
`CAM_DIR_RESV`	Data direction (00: reserved)
`CAM_DIR_IN`	Data direction (01: DATA IN)
`CAM_DIR_OUT`	Data direction (10: DATA OUT)
`CAM_DIR_NONE`	Data direction (11: no data)
`CAM_DIS_AUTOSENSE`	Disable autosense feature
`CAM_SCATTER_VALID`	Scatter/gather list is valid
`CAM_DIS_CALLBACK`	Disable callback feature
`CAM_CDB_LINKED`	CCB contains linked CDB
`CAM_QUEUE_ENABLE`	SIM queue actions are enabled
`CAM_CDB_POINTER`	CDB field contains pointer
`CAM_DIS_DISCONNECT`	Disable disconnect
`CAM_INITIATE_SYNC`	Attempt synchronous data transfer, after issuing Synchronous Data Transfer Request (SDTR)
`CAM_DIS_SYNC`	Disable synchronous mode, go to asynchronous
`CAM_SIM_QHEAD`	Place CCB at head of SIM queue
`CAM_SIM_QFREEZE`	Return SIM queue to frozen state
`CAM_SIM_QFRZDIS`	Disable the SIM Q frozen state

Flag Name	Description
CAM_ENG_SYNC	Flush residual bytes from HBA data engine before terminating I/O
CAM_ENG_SGLIST	Scatter/gather list is for HBA engine
CAM_CDB_PHYS	CDB pointer is physical address
CAM_DATA_PHYS	Scatter/gather/buffer data pointers are physical address
CAM_SNS_BUF_PHYS	Autosense data pointer is physical address
CAM_MSG_BUF_PHYS	Message buffer pointer is physical address
CAM_NXT_CCB_PHYS	Next CCB pointer is physical address
CAM_CALLBCK_PHYS	Callback function pointer is physical address
CAM_DATAB_VALID	Data buffer valid
CAM_STATUS_VALID	Status buffer valid
CAM_MSGB_VALID	Message buffer valid
CAM_TGT_PHASE_MODE	SIM will run in phase mode
CAM_TGT_CCB_AVAIL	Target CCB available
CAM_DIS_AUTODISC	Disable autodisconnect
CAM_DIS_AUTOSRP	Disable autosave/restore pointers

Description

The `cdbg_DumpCCBHeaderFlags` routine dumps the contents of the `cam_flags` member of a CAM Control Block (CCB) header structure. The CAM Control Block (CCB) header structure is defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

None

C.45 cdbg_DumpInquiryData

Name

cdbg_DumpInquiryData – Dumps the contents of an ALL_INQ_DATA structure

Syntax

void cdbg_DumpInquiryData(*inquiry*)
register ALL_INQ_DATA **inquiry*;

Arguments

inquiry Pointer to the ALL_INQ_DATA structure.

Description

The `cdbg_DumpInquiryData` routine dumps the contents of an ALL_INQ_DATA structure. The ALL_INQ_DATA structure is defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

Return Value

None

C.46 cdbg_DumpPDRVws

Name

cdbg_DumpPDRVws – Dumps the contents of a SCSI/CAM Peripheral Device Driver Working Set Structure

Syntax

void cdbg_DumpPDRVws(*pws*)
register PDRV_WS **pws*;

Arguments

pws Pointer to the SCSI/CAM Peripheral Device Driver Working Set Structure.

Description

The `cdbg_DumpPDRVws` routine dumps the contents of a SCSI/CAM Peripheral Device Driver Working Set Structure. The SCSI/CAM Peripheral Device Driver Working Set Structure is defined in the `/usr/sys/include/io/cam/pdrv.h` file.

Return Value

None

C.47 cdbg_DumpSCSIIO

Name

cdbg_DumpSCSIIO – Dumps the contents of a SCSI I/O CCB

Syntax

void cdbg_DumpSCSIIO(*ccb*)
register CCB_SCSIIO **ccb*;

Arguments

ccb	Pointer to the SCSI I/O CCB structure.

Description

The `cdbg_DumpSCSIIO` routine dumps the contents of a SCSI I/O CCB. The SCSI I/O CCB is defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

None

C.48 cdbg_DumpTERMIO

Name

cdbg_DumpTERMIO – Dumps the contents of a TERMINATE I/O CCB

Syntax

void cdbg_DumpTERMIO(*ccb*)
register CCB_TERMIO **ccb*;

Arguments

ccb Pointer to the TERMINATE I/O CCB.

Description

The `cdbg_DumpTERMIO` routine dumps the contents of a TERMINATE I/O CCB. The TERMINATE I/O CCB is defined in the `/usr/sys/include/io/cam/cam.h` file.

Return Value

None

C.49 cdbg_GetDeviceName

Name

cdbg_GetDeviceName – Returns a pointer to a character string describing the `dtype` member of an ALL_INQ_DATA structure

Syntax

char * cdbg_GetDeviceName(*device_type*)
register *device_type*;

Arguments

device_type SCSI device type value from Inquiry data.

Description

The `cdbg_GetDeviceName` routine returns a pointer to a character string describing the `dtype` member of an ALL_INQ_DATA structure. The ALL_INQ_DATA structure is defined in the `/usr/sys/include/io/cam/scsi_all.h` file.

Return Value

Returns a character pointer to a text string.

C.50 cdbg_ScsiStatus

Name

cdbg_ScsiStatus – Reports SCSI status codes

Syntax

char * cdbg_ScsiStatus(*scsi_status*, *report_format*)
register u_char *scsi_status*;
int *report_format*;

Arguments

scsi_status The SCSI status from the CAM SCSI I/O CCB.

report_format
The format of the message text returned, which can be CDBG_BRIEF or CDBG_FULL.

Description

The `cdbg_ScsiStatus` routine reports SCSI status codes. Program constants are defined to allow either the status code name only or a brief explanation to be printed. The SCSI status codes are defined in the `/usr/sys/include/io/cam/scsi_status.h` file.

Return Value

Returns a character pointer to a text string.

C.51 cdbg_SystemStatus

Name

cdbg_SystemStatus – Reports system error codes

Syntax

char * cdbg_SystemStatus(*errno*)
int *errno*;

Arguments

errno The error number.

Description

The cdbg_SystemStatus routine reports system error codes. The system error codes are defined in the `/usr/sys/include/sys/errno.h` file.

Return Value

Returns a character pointer to a text string.

C.52 cgen_async

Name

cgen_async – Handles notification of asynchronous events

Syntax

void cgen_async(*opcode*, *path_id*, *target*, *lun*, *buf_ptr*, *data_cnt*)
u_long *opcode*;
u_char *path_id*;
u_char *target*;
u_char *lun*;
caddr_t *buf_ptr*;
u_char *data_cnt*;

Arguments

opcode	SCSI asynchronous callback operation code.
path_id	SCSI target's bus controller number.
target	SCSI target's ID number.
lun	SCSI target's logical unit number.
buf_ptr	Buffer address for Asynchronous Event Notification (AEN).
data_cnt	Number of bytes the XPT had to transfer from the SIM's buffer or the limit of the SCSI/CAM peripheral buffer.

Description

The `cgen_async` routine handles notification of asynchronous events. The routine is called when an Asynchronous Event Notification(AEN), Bus Device Reset (BDR), or Bus Reset (BR) occurs. The routine sets the CGEN_RESET_STATE flag and clears the CGEN_RESET_PEND_STATE flag for BDRs and bus resets. The routine sets the CGEN_UNIT_ATTEN_STATE flag for AENs.

Return Value

None

C.53 cgen_attach

Name

cgen_attach – Is called for each bus, target, and LUN after the `cgen_slave` routine returns SUCCESS

Syntax

cgen_attach(*device*)
struct device **device*;

Arguments

device	Pointer to the device information contained in the `device` structure.

Description

The `cgen_attach` routine is called for each bus, target, and LUN after the `cgen_slave` routine returns SUCCESS. The routine calls the `ccmn_open_unit` routine, passing the bus, target, and LUN information.

The `cgen_attach` routine calls the `ccmn_close_unit` routine to close the device. If a device of the specified type is found, the device identification string is printed.

Return Value

PROBE_FAILURE
PROBE_SUCCESS

See Also

`ccmn_close_unit`, `ccmn_open_unit`, `cgen_slave`

C.54 cgen_ccb_chkcond

Name

cgen_ccb_chkcond – Decodes the autosense data for a device driver

Syntax

cgen_ccb_chkcond(*pdrv_dev*, *ccb*)
PDRV_DEVICE **pdrv_dev*;
CCB_SCSIIO **ccb*;

Arguments

pdrv_dev Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.

ccb Pointer to the SCSI I/O CCB structure.

Description

The `cgen_ccb_chkcond` routine decodes the autosense data for a device driver and returns the appropriate status to the calling routine. The routine is called when a SCSI I/O CCB is returned with a CAM status of CAM_REQ_CMP_ERR (request completed with error) and a SCSI status of SCSI_STAT_CHECK_CONDITION. The routine also sets the appropriate flags in the Generic-Specific Structure.

Return Value

An integer indicating one of the following values:

Flag Name	Description
`CHK_CHK_NOSENSE`	Request sense did not complete without error. Sense buffer contents cannot be used to determine error condition.
`CHK_SENSE_NOT_VALID`	Valid bit in sense buffer is not set; sense data is useless.
`CHK_EOM`	End of media detected.
`CHK_FILEMARK`	Filemark detected.
`CHK_ILI`	Incorrect record length detected.

Flag Name	Description
CHK_NOSENSE_BITS	Sense key equals no sense, but there are no bits set in byte 2 of sense data.
CHK_SOFTERR	Soft error detected; corrected by unit.
CHK_NOT_READY	Unit is not ready.
CHK_HARDERR	Unit has detected a hard error.
CHK_UNIT_ATTEN	Unit has either had media change or just powered up.
CHK_DATA_PROT	Unit is write protected.
CHK_UNSUPPORTED	Sense key that is unsupported has been returned.
CHK_CMD_ABORTED	Unit aborted this command.
CHK_INFORMATIONAL	Unit is reporting informational message.
CHK_UNKNOWN_KEY	Unit has returned sense key that is not supported by SCSI 2 specification.

C.55 cgen_close

Name

cgen_close – Closes the device

Syntax

cgen_close(*dev*, *flags*, *fmt*)
dev_t *dev*;
int *flags*;
int *fmt*;

Arguments

dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.
flags	Flags set when a file is open.
fmt	Indicates whether to close the character or block device.

Description

The `cgen_close` routine closes the device. The routine checks any device flags that are defined to see if action is required, such as rewind on close or release the unit. The `cgen_close` closes the device by calling the `ccmn_close_unit` routine.

Return Value

The `cgen_close` routine returns GENERIC_SUCCESS upon successful completion.

Diagnostics

The `cgen_close` routine fails under the following condition:

[ENOMEM]	Resource problem

See Also

`ccmn_close_unit`

C.56 cgen_done

Name

cgen_done – Serves as the entry point for all nonread and nonwrite I/O callbacks

Syntax

cgen_done(*ccb*)
CCB_SCSIIO **ccb*;

Arguments

ccb Pointer to the SCSI I/O CCB structure.

Description

The `cgen_done` routine is the the entry point for all nonread and nonwrite I/O callbacks. The generic device driver uses two callback entry points, one for all nonuser I/O requests and one for all user I/O requests. The SCSI/CAM peripheral device driver writer can declare multiple callback routines for each type of command and can fill the CCB with the address of the appropriate callback routine.

This is a generic routine for all nonread and nonwrite SCSI I/O CCBs. The SCSI I/O CCB should not contain a pointer to a `buf` structure in the `cam_req_map` member of the structure. If it does, then a wake-up call is issued on the address of the CCB and the error is reported. If the SCSI I/O CCB does not contain a pointer to a `buf` structure in the `cam_req_map` member, then a wake-up call is issued on the address of the CCB and the CCB is removed from the active queues. No CCB completion status is checked because that is the responsibility of the routine that created the CCB and is waiting for completion status. When this routine is entered, context is on the interrupt stack and the driver cannot sleep waiting for an event.

Return Value

None

C.57 cgen_ioctl

Name

cgen_ioctl – Handles user process requests for specific actions other than read, write, open, or close for SCSI tape devices

Syntax

cgen_ioctl(*dev*, *cmd*, *data*, *flags*)
dev_t *dev*;
int *cmd*;
caddt_t *data*;
int *flags*;

Arguments

dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.
cmd	The ioctl command, DEVIOCGET.
data	Pointer to the kernel copy of the structure passed by the user process.
flags	User process flags.

Description

The `cgen_ioctl` routine handles user process requests for specific actions other than read, write, open, or close for SCSI tape devices. The routine currently issues a DEVIOCGET `ioctl` command for the device, which fills out the `devget` structure passed in, and then calls the `cgen_mode_sns` routine which issues a SCSI_MODE_SENSE to the device to determine the device's state. The routine then calls the `ccmn_rel_ccb` routine to release the CCB. When the call to `cgen_mode_sns` completes, the `cgen_ioctl` routine fills out the rest of the `devget` structure based on information contained in the mode sense data.

Return Value

[EINVAL] Invalid command.

See Also

`ccmn_rel_ccb`, `cgen_mode_sns`, `ioctl`(2)

C.58 cgen_iodone

Name

cgen_iodone – Serves as the entry point for all read and write I/O callbacks

Syntax

cgen_iodone(*ccb*)
CCB_SCSIIO **ccb*;

Arguments

ccb	Pointer to the SCSI I/O CCB structure.

Description

The `cgen_iodone` routine is the entry point for all read and write I/O callbacks. This is a generic routine for all read and write SCSI I/O CCBs. The SCSI I/O CCB should contain a pointer to a `buf` structure in the `cam_req_map` member of the structure. If it does not, then a wake-up call is issued on the address of the CCB and the error is reported. If the SCSI I/O CCB does contain a pointer to a `buf` structure in the `cam_req_map` member, as it should, then the completion status is decoded. Depending on the CCB's completion status, the correct fields within the `buf` structure are filled out.

The device's active queues may need to be aborted because of errors or because the device is a sequential access device and the transaction was an asynchronous request.

The CCB is removed from the active queues by a call to the `ccmn_rem_ccb` routine and is released back to the free CCB pool by a call to the `ccmn_rel_ccb` routine. When the `cgen_iodone` routine is entered, context is on the interrupt stack and the driver cannot sleep waiting for an event.

Return Value

None

See Also

ccmn_rem_ccb, ccmn_rel_ccb

C.59 cgen_minphys

Name

cgen_minphys – Compares the `b_bcount` with the maximum transfer limit for the device

Syntax

cgen_minphys(*bp*)
register struct buf **bp*;

Arguments

bp A `buf` structure pointer, which is used for request mapping.

Description

The `cgen_minphys` routine compares the `b_bcount` with the maximum transfer limit for the device. The routine compares the `b_bcount` field in the `buf` structure with the maximum transfer limit for the device in the Device Descriptor Structure. The count is adjusted if it is greater than the limit.

Return Value

None

C.60 cgen_mode_sns

Name

cgen_mode_sns – Issues a SCSI_MODE_SENSE command to the unit defined

Syntax

cgen_mode_sns(*pdrv_dev*, *action*, *done*, *page_code*, *page_ctrl*, *sleep*)
PDRV_DEVICE **pdrv_dev*;
CGEN_ACTION **action*;
void (**done*) ();
u_char *page_code*;
u_char *page_ctrl*;
u_long *sleep*;

Arguments

pdrv_dev	Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.
action	Pointer to the caller's Generic Action Structure.
done	The address of the completion routine to be called when the SCSI command completes.
page_code	The user process's target page.
page_ctrl	The page control settings field.
sleep	Whether or not the GENERIC_SLEEP flag is set.

Description

The `cgen_mode_sns` routine issues a SCSI_MODE_SENSE command to the unit defined. The CGEN_ACTION structure is filled in for the calling routine based on the completion status of the CCB.

Return Value

NULL – command could not be issued
CCB_SCSIIO pointer

See Also

`ccmn_ccb_status`

C.61 cgen_open

Name

cgen_open – Is called by the kernel when a user process requests an open of the device

Syntax

cgen_open(*dev*, *flags*, *fmt*)
dev_t *dev*;
int *flags*;
int *fmt*;

Arguments

dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.
flags	Flags set when a file is open.
fmt	Indicates whether to open the character or block device.

Description

The `cgen_open` routine is called by the kernel when a user process requests an open of the device. The `cgen_open` routine calls the `ccmn_open_unit` routine, which manages the SMP_LOCKS and, if passed the exclusive use flag for SCSI devices, makes sure that no other process has opened the device. If the `ccmn_open_unit` routine returns success, the necessary data structures are allocated.

The `cgen_open` routine calls the `ccmn_sasy_ccb_bld` routine to register for asynchronous event notification for the device. The `cgen_open` routine then enters a `for` loop based on the power-up time specified in the Device Descriptor Structure for the device. Within the loop, calls are made to the `cgen_ready` routine, which calls the `ccmn_tur` routine to issue a TEST UNIT READY command to the device.

The `cgen_open` routine calls the `ccmn_rel_ccb` routine to release the CCB. The `cgen_open` routine checks certain state flags for the device to decide whether to send the initial SCSI mode select pages to the device. Depending on the setting of the state flags CGEN_UNIT_ATTEN_STATE and CGEN_RESET_STATE, the `cgen_open` routine calls the `cgen_open_sel` routine for each mode select page to be sent to the device. The `cgen_open_sel` routine fills out the Generic Action

Structure based on the completion status of the CCB for each mode select page it sends.

