IBM Ultrium Device Drivers

Programming Reference

GC35-0483-01

IBM Ultrium Device Drivers

Programming Reference

GC35-0483-01

Note!

Before using this information and the product that it supports, be sure to read the general information under “Notices” on page 211.

Second Edition (January 2003)

This edition replaces and makes obsolete GC35-0483-00. Changes or additions are indicated by a vertical line in the left margin.

Contents

Preface

This publication provides programming reference information for IBM® Ultrium™ tape drive, medium changer, and library device drivers.

Special Printing Instructions

This SCSI Device Driver Manual contains different sections for each type of operating platform; for example, AIX, HP-UX, Linux, Sun Solaris, Windows®, and a separate section on these operating systems for the 3494 Enterprise Tape Library.

Note: When selecting the page range for the section you wish to print, note that the print page range is based on the page controls for Adobe Acrobat, not the page printed on the actual document. Enter the Adobe page numbers to print.

If you wish to print one or more separate sections of the manual, follow these steps:

1. Navigate to the beginning of the section and note the page number.
2. Navigate to the last page in the section and note that page number.
3. Select File —>Print, then choose ″Pages″ and enter the page range for the section. Only the page range entered will print.
4. Repeat these steps to print additional sections.

Important printer note

This area indicates the pages that will actually print in your specified range of pages.

Ignore the page number appearing on the page itself when entering page ranges for your printer.

Attention: There is only one Table of Contents and one Index for this entire book. If you wish to print those items, you must repeat the process above, entering the page range of the Table of Contents and the Index page range, respectively.

Related Information

The following sections contain lists of sources that you may need for information related to the IBM Ultrium tape drive, medium changer, and library device drivers.

IBM 3580 Ultrium Tape Drive

The following publication relates to the IBM 3580 Ultrium Tape Drive:

- *IBM 3580 Ultrium Tape Drive Setup, Operator, and Service Guide*, GA32-0415

- *IBM 3580 Ultrium Tape Drive and StorageSmart™ by IBM Ultrium Tape Drive TX200 SCSI Reference*, WB1109

IBM 3581 Ultrium Tape Autoloader

The following publication relates to the IBM 3581 Ultrium Tape Autoloader:

- *IBM 3581 Ultrium Tape Autoloader Setup, Operator, and Service Guide*, GA32-0412

IBM 3583 Ultrium Scalable Tape Library

The following publications relate to the IBM 3583 Ultrium Scalable Tape Library:

- *IBM 3583 Ultrium Scalable Tape Library Setup and Operator Guide*, GA32-0411
- *IBM 3583 Ultrium Scalable Tape Library Service Guide*, GA32-0425

IBM 3584 UltraScalable Tape Library

The following publications relate to the IBM 3584 UltraScalable Tape Library:

- *IBM 3584 UltraScalable Tape Library Planning and Operator Guide*, GA32-0408
- *IBM 3584 UltraScalable Tape Library Maintenance Information*, 19P2440

StorageSmart by IBM Ultrium External Tape Drive TX200 Machine Type 3585

The following publications relate to the StorageSmart by IBM Ultrium External Tape Drive TX200 Machine Type 3585:

- *StorageSmart by IBM Ultrium External Tape Drive TX200 Machine Type 3585 Setup, Operator, and Service Guide*, GA32-0421
- *StorageSmart by IBM Ultrium External Tape Drive TX200 Machine Type 3585 Quick Reference*, GX35-5061
- *IBM 3580 Ultrium Tape Drive and StorageSmart by IBM Ultrium Tape Drive TX200 SCSI Reference*, WB1109

StorageSmart by IBM Ultrium Tape Autoloader SL7 Machine Type 3586

The following publications relate to the StorageSmart by IBM Ultrium Tape Autoloader SL7 Machine Type 3586:

- *StorageSmart by IBM Ultrium Tape Autoloader SL7 Machine Type 3586 Setup, Operator, and Service Guide*, GA32-0423
- *StorageSmart by IBM Ultrium Tape Autoloader SL7 Machine Type 3586 Quick Reference*, GX35-5058
- *StorageSmart by IBM Ultrium Tape Autoloader SL7 Machine Type 3586 SCSI Reference*, WB1105

StorageSmart by IBM Ultrium Scalable Tape Library SL72 Machine Type 3587

The following publications relate to the StorageSmart by IBM Ultrium Scalable Tape Library SL72 Machine Type 3587:

- *StorageSmart by IBM Ultrium Scalable Tape Library SL72 Machine Type 3587 Setup, Operator, and Service Guide*, GA32-0425
- *StorageSmart by IBM Ultrium Scalable Tape Library SL72 Machine Type 3587 Quick Reference*, GX35-5059
- *StorageSmart by IBM Ultrium Scalable Tape Library SL72 Machine Type 3587 Maintenance Information*, SA37-0427

- *StorageSmart by IBM Ultrium Scalable Tape Library SL72 Machine Type 3587 SCSI Reference*, WB1106

AIX®

The following publication relates to AIX systems:

- *IBM AIX Parallel and ESCON® Channel Tape Attachment/6000 Installation and User's Guide*, GA32-0311

HP-UX

The following publication relates to HP-UX systems:

- *HP-UX Reference, Volumes 1, 2, and 3*
  Hewlett-Packard Company, Part B2355-90033
- *System Administration Tasks, HP-UX Release 9.0*
  Hewlett-Packard Company, Part B2355-90040

Linux

The following publication relates to Linux systems:

- *System Administration Tasks, HP-UX Release 9.0*

Solaris (Sun)

The following publication relates to Solaris (Sun) systems:

- *Solaris 2.x: Adding and Maintaining Peripherals*
- *SunOS 5.x: User's Guide to System Administration*
- *SunOS 5.x: Reference Manual (Sections 1 through 9)*

Additional Information

The following publications contain additional information that to relates to the IBM Ultrium™ tape drive, medium changer, and library device drivers:

- *IBM Ultrium Tape Drive, Medium Changer, and Library Device Drivers: Installation and User's Guide*, GA32-0430
- *American National Standards Institute Small Computer System Interface*
  X3T9.2/86-109 X3.180, X3B5/91-173C, X3B5/91-305, X3.131-199X Revision 10H, and X3T9.9/91-11 Revision 1

Part 1. AIX Tape and Medium Changer Device Driver

Chapter 1. Software Interface

Tape Devices

The AIX tape and medium changer device driver supports the following AIX entry points for tape devices:

- Open

  This entry point is driven by *open*, *openx*, and *creat* subroutines.
- Write

  This entry point is driven by *write*, *writev*, *writex*, and *writevx* subroutines.
- Read

  This entry point is driven by *read*, *readv*, *readx*, and *readvx* subroutines.
- Close

  This entry point is driven explicitly by the *close* subroutine and implicitly by the operating system at program termination.
- IOCTL

  This entry point provides a set of tape and SCSI specific functions. It allows AIX applications to access and control the features and attributes of the tape device programmatically. For the medium changer devices, it also provides a set of medium changer functions that is accessed through the tape device special files or independently through an additional special file for the medium changer only.
- Dump

  This entry point allows the use of the AIX dump facility with the driver.

The standard set of AIX device management commands is available. The *chdev*, *rmdev*, *mkdev*, and *lsdev* commands are used to bring the device online or change the attributes that determine the status of the tape device.

Medium Changer Devices

The AIX tape and medium changer device driver supports the following AIX entry points for the medium changer devices:

- Open

  This entry point is driven by *open* and *openx* subroutines.
- Close

  This entry point is driven explicitly by the *close* subroutine and implicitly by the operating system at program termination.
- IOCTL

  This entry point provides a set of medium changer and SCSI specific functions. It allows AIX applications to access and control the features and attributes of the tape system robotic device programmatically.

The standard set of AIX device management commands is available. The *chdev*, *rmdev*, *mkdev*, and *lsdev* commands are used to bring the device online or change the attributes that determine the status of the tape system robotic device.

Chapter 2. Special Files

After the driver is installed and a tape device is configured and made available for use, access is provided through the special files. These special files, which consist of the standard AIX special files for tape devices (along with other files unique to the Atape driver), are in the */dev* directory.

Special Files for Tape Devices

Each tape device has a set of special files that provides access to the same physical drive, but to different types of functions. As shown in Table 1, in addition to the tape special files, a special file is provided for tape devices, which allows access to the medium changer as a separate device. The asterisk (*) represents a number assigned to a particular device (such as *rmt0*).

Table 1. Special Files for Tape Devices

Special File Name	Rewind on Close	Retension on Open	Bytes per Inch	Trailer Label	Unload on Close
/dev/rmt*	Yes	No	N/A	No	No
/dev/rmt*.1	No	No	N/A	No	No
/dev/rmt*.2	Yes	Yes	N/A	No	No
/dev/rmt*.3	No	Yes	N/A	No	No
/dev/rmt*.4	Yes	No	N/A	No	No
/dev/rmt*.5	No	No	N/A	No	No
/dev/rmt*.6	Yes	Yes	N/A	No	No
/dev/rmt*.7	No	Yes	N/A	No	No
/dev/rmt*.10	No	No	N/A	No	No
/dev/rmt*.20	Yes	No	N/A	No	Yes
/dev/rmt*.40	Yes	No	N/A	Yes	No
/dev/rmt*.41	No	No	N/A	Yes	No
/dev/rmt*.60	Yes	No	N/A	Yes	Yes
/dev/rmt*.null	Yes	No	N/A	No	No
/dev/rmt*.smc	N/A	N/A	N/A	N/A	N/A

Notes:

1. The Rewind on Close special files write filemarks under certain conditions before rewinding. See “Closing the Special File” on page 9.
2. The Retension on Open special files rewind the tape only on open. Retensioning is not performed because these tape products perform the retension operation automatically when needed.
3. The Bytes per Inch options are ignored for the tape devices supported by this driver. The density selection is automatic.
4. The *rmt*.null* file is a pseudo device similar to the */dev/null* AIX special file. The *ioctl* calls can be issued to this file without a real device attached to it, and the device driver will return a successful completion. Read and write system calls will return the requested number of bytes. This file can be used for application development or debugging problems.
5. The *rmt*.smc* file can be opened independently of the other tape special files.
6. The *rmt*.10* file bypasses normal close processing, and the tape is left at the current position.

AIX Device Driver (Atape)

For tape drives with attached SCSI medium changer devices, the *rmt*.smc* special file provides a separate path for issuing commands to the medium changer. When this special file is opened, the application can view the medium changer as a separate SCSI device.