Return Value

The `cgen_open` routine returns GENERIC_SUCCESS upon successful completion.

Diagnostics

The `cgen_open` routine fails under the following conditions:

[EBUSY]	The device is already opened and the exclusive use bit is set.
[ENOMEM]	Resource problem
[EINVAL]	The *scsi_dev_type* parameter does not match the device type in the Inquiry data returned by GET DEVICE TYPE CCB. The *scsi_dev_type* was not configured.
[ENXIO]	The device does not exist.
[EIO]	Check device conditions.

See Also

`ccmn_close_unit`, `ccmn_open_unit`, `ccmn_rel_ccb`, `ccmn_sasy_ccb_bld`, `ccmn_tur`, `cgen_open_sel`, `cgen_close`

C.62 cgen_open_sel

Name

cgen_open_sel – Issues a SCSI_MODE_SELECT command to the SCSI device

Syntax

cgen_open_sel(*pdrv_dev*, *action*, *ms_index*, *done*, *sleep*)
PDRV_DEVICE **pdrv_dev*;
CGEN_ACTION **action*;
u_long *ms_index*;
void (**done*) ();
u_long *sleep*;

Arguments

pdrv_dev	Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.
action	Pointer to the caller's Generic Action Structure.
ms_index	An index into a page in the Mode Select Table that is pointed to in the Device Descriptor Structure.
done	The address of the completion routine to be called when the SCSI command completes.
sleep	Whether or not the GENERIC_SLEEP flag is set.

Description

The `cgen_open_sel` routine issues a SCSI_MODE_SELECT command to the SCSI device. The mode select data sent to the device is based on the data contained in the Mode Select Table Structure for the device, if one is defined. The CGEN_ACTION structure is filled in for the calling routine based on the completion status of the CCB.

The `cgen_open_sel` routine calls the `ccmn_mode_select` routine to create a SCSI I/O CCB and send it to the XPT for processing.

Return Value

None

See Also

ccmn_ccb_status, ccmn_mode_select

C.63 cgen_read

Name

cgen_read – Handles synchronous read requests for user processes

Syntax

cgen_read(*dev*, *uio*)
dev_t *dev*;
struct uio **uio*;

Arguments

dev	The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.
uio	Pointer to the device information contained in the `uio` I/O structure.

Description

The `cgen_read` routine handles synchronous read requests for user processes. It passes the user process requests to the `cgen_strategy` routine. The `cgen_read` routine calls the `ccmn_get_bp` routine to allocate a `buf` structure for the user process read request. When the I/O is complete, the `cgen_read` routine calls the `ccmn_rel_bp` routine to deallocate the `buf` structure.

Return Value

The `cgen_read` routine passes the return from the `physio` routine.

See Also

`ccmn_get_bp`, `ccmn_rel_bp`, `cgen_strategy`

C.64 cgen_ready

Name

cgen_ready – Issues a TEST UNIT READY command to the unit defined

Syntax

cgen_ready(*pdrv_dev*, *action*, *done*, *sleep*)
PDRV_DEVICE **pdrv_dev*;
CGEN_ACTION **action*;
void (**done*) ();
u_long *sleep*;

Arguments

pdrv_dev	Pointer to the CAM Peripheral Device Structure allocated for each SCSI device in the system.
action	Pointer to the caller's Generic Action Structure.
done	The address of the completion routine to be called when the SCSI command completes.
sleep	Whether or not the GENERIC_SLEEP flag is set.

Description

The `cgen_ready` routine issues a TEST UNIT READY command to the unit defined. The routine calls the `ccmn_tur` routine to issue the TEST UNIT READY command and sleeps waiting for command status.

Return Value

None

See Also

`ccmn_tur`

C.65 cgen_slave

Name

cgen_slave – Is called at system boot to initialize the lower levels

Syntax

cgen_slave(*device*, *reg*)
struct device **device*;
caddr_t *reg*;

Arguments

device	Pointer to the device information contained in the `device` structure.
reg	The virtual address of the controller.

Description

The `cgen_slave` routine is called at system boot to initialize the lower levels. The routine also checks the bounds for the unit number to ensure it is within the allowed range and sets the device-configured bit for the device at the specified bus, target, and LUN.

Return Value

PROBE_FAILURE
PROBE_SUCCESS

See Also

`ccmn_close_unit`, `ccmn_init`, `ccmn_open_unit`

C.66 cgen_strategy

Name

cgen_strategy – Handles all I/O requests for user processes

Syntax

cgen_strategy(*bp*)
struct buf **bp*;

Arguments

bp A `buf` structure pointer, which is used for request mapping.

Description

The `cgen_strategy` routine handles all I/O requests for user processes. It performs specific checks, depending on whether the request is synchronous or asynchronous and on the SCSI device type. The `cgen_strategy` routine calls the `ccmn_io_ccb_bld` routine to obtain an initialized SCSI I/O CCB and build either a read or a write command based on the information contained in the `buf` structure. The `cgen_strategy` routine then calls the `ccmn_send_ccb` to place the CCB on the active queue and send it to the XPT layer.

Return Value

[EINVAL] Device not ready.
[EIO]

See Also

`ccmn_io_ccb_bld`, `ccmn_send_ccb`, `cgen_iodone`

C.67 cgen_write

Name

cgen_write – Handles synchronous write requests for user processes

Syntax

cgen_write(*dev*, *uio*)
dev_t *dev*;
struct uio **uio*;

Arguments

dev The major/minor device number pair that identifies the bus number, target ID, and LUN associated with this SCSI device.

uio Pointer to the device information contained in the `uio` I/O structure.

Description

The `cgen_write` routine handles synchronous write requests for user processes. The routine passes the user process requests to the `cgen_strategy` routine. The `cgen_write` routine calls the `ccmn_get_bp` routine to allocate a `buf` structure for the user process write request. When the I/O is complete, the `cgen_write` routine calls the `ccmn_rel_bp` routine to deallocate the `buf` structure.

Return Value

The `cgen_write` routine passes the return from the `physio` routine.

See Also

`ccmn_get_bp`, `ccmn_rel_bp`, `cgen_strategy`

C.68 sim_action

Name

sim_action – Initiates an I/O request from a SCSI/CAM peripheral device driver

Syntax

sim_action(*ccb_hdr*)
CCB_HEADER **ccb_hdr*;

Arguments

ccb_hdr Address of the header for the ccb.

Description

The `sim_action` routine initiates an I/O request from a SCSI/CAM peripheral device driver. The routine is used by the XPT for immediate as well as for queued operations. When the operation completes, the SIM calls back directly to the peripheral driver using the CCB callback address, if callbacks are enabled and the operation is not to be carried out immediately.

The SIM determines whether an operation is to be carried out immediately or to be queued according to the function code of the CCB structure. All queued operations, such as ''Execute SCSI I/O'' (reads or writes), are placed by the SIM on a nexus-specific queue and return with a CAM status of CAM_INPROG.

Some immediate operations, as described in the American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module*, working draft, X3T9.2/90-186, may not be executed immediately. However, all CCBs to be carried out immediately return to the XPT layer immediately. For example, the ABORT CCB command does not always complete synchronously with its call; however, the CCB_ABORT is returned to the XPT immediately. An XPT_RESET_BUS CCB returns to the XPT following the reset of the bus.

Return Value

CAM_REQ_INPROG for queued commands
CAM_REQ_CMP for immediate commands
A valid CAM error value

See Also

American National Standard for Information Systems, *SCSI-2 Common Access Method: Transport and SCSI Interface Module*, working draft, X3T9.2/90-186

C.69 sim_init

Name

sim_init – Initializes the SIM

Syntax

sim_init(*pathid*)
u_long *pathid*;

Arguments

pathid SCSI target's bus controller number.

Description

The `sim_init` routine initializes the SIM. The SIM clears all its queues and releases all allocated resources in response to this call. This routine is called using the function address contained in the CAM_SIM_ENTRY structure. This routine can be called at any time; the SIM layer must ensure that data integrity is maintained.

Return Value

CAM_REQ_CMP

C.70 uagt_close

Name

uagt_close – Handles the close of the User Agent driver

Syntax

uagt_close(*dev*, *flag*)
dev_t *dev*;
int *flag*;

Arguments

dev	The major/minor device number pair that identifies the User Agent.
flag	Unused.

Description

The `uagt_close` routine handles the close of the User Agent driver. For the last close operation for the driver, if any queues are frozen, a RELEASE SIM QUEUE CCB is sent to the XPT layer for each frozen queue detected by the User Agent.

Return Value

None

See Also

`uagt_open`, `xpt_ccb_free`

C.71 uagt_ioctl

Name

uagt_ioctl – Handles the `ioctl` system call for the User Agent driver

Syntax

uagt_ioctl(*dev*, *cmd*, *data*, *flag*)
dev_t *dev*;
register int *cmd*;
caddr_t *data*;
int *flag*;

Arguments

dev The major/minor device number pair that identifies the User Agent.

cmd The ioctl command, UAGT_CAM_IO.

data Pointer to the UAGT_CAM_CCB structure passed by the user process.

flag Unused.

Description

The `uagt_ioctl` routine handles the `ioctl` system call for the User Agent driver. The `ioctl` commands supported are: DEVIOCGET, to obtain the User Agent driver's SCSI device status; UAGT_CAM_IO, the `ioctl` define for calls to the User Agent driver; UAGT_CAM_SINGLE_SCAN, to scan a bus, target, and LUN; and UAGT_CAM_FULL_SCAN, to scan a bus.

For SCSI I/O CCB requests, the user data area is locked before passing the CCB to the XPT. The User Agent sleeps waiting for the I/O to complete and issues an ABORT CCB if a signal is caught while sleeping.

Return Value

The `uagt_ioctl` routine returns a value of 0 (zero) upon successful completion.

Diagnostics

The `uagt_ioctl` routine fails under the following conditions:

[EFAULT]	Copy to or from user space failed.
[EINVAL]	An unsupported cmd value was passed to ioctl(). The CCB copied from the user process contained an invalid XPT function code, or an invalid target or LUN.
[EBUSY]	The maximum allowable number of User Agent requests has been reached (MAX_UAGT_REQ).

See Also

`ioctl`(2), `xpt_action`, `xpt_ccb_alloc`

C.72 uagt_open

Name

uagt_open – Handles the open of the User Agent driver

Syntax

uagt_open(*dev*, *flag*)
dev_t *dev*;
int *flag*;

Arguments

dev	The major/minor device number pair that identifies the User Agent.
flag	Unused.

Description

The `uagt_open` routine handles the open of the User Agent driver.

The character device special file name used for the open is `/dev/cam`.

Return Value

The `uagt_open` routine returns a value of 0 (zero) upon successful completion.

See Also

`uagt_close`, `xpt_init`

C.73 xpt_action

Name

xpt_action – Calls the appropriate XPT/SIM routine

Syntax

I32 xpt_action (*ch*)
CCB_HEADER * *ch*;

Arguments

ch Specifies a pointer to the CAM Control Block (CCB) on which to act.

Description

The `xpt_action` routine calls the appropriate XPT/SIM routine. The routine routes the specified CCB to the appropriate SIM module or to the Configuration driver, depending on the CCB type and on the path ID specified in the CCB. Vendor-unique CCBs are also supported. Those CCBs are passed to the appropriate SIM module according to the path ID specified in the CCB.

Return Value

Upon completion, the `xpt_action` routine returns a valid CAM status value.

See Also

`xpt_ccb_alloc`, `xpt_ccb_free`

C.74 xpt_ccb_alloc

Name

xpt_ccb_alloc – Allocates a CAM Control Block (CCB)

Syntax

CCB_HEADER *xpt_ccb_alloc ()

Arguments

None

Description

The `xpt_ccb_alloc` routine allocates a CAM Control Block (CCB) for use by a SCSI/CAM peripheral device driver. The `xpt_ccb_alloc` routine returns a pointer to a preallocated data buffer large enough to contain any CCB structure. The peripheral device driver uses this structure for its XPT/SIM requests. The routine also ensures that the SIM private data space and peripheral device driver pointer, `cam_pdrv_ptr` , are set up.

Return Value

Upon successful completion, `xpt_ccb_alloc` returns a pointer to a preallocated data buffer. The data buffer returned by `xpt_ccb_alloc` is initialized to be a SCSI I/O CCB. For other types of CCBs, some fields may have to be reinitialized for the specific CCB.

See Also

`xpt_ccb_free`

C.75 xpt_ccb_free

Name

xpt_ccb_free – Frees a previously allocated CCB

Syntax

I32 xpt_ccb_free(*ch*)
CCB_HEADER **ch*;

Arguments

ch Specifies a pointer to the CCB to be freed. This CCB was allocated in a call to `xpt_ccb_alloc`.

Description

The `xpt_ccb_free` routine frees a previously allocated CCB. The routine returns a CCB, previously allocated by a peripheral device driver, to the CCB pool.

Return Value

XPT_CCB_INVALID or CAM_SUCCESS

See Also

`xpt_ccb_alloc`

C.76 xpt_init

Name

xpt_init – Validates the initialized state of the CAM subsystem

Syntax

long xpt_init()

Arguments

None

Description

The `xpt_init` routine validates the initialized state of the CAM subsystem. The routine initializes all global and internal variables used by the CAM subsystem through a call to the Configuration driver. Peripheral device drivers must call this routine either during or prior to their own initialization. The `xpt_init` routine simply returns to the calling SCSI/CAM peripheral device driver if the CAM subsystem was previously initialized.

Return Value

Upon completion, `xpt_init` returns one of the following values:

Return Value	Meaning
CAM_SUCCESS	The `xpt_init` routine initialized the CAM subsystem.
CAM_FAILURE	The `xpt_init` routine did not initialize the CAM subsystem and the CAM subsystem cannot be used.

Sample Generic CAM Peripheral Driver D

This chapter contains a sample generic CAM peripheral driver. There are two sample files: the first contains the `cam_generic.h` header file; the second contains the driver source file `cam_generic.c`.

Example D-1: cam_generic.h

```
==================================================================

/*****************************************************************
==================================================================
 *                                                               *
 *                  Copyright (c) 1990 by                        *
 *          Digital Equipment Corporation, Maynard, MA           *
 *                  All rights reserved.                         *
 *                                                               *
 *   This software is furnished under a license and may be used  *
 *   and copied  only  in accordance with the terms of such      *
 *   license and with the  inclusion  of  the  above  copyright  *
 *   notice.  This software or any other copies thereof may not  *
 *   otherwise made available to any other person.  No title to  *
 *   and ownership of the software is hereby transferred.        *
 *                                                               *
 *                                                               *
 *   The information in this software is subject to change       *
 *   without notice and should not be construed as a commitment  *
 *   by Digital Equipment Corporation.                           *
 *                                                               *
 *   Digital assumes no responsibility for the use or reliability*
 *   of its software on equipment which is not supplied by       *
 *   Digital.                                                    *
 *                                                               *
 *****************************************************************/

/* ----------------------------------------------------------- */

/*
This file contains examples of a CAM generic driver's defines.

Modification History

        Version       Date         Who           Reason

*/
```

Example D-1: (continued)

```
/* ---------------------------------------------------------- */

/* Include Files         */
/*     None              */


/* ---------------------------------------------------------- */

/* Defines               */


/*
The following flags are used in the CGEN_SPECIFIC structure in
member gen_state_flags. The state flags are used to determine and
indicate certain states of the driver and the SCSI unit.
*/

#define CGEN_NOT_READY_STATE            0x00000001
        /* Indicates that the unit was opened with the FNDELAY
         * flag and the unit had a failure during the open, but
         * was seen
         */
#define CGEN_UNIT_ATTEN_STATE           0x00000002
        /* Indicates that a check condition occurred and the
         * sense key was UNIT ATTENTION. This usually indicates
         * that a media change has occurred, but it could
         * indicate power up or reset. Either way, current
         * settings are lost.
         */
#define CGEN_RESET_STATE                0x00000004
        /* Indicates notification of a reset set condition
         * on the device or bus.
         */
#define CGEN_RESET_PENDING_STATE        0x00000008
        /*
         * A reset is pending will be notified shortly
         */
#define CGEN_OPENED_STATE               0x00000010
        /*
         * The unit is opened
         */
#define CGEN_XXX_STATE                  0x00000020
        /*
         * Sample state used in generic driver.
         */

/* ---------------------------------------------------------- */

/*
The following flags are used in the CGEN_SPECIFIC structure in
member gen_flags. The flags are used to determine and indicate
certain conditions of the SCSI unit.
*/
```

Example D-1: (continued)

```
#define CGEN_EOM                                0x00000001
        /* At End of Tape
         */
#define CGEN_OFFLINE                            0x00000002
        /* Indicates the device is returning DEVICE NOT READY
         * in response to a command.
         */
#define CGEN_WRT_PROT                           0x00000004
        /* Hardware write protected or opened read only
         */
#define CGEN_SOFTERR                            0x00000008
        /* Indicates that a soft error has been reported by the
         * SCSI unit.
         */
#define CGEN_HARDERR                            0x00000010
        /* Indicates a hard error has occurred. This flag can be
         * reported to the user process either through an ioctl
         * orby the buf struct being marked as EIO.
         */
#define CGEN_XXX                                0x00000020
        /* Sample flag used in generic driver.
         */
#define CGEN_YYY                                0x00000040
        /* Sample flag used in generic driver.
         */


/* ----------------------------------------------------------- */

/*
Generic Structure Declarations
*/

/* Generic-Specific Structure  */

typedef struct generic_specific {
    u_long gen_flags;          /* flags - EOM, write locked */
    u_long gen_state_flags;   /* STATE - UNIT_ATTEN, RESET etc. */
    u_long gen_resid;          /* Last operation residual count */
}CGEN_SPECIFIC;

/*
 * Generic Action Structure
 *
 * The generic_action struct is passed down to the action
 * routines to be filled in based on success or failure of the
 * command.
 */
typedef struct generic_action {
    CCB_SCSIIO       *ccb;      /* CCB that is returned to caller*/
    long             ret_error;      /* Error code if any*/
    u_long           fatal;          /* Is this considered fatal?*/
    u_long           ccb_status;     /* The CCB status code*/
```

Example D-1: (continued)

```
    u_long          scsi_status;    /* The SCSI error code*/
    u_long          chkcond_error; /* The check condition error*/
}CGEN_ACTION;

/*
 * CGEN_ACTION defines
 * action.fatal flags;
 */
#define ACT_FAILED      0x00000001  /* This action has failed */
#define ACT_RESOURCE    0x00000002  /* Resource problem (memory)*/
#define ACT_PARAMETER   0x00000004  /* Invalid parameter */
#define ACT_RETRY_EXCEDED 0x00000008 /* The retry operation count
                                      * has been exceeded
                                      */


/*
 * CGEN_REL_MEM will examine a SCSI I/O CCB to see if the data
 * buffer pointer is non NULL. If so, the macro will call
 * ccmn_rel_dbuf with the size of the buffer, to release the
 * memory back to the pools.
 */
#define CGEN_REL_MEM(ccb); { \
   if(((CCB_SCSIIO *)(ccb))->cam_data_ptr != (u_char *)NULL ) { \
      ccmn_rel_dbuf(((CCB_SCSIIO *)(ccb))->cam_data_ptr, \
                 ((CCB_SCSIIO *)(ccb))->cam_dxfer_len ); \
      ((CCB_SCSIIO *)(ccb))->cam_data_ptr = (u_char *)NULL; \
      ((CCB_SCSIIO *)(ccb))->cam_dxfer_len = (u_long)NULL; \
   } \
}


/*
 * Maximum I/O size.
 */
#define CGEN_MAXPHYS    (16 * (1024 * 1024))   /* 16 meg */

/*
 * Default time-out value for NON read/write operations
 * (rewind,space)
 */
#define CGEN_DEF_TIMEO 600

/*
 * 5-second time
 */
#define CGEN_TIME_5    5

/*
 * Whether to sleep in the work routines
 */
#define CGEN_SLEEP      0x00000000
#define CGEN_NOSLEEP    0x00000001
```

Example D-1: (continued)

```
/*
 * Success and failure defines
 */
#define CGEN_SUCCESS    00
#define CGEN_FAIL       -1

/*
 * Defines for return values from CGEN_ccb_chkcond
 */
#define CHK_SENSE_NOT_VALID      0x0001
        /* Valid bit is not set in sense */
#define CHK_EOM                 0x0002  /* End of media */
#define CHK_FILEMARK            0x0003  /* File mark detected */
#define CHK_ILI                 0x0004  /* Incorrect length */
#define CHK_NOSENSE_BITS        0x0005  /* NOSENSE key and no bits */
#define CHK_SOFTERR             0x0006  /* Soft error reported */
#define CHK_NOT_READY           0x0007  /* Device is not ready */
#define CHK_HARDERR             0x0008  /* Device reported */
                                               /* hard error */
#define CHK_UNIT_ATTEN          0x0009  /* Unit attention (ready?) */
#define CHK_DATA_PROT           0x000a  /* Write protected */
#define CHK_CMD_ABORTED         0x000b  /* Command has been aborted */
#define CHK_UNSUPPORTED         0x000c  /* We don't handle them */
#define CHK_UNKNOWN_KEY         0x000d  /* Bogus sense key */
#define CHK_CHK_NOSENSE         0x000e  /* Sense Auto sense */
                                               /* valid bit 0 */
#define CHK_INFORMATIONAL       0x000f  /* Informational message.. */

/*
 * Clear the fields in the CCB which will be filled in on a retry
 * of the CCB.
 */
#define CGEN_CLEAR_CCB(ccb)  {                  \
        (ccb)->cam_ch.cam_status = 0;           \
        (ccb)->cam_scsi_status = 0;             \
        (ccb)->cam_resid = 0;                   \
}

#define CGEN_BTOL(ptr, long_val) {              \
        char *p = (char *)(ptr);                \
        union   {                               \
                unsigned char   c[4];           \
                unsigned long   l;              \
        }tmp;                                   \
        tmp.c[3] = *p++;                        \
        tmp.c[2] = *p++;                        \
        tmp.c[1] = *p++;                        \
        tmp.c[0] = *p++;                        \
        (long_val) = tmp.l;                     \
}

#define CGEN_LOCK_OR_STATE(pd, ts, flags) {             \
        int         ipl;                                \
```

Example D-1: (continued)

```
        PDRV_IPLSMP_LOCK( (pd), LK_RETRY, ipl );  \
        (ts)->gen_state_flags |= (flags);         \
        PDRV_IPLSMP_UNLOCK( (pd), ipl );          \
}

#define CGEN_LOCK_OR_FLAGS(pd, ts, flags) {       \
        int         ipl;                          \
        PDRV_IPLSMP_LOCK( (pd), LK_RETRY, ipl );  \
        (ts)->gen_flags |= (flags);               \
        PDRV_IPLSMP_UNLOCK( (pd), ipl );          \
}

#define CGEN_BERROR(buf , count, error ) {        \
        (buf)->b_resid = (count);                 \
        (buf)->b_error = (error);                 \
        (buf)->b_flags |= B_ERROR;                \
}

#define CGEN_NULLCCB_ERR( act_ptr, pd, mod ) {    \
    int         ipl;                              \
        PDRV_IPLSMP_LOCK( (pd), LK_RETRY, ipl );  \
        CAM_ERROR((mod), "NULL CCB returned", CAM_SOFTWARE,  \
                (CCB_HEADER *)NULL, (pd)->pd_dev, \
                (u_char *)NULL);                  \
        PDRV_IPLSMP_UNLOCK((pd), ipl);            \
        (act_ptr)->fatal |= (ACT_RESOURCE | ACT_FAILED);  \
        (act_ptr)->ret_error = ENOMEM;            \
}

/*
You should inplement your own error logging.
*/
#define LOG_ERR         printf

=================================================================
```

The following file contains source code for a generic peripheral driver.

Example D-2: cam_generic.c Source File

```
=================================================================

/****************************************************************
 *                                                              *
 *                   Copyright (c) 1990 by                      *
 *          Digital Equipment Corporation, Maynard, MA          *
 *                   All rights reserved.                       *
 *                                                              *
 *  This software is furnished under a license and may be used  *
 *  and copied only in accordance with the terms of such        *
 *  license and with the inclusion of the above copyright       *
 *  notice.  This software or any other copies thereof may not  *
```

Example D-2: (continued)

```
 *  be provided or otherwise made available to any other person.*
 *  No title to and ownership of the software is hereby          *
 *  transferred.                                                 *
 *                                                               *
 *  The information in this software is subject to change       *
 *  without notice and should not be construed as a commitment  *
 *  by Digital Equipment Corporation.                            *
 *                                                               *
 *  Digital assumes no responsibility for the use or            *
 *  reliability of its software on equipment which is not       *
 *  supplied by Digital.                                         *
 *                                                               *
 *****************************************************************/

/* ----------------------------------------------------------- */

/* cam_generic.c      Version 1.00        Aug. 05, 1991

    This module is the upper layer (class) for a generic SCSI
    device driver.
    The module is an example of a device driver for the CAM
    interface only.

Modification History

    Version    Date     Who    Reason


*/

/* ----------------------------------------------------------- */

/* Include files. */

#include <sys/types.h>
#include <sys/file.h>
#include <sys/param.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <sys/buf.h>
#include <sys/ioctl.h>
#include <sys/mtio.h>
#include <sys/errno.h>
#include <io/common/devio.h>
#include <io/common/devdriver.h>
#include <io/common/iotypes.h>
#include <io/cam/cam_debug.h>
#include <io/cam/cam.h>
#include <io/cam/dec_cam.h>
#include <io/cam/scsi_status.h>
#include <io/cam/scsi_all.h>
#include <io/cam/pdrv.h>
#include <io/cam/scsi_sequential.h>
```

Example D-2: (continued)

```
#include "cam_generic.h"


/* ---------------------------------------------------------- */

/* Local defines. */

void cgen_done();
void cgen_async();
void cgen_iodone();
void ccmn_minphys();
void cgen_strategy();
void cgen_ready();
void cgen_open_sel();
void cgen_mode_sns();
void cgen_minphys();
u_long cgen_ccb_chkcond();


/* ---------------------------------------------------------- */

/* External declarations. */
extern int lbolt;
extern int nCAMBUS;
extern void ccmn_init();
extern long ccmn_open_unit();
extern void ccmn_close_unit();
extern u_long ccmn_send_ccb();
extern void ccmn_rem_ccb();
extern void ccmn_abort_que();
extern void ccmn_term_que();
extern CCB_HEADER *ccmn_getccb();
extern void ccmn_rel_ccb();
extern CCB_SCSIIO *ccmn_io_ccb_bld();
extern CCB_GETDEV *ccmn_gdev_ccb_bld();
extern CCB_SETDEV *ccmn_sdev_ccb_bld();
extern CCB_SETASYNC *ccmn_sasy_ccb_bld();
extern CCB_RELSIM *ccmn_rsq_ccb_bld();
extern CCB_PATHINQ *ccmn_pinq_ccb_bld();
extern CCB_ABORT *ccmn_abort_ccb_bld();
extern CCB_TERMIO *ccmn_term_ccb_bld();
extern CCB_RESETDEV *ccmn_bdr_ccb_bld();
extern CCB_RESETBUS *ccmn_br_ccb_bld();
extern CCB_SCSIIO *ccmn_tur();
extern CCB_SCSIIO *ccmn_mode_select();
extern u_long ccmn_ccb_status();
extern struct buf *ccmn_get_bp();
extern void ccmn_rel_bp();
extern u_char *ccmn_get_dbuf();
extern void ccmn_rel_dbuf();

extern struct device *camdinfo[];
```

Example D-2: (continued)

```
extern struct controller *camminfo[];

extern PDRV_UNIT_ELEM pdrv_unit_table[];


/* ---------------------------------------------------------------------- */

/* Initialized and uninitialized data. */


/* ---------------------------------------------------------------------- */
/* Function description.
 *
 * Routine name cgen_slave
 *
 *  This routine is called at boot. The main purposes of the
 *  routine are to initialize the lower levels, to check the unit
 *  number to make sure it falls within range, and to set the
 *  device-configured bit for this device type at this
 *  bus/target/lun.
 *
 * Call syntax
 *  cgen_slave(attach)
 *      struct device *attach       Pointer to the device struct
 *      caddr_t       reg           Virtual address of controller
 *                                  DO NOT USE
 *
 * Implicit inputs
 *     NONE
 *
 * Implicit outputs
 *     NONE
 *
 * Return values
 *     PROBE_FAILURE
 *     PROBE_SUCCESS
 *
 */

int
cgen_slave(attach, reg)
struct device *attach;              /* Pointer to device struct */
caddr_t reg;               /* Virtual register address - unused */

{

    /*
     * Local variables
     */
    u_long      unit;               /* Unit number */

    PDRV_DEVICE *pdrv_dev;
                           /* Peripheral Device Structure pointer  */
```

Example D-2: (continued)

```
    dev_t          dev;   /* For the PRINTD statements */
    static u_char    module[] = "cgen_slave";    /* Module name */

    /*
     * The UBA_UNIT_TO_DEV_UNIT macro assumes unit has bits
     * 0-2 = lun, bits 3-5 = target id, and 6-7 = bus num.
     */
    dev = makedev(0, MAKEMINOR(UBA_UNIT_TO_DEV_UNIT(attach), 0));
    unit = DEV_UNIT(dev);

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: entry\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

    /*
     * Call common initialization routine because we do not know
     * if the subsystem has been initialized
     */
    ccmn_init();

    if( unit > MAX_UNITS){
        /*
         * Unit number is greater than maximum allowed.
         */
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            (CAMD_GENERIC |CAMD_INOUT),
            ("[%d/%d/%d] %s: Unit number too large %d\n",
            DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
                module, unit));
            return(PROBE_FAILURE);
    }
    /*
     * Set the configured bit in the unit table with the device
     * type of your device - we will use sequential access
     * devices.
     */
 pdrv_unit_table[unit].pu_config |= ( 1 << ALL_DTYPE_SEQUENTIAL);


    /*
     * Call the common open unit routine to see if a device is
     * there.  Shift the unit number left by 4 to move over the
     * device-specific bits such as density, no rewind, disk's
     * partition number, etc.
     */
    unit = (unit << 4);

    /*
     * ccmn_open_unit args = device number (major/minor pair);
     * SCSI device type; exclusive use flag if exclusive access
     * is desired; and the size of the device-specific structure.
     */
```

Example D-2: (continued)

```
    if( ccmn_open_unit( (dev_t)unit, ALL_DTYPE_SEQUENTIAL,
          CCMN_EXCLUSIVE, sizeof(CGEN_SPECIFIC)) != (long)NULL){
        /*
         * Could not open unit.
         */
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: ccmn_open_unit failed\n",
=================================================================
         DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));
        return(PROBE_FAILURE);
    }

    /*
     * Close the unit.
     */
    ccmn_close_unit((dev_t)unit);

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));


    return(PROBE_SUCCESS);
}


/* ---------------------------------------------------------- */
/* Function description.
 *
 * Routine name cgen_attach
 *
 *    This routine is called at boot to find out if there are any
 *    devices at this BUS/TARGET/LUN. If a device is found for
 *    our device type print out unit identification
 *
 * Call syntax
 *  cgen_attach(attach)
 *      struct device *attach          Pointer to the uba struct
 *
 * Implicit inputs
 *      NONE
 *
 * Implicit outputs
 *      NONE
 *
 * Return values
 *      PROBE_FAILURE
 *      PROBE_SUCCESS
 *
 */
```

Example D-2: (continued)

```
int
cgen_attach(attach)
    struct device *attach;      /* Pointer to device struct */
{

    /* Local Varibles */

    PDRV_DEVICE *pdrv_dev;
                            /* Peripheral Device Structure pointer */
    dev_t       dev;    /* For the PRINTD statements */
    u_long      unit;
    static      u_char module[] = "cgen_attach"; /* Module name */

    /*
     * The UBA_UNIT_TO_DEV_UNIT macro assumes unit
     * has bits 0-2 = lun, bits 3-5 = target id,
     * and 6-7 = bus num.
     */
    dev = makedev(0, MAKEMINOR(UBA_UNIT_TO_DEV_UNIT(attach), 0));
    unit = DEV_UNIT(dev);

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: entry\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));


    /*
     * Determine whether a device exists at this address by
     * calling ccmn_open_unit which checks the Equipment Device
     * Table.
     */
    if( ccmn_open_unit(dev, (u_long)ALL_DTYPE_SEQUENTIAL,
        CCMN_EXCLUSIVE, (u_long)sizeof(CGEN_SPECIFIC)) != 0L)   {
        PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            (CAMD_GENERIC ),
            ("[%d/%d/%d] %s: ccmn_open_unit failed\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
        return(PROBE_FAILURE);
    }

    /*
     * Get the pointer to the PDRV_DEVICE structure
     */

 if( (pdrv_dev = GET_PDRV_PTR(dev)) == (PDRV_DEVICE *)NULL)  {
  ccmn_close_unit(dev);
  PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
    (CAMD_GENERIC ),
    ("[%d/%d/%d] %s: No peripheral device structure allocated\n",
    DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
  return(PROBE_FAILURE);
    }
```

Example D-2: (continued)

```
    /* Output the identification string */
    printf(" (%s %s)", pdrv_dev->pd_dev_desc->dd_dev_name,
            pdrv_dev->pd_dev_desc->dd_pv_name);

    /*
     * Close the unit.
     */
    ccmn_close_unit(dev);

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

    return(PROBE_SUCCESS);
}




/* ---------------------------------------------------------- */
/* Function description.
 *
 * Routine name cgen_open
 *
 *   This routine opens the unit.
 *   For a flag of FNDELAY and all errors other than
 *   reservation conflicts and memory resource conflicts, always
 *   return success. This is based on the POSIX standard.
 *
 *   First do a TEST UNIT READY command to see if the device is
 *   ready to use.
 *
 *
 *
 * Call syntax
 *  cgen_open( dev, flags )
 *
 * Implicit inputs
 *      Flags of CGEN_UNIT_ATTEN_STATE, CGEN_RESET_STATE from
 *      last open.
 *
 * Implicit outputs
 *      Flags of CGEN_NOT_READY_STATE, if the FNDELAY flag was
 *      passed in this routine and the device had an error.
 *
 * Return values
 *      CGEN_SUCCESS
 *      EBUSY    Device reserved by another initiator
 *      ENOMEM   Resource problem
 *      EINVAL   CCB problems
 *      ENXIO    Device path problems.
 *      EIO      Device check conditions
```