Both this special file and the *rmt** special file can be opened at the same time. The file descriptor that results from opening the *rmt*.smc* special file does not support the following operations:

- Read
- Write
- Open in diagnostic mode
- Commands designed for a tape device

If a tape drive has a SCSI medium changer device attached, then all operations (including the medium changer operations) are supported through the interface to the *rmt** special file.

Special Files for Medium Changer Devices

After the driver is installed and a medium changer device is configured and made available for use, access to the robotic device is provided through the *smc** special file in the */dev* directory.

Table 2 shows the attributes of the special file. The asterisk (*) represents a number assigned to a particular device (such as *smc0*). The term *smc* is used for a SCSI medium changer device. The *smc** special file provides a path for issuing commands to control the medium changer robotic device.

Table 2. Special Files

Special File Name	Description
/dev/smc*	Access to the medium changer robotic device
/dev/smc*.null	Pseudo medium changer device
Note: The *smc*.null* file is a pseudo device similar to the */dev/null* AIX special file. The commands can be issued to this file without a real device attached to it, and the device driver will return a successful completion. This file can be used for application development or debugging problems.	

The file descriptor that results from opening the *smc* special file does not support the following operations:

- Read
- Write
- Commands designed for a tape device

Opening the Special File for I/O

Several options are available when a file is opened for access. These options, known as O_FLAGS, affect the characteristics of the opened tape device or the result of the open operation. The open command is:

```
tapefd=open("/dev/rmt0",O_FLAGS);
smcfd=open("/dev/smc0",O_FLAGS);
```

The O_FLAGS parameter has the following flags:

- O_RDONLY

  This flag allows only operations that do not change the content of the tape. The flag is ignored if it is used to open the *smc* special files.

- O_RDWR

  This flag allows complete access to the tape. The flag is ignored if it is used to open the *smc* special files.

- O_WRONLY

  This flag does not allow the tape to be read. All other operations are allowed. The flag is ignored if it is used to open the *smc* special files.

- O_NDELAY or O_NONBLOCK

  These two flags perform the same function. The driver does not wait until the device is ready before opening and allowing commands to be sent. If the device is not ready, then subsequent commands (that require that the device is ready or a physical tape is loaded) fail with ENOTREADY. Other commands (such as gather the inquiry data) complete successfully.

- O_APPEND

  When a file is opened, the tape is positioned after the last file that was written to the tape. This status is the same if the tape was opened previously with a No Rewind on Close special file. This process can take several minutes for a full tape. The flag is ignored if it is used to open the *smc* special files.

  This flag must be used in conjunction with the O_WRONLY flag to append data to the end of the current data on the tape. The O_RDONLY or O_RDWR flag is illegal in combination with the O_APPEND flag.

  Note: This flag cannot be used with the Retension on Open special files, such as *rmx.2*.

If the *open()* system call fails, the *errno* value contains the error code. See Chapter 7, “Return Codes”, on page 49 for a description of the *errno* values.

Using the Extended Open Operation

An extended open operation is also supported on the device. This operation allows special types of processing to occur during the opening and subsequent closing of the tape device. The extended open command is:

```
tapefd=openx("/dev/rmt0",O_FLAGS,NULL,E_FLAGS);
smcfd=openx("/dev/smc0",O_FLAGS,NULL,E_FLAGS);
```

The O_FLAGS parameter provides the same options described in “Opening the Special File for I/O” on page 6. The third parameter is always NULL. The E_FLAGS parameter provides the extended options. The E_FLAGS values can be combined during an open operation, or they can be used with an OR operation.

The E_FLAGS parameter has the following flags:

- SC_RETAIN_RESERVATION

  This flag prevents the SCSI Release command from being sent during a close operation. You must have *root* or *superuser* authority to perform this function.

- SC_FORCED_OPEN

  The Bus Device Reset command is sent to the target device during an open operation. This flag forces the release of any reservation obtained by any initiator on the bus. You must have *root* or *superuser* authority to perform this function.

- SC_DIAGNOSTIC

The device is opened in diagnostic mode, and no SCSI commands are sent to the device during an open operation or a close operation. All operations (such as reserve and mode select) must be processed by the application. You must have *root* or *superuser* authority to perform this function.

- SC_NO_RESERVE

  This flag prevents the SCSI Reserve command from being sent during an open operation. You must have *root* or *superuser* authority to perform this function.
- SC_PASSTHRU

  No SCSI commands are sent to the device during an open operation or a close operation. All operations (such as reserve the device, release the device, and set the tape parameters) must be processed explicitly by the application.
- SC_FEL

  This flag turns the forced error logging on in the tape device for read and write operations.
- SC_NO_ERRORLOG

  This flag turns off the AIX error logging for all read, write, or *ioctl* operations.
- SC_TMCP

  This flag allows a single process to open a device even if the device is already open by another process. Only a limited set of *ioctl* commands can be used when opening with this flag, and, currently, only the following commands are available:
  - SIOC_QUERY_OPEN
  - SIOC_QUERY_PATH

  If another process already has the device open with this flag, the open will fail and the errno is set to EAGAIN.

If the *open()* system call fails, the *errno* value contains the error code. See Chapter 7, “Return Codes”, on page 49 for a description of the *errno* values.

Writing to the Special File

Several subroutines allow writing the data to a tape. The basic write command is:

```
count=write(tapefd, buffer, numbytes);
```

The write operation returns the number of bytes written during the operation. It can be less than the value in *numbytes*. If the block size is fixed (block_size≠0), then the *numbytes* value must be a multiple of the block size. If the block size is variable, then the value specified in *numbytes* is written. If the *count* is less than 0, then the *errno* value contains the error code returned from the driver.

See Chapter 7, “Return Codes”, on page 49 for a description of the *errno* values.

The *writev*, *writex*, and *writevx* subroutines are also supported. Any values passed in the *ext* field by using the extended write operation are ignored.

Reading from the Special File

Several subroutines allow reading the data from a tape. The basic read command is:

```
count=read(tapefd, buffer, numbytes);
```

The read operation returns the number of bytes read during the operation. It can be less than the value in *numbytes*. If the block size is fixed (block_size≠0), then the *numbytes* value must be a multiple of the block size. If the *count* is less than 0, then the *errno* value contains the error code returned from the driver.

See Chapter 7, “Return Codes”, on page 49 for a description of the *errno* values.

If the block size is variable, then the value specified in *numbytes* is read. If the blocks read are smaller than requested, then the block is returned up to the maximum size of one block. If the blocks read are greater than requested, then an error occurs with the error set to ENOMEM.

Reading a filemark returns a value of zero and positions the tape after the filemark. Continuous reading (after EOM is reached) results in a value of zero and no further change in the tape position.

The *readv* subroutine is also supported.

Reading with the TAPE_SHORT_READ Extended Parameter

For normal read operations, if the block size is set to variable (0) and the amount of data in a block on the tape is more than the number of bytes requested in the call, an ENOMEM error is returned to the application. An application can read fewer bytes without an error by using the *readx* or *readvx* subroutine and specifying the TAPE_SHORT_READ extended parameter:

```
count=readx(tapefd, buffer, numbytes, TAPE_SHORT_READ);
```

The TAPE_SHORT_READ parameter is defined in the */usr/include/sys/tape.h* header file.

Closing the Special File

Closing a special file is a simple process. The file descriptor that is returned by the Open command is used to close the command:

```
rc=close(tapefd);
rc=close(smcfd);
```

The return code from the close operation should be checked by the application. If the return code is not zero, the *errno* value is set during a close operation to indicate that a problem occurred while closing the special file. The *close* subroutine tries to perform as many operations as possible even if there are failures during portions of the close operation. If the device driver cannot terminate the file correctly with filemarks, then it tries to close the connection. If the close operation fails, then consider the device closed and try another open operation to continue processing the tape. After a close failure, assume that the data or the tape is inconsistent.

For the tape drives, the result of a close operation depends on which special file was used during the open operation and which tape operation was performed while it was opened. The SCSI commands are issued according to the following logic:

```
If the last tape operation was a WRITE command
   Write 2 filemarks on tape
   If special file is Rewind on Close (Example: /dev/rmt0)
     Rewind tape
   If special file is a No-Rewind on Close (Example: /dev/rmt0.1)
     Backward space 1 filemark (tape is positioned to append next file)
```

```
If the last tape operation was a WRITE FILEMARK command
   Write 1 filemark on tape
   If special file is Rewind on Close (Example: /dev/rmt0)
     Rewind tape
   If special file is a No-Rewind on Close (Example: /dev/rmt0.1)
     Backward space 1 filemark (tape is positioned to append next file)

If the last tape operation was a READ command
   If special file is Rewind on Close (Example: /dev/rmt0)
     Rewind tape
   If special file is a No-Rewind on Close (Example: /dev/rmt0.1)
     Forward space to next filemark (tape is positioned to read or append next file)

If the last tape operation was NOT a READ, WRITE, or WRITE FILEMARK command
   If special file is Rewind on Close (Example: /dev/rmt0)
     Rewind tape
  If special file is a No-Rewind on Close (Example: /dev/rmt0.1)
     No commands are issued, tape remains at the current position
```

Chapter 3. Device and Volume Information Logging

The device driver provides a logging facility that saves information about the device and the media. The information is extensive for some devices and limited for other devices. If this feature is set to on either by configuration or the STIOCSETP *ioctl*, the device driver logging facility gathers all available information through the SCSI Log Sense command.

This process is a separate facility from error logging. Error logging is routed to the system error log. Device information logging is sent to a separate file.

The following parameters control this utility:

- Logging
- Maximum size of the log file
- Volume ID for logging

See the *IBM SCSI Tape Drive, Medium Changer, and Library Device Drivers: Installation and User's Guide* for a description of these parameters.

Each time an unload command or the STIOC_LOG_SENSE *ioctl* command is issued, the log sense data is collected and an entry is added to the log. Each time a new cartridge is loaded, the log sense data in the tape device is reset; therefore, the log data is gathered on a per-volume basis.

Log File

The data is logged in the */usr/adm/ras* directory. The file name is dependent on each device; therefore, each device has a separate log. An example of the *rmt1* device file is:

```
/usr/adm/ras/Atape.rmt1.log
```

The files are in binary format. Each entry has a header followed by the raw Log Sense pages as defined for a particular device.

The first log page is always page 0x00. This page, as defined in the SCSI-2 ANSI specification, contains all of the pages supported by the device. Page 0x00 is followed by all of the pages specified in page 0x00. The format of each following page is defined in the SCSI specification and the device manual.