Example D-2: (continued)

```
 *
 * TO DO:
 */


int
cgen_open(dev, flags)


    dev_t dev;      /* Major/minor number pair */
    int flags;      /* Flags RDONLY, READ, WRITE, FNDELAY, etc. */

{

    /*
     * LOCAL VARIABLES
     */


    PDRV_DEVICE         *pdrv_dev;
                        /* Peripheral Device Structure pointer */

    DEV_DESC            *dev_desc;
                        /* Device Descriptor Structure pointer */

    MODESEL_TBL         *modsel_tab;
                              /* Pointer to Mode Select Table
                               * structure to read for the open.
                               */

    CGEN_SPECIFIC       *gen_spec;
                        /* Generic-Specific Structure pointer */

    CCB_SETASYNC        *ccb_async;
                        /* CAM SET ASYNCHRONOUS CALLBACK CCB */
    CGEN_ACTION         action;
                        /* Generic Action Structure */


    long                ret_val;
                        /* Return value from sub-routines */


    u_long              ready_time;
                        /* Time it takes for this type
                         * unit to become ready (seconds)
                         */


    u_long              state_flags;
                        /* Saved state */
    long                success;
                        /* Test unit ready loop indicator */
    long                fndelay;
```

Example D-2: (continued)

```
                        /* Test unit ready loop indicator */
long                    fatal;
                        /* Test unit ready loop indicator */

int                     i;          /*  For loop counter */
int                     s;          /* For our saved IPL */
int                     s1;         /* Throwaway IPL */
static u_char           module[] = "cgen_open"; /* Module name */


/*
 *END OF LOCAL VARIABLES
 */

PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
    (CAMD_GENERIC |CAMD_INOUT),
    ("[%d/%d/%d] %s: entry\n",
    DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

/*
 * Call peripherial driver common routine to
 * open the device. This routine locks the Peripheral
 * Device Unit Table and makes sure everything matches.
 * Arguments are dev; device_type (tape ,disk,scanner);
 * whether exclusive use or not; size of device-specific
 * struct (CGEN_SPECIFIC).
 *
 * Refer to ccmn_open_unit() for a full description.
 */

ret_val = ccmn_open_unit( dev, ALL_DTYPE_SEQUENTIAL,
            CCMN_EXCLUSIVE, sizeof(CGEN_SPECIFIC) );

if ( ret_val != NULL ){
    /*
     * Return ERRNO based on return value:
     * EBUSY - Device is already opened exclusive use
     * EINVAL- Device types do not match
     * ENXIO - Device does not exist even after rescan.
     */

PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
    (CAMD_GENERIC ),
    ("[%d/%d/%d] %s: Dev failed ccmn_open_unit dev = %d\n",
    DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
    dev));

return( ret_val );

}

/*
 * Can now set up structure pointers
```

Example D-2: (continued)

```
 */

/*
 * Get Peripherial Device Structure pointer
 */
if( (pdrv_dev = GET_PDRV_PTR(dev)) == (PDRV_DEVICE *)NULL){
    /*
     * This should not happen - no PDRV_DEVICE struct.
     */
    LOG_ERROR("Implement your error logging");

    return(ENOMEM);
}


/*
 * Get pointer to Device Descriptor Structure
 */
dev_desc = pdrv_dev->pd_dev_desc;

/*
 * Get pointer to Mode Select Table Structure
 */
modsel_tab = dev_desc->dd_modesel_tbl;


/*
 * Get pointer to Generic-Specific Structure
 */

if( (gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific) ==
            (CGEN_SPECIFIC *)NULL){
    PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
    LOG_ERROR("Implement your error logging");
    PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
    return(ENOMEM);
}

PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
    (CAMD_GENERIC ),
    ("[%d/%d/%d] %s: state flags = %X\n",
    DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
    gen_spec->gen_state_flags));


/*
 * Initialize state flags and regular flags
 * The flags of CGEN_UNIT_ATTEN_STATE and CGEN_RESET_STATE,
 * will be preserved across opens if the open has failed due
 * to device problems.
 */

PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
```

Example D-2: (continued)

```
state_flags = gen_spec->gen_state_flags;

if((state_flags &
    (CGEN_UNIT_ATTEN_STATE | CGEN_RESET_STATE )) != NULL) {

    /*
     * Flags for known state.
     */
    gen_spec->gen_flags = CGEN_XXX;
}
else {
    /*
     * Do you want to save any flags set from
     * the last unit open. CGEN_XXX and CGEN_YYY are
     * example flags.
     */
     gen_spec->gen_flags &= (CGEN_XXX | CGEN_YYY);

}

PDRV_IPLSMP_UNLOCK( pdrv_dev, s);


/*
 * Register for a SET ASYNCHRONOUS CALLBACK CCB
 * The events to notice are:
 *
 * Bus Device resets, SCSI_Attens, Asynchronous Event
 * Notifications (AEN), Bus Resets
 */
 ccb_async = ccmn_sasy_ccb_bld( dev, (u_long)CAM_DIR_NONE,
                    (AC_SENT_BDR |
                    AC_SCSI_AEN | AC_BUS_RESET), cgen_async,
                    (u_char *)NULL, NULL);

/*
 * This command is carried out immediately, so status should
 * be valid
 */
if( CAM_STATUS(ccb_async) != CAM_REQ_CMP ){
    /*
     * The SET ASYNCHRONOUS CALLBACK CCB can not be
     * registered.  If FNDELAY is set, continue.
     */
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
      (CAMD_GENERIC ),
      ("[%d/%d/%d] %s: Can't set async ccb status = %x\n",
       Cannot SET ASYNCHRONOUS CALLBACK CCB; status =
      DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
      ccb_async->cam_ch.cam_status));

    LOG_ERROR("Implement your error logging");
```

Example D-2: (continued)

```
        /*
         * Release the CCB
         */
        ccmn_rel_ccb((CCB_HEADER *)ccb_async );

        if((flags & FNDELAY) == NULL){
            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags);
            ccmn_close_unit(dev);
            return(EIO);
         }
    }
                    /* end of if status != CAM_REQ_CMP for SET */
                    /*    ASYNCHRONOUS CALLBACK CCB */

    else {

       ccmn_rel_ccb((CCB_HEADER *)ccb_async );

    }
                    /* end of else (SET ASYNCHRONOUS CALLBACK */
                    /*   CCB status == CAM_REQ_CMP) */



    /*
     * Everything is in place to start operations
     * Check the dev descriptor to get the device ready
     * time in seconds. If null, take the default of 45 seconds.
     */
    ready_time = dev_desc->dd_ready_time;

    if( ready_time == NULL){
        ready_time = 45;
    }

    /*
     * The following 3 variables are VERY important. They direct
     * actions at the bottom of the for loop that issues the TEST
     * UNIT READY command. If success is nonzero, then the TUR
     * succeeded with no errors. If fndelay is non zero the TUR
     * failed but the FNDELAY flag was set. Either way, get out
     * of for loop.  If fatal is ever positive, then either the
     * unit is reserved to another initiator or there is a driver
     * problem.
     */

    success = 0;
    fndelay = 0;
    fatal = 0;


    /*
     * Start of the for loop that looks for the device to become
```

Example D-2: (continued)

```
     * ready.  Take into account the FNDELAY flag and SCSI BUSY
     * status. POSIX definition of the FNDELAY flags say don't
     * wait for the unit to become ready. If the unit is not
     * there or is reserved by another initiator, return failure;
     * else return success. SCSI BUSY status indicates that the
     * device is unable to accept the command at this time.
     */
    for ( i = 0; i < ready_time; i++) {
        /*
         * Zero out action structure
         */
        bzero( &action, sizeof(CGEN_ACTION));

        /*
         * Issue a TEST UNIT READY command. The autosense feature
         * performs the REQUEST SENSE operation, if there is a
         * SCSI status of check condition.
         */

    cgen_ready( pdrv_dev, &action, cgen_done, CGEN_SLEEP);

    if( action.ccb == NULL ) {
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: TUR, CCB_IO = NULL\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
            /*
             * Resource problem? If so, get out.
             */
            if(( action.fatal & ACT_RESOURCE ) != NULL){
                fatal++;
                break;
            }
            /*
             * Some other gross error
             */
            else if(( flags & FNDELAY ) != NULL){
                fndelay++;
                break;

            }
            else {
                fatal++;
                break;
            }

        }

      /*
       * Check to see if this is a successfully completed CCB
       */
      if(action.ccb_status == CAT_CMP) {
```

Example D-2: (continued)

```
        PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          (CAMD_GENERIC ),
          ("[%d/%d/%d] %s: TUR, SUCCESS\n",
          DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

        success++;

    } /* end if status == CAT_CMP */

    else {
        /*
         * If the CCB status does not equal
         * CAT_CMP_ERR, then this open failed.
         */
        if( action.ccb_status != CAT_CMP_ERR ){
            fatal++;
        }

        /*
         * The only error that will cause an open to fail with
         * the FNDELAY flag set is EBUSY (reservation
         * conflict) to return.
         * Check for Reservation conflict
         */
 else if( action.scsi_status == SCSI_STAT_RESERVATION_CONFLICT ){
        fatal++;
    }


    /*
     * Check the device state for SCSI_STAT_BUSY. If the status
     * is BUSY, the device could be powering up or rewinding.
     * If the status is BUSY, then retry the TUR operation again.
     * If the status is not BUSY, a UNIT ATTENTION status may have
     * been seen.
     * The (( flags & FNDELAY ) != NULL)  && ( i > 1 )) statement
     * covers this possibility.
     * The UNIT ATTENTION status is also a common condition
     * with power up, resets and cartridge changes. Just retry
     * the TUR operation again.
     *

     */

        if( (action.scsi_status != SCSI_STAT_BUSY ) &&
            (( flags & FNDELAY ) != NULL)  && ( i > 1 )) {
            fndelay++;
        }
    }

        /*
         * Check the release queue prior to releasing the CCB
         */
```

Example D-2: (continued)

```
        CHK_RELEASE_QUEUE(pdrv_dev, action.ccb);

        /*
         * Release the CCB
         */
        ccmn_rel_ccb((CCB_HEADER *)action.ccb );

    /*
     * Check for POSIX nodelay or sucessful open or fatal error
     */
    if((fndelay != NULL) || (success != NULL) ||(fatal != NULL)){
    /*
     * Break out of for loop
     */
    break;
    }


    /*
     * No success, so sleep
     */

if( mpsleep( &lbolt, (PCATCH | (PZERO+1)), \
                                    "Zzzzzz",0,(void *)0,0) )  {
            /*
             * Set interruptable sleeps. If
             * non zero comes back, a signal was delivered.
             * Restore the flags as they were at the start
             * of the for loop.
             */
            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags);
            ccmn_close_unit(dev);
            return( EINTR );
    }


    } /* end of TUR for loop */

  /*
   * Check to see if fatal is set (reservation conflict)
   */
  if ( fatal != NULL ){
      PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
      (CAMD_GENERIC ),
      ("[%d/%d/%d] %s: TUR, FATAL\n",
      DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
    CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags);
    ccmn_close_unit(dev);
    return(action.ret_error);
    }
    else if( fndelay != NULL ){
        /*
         * Broke out of loop because of some failure
```

Example D-2: (continued)

```
         * and the FNDELAY flag is set.
         * Set the flag indicating device has not gone through
         * the full online sequence.
         */
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: TUR, FNDELAY\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
    CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec,
        (state_flags | CGEN_NOT_READY_STATE));

    return(CGEN_SUCCESS);
  }
  else if ( success == NULL){
        /*
         * The TUR command never completed successfully and
         * FNDELAY flag WAS NOT set, so return the last error
         * value
         */
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: TUR, NO_SUCCESS\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
    CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags );
    ccmn_close_unit(dev);
    return(action.ret_error);
  }



    /*
     * If there has been a reset or unit attention, do
     * as directed in the Mode Select Table.
     */

    PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );

    /*
     * The if statement checks to see if there was a reset when
     * the unit was closed and if one has occurred while it was
     * opening.
     */
if(((state_flags & (CGEN_UNIT_ATTEN_STATE | CGEN_RESET_STATE)) !=
    NULL ) || ((gen_spec->gen_state_flags & (CGEN_RESET_STATE |
    CGEN_UNIT_ATTEN_STATE)) != NULL)) {

        /*
         * There was a unit attention or reset, so the
         * Mode Select Table page  must be sent to the
         * device.
         */
    PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
```

Example D-2: (continued)

```
        /*
         * Read the Mode Select Table for this device,
         * passing the index into the Mode Select Table to send
         * to the device.
         */


    if(modsel_tab != NULL) {
            /*
             * The Mode Select Table contains a pointer to the
             * page definition for this device.
             */
        for( i = 0; (modsel_tab->ms_entry[i].ms_data != NULL) &&
                                ( i < MAX_OPEN_SELS); i++) {
        /*
         * Zero out the action structure
         */
        bzero( &action, sizeof(CGEN_ACTION));
        cgen_open_sel( pdrv_dev, &action, i,
                cgen_done, CGEN_SLEEP);

        if(action.ccb == (CCB_SCSIIO *)NULL) {
            PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            (CAMD_GENERIC ),
            ("[%d/%d/%d] %s: MODSEL, CCB = NULL\n",
            DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            module));

        if(( action.fatal & ACT_RESOURCE ) != NULL ){
        /*
         * Could not get resources (ccb's) needed;
         */

            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags);
            ccmn_close_unit(dev);
            return(action.ret_error); /* driver/resource problem */
        }

            if( (flags & FNDELAY) == NULL ) {
                /*
                 * Close the unit and return errno
                 */
            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags);
            ccmn_close_unit(dev);
            return( action.ret_error );
        }

            /*
             * The FNDELAY flag is set; must return success
             */
        else {
            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec,
                ( state_flags | CGEN_NOT_READY_STATE));
```

Example D-2: (continued)

```
            return(CGEN_SUCCESS);

    }

 }

        /*
         * Check to see if the CCB completed successfully
         */
        if(action.ccb_status == CAT_CMP){

            /*
             * Release the CCB back to the pool
             */

            ccmn_rel_ccb((CCB_HEADER *)action.ccb );

            /* do next page if any */
            continue;
        }

        /*
         * The Mode Select for this page failed.
         * The only error that causes an open to fail with
         * the FNDELAY flag set is a SCSI statua of
         * RESERVATION CONFLICT.
         */

        else {
            PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            (CAMD_GENERIC ),
            ("[%d/%d/%d] %s: MODSEL FAILED index = 0x%x\n",
            DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            module, i));

            /*
             * Check the release queue prior to releasing the CCB
             */
            CHK_RELEASE_QUEUE(pdrv_dev, action.ccb);

            /*
             * Release the ccb
             */
            ccmn_rel_ccb((CCB_HEADER *)action.ccb );


           /*
            * If the returned SCSI status is RESERVATION CONFLICT
            * or FNDELAY == NULL, then fail this open.
            */
     if((action.scsi_status == SCSI_STAT_RESERVATION_CONFLICT) ||
            ((flags & FNDELAY) == NULL) ){
            /*
```

Example D-2: (continued)

```
             * Close the unit and return EBUSY
             */
            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, state_flags);
            ccmn_close_unit(dev);
            return( action.ret_error );
    }

            /*
             * The FNDELAY flag is set; must return success
             */
    else {
            CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec,
                    (state_flags | CGEN_NOT_READY_STATE));

            return(CGEN_SUCCESS);

    }
    }
    } /* end of for loop */
  } /* End of if modsel != NULL */
} /* End of reset or unit attention */

    /*
     * Unlock the struct
     */
    else {
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
    }

    /*
     * At this point, you can set up any other device
     * features you need.
     */

    /*
     * Add your device-specific code here.
     */



    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

    return(CGEN_SUCCESS);
} /* End of cgen_open() */



/* ------------------------------------------------------------ */
/* Function description.
```

Example D-2: (continued)

```
 *
 * Routine name cgen_close
 *
 *     This routine closes the unit.
 *
 *
 * Call syntax
 *  cgen_close( dev, flags )
 *
 * Implicit inputs
 *      Flags of  XXXX
 *
 * Implicit outputs
 *      NONE
 *
 * Return values
 *      CGEN_SUCCESS
 *      ENOMEM    Resource problem
 *
 * TO DO:
 */


int
cgen_close(dev, flags)

    dev_t dev;          /*  Major/minor number pair */
    int flags;          /* Flags RDONLY READ WRITE FNDELAY etc. */

{

    /*
     * LOCAL VARIABLES
     */

    PDRV_DEVICE           *pdrv_dev;
                          /* Peripheral Device Structure pointer */


    CGEN_SPECIFIC         *gen_spec;
                          /* Generic-Specific Structure pointer */
    CGEN_ACTION              action;
                          /* Generic Action Structure */


    u_long                s;     /* For saved IPL */

    static u_char         module[] = "cgen_close"; /* Module name */

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
```