The format of the file is defined by the data structure. The *logfile_header* is followed by *max_log_size* (or a fewer number of entries for each file). The *log_record_header* is followed by a log entry.

The data structure for log recording is:

```
struct logfile_header
  {
  char owner[16];          /* module that created the file */
  time_t when;             /* time when file created */
  unsigned long count;     /* number of entries in file */
  unsigned long first;     /* first entry number in wrap queue */
  unsigned long max;       /* maximum entries allowed before wrap */
  unsigned long size;      /* size of entry (bytes), entry size is fixed */
  };
struct log_record_header
  {
```

```
  time_t when;              /* time when log entry made */
  ushort type;              /* log entry type */
    #define LOGDEMOUNT   1 /* demount log entry */
    #define LOGSENSE     2 /* log sense ioctl entry */
    #define LOGOVERFLOW  3 /* log overflow entry */
  char device_type[8];      /* device type that made entry */
  char volid[16];           /* volume ID of entry */
  char serial[12];          /* serial number of device */
  reserved[12];
  };
```

The format of the log file is:

logfile_header
log_record_header
log_record_entry
•
•
•
•
log_record_header
log_record_entry

Each log_record_entry contains multiple log sense pages. The log pages are placed in order one after another. Each log page contains a header, which is followed by the page contents.

The data structure for the header of the log page is:

```
struct log_page_header
  {
  char code;                /* page code */
  char res;                 /* reserved */
  unsigned short len;       /* length of data in page after header */
  };
```

Chapter 4. General IOCTL Operations

This chapter describes the set of *ioctl* commands that provides control and access to the SCSI tape and medium changer devices. These commands are available for all of the tape and medium changer devices. They can be issued to any *rmt**, *rmt*.smc*, or *smc** special file.

Overview

The following *ioctl* commands are supported:

IOCINFO	Query the device information.
STIOCMD	Issue the pass-through command.
SIOC_INQUIRY	Return the inquiry data.
SIOC_REQSENSE	Return the sense data.
SIOC_RESERVE	Reserve the device.
SIOC_RELEASE	Release the device.
SIOC_TEST_UNIT_READY	Issue the SCSI Test Unit Ready command.
SIOC_LOG_SENSE_PAGE	Return the log sense data.
SIOC_MODE_SENSE_PAGE	Return the mode sense data.
SIOC_INQUIRY_PAGE	Return the inquiry data for a specific page.
SIOC_QUERY_PATH	Query device and SCSI path information.
SIOC_QUERY_OPEN	Returns the ID of the process that currently has a device opened.

These *ioctl* commands and their associated structures are defined in the */usr/include/sys/Atape.h* header file, which is included in the corresponding C program using the functions.

A sample *tapeutil.c* program is provided with the device driver in the */usr/lpp/Atape/samples* directory that demonstrates the use of these *ioctl* commands.

IOCINFO

This *ioctl* command provides access to information about the tape or medium changer device. It is a standard AIX *ioctl* function.

An example of the IOCINFO command is:

```
#include <sys/devinfo.h>
#include <sys/Atape.h>
struct devinfo info;

if (!ioctl (fd, IOCINFO, &info))
  {
      printf ("The IOCINFO ioctl succeeded\n");
  }
  else
  {
      perror ("The IOCINFO ioctl failed");
  }
```

An example of the output data structure for a tape drive *rmt** special file is:

```
info.devtype=DD_SCTAPE
info.un.scmt.type=DT_STREAM
info.un.scmt.blksize=tape block size (0=variable)
```

An example of the output data structure for an integrated medium changer *rmt*.smc* special file is:

```
info.devtype=DD_MEDIUM_CHANGER;
```

An example of the output data structure for an independent medium changer *smc** special file is:

```
info.devtype=DD_MEDIUM_CHANGER;
```

See the *Atape.h* header file for the defined *devsubstype* values.

STIOCMD

This *ioctl* command issues the SCSI pass-through command. It is used by the diagnostic and service aid routines. The structure for this command is in the */usr/include/sys/scsi.h* file.

This *ioctl* is supported on both SCSI adapter attached devices and FCP adapter attached devices. For FCP adapter devices, the *adapter_status* field returned is converted from the FCP codes defined in */usr/include/sys/scsi_buf.h* to the SCSI codes defined in */usr/include/sys/scsi.h*, if possible. This is to provide downward compatibility with existing applications that use the STIOCMD *ioctl* for SCSI attached devices.

Note: There is no interaction by the device driver with this command. The error-handling and logging functions are disabled. If the command results in a check condition, then the application must issue a Request Sense command to clear any contingent allegiance with the device.

An example of the STIOCMD command is:

```
struct sc_iocmd sciocmd;
struct inquiry_data inqdata;

bzero(&sciocmd, sizeof(struct sc_iocmd));
bzero(&inqdata, sizeof(struct inquiry_data));

/* issue inquiry */
sciocmd.scsi_cdb[0]=0x12;
sciocmd.timeout_value=200;    /* SECONDS */
sciocmd.command_length=6;
sciocmd.buffer=(char *)&inqdata;
sciocmd.data_length=sizeof(struct inquiry_data);
sciocmd.scsi_cdb[4]=sizeof(struct inquiry_data);
sciocmd.flags=B_READ;

if (!ioctl (sffd, STIOCMD, &sciocmd))
  {
      printf ("The STIOCMD ioctl for Inquiry Data succeeded\n");
      printf ("\nThe inquiry data is:\n");
      dump_bytes (&inqdata, sizeof(struct inquiry_data),"Inquiry Data");
  }
  else
  {
      perror ("The STIOCMD ioctl for Inquiry Data failed");
  }
```

SIOC_INQUIRY

This *ioctl* command collects the inquiry data from the device.

The data structure is:

```
struct inquiry_data
{
    uint  qual:3,                /* peripheral qualifier */
          type:5;                /* device type */
    uint  rm:1,                  /* removable medium */
          mod:7;                 /* device type modifier */
    uint  iso:2,                 /* ISO version */
          ecma:3,                /* EMCA version */
          ansi:3;                /* ANSI version */
    uint  aenc:1,                /* asynchronous event notification */
          trmiop:1,              /* terminate I/O process message */
          :2,                    /* reserved */
          rdf:4;                 /* response data format */
    uchar len;                   /* additional length */
    uchar resvd1;                /* reserved */
    uint  :4,                    /* reserved */
          mchngr:1,              /* medium changer mode (SCSI-3 only) */
          :3;                    /* reserved */
    uint  reladr:1,              /* relative addressing */
          wbus32:1,              /* 32-bit wide data transfers */
          wbus16:1,              /* 16-bit wide data transfers */
          sync:1,                /* synchronous data transfers */
          linked:1,              /* linked commands */
          :1,                    /* reserved */
          cmdque:1,              /* command queueing */
          sftre:1;               /* soft reset */
    uchar vid[8];                /* vendor ID */
    uchar pid[16];               /* product ID */
    uchar revision[4];           /* product revision level */
    uchar vendor1[20];           /* vendor specific */
    uchar resvd2[40];            /* reserved */
    uchar vendor2[31];           /* vendor specific (padded to 127) */
};
```

An example of the SIOC_INQUIRY command is:

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

struct inquiry_data inquiry_data;

if (!ioctl (fd, SIOC_INQUIRY, &inquiry_data))
{
    printf ("The SIOC_INQUIRY ioctl succeeded\n");
    printf ("\nThe inquiry data is:\n");
    dump_bytes ((uchar *)&inquiry_data, sizeof (struct inquiry_data));
}
else
{
    perror ("The SIOC_INQUIRY ioctl failed");
    sioc_request_sense();
}
```

SIOC_REQSENSE

This *ioctl* command returns the device sense data. If the last command resulted in an EIO error, then the sense data is returned for the error. Otherwise, a new sense command is issued to the device.

The data structure is:

```
struct request_sense
{
    uint        valid:1,          /* sense data is valid */
                err_code:7;       /* error code */
    uchar       segnum;           /* segment number */
    uint        fm:1,             /* filemark detected */
                eom:1,            /* end of medium */
                ili:1,            /* incorrect length indicator */
                resvd1:1,         /* reserved */
                key:4;            /* sense key */
    signed int  info;             /* information bytes */
    uchar       addlen;           /* additional sense length */
    uint        cmdinfo;          /* command specific information */
    uchar       asc;              /* additional sense code */
    uchar       ascq;             /* additional sense code qualifier */
    uchar       fru;              /* field replaceable unit code */
    uint        sksv:1,           /* sense key specific valid */
                cd:1,             /* control/data */
                resvd2:2,         /* reserved */
                bpv:1,            /* bit pointer valid */
                sim:3;            /* system information message */
    uchar       field[2];         /* field pointer */
    uchar       vendor[109];      /* vendor specific (padded to 127) */
};
```

An example of the SIOC_REQSENSE command is:

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

struct request_sense sense_data;

if (!ioctl (smcfd, SIOC_REQSENSE, &sense_data))
{
    printf ("The SIOC_REQSENSE ioctl succeeded\n");
    printf ("\nThe request sense data is:\n");
    dump_bytes ((uchar *)&sense_data, sizeof (struct request_sense));
}
else
{
    perror ("The SIOC_REQSENSE ioctl failed");
}
```

SIOC_RESERVE

This *ioctl* command explicitly reserves the device and prevents the device from being released after a close operation. The device is not released until an SIOC_RELEASE *ioctl* command is issued. The *ioctl* command can be used for applications that require multiple open and close processing in a host-sharing environment.

There are no arguments for this *ioctl* command.

An example of the SIOC_RESERVE command is:

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

if (!ioctl (fd, SIOC_RESERVE, NULL))
{
    printf ("The SIOC_RESERVE ioctl succeeded\n");
}
else
{
    perror ("The SIOC_RESERVE ioctl failed");
    sioc_request_sense();
}
```

SIOC_RELEASE

This *ioctl* command explicitly releases the device and allows other hosts to access the device. The *ioctl* command can be used with the SIOC_RESERVE *ioctl* command for applications that require multiple open and close processing in a host-sharing environment.

There are no arguments for this *ioctl* command.

An example of the SIOC_RELEASE command is:

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

if (!ioctl (fd, SIOC_RELEASE, NULL))
{
    printf ("The SIOC_RELEASE ioctl succeeded\n");
}
else
{
    perror ("The SIOC_RELEASE ioctl failed");
    sioc_request_sense();
}
```

SIOC_TEST_UNIT_READY

This *ioctl* command issues the SCSI Test Unit Ready command to the device.

There are no arguments for this *ioctl* command.

An example of the SIOC_TEST_UNIT_READY command is:

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

if (!ioctl (fd, SIOC_TEST_UNIT_READY, NULL))
{
    printf ("The SIOC_TEST_UNIT_READY ioctl succeeded\n");
}
else
{
    perror ("The SIOC_TEST_UNIT_READY ioctl failed");
    sioc_request_sense();
}
```

SIOC_LOG_SENSE_PAGE

This *ioctl* command returns a log sense page from the device. The desired page is selected by specifying the page_code in the log_sense_page structure. Optionally, a specific parm pointer, also known as a parm code, and the number of parameter bytes can be specified with the command.

To obtain the entire log page, the len and parm_pointer fields should be set to zero. To obtain the entire log page starting at a specific parameter code, set the parm_pointer field to the desired code and the len field to zero. To obtain a specific number of parameter bytes, set the parm_pointer field to the desired code and set the len field to the number of parameter bytes plus the size of the log page header (four bytes). The first four bytes of returned data will always be the log page header.

See the appropriate device manual to determine the supported log pages and content.

The data structure is:

```
struct log_sense_page
{
    char page_code;
    unsigned short len;
    unsigned short parm_pointer;
    char data[LOGSENSEPAGE];
};
```

An example of the SIOC_LOG_SENSE_PAGE command is:

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

struct log_sense_page log_page;
int temp;

/* get log page 0, list of log pages */
log_page.page_code = 0x00;
log_page.len = 0;
log_page.parm_pointer = 0;

if (!ioctl (fd, SIOC_LOG_SENSE_PAGE, &log_page))
{
    printf ("The SIOC_LOG_SENSE_PAGE ioctl succeeded\n");
    dump_bytes(log_page.data, LOGSENSEPAGE);
}
else
{
    perror ("The SIOC_LOG_SENSE_PAGE ioctl failed");
    sioc_request_sense();
}


/* get fraction of volume traversed */
log_page.page_code = 0x38;
log_page.len = 0;
log_page.parm_pointer = 0x000F;

if (!ioctl (fd, SIOC_LOG_SENSE_PAGE, &log_page))
{
    temp = log_page.data[sizeof(log_page_header) + 4)];
    printf ("The SIOC_LOG_SENSE_PAGE ioctl succeeded\n");
    printf ("Fractional Part of Volume Traversed %x\n",temp);
}
else
{
    perror ("The SIOC_LOG_SENSE_PAGE ioctl failed");
    sioc_request_sense();
}
```

SIOC_MODE_SENSE_PAGE

This *ioctl* command returns a mode sense page from the device. The desired page is selected by specifying the page_code in the mode_sense_page structure.

See the appropriate device manual to determine the supported mode pages and content.

The data structure is:

```
struct mode_sense_page
{
    char page_code;
    char data[MODESENSEPAGE];
};
```

An example of the SIOC_MODE_SENSE_PAGE command is:

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

struct mode_sense_page mode_page;

/* get medium changer mode */
mode_page.page_code = 0x20;
if (!ioctl (fd, SIOC_MODE_SENSE_PAGE, &mode_page))
{
    printf ("The SIOC_MODE_SENSE_PAGE ioctl succeeded\n");
    if (mode_page.data[2] == 0x02)
      printf ("The library is in Random mode.\n");
    else
      if (mode_page.data[2] == 0x05)
        printf ("The library is in Automatic (Sequential) mode.\n");
}
else
{
    perror ("The SIOC_MODE_SENSE_PAGE ioctl failed");
    sioc_request_sense();
}
```

SIOC_QUERY_OPEN

This *ioctl* command returns the ID of the process that currently has a device opened. There is no associated data structure. The arg parameter specifies the address of an int for the return process id.

If the application opened the device using the extended open parameter SC_TMCP, the process ID returned will be for any other process that currently has the device open or 0 is returned if the device is not currently open. If the application opened the device without using the extended open parameter SC_TMCP, the process id of the current application is returned.

An example of the SIOC_QUERY_OPEN command is:

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

int sioc_query_open (void)
{
int pid = 0;

if (ioctl(fd, SIOC_QUERY_OPEN, &pid) == 0)
  {
  if (pid)
    printf("Device is currently open by process id %d\n",pid)
  else
    printf("Device is not open\n");
   }
else
printf("Error querying device open...\n");

return errno
}
```

SIOC_INQUIRY_PAGE

This *ioctl* command returns an inquiry page from the device. The desired page is selected by specifying the page_code in the inquiry_page structure.

See the appropriate device manual to determine the supported inquiry pages and content.

The data structure is:

```
struct inquiry_page
{
 char page_code;
 char data[INQUIRYPAGE];
};
```

An example of the SIOC_INQUIRY_PAGE command is:

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

struct inquiry_page inq_page;

/* get inquiry page x83 */
inq_page.page_code = 0x83;
if (!ioctl (fd, SIOC_INQUIRY_PAGE, &inq_page))
  {
  printf ("The SIOC_INQUIRY_PAGE ioctl succeeded\n");
  }
 else
  {
  perror ("The SIOC_INQUIRY_PAGE ioctl failed");
  sioc_request_sense();
  }
```

SIOC_QUERY_PATH

This *ioctl* command returns information about the device and SCSI paths, such as logical parent, SCSI IDs, and status of the SCSI paths.

The data structure is:

```
struct scsi_path {
  char  primary_name[15];                   /* Primary logical device name */
  char  primary_parent[15];                 /* Primary SCSI parent name */
  uchar primary_id;                         /* Primary target address of device */
  uchar primary_lun;                        /* Primary logical unit of device */
  uchar primary_bus;                        /* Primary SCSI bus for device */
  unsigned long long primary_fcp_scsi_id;   /* Primary FCP SCSI id of device */
  unsigned long long primary_fcp_lun_id;    /* Primary FCP logical unit of device */
  unsigned long long primary_fcp_ww_name;   /* Primary FCP world wide name */
  uchar primary_enabled;                    /* Primary path enabled */
  uchar primary_id_valid;                   /* Primary id/lun/bus fields valid */
  uchar primary_fcp_id_valid;               /* Primary FCP scsi/lun id fields valid */
  uchar alternate_configured;               /* Alternate path configured */
  char  alternate_name[15];                 /* Alternate logical device name */
  char  alternate_parent[15];               /* Alternate SCSI parent name */
  uchar alternate_id;                       /* Alternate target address of device */
  uchar alternate_lun;                      /* Alternate logical unit of device */
  uchar alternate_bus;                      /* Alternate SCSI bus for device */
  unsigned long long alternate_fcp_scsi_id; /* Alternate FCP SCSI id of device */
  unsigned long long alternate_fcp_lun_id;  /* Alternate FCP logical unit of device */
  unsigned long long alternate_fcp_ww_name; /* Alternate FCP world wide name */
  uchar alternate_enabled;                  /* Alternate path enabled */
  uchar alternate_id_valid;                 /* Alternate id/lun/bus fields valid */
  uchar alternate_fcp_id_valid;             /* Alternate FCP scsi/lun id fields valid*/
  uchar primary_drive_port_valid;           /* Primary drive port field valid */
  uchar primary_drive_port;                 /* Primary drive port number */
  uchar alternate_drive_port_valid;         /* Alternate drive port field valid */
  uchar alternate_drive_port;               /* Alternate drive port number */
  char reserved[60];
  };
```

An example of the SIOC_QUERY_PATH command is:

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

int sioc_query_path(void)
 {
 struct scsi_path path;

 printf("Querying SCSI paths...\n");

 if (ioctl(fd, SIOC_QUERY_PATH, &path) == 0)
   show_path(&path);

 return errno;
 }

void show_path(struct scsi_path *path)
 {

 printf("\n");
 if (path->alternate_configured)
 printf("Primary Path Information:\n");
 printf("  Logical Device................. %s\n",path->primary_name);
 printf("  SCSI Parent.................... %s\n",path->primary_parent);
 if (path->primary_fcp_id_valid)
  {
  if (path->primary_id_valid)
  {
  printf("  Target ID...................... %d\n",path->primary_id);
  printf("  Logical Unit................... %d\n",path->primary_lun);
  printf("  SCSI Bus....................... %d\n",path->primary_bus);
  }
  printf("  FCP SCSI ID.................... 0x%llx\n",path->primary_fcp_scsi_id);
  printf("  FCP Logical Unit............... 0x%llx\n",path->primary_fcp_lun_id);
  printf("  FCP World Wide Name............ 0x%llx\n",path->primary_fcp_ww_name);
  }
 else
  {
  printf("  Target ID...................... %d\n",path->primary_id);
  printf("  Logical Unit................... %d\n",path->primary_lun);
  }
 if (path->primary_drive_port_valid)
  printf("  Drive Port Number.............. %d\n",path->primary_drive_port);
 if (path->primary_enabled)
  printf("  Path Enabled................... Yes\n");
 else
  printf("  Path Enabled................... No \n");

 if (!path->alternate_configured)
  printf("  Alternate Path Configured...... No\n");
 else
  {
  printf("  Alternate Path Configured...... Yes\n");
  printf("\nAlternate Path Information:\n");
  printf("  Logical Device................. %s\n",path->alternate_name);
  printf("  SCSI Parent.................... %s\n",path->alternate_parent);
   if (path->alternate_fcp_id_valid)
     {
     if (path->alternate_id_valid)
       {
       printf("  Target ID...................... %d\n",path->alternate_id);
       printf("  Logical Unit................... %d\n",path->alternate_lun);
       printf("  SCSI Bus....................... %d\n",path->alternate_bus);
       }
      printf("  FCP SCSI ID.................... 0x%llx\n",path->alternate_fcp_scsi_id);
      printf("  FCP Logical Unit............... 0x%llx\n",path->alternate_fcp_lun_id);
      printf("  FCP World Wide Name............ 0x%llx\n",path->alternate_fcp_ww_name);
      }
     else
```

```
        {
        printf("  Target ID..................... %d\n",path->alternate_id);
        printf("  Logical Unit.................. %d\n",path->alternate_lun);
        }
      if (path->alternate_drive_port_valid)
        printf("  Drive Port Number............. %d\n",path->alternate_drive_port);
      if (path->alternate_enabled)
        printf("  Path Enabled.................. Yes\n");
    else
     printf("  Path Enabled.................. No \n");
    }
  }
```

Chapter 5. Tape IOCTL Operations

The device driver supports the set of tape *ioctl* commands that is available with the base AIX operating system in addition to a set of expanded tape *ioctl* commands that give applications access to additional features and functions of the tape drives.

Overview

The following *ioctl* commands are supported:

STIOCHGP Set the block size.