Example D-2: (continued)

```
            ("[%d/%d/%d] %s: entry\n",
            DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

    /*
     * Get Peripheral Device Structure pointer
     */
    if( (pdrv_dev = GET_PDRV_PTR(dev)) == (PDRV_DEVICE *)NULL){
           /*
            * This should not happen--no Peripheral Device Structure.
            */
            LOG_ERROR("Implement your error logging");
            return(ENOMEM);
    }



    /*
     * Get device-specific structure pointer
     */

    gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific;
    if( (gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific) ==
                                (CGEN_SPECIFIC *)NULL){
            PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
            LOG_ERROR("Implement your error logging");
            PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
            return(ENOMEM);
    }



    /*
     * Check to see if a unit attention has been seen; if so,
     * close the unit. Since we can not determine what type
     * of device the driver is being written for, this is
     * only an example of UNIT ATTENTIONS and RESETS being
     * detected at the close of the device.
     */
    PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );

    if((gen_spec->gen_state_flags & (CGEN_UNIT_ATTEN_STATE |
                CGEN_RESET_STATE)) != NULL ){

            /*
             * Close unit
             */
            PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
            ccmn_close_unit(dev);
            return(CGEN_SUCCESS);

    }
```

Example D-2: (continued)

```
    /*
     * Do device-specific close steps.
     */


    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

    return( CGEN_SUCCESS );

} /* End of cgen_close() */



/* ---------------------------------------------------------- */
/* Function description.
 *
 * Routine cgen_read
 *
 * Functional Description:
 *     This routine handles user processes' synchronous read
 *     requests.  This is a pass through function that gets a buf
 *     struct allocated and then passes the work to cgen_strategy.
 *
 * Call syntax
 *  cgen_read( dev, uio)
 *      dev_t          dev;              Major/minor number pair
 *      struct         *uio              Pointer to the uio struct
 *
 * Implicit inputs
 *      NONE
 *
 * Implicit outputs
 *      NONE
 *
 * Return values
 *      Passes return from physio()
 *
 * TO DO:
 */



int
cgen_read( dev, uio)
        dev_t        dev;          /* Major/minor number pair */
        struct uio   *uio;         /* Pointer to the uio struct */

{
```

Example D-2: (continued)

```
    /*
     * Local variables
     */
    int             ret_val;        /* Value to be returned */
    struct buf      *bp;            /* Allocated buf struct */
    static u_char   module[] = "cgen_read";  /* Module name */


    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: entry\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));
    /*
     * Allocate buf struct
     */
    bp = ccmn_get_bp();

    if( bp == NULL ){
        LOG_ERROR("Implement your error logging");
        return (ENOMEM);
    }


    ret_val =  physio(cgen_strategy, bp, dev, B_READ,
                                        cgen_minphys, uio);

    /*
     * Release the buf struct
     */
    ccmn_rel_bp( bp );

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

    return( ret_val );

} /* end of cgen_read */




/* ---------------------------------------------------------- */
/* Function description.
 *
 * Routine cgen_write
 *
 * Functional Description:
 *      This routine handles synchronous write requests for user
 *      processes.  This is a pass through function, that gets a
 *      buf struct and then passes the work to cgen_strategy
```

Example D-2: (continued)

```
 *
 * Call syntax
 *  cgen_write( dev, uio)
 *      dev_t        dev;            Major/minor number pair
 *      struct       *uio;           Pointer to the uio struct
 *
 * Implicit inputs
 *      NONE
 *
 * Implicit outputs
 *      NONE
 *
 * Return values
 *      Passes return from physio()
 *
 * TO DO:
 */




int
cgen_write( dev, uio)
        dev_t        dev;         /* Major/minor number pair */
        struct uio   *uio;        /* Pointer to the uio struct */

{
    /*
     * Local variables
     */
    int                ret_val;     /* Value to be returned */
    struct buf         *bp;         /* Allocated buf struct */
    static u_char      module[] = "cgen_write";    /* Module name */


    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: entry\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

    /*
     * Allocate buf struct
     */
    bp = ccmn_get_bp();

    if( bp == NULL ){
        LOG_ERROR("Implement your error logging");
        return (ENOMEM);
    }

    ret_val =  physio(cgen_strategy, bp, dev, B_WRITE,
                                            cgen_minphys, uio);
```

Example D-2: (continued)

```
    /*
     * Release the bp
     */
    ccmn_rel_bp( bp );

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

    return( ret_val );

} /* end of cgen_write */



/* ---------------------------------------------------------------- */
/* Function description.
 *
 *      This routine handles all I/O requests for user processes.
 *      A number of checks based on whether the request
 *      is synchronous or asynchronous are made.
 *
 * Call syntax
 *  cgen_strategy( bp )
 *          struct      buf      *bp    Pointer to the struct buf
 *
 *
 * Implicit inputs
 *      In the bp, whether the request is a read or a write,
 *      synchronous or asynchronous.
 *      In the CGEN_SPECIFIC structure, various state conditions.
 *
 * Implicit outputs
 *      None.
 *
 * Return values
 *
 * TO DO:
 *
 */


void
cgen_strategy( bp )

        struct  buf     *bp;    /* Pointer to the buf struct */

{

    /*
     * Local variables.
     */
```

Example D-2: (continued)

```
PDRV_DEVICE       *pdrv_dev;
                     /* Peripheral Device Structure pointer */

DEV_DESC          *dev_desc;
                     /* Device Descriptor Structure pointer */

CGEN_SPECIFIC     *gen_spec;
                      /* Generic-Specific Structure pointer */

CCB_SCSIIO        *ccb_io;          /* SCSI I/O CCB pointer */

u_long            ccb_flags;
                          /* The flags to be set in the ccb */
SEQ_READ_CDB6     *rd_cdb;
      /* Pointer to CDB within the CCB for a read command. */
SEQ_WRITE_CDB6    *wt_cdb;
     /* Pointer to CDB within the CCB for a write command. */
u_long            send_stat;
                          /* Value send CCB routine returns */
static u_char    module[] = "cgen_strategy"; /* Module name */

int               s;                            /* Saved IPL */
u_char            sense_size;     /* The request sense size */
SEQ_WRITE_CDB6    *wt_cdb;          /* Pointer to write CDB */


PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
    (CAMD_GENERIC |CAMD_INOUT),
    ("[%d/%d/%d] %s: entry\n",
    DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

/*
 * Get Peripheral Device Structure pointer
 */
pdrv_dev = GET_PDRV_PTR(bp->b_dev);

/*
 * Get Device Descriptor Structure pointer
 */
dev_desc = pdrv_dev->pd_dev_desc;
/*
 * Get Generic-Specific Structure pointer
 */
gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific;

/*
 * Lock the structure now because throughout the routine we
 * examine flags within the CGEN_SPECIFIC structure and do
 * not want any other routine to be clearing or setting flags
 * while decisons on the flags are being made.
 */
PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
```

Example D-2: (continued)

```
    /*
     * Check to see if the device was opened with the FNDELAY
     * flag set and it was not ready.
     */
if(( gen_spec->gen_state_flags & CGEN_NOT_READY_STATE ) != NULL){
       PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            (CAMD_GENERIC ),
            ("[%d/%d/%d] %s: NOT ready state flags = %0xX\n",
            DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
            module, gen_spec->gen_state_flags));

        /*
         * Do not allow I/O operations to the unit
         */
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
        CGEN_BERROR(bp, bp->b_bcount, EINVAL);
        biodone( bp );
        return;
    }
    /*
     * This section of code notices various state conditions and
     * handles according to device.
     */

    if(( gen_spec->gen_state_flags & CGEN_XXX_STATE ) != NULL ){
        PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          (CAMD_GENERIC ),
          ("[%d/%d/%d] %s: CGEN_XXX_STATE: stateflags = 0x%X\n",
          DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          module, gen_spec->gen_state_flags));

        /*
         * Do not allow I/O operations to the unit
         */
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s );
        CGEN_BERROR(bp, bp->b_bcount, EIO);
        biodone( bp );
        return;
    }

    /*
     * Build the CDB (SCSI Command )
     * EXAMPLE of a sequential access device.
     */
    /*
     * Check the buf structure flags to determine if the user
     * has requested a read or write operation.
     */
    if(( bp->b_flags & B_READ ) != NULL){
      rd_cdb = (SEQ_READ_CDB6 *)ccb_io->cam_cdb_io.cam_cdb_bytes;

        rd_cdb->opcode = SEQ_READ_OP;
        rd_cdb->lun = 0;
```

Example D-2: (continued)

```
        SEQTRANS_TO_READ6( bp->b_bcount, rd_cdb );

        /*
         * Set the length of the CDB
         */
        ccb_io->cam_cdb_len = sizeof(SEQ_READ_CDB6);
    }
    /*
     * Must be user write command.
     */
    else {
     wt_cdb = (SEQ_WRITE_CDB6 *)ccb_io->cam_cdb_io.cam_cdb_bytes;

     wt_cdb->opcode = SEQ_WRITE_OP;

     wt_cdb->lun = 0;

     SEQTRANS_TO_WRITE6( bp->b_bcount, wt_cdb );

        /*
         * Set the length of the CDB
         */
        ccb_io->cam_cdb_len = sizeof(SEQ_WRITE_CDB6);
    }


    /*
     * Send it down to the XPT layer
     */

    send_stat = ccmn_send_ccb( pdrv_dev, (CCB_HEADER *)ccb_io,
                                                    NOT_RETRY);

    /*
     * If the CCB is not in progress...
     */
    if((send_stat & CAM_STATUS_MASK) != CAM_REQ_INPROG){
        /*
         * The CCB has been returned and has not gone through
         * cgen_iodone. Call the CCB and return.
         */
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: send status NOT inprog\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

        PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
        cgen_iodone(ccb_io);
        return;
    }

    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
```

Example D-2: (continued)

```
    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

    return;

} /* end of cgen_strategy */


/* ---------------------------------------------------------- */
/* Function description.
 *
 * Routine name cgen_ioctl
 *
 *      This routine handles specific requests for actions other
 *      than read and write.
 *
 * Call syntax
 *  cgen_ioctl( dev, cmd, data, flags )
 *
 * Implicit inputs
 *      flags of CGEN_XXX
 *
 * Implicit outputs
 *
 *
 * Return values
 *
 * TO DO:
 */

int
cgen_ioctl( dev, cmd, data, flag )
        dev_t           dev;
                                    /* Major/minor number pair */
        int             cmd;       /* The command we are doing */
        caddr_t         data;
                              /*
                               * Pointer to kernel's copy of user
                               * request struct
                               */
        int             flag;                   /* User flags */
{


    /*
     * Local Variables
     */

    PDRV_DEVICE          *pdrv_dev;
                         /* Peripheral Device Structure pointer */
    CGEN_SPECIFIC        *gen_spec;
```

Example D-2: (continued)

```
                          /* Generic-Specific Structure pointer */
DEV_DESC                 *dev_desc;
                         /* Device Descriptor Structure pointer */
SEQ_MODE_DATA6           *msdp;      /* Mode sense data pointer */
CGEN_ACTION              action;    /* Generic Action Structure */
struct devget            *devget;           /* Device get ioctl */
struct device            *device;        /* Used for devget only */
struct controller        *cont;          /* Used for devget only */
long                     retries;
                                /* The number of times to try to
                                 * do a mode sense for devget
                                 */
int                      s;                         /* Saved IPL */
                                         /* Device unit number */
u_long                   ccb_status;                /* CCB status */
u_long                   chk_status;  /* Check condition status */
static u_char         module[] = "cgen_ioctl";    /* Module name */


PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
    (CAMD_GENERIC |CAMD_INOUT),
    ("[%d/%d/%d] %s: entry\n",
    DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));


/*
 * Get pointers
 */
pdrv_dev = GET_PDRV_PTR(dev);
if( pdrv_dev == (PDRV_DEVICE *)NULL) {
    /*
     * There is no Peripheral Device Structure
     */
    PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
    LOG_ERROR("Implement your error logging");
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
    return(ENXIO);
}
gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific;
if( gen_spec == (CGEN_SPECIFIC *)NULL){
    /*
     * No Generic-Specific Structure
     */
    PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
    LOG_ERROR("Implement your error logging");
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
    return(ENXIO);
}

device = camdinfo[pdrv_dev->pd_log_unit];

cont = camminfo[device->ctlr_num];
```

Example D-2: (continued)

```
dev_desc = pdrv_dev->pd_dev_desc;

/*
 * Look at command to determine next action.
 */
switch (cmd) {

case DEVIOCGET:                /* device status */
  devget = (struct devget *)data;
  bzero(devget,sizeof(struct devget));
  devget->category = DEV_SCSI;
  devget->bus = DEV_SCSI;
  bcopy(DEV_SCSI_GEN, devget->interface,
                              strlen(DEV_SCSI_GEN));
  bcopy(dev_desc->dd_dev_name, devget->device, DEV_SIZE);
  devget->adpt_num = cont->slot;
  devget->nexus_num = 0;
  devget->bus_num = DEV_BUS_ID(dev);
  devget->ctlr_num = device->ctlr_num;
  devget->rctlr_num = 0;
  devget->slave_num = DEV_TARGET(dev) ;
  bcopy("generic", devget->dev_name, 6);
  devget->unit_num = ((pdrv_dev->pd_target << 3)
                                | pdrv_dev->pd_lun);

  PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
  devget->soft_count = pdrv_dev->pd_soft_err;
  devget->hard_count = pdrv_dev->pd_hard_err;

  devget->stat = gen_spec->gen_flags;
  devget->category_stat = gen_spec->gen_flags;
  PDRV_IPLSMP_UNLOCK(pdrv_dev, s);

    /*
     * Do a mode sense to check for write-locked drive.
     * The first SCSI mode sense command can fail due to
     * unit attention.
     */

    retries = 0;
    do {

        /*
         * Issue a mode sense command
         */
        /*
         * Clear out Generic Action Structure
         */
        bzero(&action, sizeof(CGEN_ACTION));

     cgen_mode_sns( pdrv_dev, &action, cgen_done,SEQ_NO_PAGE,
                          ALL_PCFM_CURRENT, CGEN_SLEEP);
```

Example D-2: (continued)

```
        if(action.ccb == (CCB_SCSIIO *)NULL) {
             PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev),
                    DEV_LUN(dev), (CAMD_GENERIC ),
                    ("[%d/%d/%d] %s: devget NULL CCB\n",
                    DEV_BUS_ID(dev), DEV_TARGET(dev),
                    DEV_LUN(dev), module));

        /* Must return 0 for devget */
        return(0);
        }

        if(action.ccb_status == CAT_CMP ){

            /*
             *  GOOD Status. Fill in rest of devget struct
             */
            msdp = (SEQ_MODE_DATA6 *)action.ccb->cam_data_ptr;

            if( msdp->sel_head.wp != NULL ){
                /*
                 * DEVICE is write locked.
                 */
                devget->stat |= DEV_WRTLCK;
            }
            /*
             * Do you need to set up the device's specifics?
             * For tapes, need to look at the density field
             * returned in the MODE SENSE data. Implement
             * the specifics for your device.
             */

        }

        /*
         * Release the CCB and the memory used for the
         * mode sense data back to the system.
         */
        if( action.ccb != (CCB_SCSIIO *)NULL) {
            CHK_RELEASE_QUEUE(pdrv_dev, action.ccb);

            CGEN_REL_MEM( action.ccb );

            ccmn_rel_ccb((CCB_HEADER *)action.ccb );
        }

        retries++;

    } while( (retries < 3) && (action.ccb_status != CAT_CMP));

     /*
      * Since this is a devget, always return success.
      */
       PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
```

Example D-2: (continued)

```
                (CAMD_GENERIC |CAMD_INOUT),
                ("[%d/%d/%d] %s: exit\n",
                DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
                module));
        return(0);
        break;


    default:
        return (ENXIO);
        break;
    }
    return (0);

} /* end of cgen_ioctl */



/* ---------------------------------------------------------------- */
/*
 * Generic *done routines
 */


/* ---------------------------------------------------------------- */
/* Function description.
 *
 *
 * Routine Name: cgen_done()
 *
 * Functional Description:
 *
 *      Entry point for all NON-user I/O requests.
 *      If the CCB does not contain a buf struct pointer in the
 *      Peripheral Device Driver Working Set Structure, then
 *      issue a wakeup system call on the address of the CCB.
 *
 *
 *
 * Call Syntax:
 *
 *      cgen_done( ccb )
 *
 *      CCB_SCSIIO *ccb;
 *
 *
 *
 *
 *
 * Returns :
 *      None
 */
```

Example D-2: (continued)

```
void
cgen_done (ccb)
        CCB_SCSIIO      *ccb;          /* SCSI I/O CCB pointer */
{

    /*
     * Local variable
     */
    PDRV_DEVICE         *pdrv_dev;
                        /* Peripheral Device Structure pointer */


    int                 s;                        /* Saved IPL */
    static u_char       module[] = "cgen_done"; /* Module name */


    pdrv_dev =
         (PDRV_DEVICE *)((PDRV_WS *)ccb->cam_pdrv_ptr)->pws_pdrv;


    if( pdrv_dev == NULL ){
        panic("cgen_done: NULL PDRV_DEVICE pointer");
        return;
    }
PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
       ("[%d/%d/%d] %s: entry\n",
       DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), module));

    /*
     * Remove from active lists
     */
    ccmn_rem_ccb( pdrv_dev, ccb );

    /*
     * To prevent race conditions on smp machines...
     */
    PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);

    /*
     * Check to see if buf struct pointer is filled in.
     * It should not be for this routine.
     */
    if( (struct buf *)ccb->cam_req_map == NULL){

        /*
         * This is not an user I/O CCB
         */
        wakeup(ccb);
    }
    else {
```

Example D-2: (continued)

```
        LOG_ERROR("Implement your error logging");

       wakeup(ccb);
    }
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);

PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
       ("[%d/%d/%d] %s: exit\n",
       DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), module));
    return;

} /* end of cgen_done */



/* ----------------------------------------------------------- */
/* Function description.
 *
 * Routine Name: cgen_iodone
 *
 * Functional description:
 *
 *      This routine is called by lower levels when a user I/O
 *      request has been acted on by the lower levels.
 *
 *      Due the its buffered-mode operation, the target can
 *      return good status without transferring the data to
 *      media Notifcation of media error occurs sometime later.
 *
 * Side Effects:
 *      Based on CAM status, the user buffer struct is modified
 *      to reflect either successful completion of the I/O
 *      transfer or error status.
 *
 *      Flags are set in the CGEN_SPECIFIC structure to reflect
 *      events detected.
 *
 * Call Syntax
 *      cgen_iodone( ccb )
 *              CCB_SCSIIO * ccb;
 *
 *
 * Returns:
 *      None
 *
 */

void
cgen_iodone( ccb )

    CCB_SCSIIO          *ccb;           /* SCSI I/O CCB pointer */
```

Example D-2: (continued)

```
{
    PDRV_DEVICE           *pdrv_dev;
                          /* Peripheral Device Structure pointer */
    CGEN_SPECIFIC         *gen_spec;
                           /* Generic-Specific Structure pointer */
    struct buf            *bp;
                                  /* User I/O buf struct pointer */
    u_long                ccb_status;
                                         /* Result of CCB status */
    u_long                chk_status;
                             /* Result of check condition status */
    int                   s;                         /* Saved IPL */
    dev_t                 dev;        /* Major/minor number pair */
    static u_char      module[] = "cgen_iodone";  /* Module name */


    /*
     * Peripheral Device Structure and Generic-Specific Structure
     */
    pdrv_dev =
         (PDRV_DEVICE *)((PDRV_WS *)ccb->cam_pdrv_ptr)->pws_pdrv;

    gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific;

    dev = pdrv_dev->pd_dev;

    PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: entry\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

    /*
     * Get user buf struct
     */
    bp = (struct buf *)ccb->cam_req_map;
    if( bp == (struct buf *)NULL) {
       /*
        * There should be a buf struct if this routine is called.
        */

        PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s);
        LOG_ERROR("Implement your error logging");
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s);
        ccmn_rem_ccb( pdrv_dev, ccb );
        /*
         * Issue a wakeup system call on this CCB
         */
        wakeup( ccb );
        return;
    }
    /*
     * Lock to prevent race conditions for asynchronous
     * I/O (nbuf I/O).
```

Example D-2: (continued)

```
     */
    PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s);



    /*
     * Remove this CCB from the active list
     */
    ccmn_rem_ccb( pdrv_dev, ccb );

    /* Get completion status */

    ccb_status = ccmn_ccb_status( (CCB_HEADER *)ccb );

    /*
     * Save residual counts
     */
    bp->b_resid = ccb->cam_resid;
    gen_spec->gen_resid = ccb->cam_resid;

    switch( ccb_status ) {

    case CAT_CMP:

        /*
         * The cam_resid flag indicates the number
         * of bytes that were not transferred.
         * If anything but NULL, the device has problems.
         */
    if( ccb->cam_resid != NULL){
      LOG_ERROR("Implement your error logging");
      PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
       (CAMD_GENERIC ),
       ("[%d/%d/%d] %s: Status = CMP but resid not NULL\n",
       DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

      CGEN_BERROR(bp, ccb->cam_resid, EIO);

    }

        break;

    case CAT_CMP_ERR:

        /*
         * Had some sort of SCSI status other than GOOD, so
         * must look at each SCSI status type to determine
         * how to handle.
         */

        /*
         * Reason is either a check
         * condition or reservation conflict.
```

Example D-2: (continued)

```
         */
        switch(ccb->cam_scsi_status) {
        default:
        case SCSI_STAT_GOOD:
        case SCSI_STAT_CONDITION_MET:
        case SCSI_STAT_INTERMEDIATE:
        case SCSI_STAT_INTER_COND_MET:
        case SCSI_STAT_COMMAND_TERMINATED:
        case SCSI_STAT_QUEUE_FULL:
            CGEN_BERROR(bp, ccb->cam_resid, EIO);


           PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
                 (CAMD_GENERIC ),
                 ("[%d/%d/%d] %s: default SCSI STATUS = 0x%x\n",
                 DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
                 module, ccb->cam_scsi_status));

           LOG_ERROR("Implement your error logging");

           break;


    case SCSI_STAT_BUSY:
      CGEN_BERROR(bp, ccb->cam_resid, EIO);
      PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: device BUSY STATUS\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

            LOG_ERROR("Implement your error logging");

            break;

        case SCSI_STAT_RESERVATION_CONFLICT:
            /*
             * This unit is reserved by another initiator.
             * This should not happen
             */
           CGEN_BERROR(bp, ccb->cam_resid, EBUSY);


      LOG_ERROR("Implement your error logging");
      PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
        (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: Reservation conflict.\n",
        DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module));

          break;

        case SCSI_STAT_CHECK_CONDITION:

            /*
```

Example D-2: (continued)

```
         * Call cgen_ccb_chkcond()
         * to handle the check condition
         */
        chk_status = cgen_ccb_chkcond(ccb, pdrv_dev);

        /*
         * Determine what to do.
         */
        switch ( chk_status ) {
        /*
         * Look at common conditions first.
         * Note that the gen_spec->ts_resid is handled
         * in the check condition.....
         */
        case CHK_EOM :

        case CHK_FILEMARK:

        case CHK_ILI:

        case CHK_SOFTERR:

        case CHK_INFORMATIONAL:

        case CHK_CHK_NOSENSE:

        case CHK_SENSE_NOT_VALID:

        case CHK_NOSENSE_BITS:

        case CHK_NOT_READY:

        case CHK_HARDERR:

        case CHK_UNIT_ATTEN:

        case CHK_DATA_PROT:

        case CHK_UNSUPPORTED:

        case CHK_CMD_ABORTED:

        case CHK_UNKNOWN_KEY:

        default:
            break;

        } /* end of switch for check condition */

        break; /* end of scsi_status check condition */

    } /* end of switch of SCSI status */
```

Example D-2: (continued)

```
        break;

    case CAT_INPROG:
    case CAT_UNKNOWN:
    case CAT_CCB_ERR:

        CGEN_BERROR( bp, bp->b_bcount, EIO);

        PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          (CAMD_GENERIC ),
          ("[%d/%d/%d] %s: CCB status: %s\n",
          DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
          cam_ccb_str((CCB_HEADER *)ccb)));
        break;

    case CAT_RESET:
    case CAT_BUSY:

        /*
         * Status should only be busy.
         * Don't have to abort the active queues.The  CCBs that
         * are queued will be returned to use. This action is
         * defined in the CAM specification.
         * Don't error log this because the error log will fill
         * up with reset pending messages....
         */
         if( CAM_STATUS(ccb) == CAM_BUSY) {
           gen_spec->gen_state_flags |= CGEN_RESET_PENDING_STATE;
        }
        PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          (CAMD_GENERIC ),
          ("[%d/%d/%d] %s: CCB status: %s\n",
          DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
          cam_ccb_str((CCB_HEADER *)ccb)));

        CGEN_BERROR( bp, ccb->cam_resid, EIO);

        break;

    case CAT_SCSI_BUSY:
    case CAT_BAD_AUTO:
    case CAT_DEVICE_ERR:
        /*
         * Error log this
         */
        LOG_ERROR("Implement your error logging");

        PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          (CAMD_GENERIC ),
          ("[%d/%d/%d] %s: CCB status: %s\n",
          DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
          cam_ccb_str((CCB_HEADER *)ccb)));
```

Example D-2: (continued)

```
            CGEN_BERROR( bp, ccb->cam_resid, EIO);

            break;


        case CAT_NO_DEVICE:
            /*
             * Error log this.
             */

            LOG_ERROR("Implement your error logging");
             PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
              (CAMD_GENERIC ),
              ("[%d/%d/%d] %s: CCB status: %s\n",
              DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
              cam_ccb_str((CCB_HEADER *)ccb)));

            CGEN_BERROR(bp, ccb->cam_resid, ENXIO);

            break;

        case CAT_ABORT:

            /*
             * Return is a result of walking the
             * active lists and aborting the ccb's
             */
            PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
              (CAMD_GENERIC ),
              ("[%d/%d/%d] %s: CCB status: %s\n",
              DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
              cam_ccb_str((CCB_HEADER *)ccb)));

            if( CAM_STATUS( ccb ) == CAM_REQ_ABORTED ){
            }
            else if( CAM_STATUS( ccb ) == CAM_UA_ABORT ){

            }
            else if( CAM_STATUS( ccb ) == CAM_UA_TERMIO ){
            }
            else if( CAM_STATUS( ccb ) == CAM_REQ_TERMIO ){
            }
            else {
            }
            break;


        default:
            /*
             * Error log this; should never get the default condition.
             */
            LOG_ERROR("Implement your error logging");
            PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
```

Example D-2: (continued)

```
            (CAMD_GENERIC ),
            ("[%d/%d/%d] %s: CCB status: %s\n",
            DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev), module,
            cam_ccb_str((CCB_HEADER *)ccb)));

          CGEN_BERROR( bp, bp->b_bcount, EIO);

          break;

      } /* end switch on cam status */

      /*
       * Unlock
       */
      PDRV_IPLSMP_UNLOCK(pdrv_dev, s)

      /* All flags are set; call iodone on this buf struct.
       */
      iodone( bp );

      /*
       * Do not attempt to release data buffers for user I/O,
       * because a system panic will result.
       */

      /*
       * Check the release queue prior to releasing the CCB
       */
      CHK_RELEASE_QUEUE(pdrv_dev, ccb);

      /*
       * Release the CCB
       */
      ccmn_rel_ccb((CCB_HEADER *)ccb );

      PRINTD(DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),
          (CAMD_GENERIC |CAMD_INOUT),
          ("[%d/%d/%d] %s: exit\n",
          DEV_BUS_ID(dev), DEV_TARGET(dev), DEV_LUN(dev),module));

      return;

  } /* end of cgen_iodone */



  /* ---------------------------------------------------------- */
  /*
   * Asynchronous notification routine.
   */

  /* ---------------------------------------------------------- */
```

Example D-2: (continued)

```
/* Function description.
 *      This routine is called when an AEN, BDR, or Bus reset has
 *      occurred. This routine sets CGEN_RESET_STATE and clears
 *      CGEN_RESET_PEND_STATE for BDR's and Bus resets. For AEN's
 *      set CGEN_UNIT_ATTEN_STATE.
 *
 *
 * Call syntax
 *  cgen_async( opcode, path_id, target, lun, buf_ptr, data_cnt)
 *      u_long          opcode;             Reason why called
 *      u_char          path_id;            Bus number
 *      u_char          target;             Target number
 *      u_char          lun;                Logical unit number
 *      cadd_t          buf_ptr;            Buffer address AEN's
 *      u_char          data_cnt;           Number of bytes valid;
 *
 * Implicit inputs
 *       NONE
 *
 * Implicit outputs
 *      Setting and clearing of state flags
 *
 * Return values
 *      NONE
 *
 * TO DO:
 *      Recovery for unit.
 */

void
cgen_async( opcode, path_id, target, lun, buf_ptr, data_cnt)
        u_long          opcode;       /* Reason called */
        u_char          path_id;      /* Bus number */
        u_char          target;       /* Target number */
        u_char          lun;          /* Logical unit number */
        caddr_t         buf_ptr;      /* Buffer address AEN's */
        u_char          data_cnt;     /* Number of bytes valid; */

{

    /*
     * Local Variables
     */

    PDRV_DEVICE         *pdrv_dev;
                        /* Peripheral Device Structure pointer */

    CGEN_SPECIFIC       *gen_spec;
                         /* Generic-Specific Structure pointer */

    dev_t               dev;                    /* Device number */
    int                 s;                          /* Saved IPL */
    static u_char    module[] = "cgen_async";   /* Module name */
```

Example D-2: (continued)

```
PRINTD(path_id, target, lun, (CAMD_GENERIC |CAMD_INOUT),
    ("[%d/%d/%d] %s: entry\n",
    path_id, target, lun, module));

/*
 * Get device number
 */
dev = MAKE_DEV(  path_id, target, lun );


pdrv_dev = GET_PDRV_PTR(dev);

/*
 * If pdrv_device == NUll, then the device has never been
 * opened and this section should not have been reached.
 */
if( pdrv_dev == (PDRV_DEVICE *)NULL){
    LOG_ERROR("Implement your error logging");
    PRINTD(path_id, target, lun, (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: pdrv_dev == 0\n",
        path_id, target, lun, module));
    return;
}

/*
 * If gen_spec == NUll, then the device has never been opened
 * and this section should not have been reached
 */

gen_spec = (CGEN_SPECIFIC *)pdrv_dev->pd_specific;

if( gen_spec == (CGEN_SPECIFIC *)NULL){
    PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
    LOG_ERROR("Implement your error logging");
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
    return;
}

/*
 * Find out why this section was reached
 */
if((opcode & AC_SENT_BDR ) != NULL){
    PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
    gen_spec->gen_state_flags |= CGEN_RESET_STATE;
    gen_spec->gen_state_flags &= ~CGEN_RESET_PENDING_STATE;
    LOG_ERROR("Implement your error logging");
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
}
if((opcode & AC_BUS_RESET ) != NULL){
    PDRV_IPLSMP_LOCK(pdrv_dev, LK_RETRY, s);
    gen_spec->gen_state_flags |= CGEN_RESET_STATE;
    gen_spec->gen_state_flags &= ~CGEN_RESET_PENDING_STATE;
    LOG_ERROR("Implement your error logging");
```

Example D-2: (continued)

```
        PDRV_IPLSMP_UNLOCK(pdrv_dev, s);
    }
    if((opcode & AC_SCSI_AEN ) != NULL){
   CGEN_LOCK_OR_STATE(pdrv_dev, gen_spec, CGEN_UNIT_ATTEN_STATE);
    }

    PRINTD(path_id, target, lun, (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: exit\n",
        path_id, target, lun, module));
    return;

} /* End of cgen_async() */


/* ---------------------------------------------------------------- */

/*
 * Command Support Routines
 */



/* ---------------------------------------------------------------- */
/* Function description.
 *
 * This routine issues a SCSI TEST UNIT READY command
 * to the unit.
 *
 * The varible sleep for this version will always be TRUE. This
 * directs the code to sleep waiting for comand status.
 *
 * Call syntax
 *  cgen_ready( pdrv_dev, action, done, sleep)
 *      PDRV_DEVICE      *pdrv_dev;
 *                             Peripheral Device Structure pointer
 *      CGEN_ACTION      *action; Generic Action Structure pointer
 *      void             (*done)();             Completion routine
 *      u_long           sleep;                   Whether to sleep
 *
 * Implicit inputs
 *       NONE
 *
 * Implicit outputs
 *      The various statuses into the caller's action struct.
 *
 * Return values
 *      NONE
 *
 * TO DO:
 *
 */
```

Example D-2: (continued)

```
void
cgen_ready( pdrv_dev, action, done, sleep)
        PDRV_DEVICE        *pdrv_dev;
                           /* Peripheral Device Structure pointer */
        CGEN_ACTION        *action;
                              /* Generic Action Structure pointer */
        void               (*done)();        /* Completion routine */
        u_long             sleep;              /* Whether to sleep */

{

    /*
     * LOCAL variables
     */
    DEV_DESC               *dev_desc = pdrv_dev->pd_dev_desc;
                              /*
                               * Device Descriptor Structure pointer
                               */

    int                    s;                           /* Saved IPL */
    int                    s1;                      /* Throwaway IPL */
    u_char                 sense_size;
                                        /* Request sense buffer size */
    static u_char       module[] = "cgen_ready";    /* Module name */

PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
       ("[%d/%d/%d] %s: entry\n",
       DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), module));

    /*
     * See if the System administrator has set the request sense
     * size. This is for autosense. If there is an error,
     * the lower levels will do a request sense.
     */
    sense_size = GET_SENSE_SIZE( pdrv_dev );

    /*
     * Call the common routine to create the CCB for the test
     * unit ready. It will return a CCB that is already being
     * processed.
     */
    action->act_ccb =
        ccmn_tur(pdrv_dev, sense_size,(u_long)CAM_DIR_NONE, done,
                (u_char)NULL, CGEN_TME_5);

   /*
    * Check if CCB is NULL. If so, the generic macro fills out
    * the error logs and the action return values.
    */
    if(action->act_ccb == (CCB_SCSIIO *)NULL){
        CGEN_NULLCCB_ERR(action, pdrv_dev, module);
```

Example D-2: (continued)

```
        return;
    }

    /*
     * Check to see if sleep is not set
     */
    if( sleep != CGEN_SLEEP){
        return;
    }


    /*
     * Check the CCB to make sure it is in progress before
     * going to sleep. Raise the IPL to block the
     * interrupt; the sleep will lower it.
     */

    PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
    while( CAM_STATUS( action->act_ccb) == CAM_REQ_INPROG ){
        /*
         * Sleep on address of CCB, but NON interruptable
         */
        PDRV_SMP_SLEEPUNLOCK( action->act_ccb, PRIBIO, pdrv_dev);

        /*
         * Get the lock again
         */
         PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s1 );
    }



    /*
     * At this point, the command has been sent down and
     * completed.  Now check for status.
     */


    action->act_ccb_status =
                  ccmn_ccb_status((CCB_HEADER *)action->act_ccb);

    switch( action->act_ccb_status ) {

    case CAT_CMP:

        /*
         *  GOOD status; just return.
         */
        break;

    case CAT_CMP_ERR:

        /*
```