STIOCTOP Perform the *ioctl* tape operation.

STIOCQRYP Query the tape device, device driver, and media parameters.

STIOCSETP Change the tape device, device driver, and media parameters.

STIOCSYNC Synchronize the tape buffers with the tape.

STIOCQRYPOS Query the tape position and the buffered data.

STIOCSETPOS Set the tape position.

STIOCQRYSENSE Query the sense data from the tape device.

STIOCQRYINQUIRY Return the inquiry data.

STIOC_LOG_SENSE Return the log sense data.

STIOC_LOCATE Locate to the tape position.

STIOC_READ_POSITION Read the current tape position.

STIOC_SET_VOLID Set the volume name for the current mounted tape. The name is used for tape volume logging only.

STIOC_DUMP Force and read a dump from the device

STIOC_FORCE_DUMP Force a dump on the device.

STIOC_READ_DUMP Read a dump from the device.

STIOC_LOAD_UCODE Download the microcode to the device.

STIOC_RESET_DRIVE Issue a SCSI Send Diagnostic command to reset the tape drive

STIOC_PREVENT_MEDIUM_REMOVAL
Prevent medium removal by an operator.

STIOC_ALLOW_MEDIUM_REMOVAL
Allow medium removal by an operator.

STIOC_REPORT_DENSITY_SUPPORT
Return supported densities from the tape device.

These *ioctl* commands and their associated structures are defined in the */usr/include/sys/Atape.h* header file, which is included in the corresponding C program using the functions.

A sample *tapeutil.c* program is provided with the device driver in the */usr/lpp/Atape/samples* directory that demonstrates the use of these *ioctl* commands.

STIOCHGP

This *ioctl* command sets the current block size. A block size of zero is a variable block. Any other value is a fixed block.

An example of the STIOCHGP command is:

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

struct stchgp stchgp;

stchgp.st_blksize = 512;

if (ioctl(tapefd,STIOCHGP,&stchgp)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
```

STIOCTOP

This *ioctl* command performs the basic tape operations. The *st_count* variable is used for many of its operations. Normal error recovery applies to these operations. The device driver can issue several tries to complete them.

For all space operations, the tape position finishes on the end-of-tape side of the record or filemark for forward movement and on the beginning-of-tape side of the record or filemark for backward movement. The only exception occurs for forward and backward space record operations over a filemark if the device is configured for the AIX record space mode.

The input data structure is:

```
struct stop
 {
 short st_op;           /* operations defined below */
 daddr_t st_count;      /* how many of them to do (if applicable) */
 };
```

The *st_op* variable is set to one of the following operations:

STOFFL Unload the tape. The *st_count* parameter does not apply.

STREW Rewind the tape. The *st_count* parameter does not apply.

STERASE Erase the entire tape. The *st_count* parameter does not apply.

STRETEN Perform the rewind operation. The tape devices perform the retension operation automatically when needed.

STWEOF Write the *st_count* number of filemarks.

STFSF Space forward the *st_count* number of filemarks.

STRSF Space backward the *st_count* number of filemarks.

STFSR Space forward the *st_count* number of records.

STRSR Space backward the *st_count* number of records.

STTUR Issue the Test Unit Ready command. The *st_count* parameter does not apply.

STLOAD Issue the SCSI Load command. The *st_count* parameter does not apply. The operation of the SCSI Load command varies depending on the type of device. See the appropriate hardware reference manual.

STSEOD Space forward to the end of the data. The *st_count* parameter does not apply.

STEJECT Unload the tape. The *st_count* parameter does not apply.

STINSRT Issue the SCSI Load command. The *st_count* parameter does not apply.

Note: If zero is used for operations that require the count parameter, then the command is not issued to the device, and the device driver will return a successful completion.

An example of the STIOCTOP command is:

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

struct stop stop;

stop.st_op=STWEOF;

stop.st_count=3;

if (ioctl(tapefd,STIOCTOP,&stop)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
```

STIOCQRYP or STIOCSETP

The STIOCQRYP *ioctl* command allows the program to query the tape device, device driver, and media parameters. The STIOCSETP *ioctl* command allows the program to change the tape device, device driver, and media parameters. Before issuing the STIOCSETP *ioctl* command, use the STIOCQRYP *ioctl* command to query and fill the fields of the data structure that you do not want to change. Then issue the STIOCSETP command to change the selected fields.

Changing certain fields (such as *buffered_mode*) impacts performance. If the *buffered_mode* field is false, then each record written to the tape is immediately transferred to the tape. This operation guarantees that each record is on the tape, but it impacts performance.

The configuration parameters changed by this *ioctl* command are effective only during the corresponding open or close process. They revert to their current default values after closing.

The following parameters returned by the STIOCQRYP *ioctl* command cannot be changed by the STIOCSETP *ioctl* command:

- trace

  This parameter is the current setting of the AIX system tracing for channel 0. All *Atape* device driver events are traced in channel 0 with other kernel events. If it is set to on, then the device driver tracing is active.

- hkwrd

  This parameter is the trace hookword used for *Atape* device driver events.
- write_protect

  If the current mounted tape is write protected, then this field is set to TRUE. Otherwise, it is set to FALSE.
- min_blksize

  This parameter is the minimum block size for the device. The driver sets this field by issuing the SCSI Read Block Limits command to the device.
- max_blksize

  This parameter is the maximum block size for the device. The driver sets this field by issuing the SCSI Read Block Limits command to the device.
- max_scsi_xfer

  This parameter is the maximum transfer size of the parent SCSI adapter for the device.
- retain_reservation

  This parameter is the current setting of the retain reservation flag that is set or reset by the STIOC_RESERVE or STIOC_RELEASE *ioctl*.
- medium_type

  This parameter is the media type of the current loaded tape. Some tape devices support multiple media types and will report different values in this field. See the documentation for the specific tape device to determine the possible values.
- density_code

  This parameter is the density setting for the current loaded tape. Some tape devices support multiple densities and will report the current setting in this field. See the documentation for the specific tape device to determine the possible values.

The following parameters can be changed by using the STIOCSETP *ioctl* command:

- autoload

  This parameter turns the autoload feature On and Off in the device driver. If set to On, the tapes in the attached library are treated as a large virtual tape.
- blksize

  This parameter specifies the effective block size for the tape device.
- compression

  This parameter turns the hardware compression on and off.
- trailer_labels

  If this parameter is set to on, then writing a record past the early warning mark on the tape is allowed. The first write operation to detect EOM returns the ENOSPC error code. This write operation will not complete successfully. All subsequent write operations are allowed to continue despite the check conditions that result from EOM. When the end of the physical volume is reached, EIO is returned. This parameter can be used before reaching EOM or after EOM is reached.
- rewind_immediate

  This parameter turns the immediate bit on and off in rewind commands. If it is set to on, then the STREW tape operation executes faster, but the next command takes a long time to finish unless the rewind operation is physically complete.
- logging

This parameter turns the volume logging on and off. If it is set to on, then the volume log data is collected and saved in the tape log file when the Rewind and Unload command is issued to the tape drive.

- volid

  This parameter is the volume ID of the current loaded tape. If it is not set, then the device driver initializes the *volid* to UNKNOWN. If logging is active, the parameter identifies the volume in the tape log file entry. It is reset to UNKNOWN when the tape is unloaded.

- record_space_mode

  This parameter specifies how the device driver operates when a forward or backward space record operation encounters a filemark. The two modes of operation are SCSI and AIX.

- read_sili_bit

  This parameter turns the Suppress Incorrect Length Indication (SILI) bit on and off for variable length read commands. The device driver sets this bit when the device is configured if it detects that the adapter can support this setting. When this bit is off, variable length read commands will result in a SCSI check condition if there is less data read than what the read system call requested. This can have a significant impact on read performance.

The input or output data structure is:

```
struct stchgp_s
  {
  int blksize;                /* new block size */
  boolean trace;              /* TRUE=trace on */
  ulong hkwrd;                /* trace hookword */
  int sync_count;             /* obsolete (not used) */
  boolean autoload;           /* on/off autoload feature */
  boolean buffered_mode;      /* on/off buffered mode */
  boolean compression;        /* on/off compression */
  boolean trailer_labels;     /* on/off allow writing after EOM */
  boolean rewind_immediate;   /* on/off immediate rewinds */
  boolean bus_domination;     /* obsolete (not used) */
  boolean logging;            /* volume logging */
  boolean write_protect;      /* write-protected media */
  uint min_blksize;           /* minimum block size */
  uint max_blksize;           /* maximum block size */
  uint max_scsi_xfer;         /* maximum SCSI transfer length */
  char volid[16];             /* volume ID */
  uchar acf_mode;             /* automatic cartridge facility mode */
    #define ACF_NONE            0
    #define ACF_MANUAL          1
    #define ACF_SYSTEM          2
    #define ACF_AUTOMATIC       3
    #define ACF_ACCUMULATE      4
    #define ACF_RANDOM          5
  uchar record_space_mode;    /* fsr/bsr space mode */
    #define SCSI_SPACE_MODE     1
    #define AIX_SPACE_MODE      2
  uchar logical_write_protect; /* logical write protect */
    #define NO_PROTECT          0
    #define ASSOCIATED_PROTECT  1
    #define PERSISTENT_PROTECT  2
    #define WORM_PROTECT        3
  uchar capacity_scaling;     /* capacity scaling */
    #define SCALE_100           0
    #define SCALE_75            1
    #define SCALE_50            2
    #define SCALE_25            3
  uchar retain_reservation;   /* retain reservation */
  uchar alt_pathing;          /* alternate pathing active */
```

```
  uchar emulate_autoloader;    /* emulate autoloader in random mode */
  uchar medium_type;           /* medium type */
  uchar density_code;          /* density code */
  boolean disable_sim_logging; /*disable sim/mim error logging*/
  boolean read_sili_bit;       /*SILI bit setting for read commands*/
  uchar reserved [22];
 };
```

An example of the STIOCQRYP and STIOCSETP commands is:

```
#include <sys/Atape.h>
 struct stchgp_s stchgp;

 /* get current parameters */
 if (ioctl(tapefd,STIOCQRYP,&stchgp)<0)
   {
   printf("IOCTL failure. errno=%d",errno);
   exit(errno);
   }

 /* set new parameters */
 stchgp.rewind_immediate=1;
 stchgp.trailer_labels=1;
 if (ioctl(tapefd,STIOCSETP,&stchgp)<0)
   {
   printf("IOCTL failure. errno=%d",errno);
   exit(errno);
   }
```

STIOCSYNC

This *ioctl* command immediately flushes the tape buffers to the tape.