Example D-2: (continued)

```
         * Had a SCSI status other than good;
         * must look at each possible SCSI status to
         * determine our action.
         */


      action->act_scsi_status = action->act_ccb->cam_scsi_status;
      switch(action->act_scsi_status)
            {
            default:
            case SCSI_STAT_GOOD:
            case SCSI_STAT_CONDITION_MET:
            case SCSI_STAT_BUSY:
            case SCSI_STAT_INTERMEDIATE:
            case SCSI_STAT_INTER_COND_MET:
            case SCSI_STAT_COMMAND_TERMINATED:
            case SCSI_STAT_QUEUE_FULL:
            case SCSI_STAT_RESERVATION_CONFLICT:
            case SCSI_STAT_CHECK_CONDITION:

                /* Call cgen_ccb_chkcond()
                 * to handle the check condition.
                 */
    action->act_chkcond_error = cgen_ccb_chkcond(action->act_ccb,
                   pdrv_dev);

                /*
                 * Now determine what to do.
                 */
                switch ( action->act_chkcond_error ) {

                case CHK_UNIT_ATTEN:
                case CHK_NOT_READY:
                case CHK_INFORMATIONAL:
                case CHK_SOFTERR:
                case CHK_EOM :
                case CHK_FILEMARK:
                case CHK_ILI:
                case CHK_CHK_NOSENSE:
                case CHK_SENSE_NOT_VALID:
                case CHK_NOSENSE_BITS:
                case CHK_HARDERR:
                case CHK_DATA_PROT:
                case CHK_UNSUPPORTED:
                case CHK_CMD_ABORTED:
                case CHK_UNKNOWN_KEY:
                default:
                        break;
                }
            break; /* end of scsi_status check condition */

            } /* end of switch of scsi status */
```

Example D-2: (continued)

```
            break; /* End of CAM_CMP_ERR */

        case CAT_INPROG:
        case CAT_UNKNOWN:
        case CAT_CCB_ERR:
        case CAT_RESET:
        case CAT_BUSY:
        case CAT_SCSI_BUSY:
        case CAT_BAD_AUTO:
        case CAT_DEVICE_ERR:
        case CAT_NO_DEVICE:
        case CAT_ABORT:
            action->act_fatal |=  ACT_FAILED;
            action->act_ret_error = EIO;
        default:
        /*
         * Error log this; it should never occur.
         */
            action->act_fatal |=  ACT_FAILED;
            action->act_ret_error = EIO;
            LOG_ERROR("Implement your error logging");

            break;

        } /* end switch on cam status */

        /*
         * Now unlock
         */
        PDRV_IPLSMP_UNLOCK(pdrv_dev, s);

PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
       ("[%d/%d/%d] %s: exit\n",
       DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), module));

        return;

} /* End of cgen_ready() */



/* ---------------------------------------------------------- */
/* Function description.
 * This routine runs down the Mode Select Table Structure for
 * this device, if one is defined.
 *
 * Call syntax
 *  cgen_open_sel( pdrv_dev, action,index, done, sleep)
 *      PDRV_DEVICE      *pdrv_dev;
 *                      Pointer to the Peripheral Device Structure
 *      CGEN_ACTION      *action;
```

Example D-2: (continued)

```
                              Generic Action Structure pointer
 *      long            ms_index;
                            The index of the Mode Select Table
 *      void            (*done)();              Completion routine
 *      u_long          sleep;                    Whether to sleep
 *
 * Implicit inputs
 *      NONE
 *
 * Implicit outputs
 *      Return values of status of command placed in the action
 *      struct.
 *
 * Return values
 *      NONE
 *
 * TO DO:
 *      No sleep and state step
 *      Interrupted sleeps
 */


void
cgen_open_sel( pdrv_dev, action, index, done, sleep)
        PDRV_DEVICE     *pdrv_dev;
                        /* Peripheral Device Structure pointer */
        CGEN_ACTION     *action;
                           /* Generic Action Structure pointer */
        void            (*done)();       /* Completion routine */
        u_long          sleep;             /* Whether to sleep */

{

    /*
     * Local Variables
     */
    DEV_DESC                *dev_desc = pdrv_dev->pd_dev_desc;
                    /*
                     * Device Descriptor Structure pointer
                     */

    MODESEL_TBL *mod_tbl = pdrv_dev->pd_dev_desc->dd_modesel_tbl;
                    /*
                     * Pointer to Mode Select Table Structure
                     */


    int                     s;                      /* Saved IPL */
    int                     s1;                 /* Throwaway IPL */
    u_char                  sense_size;
                                 /* Request sense buffer size */
    static u_char   module[] = "cgen_open_sel"; /* Module name */
```

Example D-2: (continued)

```
PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
      ("[%d/%d/%d] %s: entry\n",
      DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), module));

    /*
     * Validate this Mode Select Table index
     */
  if(( index >=
     MAX_OPEN_SELS)||(mod_tbl->ms_entry[index].ms_data == NULL)){

        /*
         * The caller of this routine passed a invalid index.
         */

        PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
        LOG_ERROR("Implement your error logging");
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s );

PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC ),
       ("[%d/%d/%d] %s: Data pointer 0 or excede OPEN_SELS\n",
       DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
       DEV_LUN(pdrv_dev->pd_dev), module));

       action->act_fatal |= (ACT_PARAMETER| ACT_FAILED);
       action->act_ret_error = EINVAL;
       return;
    }

    /*
     * See if the System Administrator has set the request sense
     * size. This is for autosense. If there is an error,
     * the lower levels will do a request sense.
     */
    sense_size = GET_SENSE_SIZE( pdrv_dev );

   action->act_ccb =
     ccmn_mode_select( pdrv_dev, sense_size, (u_long)CAM_DIR_OUT,
              done, (u_char)NULL, CGEN_TIME_5, index);

   /*
    * Check if CCB is NULL. If so, the macro fills out
    * the error logs and the action return values.
    */
    if(action->act_ccb == (CCB_SCSIIO *)NULL){
        CGEN_NULLCCB_ERR(action, pdrv_dev, module);
        return;
    }

    /*
     * Check to see if sleep is set...
```

Example D-2: (continued)

```
         */
        if( sleep == CGEN_NOSLEEP ){
            return;
        }


        /*
         * Check the CCB to make sure it is in progress
         * before going to sleep. Raise the
         * IPL to block the interrupt, the sleep
         * will lower it.
         */
        PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
        while( CAM_STATUS( action->act_ccb) == CAM_REQ_INPROG ){
            /*
             * Sleep NON interruptable on address of CCB
             */
            PDRV_SMP_SLEEPUNLOCK( action->act_ccb, PRIBIO, pdrv_dev);

            /*
             * Get the lock again
             */
            PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s1 );
        }

        action->act_ccb_status =
                    ccmn_ccb_status((CCB_HEADER *)action->act_ccb);

        switch( action->act_ccb_status ) {

        case CAT_CMP:

            /*
             *  GOOD status; just return.
             */
            break;

        case CAT_CMP_ERR:

            /*
             * Had SCSI status other than GOOD;
             * must look at each possibile status and
             * determine what to do..
             */


          action->act_scsi_status = action->act_ccb->cam_scsi_status;
          switch(action->act_scsi_status)
                {
                default:
                case SCSI_STAT_GOOD:
                case SCSI_STAT_CONDITION_MET:
                case SCSI_STAT_BUSY:
```

Example D-2: (continued)

```
            case SCSI_STAT_INTERMEDIATE:
            case SCSI_STAT_INTER_COND_MET:
            case SCSI_STAT_COMMAND_TERMINATED:
            case SCSI_STAT_QUEUE_FULL:
            case SCSI_STAT_RESERVATION_CONFLICT:

            case SCSI_STAT_CHECK_CONDITION:

                /* call cgen_ccb_chkcond()
                 * to handle the check condition
                 */
                action->act_chkcond_error =
                     cgen_ccb_chkcond(action->act_ccb, pdrv_dev);

                /*
                 * Now determine what to do.
                 */
                switch ( action->act_chkcond_error ) {
                /*
                 * Look at conditions.
                 */

                case CHK_INFORMATIONAL:
                case CHK_SOFTERR:
                case CHK_EOM :
                case CHK_FILEMARK:
                case CHK_ILI:
                case CHK_CHK_NOSENSE:
                case CHK_SENSE_NOT_VALID:
                case CHK_NOSENSE_BITS:
                case CHK_NOT_READY:
                case CHK_HARDERR:
                case CHK_UNIT_ATTEN:
                case CHK_DATA_PROT:
                case CHK_UNSUPPORTED:
                case CHK_CMD_ABORTED:
                case CHK_UNKNOWN_KEY:
                default:
                        break;
                } /* end of switch for check condition */

            break; /* end of scsi_status check condition */

            } /* end of switch of scsi status */

        break; /* End of CAM_CMP_ERR */

    case CAT_INPROG:
    case CAT_UNKNOWN:
    case CAT_CCB_ERR:
    case CAT_RESET:
    case CAT_BUSY:
    case CAT_SCSI_BUSY:
```

Example D-2: (continued)

```
    case CAT_BAD_AUTO:
    case CAT_DEVICE_ERR:
    case CAT_NO_DEVICE:
    case CAT_ABORT:
        action->act_fatal |=  ACT_FAILED;
        action->act_ret_error = EIO;
        break;
    default:
    /*
     * Error log this; it should never occur
     */
        action->act_fatal |=  ACT_FAILED;
        action->act_ret_error = EIO;
        LOG_ERROR("Implement your error logging");

        break;

    } /* end switch on cam status */

    /*
     * Now unlock
     */
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);

PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
      ("[%d/%d/%d] %s: exit\n",
      DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), module));

    return;

} /* End of cgen_open_sel() */



/* ---------------------------------------------------------------- */
/* Function description.
 *
 * This routine issues a SCSI_MODE_SENSE command
 * to the unit. The CGEN_ACTION structure is filled in for the
 * the caller.  The varible sleep directs the code to sleep
 * waiting for comand status.
 *
 * Call syntax
 *  cgen_mode_sns
         ( pdrv_dev, action, done, page_code, page_cntl, sleep)
 *      PDRV_DEVICE     *pdrv_dev;
                             Peripheral Device Structure pointer
 *      CGEN_ACTION     *action;
                                Generic Action Structure pointer
 *      void            (*done)();             Completion routine
 *      u_char          page_code;               The page we want
```

Example D-2: (continued)

```
 *      u_char          page_cntl;          The page control field
 *      u_long          sleep;                    Whether we sleep
 *
 * Implicit inputs
 *      NONE
 *
 * Implicit outputs
 *      CGEN_ACTION structure is filled in based on the CCB's
 *      completion status.
 *
 * Return values
 *
 * TO DO:
 *      No sleep and state step
 *      Interrupted sleeps
 */


void
cgen_mode_sns
          ( pdrv_dev, action, done, page_code, page_cntl, sleep)
        PDRV_DEVICE     *pdrv_dev;
                        /* Peripheral Device Structure pointer */
        CGEN_ACTION     *action;
                           /* Generic Action Structure pointer */
        void            (*done)();        /* Completion routine */
        u_char          page_code;           /* The page wanted */
        u_char          page_cntl;   /* The page control field */
        u_long          sleep;              /* Whether to sleep */

{


    /*
     * Local Variables
     */

    DEV_DESC         *dev_desc = pdrv_dev->pd_dev_desc;
                        /*
                         * Device Descriptor Structure pointer
                         */
    ALL_MODE_SENSE_CDB6  *mod_cdb;  /* Mode sense CDB pointer */
    u_char               *data_buf;    /* Data buffer pointer */
    u_long               data_buf_size;
                                 /* Size of the data buffer */
    int                  s;                    /* Saved IPL */
    int                  s1;               /* Throwaway IPL */
    u_char               sense_size;
                             /* Size of request sense buffer */
    static u_char   module[] = "cgen_mode_sns"; /* Module name */


PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
```

Example D-2: (continued)

```
        DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
        ("[%d/%d/%d] %s: entry\n",
        DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
        DEV_LUN(pdrv_dev->pd_dev), module));
    /*
     * Get data_buffer size for the mode sense command which will
     * use the 6-byte mode select CDB. The mode sense data will
     * have a 4-byte parameter header and an 8-byte descriptor.
     */
  data_buf_size = sizeof(SEQ_MODE_HEAD6) + sizeof(SEQ_MODE_DESC);

     /*
      * Now get the page size
      */
    switch( page_code ) {

    case SEQ_NO_PAGE:
        /*
         * The caller of the routine does not want any page data
         * for the device. Get only the mode parameter header and
         * mode descriptor.
         */
        break;
    /*
     * Check on generic pages first.
     */

    case ALL_PGM_DISCO_RECO:
        data_buf_size += sizeof( ALL_DISC_RECO_PG);
        break;

    case ALL_PGM_PERIPH_DEVICE:
        data_buf_size += sizeof( ALL_PERIPH_DEV_PG);
        break;

    case ALL_PGM_CONTROL_MODE:
        data_buf_size += sizeof( ALL_CONTROL_PG);
        break;

    /*
     * Check on the sequential pages (tapes).
     */
    case SEQ_PGM_ERR_RECOV:
        data_buf_size += sizeof( SEQ_ERR_RECOV_PG);
        break;

    case SEQ_PGM_DEV_CONF:
        data_buf_size += sizeof( SEQ_DEV_CONF_PG);
        break;

    case SEQ_PGM_PART1:
        data_buf_size += sizeof( SEQ_PART1_PG);
        break;
```

Example D-2: (continued)

```
    case SEQ_PGM_PART2:
        data_buf_size += sizeof( SEQ_PART1_PG);
        break;

    case SEQ_PGM_PART3:
        data_buf_size += sizeof( SEQ_PART1_PG);
        break;

    case SEQ_PGM_PART4:
        data_buf_size += sizeof( SEQ_PART1_PG);
        break;

    default:
        /*
         * Invalid PAGE code.
         */
        PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
        LOG_ERROR("Implement your error logging");
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s );

        action->act_fatal |= (ACT_PARAMETER | ACT_FAILED);
        action->act_ret_error = EINVAL;
        return;
        break;

    } /* end switch */

if(( data_buf = ccmn_get_dbuf(data_buf_size)) == (u_char *)NULL){
        /*
         *  Log the error
         */
        PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
        LOG_ERROR("Implement your error logging");
        PDRV_IPLSMP_UNLOCK( pdrv_dev, s );

        action->act_fatal |= ( ACT_RESOURCE | ACT_FAILED);
        action->act_ret_error = ENOMEM;
        return;
    }
    /*
     * See if the System Administrator has set its request sense
     * size. This is for autosense. If there is an error,
     * the lower levels will do a request sense.
     */
    sense_size = GET_SENSE_SIZE( pdrv_dev );

    /*
     * Get a SCSI I/O CCB
     */
    action->act_ccb = ccmn_io_ccb_bld( pdrv_dev->pd_dev,data_buf,
        data_buf_size, sense_size,
        (u_long)CAM_DIR_IN, done,(u_char)NULL, CGEN_TIME_5,
        (struct buf *)NULL);
```

Example D-2: (continued)

```
/*
 * Check if CCB is NULL. If so, the macro
 * error logs it and fills out action return values.
 */
if(action->act_ccb == (CCB_SCSIIO *)NULL){
    CGEN_NULLCCB_ERR(action, pdrv_dev, module);
    /*
     * Release data buffer
     */
    ccmn_rel_dbuf( data_buf, data_buf_size);
    return;
}

/*
 * Build  6-byte mode select command in the CDB.
 */
mod_cdb = (ALL_MODE_SENSE_CDB6 *)
                    action->act_ccb->cam_cdb_io.cam_cdb_bytes;
mod_cdb->opcode = ALL_MODE_SENSE6_OP;
mod_cdb->lun = 0;
mod_cdb->page_code = page_code;
mod_cdb->pc = page_cntl;
mod_cdb->alloc_len = data_buf_size;

/*
 * set CDB length
 */
action->act_ccb->cam_cdb_len = sizeof(ALL_MODE_SENSE_CDB6);


/*
 * Send the mode sense command down to the lower levels.
 */
PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s);
ccmn_send_ccb( pdrv_dev, (CCB_HEADER *)
                              action->act_ccb, NOT_RETRY);
PDRV_IPLSMP_UNLOCK( pdrv_dev, s);

/*
 * Do we go to sleep.
 */
if( sleep == CGEN_NOSLEEP) {
    return;
}

/*
 * Check the CCB to make sure it is in progress
 * before going to sleep. Raise the IPL to
 * block the interrupt; the sleep will lower it.
 */
PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s );
while( CAM_STATUS( action->act_ccb) == CAM_REQ_INPROG ){
    /*
```

Example D-2: (continued)

```
             * Sleep NON-interruptable on address of CCB
             */
            PDRV_SMP_SLEEPUNLOCK( action->act_ccb, PRIBIO, pdrv_dev);

            /*
             * Get the lock again
             */
            PDRV_IPLSMP_LOCK( pdrv_dev, LK_RETRY, s1 );
        }

        action->act_ccb_status =
                    ccmn_ccb_status((CCB_HEADER *)action->act_ccb);

        switch( action->act_ccb_status ) {

        case CAT_CMP:

            /*
             *  GOOD Status; just return.
             */
            break;

        case CAT_CMP_ERR:

            /*
             * Received SCSI status other than GOOD
             * must look at each of the SCSI statuses to determine
             * our action.
             */


            action->act_scsi_status = action->act_ccb->cam_scsi_status;
            switch(action->act_scsi_status)
                {
                default:
                case SCSI_STAT_GOOD:
                case SCSI_STAT_CONDITION_MET:
                case SCSI_STAT_BUSY:
                case SCSI_STAT_INTERMEDIATE:
                case SCSI_STAT_INTER_COND_MET:
                case SCSI_STAT_COMMAND_TERMINATED:
                case SCSI_STAT_QUEUE_FULL:
                    LOG_ERROR("Implement your error logging");
                    action->act_fatal |=  ACT_FAILED;
                    action->act_ret_error = EIO;
                    break;

                case SCSI_STAT_RESERVATION_CONFLICT:
                    /*
                     * This unit reserved by another
                     * initiator this should not
                     * happen
                     */
```

Example D-2: (continued)

```
                LOG_ERROR("Implement your error logging");
                action->act_fatal |=  ACT_FAILED;
                action->act_ret_error = EBUSY;
                break;

            case SCSI_STAT_CHECK_CONDITION:

                /*
                 * Call cgen_ccb_chkcond()
                 * to handle the check condition
                 */
        action->act_chkcond_error = cgen_ccb_chkcond(action->act_ccb,
                     pdrv_dev);

                /*
                 * Now determine what to do.
                 */
                switch ( action->act_chkcond_error ) {
                /*
                 * Look at conditions.
                 */

                case CHK_INFORMATIONAL:
                    LOG_ERROR("Implement your error logging");
                    break;
                case CHK_SOFTERR:
                    LOG_ERROR("Implement your error logging");
                    break;

                case CHK_EOM :
                case CHK_FILEMARK:
                case CHK_ILI:
                case CHK_CHK_NOSENSE:
                case CHK_SENSE_NOT_VALID:
                case CHK_NOSENSE_BITS:
                case CHK_NOT_READY:
                case CHK_HARDERR:
                case CHK_UNIT_ATTEN:
                case CHK_DATA_PROT:
                case CHK_UNSUPPORTED:
                case CHK_CMD_ABORTED:
                case CHK_UNKNOWN_KEY:
                default:
                    LOG_ERROR("Implement your error logging");
                    action->act_fatal |=  ACT_FAILED;
                    action->act_ret_error = EIO;

                } /* end of switch for check condition */

            break; /* end of scsi_status check condition */

            } /* end of switch of scsi status */
```

Example D-2: (continued)

```
        break; /* End of CAM_CMP_ERR */

    case CAT_INPROG:
    case CAT_UNKNOWN:
    case CAT_CCB_ERR:
    case CAT_RESET:
    case CAT_BUSY:
    case CAT_SCSI_BUSY:
    case CAT_BAD_AUTO:
    case CAT_DEVICE_ERR:
    case CAT_NO_DEVICE:
    case CAT_ABORT:
        action->act_fatal |=  ACT_FAILED;
        action->act_ret_error = EIO;
        /*
         * Error log this; should never get this error.
         */
        LOG_ERROR("Implement your error logging");
        break;
    default:
        /*
         * Error log this; should never get this error.
         */
        action->act_fatal |=  ACT_FAILED;
        action->act_ret_error = EIO;
        LOG_ERROR("Implement your error logging");

        break;

    } /* end switch on cam status */

    /*
     * Now unlock
     */
    PDRV_IPLSMP_UNLOCK(pdrv_dev, s);


PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
      ("[%d/%d/%d] %s: exit\n",
      DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), module));

    return;

}



/* ----------------------------------------------------------- */
/*
 * Error Checking Routines
```

Example D-2: (continued)

```
 */


/* ------------------------------------------------------------ */
/* Function description. */

/*
 * cgen_ccb_chkcond()
 *
 * Routine Name : cgen_ccb_chkcond
 *
 * Functional Description:
 *
 *
 *      This routine handles the sns_data (sense data) for the
 *      GENERIC driver and returns the appropriate status to the
 *      caller.  The routine is called when a CCB_SCSIIO is
 *      returned with a CAM_STATUS of CAM_REQ_CMP_ERR ( request
 *      completed with error) and the cam_scsi_status equals
 *      SCSI_CHECK_CONDITION.
 *      NOTE...
 *      This routine must be called with the device SMP LOCKED.
 *
 *      Call Syntax:
 *
 *              cgen_ccb_chkcond( ccb, pdrv_dev )
 *                      PDRV_DEVICE *pdrv_dev;
 *                      CCB_SCSIIO  *ccb;
 *
 *      Return Values:
 *              int:
 *
 *              CHK_CHK_NOSENSE
 *                  The AUTO SENSE code, in the lower levels could
 *                  not get the request sense to complete without
 *                  error. Sense buffer not valid.
 *
 *              CHK_SENSE_NOT_VALID
 *                  The valid bit in the sense buffer is not set;
 *                  sense data is useless.
 *
 *              CHK_EOM
 *                  End of media detected.
 *
 *              CHK_FILEMARK
 *                  Filemark detected.
 *
 *              CHK_ILI
 *                  Incorrect length detected.
 *
 *              CHK_NOSENSE_BITS
 *                  Sense key equals no sense, but there are
 *                  no bits set in byte 2 of sense data.
```

Example D-2: (continued)

```
 *
 *              CHK_SOFTERR
 *                  Soft error detected; corrected by the
 *                  unit.
 *
 *              CHK_NOT_READY
 *                  The unit is not ready.
 *
 *              CHK_HARDERR
 *                  The unit has detected a hard error.
 *
 *              CHK_UNIT_ATTEN
 *                  The unit has either had a media change or
 *                  just powered up.
 *
 *              CHK_DATA_PROT
 *                  The unit is write protected.
 *
 *              CHK_UNSUPPORTED
 *                  A sense key that is unsupported
 *                  has been returned.
 *
 *              CHK_CMD_ABORTED
 *                  The unit aborted this command.
 *
 *              CHK_INFORMATIONAL
 *                  Unit is reporting an informational message.
 *
 *              CHK_UNKNOWN_KEY
 *                  The unit has returned a sense key that
 *                  is not supported by the SCSI 2 spec.
 *
 */




u_long
cgen_ccb_chkcond( ccb, pdrv_dev )

    PDRV_DEVICE *pdrv_dev;
                        /* Peripheral Device Structure pointer */
    CCB_SCSIIO  *ccb;
       /* Pointer to SCSI I/O CCB that had the check condition */

{

    /*
     * Local declarations
     */
```

Example D-2: (continued)

```
    /* Pointer to generic device-specific structure */
    CGEN_SPECIFIC *gen_spec =
                          (CGEN_SPECIFIC *)pdrv_dev->pd_specific;

    /* Pointer to the sense data */
    ALL_REQ_SNS_DATA *sns_data =
                          (ALL_REQ_SNS_DATA *)ccb->cam_sense_ptr;

    int         ret_val;  /* What we return */
    int i;
    u_short     asc_asq; /* The combined asc(MSB) and asq(LSB) */
    static u_char module[] = "cgen_ccb_chkcond"; /* Module name */



PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC |CAMD_INOUT),
      ("[%d/%d/%d] %s: entry\n",
      DEV_BUS_ID(pdrv_dev->pd_dev), DEV_TARGET(pdrv_dev->pd_dev),
      DEV_LUN(pdrv_dev->pd_dev), module));

    /*
     * Check to see if there is valid sense data
     */
    if(( ccb->cam_ch.cam_status & CAM_AUTOSNS_VALID) == NULL){
        /*
         * Sense data is not valid, so return CHK_CHK_NOSENSE.
         */
        return( CHK_CHK_NOSENSE );
    }
    if( sns_data == NULL ) {
        panic("cgen_ccb_chkcond:
                    CCB-AUTOSNS_VALID but data pointer = NULL");
        return(CHK_CHK_NOSENSE);
    }

    /*
     * Sense data is valid; find out why
     * and report it.
     */

    PRINTD(DEV_BUS_ID(pdrv_dev->pd_dev),
        DEV_TARGET(pdrv_dev->pd_dev),
        DEV_LUN(pdrv_dev->pd_dev), (CAMD_GENERIC ),
        ("[%d/%d/%d] %s: error_code,
                        0x%x sense_key 0x%x asc 0x%x asq 0x%x\n",
        DEV_BUS_ID(pdrv_dev->pd_dev),
        DEV_TARGET(pdrv_dev->pd_dev),
        DEV_LUN(pdrv_dev->pd_dev), module,
        sns_data->error_code, sns_data->sns_key, sns_data->asc,
        sns_data->asq));
```

Example D-2: (continued)

```
/*
 * Make sure that the error code is valid. The only valid
 * error codes defined in SCSI 2 are 0x70 and 0x71
 */
if(( sns_data->error_code != 0x70) &&
            ( sns_data->error_code != 0x71 )){

    return( CHK_SENSE_NOT_VALID );
}

/*
 * Get the sense key and check each case
 */

switch(sns_data->sns_key ){

case ALL_NO_SENSE:
    /*
     * Must look at the bit fields
     */
    if( sns_data->filemark != NULL){
        /*
         * Set flag
         */
        gen_spec->gen_flags |= CGEN_TPMARK;
        BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
        ret_val = CHK_FILEMARK;

        break;
    }
    else if( sns_data->eom != NULL){
        /*
         * Set flag
         */
    gen_spec->gen_flags |= CGEN_EOM;
        CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
        ret_val = CHK_EOM;

        break;
    }
    else if( sns_data->ili != NULL){
        /*
         * Set flag
         */
        gen_spec->gen_flags |= CGEN_SHRTREC;
        CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
        ret_val = CHK_ILI;

        break;
    }
    else {
        /*
         * Nothing is set, so more than likely an
```

Example D-2: (continued)

```
                 * informational warning has been sent. Make sure
                 * that all the data went across. If it did not,
                 * then the device has a problem.
                 *
                 * Check to see if there is a residual count.  If
                 * there is, fail it. ret_val == CHK_NOSENSE_BITS
                 * else CHK_INFORMATIONAL
                 */

                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);

                if(gen_spec->ts_resid != NULL){
                    ret_val = CHK_NOSENSE_BITS;
                }
                else {
                    ret_val = CHK_INFORMATIONAL;
                }


                break;
            }

        case ALL_BLANK_CHECK:
        case ALL_VOL_OVERFLOW:
                /*
                 * End of media, set the flag
                 */
                gen_spec->gen_flags |= CGEN_EOM;
                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
                ret_val = CHK_EOM;

                break;

        case ALL_RECOVER_ERR:
                /*
                 * Soft error
                 */
                gen_spec->gen_flags |= CGEN_SOFTERR;
                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
                ret_val = CHK_SOFTERR;

                break;

        case ALL_NOT_READY:
                gen_spec->gen_flags |= CGEN_OFFLINE;
                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
                ret_val = CHK_NOT_READY;

                break;

        case ALL_MEDIUM_ERR:
                if( sns_data->eom != NULL){
                    /*
```

Example D-2: (continued)

```
                 * Set flag
                 */
            gen_spec->gen_flags |= CGEN_EOM;
            CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
            ret_val = CHK_EOM;

                break;
            }
            /*
             * Hard error on the device
             */
            gen_spec->gen_flags |= CGEN_HARDERR;
            CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
            ret_val = CHK_HARDERR;

            break;

    case ALL_HARDWARE_ERR:
    case ALL_ILLEGAL_REQ:
    case ALL_COPY_ABORT:
    case ALL_MISCOMPARE:
            /*
             * Hard error on the device
             */
            gen_spec->gen_flags |= CGEN_HARDERR;
            CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
            ret_val = CHK_HARDERR;

            break;

    case ALL_UNIT_ATTEN:
            /*
             * Unit has had a media change or has
             * been powered up.
             */
            gen_spec->gen_state_flags |= CGEN_UNIT_ATTEN_STATE;
            CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
            ret_val = CHK_UNIT_ATTEN;

            break;

    case ALL_DATA_PROTECT:
            /*
             * Unit is write protected
             */
            gen_spec->gen_flags |= CGEN_WRT_PROT;
            CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
            ret_val = CHK_DATA_PROT;

            break;

    case ALL_VENDOR_SPEC:
    case ALL_EQUAL:
```

Example D-2: (continued)

```
                /*
                 *These are not supported for this unit.
                 */
                ret_val = CHK_UNSUPPORTED;
                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);

                break;

        case ALL_ABORTED_CMD:
                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
                ret_val = CHK_CMD_ABORTED;

                break;

        default:
                /*
                 * Unknown sense key
                 */
                CGEN_BTOL(&sns_data->info_byte3, gen_spec->ts_resid);
                ret_val = CHK_UNKNOWN_KEY;

                break;
        }

PRINTD(DEV_BUS_ID(pdrv_device->pd_dev),
      DEV_TARGET(pdrv_device->pd_dev),
      DEV_LUN(pdrv_device->pd_dev),
      (CAMD_GENERIC |CAMD_INOUT),
      ("[%d/%d/%d] %s: exit\n",
      DEV_BUS_ID(pdrv_device->pd_dev),
      DEV_TARGET(pdrv_device->pd_dev),
      DEV_LUN(pdrv_device->pd_dev), module));

        /*
         * Return result of the checks.
         */


        return(ret_val);


}




/*****************************************************************
 *
 *   ROUTINE NAME: cgen_minphys()
 *
 *   FUNCTIONAL DESCRIPTION:
 *       This function compares the b_bcount field in the buf
 *       structure with the maximum transfer limit for the device
```

Example D-2: (continued)

```
 *      (dd_max_record) in the Device Descriptor Structure. The
 *      count is adjusted if it is greater than the limit.
 *
 *  FORMAL PARAMETERS:
 *      bp - Buf structure pointer.
 *
 *  IMPLICIT INPUTS:
 *      None.
 *
 *  IMPLICIT OUTPUTS:
 *      Modified b_bcount field of buf structure.
 *
 *  RETURN VALUE:
 *      None.
 *
 *  SIDE EFFECTS:
 *      None.
 *
 *  ADDITIONAL INFORMATION:
 *      None.
 *
 ****************************************************************/

void
cgen_minphys(bp)
struct buf *bp;
{

    PDRV_DEVICE *pdrv_dev;
                    /* Peripheral Device Structure pointer */
    DEV_DESC    *dd;            /* Device Descriptor Structure */


PRINTD(DEV_BUS_ID(bp->b_dev), DEV_TARGET(bp->b_dev),
   DEV_LUN(bp->b_dev), CAMD_GENERIC,
   ("[%d/%d/%d] cgen_minphys: entry bp=%xx bcount=%xx\n",
   DEV_BUS_ID(bp->b_dev), DEV_TARGET(bp->b_dev),
   DEV_LUN(bp->b_dev),
   bp, bp->b_bcount));

if ( (pdrv_dev = GET_PDRV_PTR(bp->b_dev)) == (PDRV_DEVICE *)NULL)
   {
    PRINTD(DEV_BUS_ID(bp->b_dev), DEV_TARGET(bp->b_dev),
         DEV_LUN(bp->b_dev), CAMD_GENERIC,
    ("[%d/%d/%d] cgen_minphys: No periheral device struct\n",
    DEV_BUS_ID(bp->b_dev), DEV_TARGET(bp->b_dev),
    DEV_LUN(bp->b_dev)));
    return;
    }

    dd = pdrv_dev->pd_dev_desc;

    /*
```

Example D-2: (continued)

```
     * Get the maximun transfer size for this device. If b_bcount
     * is greater than maximum, then adjust it.
     */
    if (bp->b_bcount > dd->dd_max_record ){
        bp->b_bcount = dd->dd_max_record;
    }
      PRINTD(DEV_BUS_ID(bp->b_dev), DEV_TARGET(bp->b_dev),
        DEV_LUN(bp->b_dev), CAMD_GENERIC,
        ("[%d/%d/%d] cgen_minphys: exit - success\n",
        _BUS_ID(bp->b_dev), DEV_TARGET(bp->b_dev),
        DEV_LUN(bp->b_dev)));
}
```

Index

D

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[a]	————	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Writing Device Drivers for the
SCSI/CAM Architecture Interfaces
AA-PS3GB-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

What do you like best about this manual? ______________________________

____

What do you like least about this manual? ______________________________

____

Please list errors you have found in this manual:

Page ____ Description ______________________________

____ ______________________________

____ ______________________________

____ ______________________________

____ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

What version of the software described by this manual are you using? ____________

Name/Title ______________________ Dept. ____________

Company ______________________ Date ____________

Mailing Address ______________________________

______________ Email ______________ Phone ____________

---- Do Not Tear – Fold Here and Tape --

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here --

Cut
Along
Dotted
Line

Reader's Comments

DEC OSF/1
Writing Device Drivers for the
SCSI/CAM Architecture Interfaces
AA-PS3GB-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

What do you like best about this manual? ______________________________

What do you like least about this manual? ______________________________

Please list errors you have found in this manual:

Page	Description
______	______________________________
______	______________________________
______	______________________________
______	______________________________
______	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

What version of the software described by this manual are you using? __________

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________ Email ______________ Phone __________

---- Do Not Tear – Fold Here and Tape --

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here --

**Cut
Along
Dotted
Line**