There are no arguments for this *ioctl* command.

An example of the STIOCSYNC command is:

```
if (ioctl(tapefd,STIOCSYNC,NULL)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
```

STIOCQRYPOS or STIOCSETPOS

The STIOCQRYPOS *ioctl* command queries the position on the tape. The STIOCSETPOS *ioctl* command sets the position on the tape. Only the *curpos* field is used during a set operation. The block_type field is not used and the curpos field is always a logical tape position. The tape position is defined as where the next read or write operation occurs. The query function can be used independently or in conjunction with the set function. Also, the set function can be used independently or in conjunction with the query function.

For a query operation, the *curpos* field is set to a simple *unsigned int*. After a set operation, the position is at the logical block indicated by the *curpos* field.

After a query operation, the *lbot* field indicates the last block of the data that was transferred physically to the tape. If the application writes 12 (0 to 11) blocks and *lbot* equals 8, then three blocks are in the tape buffer. This field is valid only if the last command was a write command. This field does not reflect the number of application write operations. A write operation can translate into multiple blocks. It

reflects tape blocks as indicated by the block size. If an attempt is made to obtain this information and the last command is not a write command, then the value of LBOT_UNKNOWN is returned.

The driver sets the *bot* field to TRUE if the tape position is at the beginning of the tape. Otherwise, it is set to FALSE. The driver sets the *eot* field to TRUE if the tape is positioned between the early warning and the physical end of the tape. Otherwise, it is set to FALSE.

The number of blocks and number of bytes currently in the tape device buffers is returned in the *num_blocks* and *num_bytes* fields, respectively. The block ID of the next block of data that will be transferred to or from the physical tape is returned in the *tapepos* field.

The partition-number field returned is the current partition of the loaded tape.

The input or output data structure is:

```
typedef unsigned int blockid_t;
struct stpos_s
 {
 char block_type;                  /* format of block ID information */
 boolean eot;                      /* position is after early warning,
                                      before physical end of tape */
 blockid_t curpos;                 /* for query, current position,
                                      for set, position to go to */
 blockid_t lbot;                   /* last block written to tape */
  #define LBOT_NONE 0xFFFFFFFF     /* no blocks were written to tape */
  #define LBOT_UNKNOWN 0xFFFFFFFE  /* unable to determine information */
 uint num_blocks;                  /* number of blocks in buffer */
 uint num_bytes;                   /* number of bytes in buffer */
 boolean bot;                      /* position is at beginning of tape */
 uchar partition_number;           /* current partition number on tape */
 uchar reserved1[2];
 blockid_t tapepos;                /* next block transferred */
 uchar reserved2[48];
 };
```

An example of the STIOCQRYPOS and STIOCSETPOS commands is:

```
#include <sys/Atape.h>
struct stpos_s stpos;
if (ioctl(tapefd,STIOCQRYPOS,&stpos)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
   }
 oldposition=stpos.curpos;
    .
    .
    .
 stpos.curpos=oldposition;
  if (ioctl(tapefd,STIOCSETPOS,&stpos)<0)
   {
   printf("IOCTL failure. errno=%d",errno);
   exit(errno);
   }
```

STIOCQRYSENSE

This *ioctl* command returns the last sense data collected from the tape device, or it issues a new Request Sense command and returns the collected data. If LASTERROR is requested, then the sense data is valid only if the last tape operation has an error that issued a sense command to the device. If the sense data is valid, then the *ioctl* command completes successfully and the *len* field is set to a value greater than zero.

The *residual_count* field contains the residual count from the last operation.

The input or output data structure is:

```
#define MAXSENSE        255
struct stsense_s
 {
 /* input */
 char sense_type;          /* fresh (new sense) or sense from last error */
  #define FRESH     1      /* initiate a new sense command */
  #define LASTERROR 2      /* return sense gathered from
                              the last SCSI sense command */
 /* output */
 uchar sense[MAXSENSE];    /* actual sense data */
 int len;                  /* length of valid sense data returned */
 int residual_count;       /* residual count from last operation  */
 uchar reserved[60];
 };
```

An example of the STIOCQRYSENSE command is:

```
#include <sys/Atape.h>
struct stsense_s stsense;
stsense.sense_type=LASTERROR;
#define MEDIUM_ERROR 0x03
if (ioctl(tapefd,STIOCQRYSENSE,&stsense)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
if ((stsense.sense[2]&0x0F)==MEDIUM_ERROR)
  {
  printf("We're in trouble now!");
  exit(SENSE_KEY(&stsense.sense));
  }
```

STIOCQRYINQUIRY

This *ioctl* command returns the inquiry data from the device. The data is divided into standard and vendor-specific portions.

The output data structure is:

```
/* inquiry data info */
struct inq_data_s
 {
 BYTE b0;
 /* macros for accessing fields of byte 1 */
  #define PERIPHERAL_QUALIFIER(x)   ((x->b0 & 0xE0)>>5)
   #define PERIPHERAL_CONNECTED          0x00
   #define PERIPHERAL_NOT_CONNECTED      0x01
   #define LUN_NOT_SUPPORTED             0x03

  #define PERIPHERAL_DEVICE__TYPE(x) (x->b0 & 0x1F)
   #define DIRECT_ACCESS                 0x00
   #define SEQUENTIAL_DEVICE             0x01
```

```
 #define PRINTER_DEVICE               0x02
 #define PROCESSOR_DEVICE             0x03
 #define CD_ROM_DEVICE                0x05
 #define OPTICAL_MEMORY_DEVICE        0x07
 #define MEDIUM_CHANGER_DEVICE        0x08
 #define UNKNOWN                      0x1F

BYTE b1;
/* macros for accessing fields of byte 2 */
#define RMB(x) ((x->b1 & 0x80)>>7)              /* removable media bit */
#define FIXED     0
#define REMOVABLE 1
#define device_type_qualifier(x) (x->b1 & 0x7F)  /* vendor specific */

BYTE b2;
/* macros for accessing fields of byte 3 */
#define ISO_Version(x)  ((x->b2 & 0xC0)>>6)
#define ECMA_Version(x) ((x->b2 & 0x38)>>3)
#define ANSI_Version(x) (x->b2 & 0x07)
 #define NONSTANDARD     0
 #define SCSI1           1
 #define SCSI2           2
BYTE b3;
/* macros for accessing fields of byte 4 */
#define AENC(x)    ((x->b3 & 0x80)>>7)   /* asynchronous event notification */
#ifndef TRUE
 #define TRUE 1
#endif
#ifndef FALSE
 #define FALSE 0
#endif
#define TrmIOP(x)  ((x->b3 & 0x40)>>6)   /* support terminate I/O
                                            process message? */
#define Response_Data_Format(x)  (x->b3 & 0x0F)
 #define SCSI1INQ      0                 /* SCSI-1 standard inquiry
                                            data format */
 #define CCSINQ        1                 /* CCS standard inquiry
                                            data format */
 #define SCSI2INQ      2                 /* SCSI-2 standard inquiry
                                            data format */

BYTE additional_length;                  /* number of bytes following
                                            this field minus 4 */
BYTE res56[2];

BYTE b7;
/* macros for accessing fields of byte 7 */
#define RelAdr(x)  ((x->b7 & 0x80)>>7)   /* the following fields are
                                            true or false */
#define WBus32(x)  ((x->b7 & 0x40)>>6)
#define WBus16(x)  ((x->b7 & 0x20)>>5)
#define Sync(x)    ((x->b7 & 0x10)>>4)
#define Linked(x)  ((x->b7 & 0x08)>>3)
#define CmdQue(x)  ((x->b7 & 0x02)>>1)
#define SftRe(x)   ((x->b7 & 0x01)

char vendor_identification[8];
char product_identification[16];
char product_revision_level[4];
};
struct st_inquiry
 {
 struct inq_data_s standard;
 BYTE vendor_specific[255-sizeof(struct inq_data_s)];
 };
```

An example of the STIOCQRYINQUIRY command is:

```
struct st_inquiry inqd;
if (ioctl(tapefd,STIOCQRYINQUIRY,&inqd)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
 if (ANSI_Version(((struct inq_data_s *)&(inqd;standard)))==SCSI2)
   printf("Hey! We have a SCSI-2 device\n");
```

STIOC_LOG_SENSE

This *ioctl* command returns the log sense data from the device. If the volume logging is set to on, then the log sense data is also saved in the tape log file.

The output data structure is:

```
struct log_sense
  {
  struct log_record_header header;
  char data[MAXLOGSENSE];
  }
```

An example of the STIOC_LOG_SENSE command is:

```
struct log_sense logdata;

if (ioctl(tapefd,STIOC_LOG_SENSE,&logdata)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
```

STIOC_LOCATE

This *ioctl* command causes the tape to be positioned at the specified block ID. The block ID used for the command must be obtained by using the STIOC_READ_POSITION command.

An example of the STIOC_LOCATE command is:

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

unsigned int current_blockid;

/* read current tape position */
if (ioctl(tapefd,STIOC_READ_POSITION,&current_blockid)<0)
  {
  printf("IOCTL failure. errno=%d"n,errno);
  exit(1);
   {

/* restore current tape position */
if (ioctl(tapefd,STIOC_LOCATE,current_blockid)<0)
  {
  printf("IOCTL failure. errno=%d"n,errno);
  exit(1);
  }
```

STIOC_READ_POSITION

This *ioctl* command returns the block ID of the current position of the tape. The block ID returned from this command can be used with the STIOC_LOCATE command to set the position of the tape.

An example of the STIOC_READ_POSITION command is:

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

unsigned int current_blockid;

/* read current tape position */
if (ioctl(tapefd,STIOC_READ_POSITION,&current_blockid)<0)
  {
  printf("IOCTL failure. errno=%d"n,errno);
  exit(1);
  }

/* restore current tape position */
if (ioctl(tapefd,STIOC_LOCATE,current_blockid)<0)
  {
  printf("IOCTL failure. errno=%d"n,errno);
  exit(1);
  }
```

STIOC_SET_VOLID

This *ioctl* command sets the volume name for the current mounted tape. The volume name is used by the device driver for tape volume logging only and is not written or stored on the tape. The volume name is reset to unknown whenever an unload command is issued to the device driver to unload the current tape. The volume name can be queried and also set using the STIOCQRYP and STIOCSETP *ioctls*, respectively.

The argument used for this command is a character pointer to a buffer that contains the name of the volume to be set.

An example of the STIOC_SET_VOLID command is:

```
/* set the volume id for the current tape to VOL001 */
  char *volid = "VOL001";
  if (ioctl(tapefd,STIOC_SET_VOLID,volid)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }
```

STIOC_DUMP

This *ioctl* command forces a dump on the tape device, then stores the dump to either a host-specified file or, if not specified, in the */var/adm/ras* system directory. The device driver stores up to three dumps in this directory. The first dump created is named *Atape.rmtx.dump1*, where *x* is the device number, for example, rmt0. The second and third dumps are named “dump2” and “dump3”, respectively. After the third dump file is created, the next dump starts at “dump1” again and overlays the previous “dump1” file.

The argument used for this command is either NULL to dump to the system directory or a character pointer to a buffer that contains the path and filename for the dump file. The dump can also be stored on a diskette by specifying /dev/rfd0 for the name.

An example of the STIOC_DUMP command is:

```
/* generate drive dump and store in the system directory */
  if (ioctl(tapefd,STIOC_DUMP,NULL)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }

  /* generate drive dump and store in file MY.dump */
  char *dump_name = "MY.dump";
  if (ioctl(tapefd,STIOC_DUMP,dump_name)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }
```

STIOC_FORCE_DUMP

This *ioctl* command forces a dump on the tape device. The dump can then be retrieved from the device using the STIOC_READ_DUMP *ioctl*.

There are no arguments for this *ioctl* command.

An example of the STIOC_FORCE_DUMP command is:

```
/* generate a drive dump */
  if (ioctl(tapefd,STIOC_FORCE_DUMP,NULL)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
   }
```

STIOC_READ_DUMP

This *ioctl* command reads a dump from the tape device, then stores the dump to either a host-specified file or, if not specified, in the */var/adm/ras* system directory. The device driver stores up to three dumps in this directory. The first dump created is named *Atape.rmtx.dump1*, where *x* is the device number, for example, rmt0. The second and third dumps are named “dump2” and “dump3”, respectively. After the third dump file is created, the next dump starts at “dump1” again and overlays the previous “dump1” file.

Dumps are either generated internally by the tape drive or can be forced using the STIOC_FORCE_DUMP *ioctl*.

The argument used for this command is either NULL to dump to the system directory or a character pointer to a buffer that contains the path and filename for the dump file. The dump can also be stored on a diskette by specifying /dev/rfd0 for the name.

An example of the STIOC_READ_DUMP command is:

```
/* read drive dump and store in the system directory */
 if (ioctl(tapefd,STIOC_READ_DUMP,NULL)<0)
   {
   printf("IOCTL failure. errno=%d",errno);
```

```
  exit(errno);
  }

/* read drive dump and store in file tape.dump */
char *dump_name = "tape.dump";
if (ioctl(tapefd,STIOC_READ_DUMP,dump_name)<0)
  {
  printf("IOCTL failure. errno=%d",errno);
  exit(errno);
  }
```

STIOC_LOAD_UCODE

This *ioctl* command will download microcode to the device. The argument used for this command is a character pointer to a buffer that contains the path and filename of the microcode. Microcode can also be loaded from a diskette by specifying /dev/rfd0 for the name.

An example of the STIOC_LOAD_UCODE command is:

```
/* download microcode from file */
  char *name = "/etc/microcode/D0I4_BB5.fmrz";
  if (ioctl(tapefd,STIOC_LOAD_UCODE,name)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }

  /* download microcode from diskette */
  if (ioctl(tapefd,STIOC_LOAD_UCODE,"/dev/rfd0")<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }
```

STIOC_RESET_DRIVE

This *ioctl* command issues a SCSI Send Diagnostic command to reset the tape drive. There are no arguments for this *ioctl* command.

An example of the STIOC_RESET_DRIVE command is:

```
/* reset the tape drive */
  if (ioctl(tapefd,STIOC_RESET_DRIVE,NULL)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }
```

STIOC_PREVENT_MEDIUM_REMOVAL

This *ioctl* command prevents an operator from removing medium from the device until the STIOC_ALLOW_MEDIUM_REMOVAL command is issued or the device is reset.

There is no associated data structure.

An example of the STIOC_PREVENT_MEDIUM_REMOVAL command is:

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

if (!ioctl (tapefd, STIOC_PREVENT_MEDIUM_REMOVAL, NULL))
 printf ("The STIOC_PREVENT_MEDIUM_REMOVAL ioctl succeeded\n");
```

```
else
 {
 perror ("The STIOC_PREVENT_MEDIUM_REMOVAL ioctl failed");
 smcioc_request_sense();
 }
```

STIOC_ALLOW_MEDIUM_REMOVAL

This *ioctl* command allows an operator to remove medium from the device. This command is normally used after an STIOC_PREVENT_MEDIUM_REMOVAL command to restore the device to the default state.

There is no associated data structure.

An example of the STIOC_ALLOW_MEDIUM_REMOVAL command is:

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

if (!ioctl (tapefd, STIOC_ALLOW_MEDIUM_REMOVAL, NULL))
 printf ("The STIOC_ALLOW_MEDIUM_REMOVAL ioctl succeeded\n");
else
 {
 perror ("The STIOC_ALLOW_MEDIUM_REMOVAL ioctl failed");
 smcioc_request_sense();
 }
```

STIOC_REPORT_DENSITY_SUPPORT

This *ioctl* command issues the SCSI Report Density Support command to the tape device and returns either all supported densities or supported densities for the currently mounted media. The media field specifies which type of report is requested. The number_reports field is returned by the device driver and indicates how many density reports in the reports array field were returned.

The data structures used with this *ioctl()* are:

```
struct density_report
{
   uchar  primary_density_code;   /* primary density code */
   uchar  secondary_density_code; /* secondary density code */
   uint   wrtok:1,                /* write ok, device can write this format */
          dup:1,                  /* zero if density only reported once */
          defLt:1,                /* current density is default format */
          :5;                     /* reserved */
   char   reserved[2];            /* reserved */
   uint   bits_per_mm:24;         /* bits per mm */
   ushort media_width;            /* media width in millimeters */
   ushort tracks;                 /* tracks */
   uint   capacity;               /* capacity in megabytes */
   char   assigning_org[8];       /* assigning organization in ASCII */
   char   density_name[8];        /* density name in ASCII */
   char   description[20];        /* description in ASCII */
};

struct report_density_support
{
   uchar  media;                  /* report all or current media as defined above */
   ushort number_reports;         /* number of density reports returned in array */
   struct density_report reports[MAX_DENSITY_REPORTS];
};
```

Examples of the STIOC_REPORT_DENSITY_SUPPORT command are:

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

int stioc_report_density_support(void)
{
int i;
struct report_density_support density;

   printf("Issuing Report Density Support for ALL supported media...\n");

   density.media = ALL_MEDIA_DENSITY;

   if (ioctl(fd, STIOC_REPORT_DENSITY_SUPPORT, &density) != 0)
     return errno;
   printf("Total number of densities reported: %d\n",density.number_reports);
   for (i = 0; i  density.number_reports; i++)
    {
    printf("\n");
    printf("  Density Name.............. %0.8s\n",density.reports[i].density_name);
    printf("  Assigning Organization..... %0.8s\n",density.reports[i].assigning_org);
    printf("  Description............... %0.20s\n",density.reports[i].description);
    printf("  Primary Density Code...... %02X\n",density.reports[i].primary_density_code);
    printf("  Secondary Density Code..... %02X\n",density.reports[i].secondary_density_code);

    if (density.reports[i].wrtok)
      printf("  Write OK.................. Yes\n");
    else
      printf("  Write OK.................. No\n");

    if (density.reports[i].dup)
    printf("  Duplicate................. Yes\n");
    else
      printf("  Duplicate................. No\n");

    if (density.reports[i].deflt)
      printf("  Default................... Yes\n");
    else
      printf("  Default................... No\n");

    printf("  Bits per MM............... %d\n",density.reports[i].bits_per_mm);
    printf("  Media Width (millimeters).... %d\n",density.reports[i].media_width);
    printf("  Tracks.................... %d\n",density.reports[i].tracks);
    printf("  Capacity (megabytes)...... %d\n",density.reports[i].capacity);
    if (opcode)
    {
    printf ("\nHit enter> to continue...");
    getchar();
    }
   }

  printf("\nIssuing Report Density Support for CURRENT media...\n");

  density.media = CURRENT_MEDIA_DENSITY;

  if (ioctl(fd, STIOC_REPORT_DENSITY_SUPPORT, &density) != 0)
    return errno;

  for (i = 0; i  density.number_reports; i++)
   {
   printf("\n");
   printf("  Density Name.............. %0.8s\n",density.reports[i].density_name);
   printf("  Assigning Organization..... %0.8s\n",density.reports[i].assigning_org);
   printf("  Description............... %0.20s\n",density.reports[i].description);
   printf("  Primary Density Code...... %02X\n",density.reports[i].primary_density_code);
   printf("  Secondary Density Code..... %02X\n",density.reports[i].secondary_density_code);

   if (density.reports[i].wrtok)
     printf("  Write OK.................. Yes\n");
```

```
 else
   printf("  Write OK..................... No\n");

 if (density.reports[i].dup)
   printf("  Duplicate.................... Yes\n");
 else
   printf("  Duplicate.................... No\n");

 if (density.reports[i].deflt)
   printf("  Default...................... Yes\n");
 else
   printf("  Default...................... No\n");

 printf("  Bits per MM.................. %d\n",density.reports[i].bits_per_mm);
 printf("  Media Width (millimeters).... %d\n",density.reports[i].media_width);
 printf("  Tracks....................... %d\n",density.reports[i].tracks);
 printf("  Capacity (megabytes)......... %d\n",density.reports[i].capacity);
 }

return errno;
}
```

Chapter 6. Medium Changer IOCTL Operations

This chapter describes the set of *ioctl* commands that provides control and access to the SCSI medium changer functions. These *ioctl* operations can be issued to the tape special file (such as *rmt0*), through a separate special file (such as *rmt0.smc*) that was created during the configuration process, or a separate special file (such as *smc0*), to access the medium changer.

When an application opens a */dev/rmt* special file that is assigned to a drive that has access to a medium changer, the *ioctl* operations described in this chapter are also available. The interface to the */dev/rmt*.smc* special file provides the application access to a separate medium changer device. When this special file is open, the medium changer is treated as a separate device. While */dev/rmt*.smc* is open, access to the *ioctl* operations described in this chapter is restricted to */dev/rmt*.smc* and any attempt to access them through */dev/rmt** fails.

The following *ioctl* commands are supported:

SMCIOC_ELEMENT_INFO Obtain the device element information.

SMCIOC_MOVE_MEDIUM Move a cartridge from one element to another element.

SMCIOC_EXCHANGE_MEDIUM
Exchange a cartridge in an element with another cartridge.

SMCIOC_POS_TO_ELEM Move the robot to an element.

SMCIOC_INIT_ELEM_STAT Issue the SCSI Initialize Element Status command.

SMCIOC_INIT_ELEM_STAT_RANGE
Issue the SCSI Initialize Element Status with Range command.

SMCIOC_INVENTORY Return the information about the four element types.

SMCIOC_LOAD_MEDIUM Load a cartridge from a slot into the drive.

SMCIOC_UNLOAD_MEDIUM Unload a cartridge from the drive and return it to a slot.

SMCIOC_PREVENT_MEDIUM_REMOVAL
Prevent medium removal by the operator.

SMCIOC_ALLOW_MEDIUM_REMOVAL
Allow medium removal by the operator.

SMCIOC_READ_ELEMENT_DEVIDS
Return the device id element descriptors for drive elements.

These *ioctl* commands and their associated structures are defined in the */usr/include/sys/Atape.h* header file, which is included in the corresponding C program using the functions.

A sample *tapeutil.c* program is provided with the device driver in the */usr/lpp/Atape/samples* directory that demonstrates the use of these *ioctl* commands.

SMCIOC_ELEMENT_INFO

This *ioctl* command obtains the device element information.

The data structure is:

```
struct element_info
{
    ushort robot_addr;          /* first robot address */
    ushort robots;              /* number of medium transport elements */
    ushort slot_addr;           /* first medium storage element address */
    ushort slots;               /* number of medium storage elements */
    ushort ie_addr;             /* first import/export element address */
    ushort ie_stations;         /* number of import/export elements */
    ushort drive_addr;          /* first data-transfer element address */
    ushort drives;              /* number of data-transfer elements */
};
```

An example of the SMCIOC_ELEMENT_INFO command is:

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

struct element_info element_info;

if (!ioctl (smcfd, SMCIOC_ELEMENT_INFO, &element_info))
{
    printf ("The SMCIOC_ELEMENT_INFO ioctl succeeded\n");
    printf ("\nThe element information data is:\n");
    dump_bytes ((uchar *)&element_info, sizeof (struct element_info));
}
else
{
    perror ("The SMCIOC_ELEMENT_INFO ioctl failed");
    smcioc_request_sense();
}
```

SMCIOC_MOVE_MEDIUM

This *ioctl* command moves a cartridge from one element to another element.

The data structure is:

```
struct move_medium
{
    ushort robot;               /* robot address */
    ushort source;              /* move from location */
    ushort destination;         /* move to location */
    char invert;                /* invert before placement bit */
};
```

An example of the SMCIOC_MOVE_MEDIUM command is:

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

struct move_medium move_medium;

move_medium.robot = 0;
move_medium.invert = 0;
move_medium.source = source;
move_medium.destination = dest;

if (!ioctl (smcfd, SMCIOC_MOVE_MEDIUM, &move_medium))
    printf ("The SMCIOC_MOVE_MEDIUM ioctl succeeded\n");
else
```

```
{
  perror ("The SMCIOC_MOVE_MEDIUM ioctl failed");
    smcioc_request_sense();
  }
```

SMCIOC_EXCHANGE_MEDIUM

This *ioctl* command exchanges a cartridge in an element with another cartridge. This command is equivalent to two SCSI Move Medium commands. The first moves the cartridge from the source element to the destination1 element, and the second moves the cartridge that was previously in the destination1 element to the destination2 element. The destination2 element can be the same as the source element.

The input data structure is:

```
struct exchange_medium
{
    ushort robot;              /* robot address */
    ushort source;             /* source address for exchange */
    ushort destination1;       /* first destination address for exchange */
    ushort destination2;       /* second destination address for exchange */
    char invert1;              /* invert before placement into destination1 */
    char invert2;              /* invert before placement into destination2 */  };
```

An example of the SMCIOC_EXCHANGE_MEDIUM command is:

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

struct exchange_medium exchange_medium;

exchange_medium.robot = 0;
exchange_medium.invert1 = 0;
exchange_medium.invert2 = 0;
exchange_medium.source = 32;           /* slot 32            */
exchange_medium.destination1 = 16;     /* drive address 16   */
exchange_medium.destination2 = 35;     /* slot 35            */

/* exchange cartridge in drive address 16 with cartridge from slot 32 and */
/* return the cartridge currently in the drive to slot 35                 */
if (!ioctl (smcfd, SMCIOC_EXCHANGE_MEDIUM, &exchange_medium))
    printf ("The SMCIOC_EXCHANGE_MEDIUM ioctl succeeded\n");
else
{
  perror ("The SMCIOC_EXCHANGE_MEDIUM ioctl failed");
  smcioc_request_sense();
}
```

SMCIOC_POS_TO_ELEM

This *ioctl* command moves the robot to an element.

The input data structure is:

```
struct pos_to_elem
{
    ushort robot;              /* robot address */
    ushort destination;        /* move to location */
    char invert;               /* invert before placement bit */
};
```

An example of the SMCIOC_POS_TO_ELEM command is:

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

char buf[10];
struct pos_to_elem pos_to_elem;

pos_to_elem.robot = 0;
pos_to_elem.invert = 0;
pos_to_elem.destination = dest;

if (!ioctl (smcfd, SMCIOC_POS_TO_ELEM, &pos_to_elem))
    printf ("The SMCIOC_POS_TO_ELEM ioctl succeeded\n");
else
{
    perror ("The SMCIOC_POS_TO_ELEM ioctl failed");
    smcioc_request_sense();
}
```

SMCIOC_INIT_ELEM_STAT

This *ioctl* command instructs the medium changer robotic device to issue the SCSI Initialize Element Status command.

There is no associated data structure.

An example of the SMCIOC_INIT_ELEM_STAT command is:

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

if (!ioctl (smcfd, SMCIOC_INIT_ELEM_STAT, NULL))
    printf ("The SMCIOC_INIT_ELEM_STAT ioctl succeeded\n");
else
{
    perror ("The SMCIOC_INIT_ELEM_STAT ioctl failed");
    smcioc_request_sense();
}
```

SMCIOC_INIT_ELEM_STAT_RANGE

This *ioctl* command issues the SCSI Initialize Element Status with Ranged command and audits specific elements in a library by specifying the starting element address and number of elements. Use the SMCIOC_INIT_ELEM_STAT *ioctl* to audit all elements.

The data structure is:

```
struct element_range
  {
    ushort element_address;     /* starting element address */
    ushort number_elements;     /* number of elements       */
  }
```

An example of the SMCIOC_INIT_ELEM_STAT_RANGE command is:

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

struct element_range elements;

/* audit slots 32 to 36  */
elements.element_address = 32;
elements.number_elements = 5;

if (!ioctl (smcfd, SMCIOC_INIT_ELEM_STAT_RANGE, &elements))
    printf ("The SMCIOC_INIT_ELEM_STAT_RANGE ioctl succeeded\n");
else
```

```
{
  perror ("The SMCIOC_INIT_ELEM_STAT_RANGE ioctl failed");
  smcioc_request_sense();
 }
```

SMCIOC_INVENTORY

This *ioctl* command returns the information about the four element types. The software application processes the input data (the number of elements about which it requires information) and allocates a buffer large enough to hold the output for each element type.

The input data structure is:

```
struct element_status
{
    ushort address;            /* element address */
    uint   :2,                 /* reserved */
           inenab:1,           /* media into changer's scope */
           exenab:1,           /* media out of changer's scope */
           access:1,           /* robot access allowed */
           impexp:1,           /* import/export placed by operator or robot */
           full  :1;           /* element contains medium */
           except:1,           /* abnormal element state */
    uchar  resvd1;             /* reserved */
    uchar  asc;                /* additional sense code */
    uchar  ascq;               /* additional sense code qualifier */
    uint   notbus:1,           /* element not on same bus as robot */
           :1,                 /* reserved */
           idvalid:1,          /* element address valid */
           luvalid:1,          /* logical unit valid */
           :1,                 /* reserved */
           lun:3;              /* logical unit number */
    uchar  scsi;               /* SCSI bus address */
    uchar  resvd2;             /* reserved */
    uint   svalid:1,           /* element address valid */
           invert:1,           /* medium inverted */
           :6;                 /* reserved */
    ushort source;             /* source storage element address */
    uchar  volume[36];         /* primary volume tag */
    uchar  resvd3[4];          /* reserved */
};

struct inventory
{
    struct element_status *robot_status; /* medium transport element pages */
    struct element_status *slot_status;  /* medium storage element pages */
    struct element_status *ie_status;    /* import/export element pages */
    struct element_status *drive_status; /* data-transfer element pages */
};
```

An example of the SMCIOC_INVENTORY command is:

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

ushort i;
struct element_status robot_status[1];
struct element_status slot_status[20];
struct element_status ie_status[1];
struct element_status drive_status[1];
struct inventory      inventory;

bzero((caddr_t)robot_status,sizeof(struct element_status));

for (i=0;i20;i++)
    bzero((caddr_t)(&slot_status[i]),sizeof(struct element_status));
```

```
bzero((caddr_t)ie_status,sizeof(struct element_status));
bzero((caddr_t)drive_status,sizeof(struct element_status));

smcioc_element_info();

inventory.robot_status = robot_status;
inventory.slot_status = slot_status;
inventory.ie_status = ie_status;
inventory.drive_status = drive_status;

if (!ioctl (smcfd, SMCIOC_INVENTORY, &inventory))
{
    printf ("\nThe SMCIOC_INVENTORY ioctl succeeded\n");
    printf ("\nThe robot status pages are:\n");

    for (i = 0; i  element_info.robots; i++)
    {
        dump_bytes ((uchar *)(inventory.robot_status+i),
                    sizeof (struct element_status));
        printf ("\n--- more ---");
        getchar();
    }

    printf ("\nThe slot status pages are:\n");

    for (i = 0; i  element_info.slots; i++)
    {
        dump_bytes ((uchar *)(inventory.slot_status+i),
                    sizeof (struct element_status));
        printf ("\n--- more ---");
        getchar();
    }
    printf ("\nThe ie status pages are:\n");

    for (i = 0; i  element_info.ie_stations; i++)
    {
        dump_bytes ((uchar *)(inventory.ie_status+i),
                    sizeof (struct element_status));
        printf ("\n--- more ---");
        getchar();
    }
    printf ("\nThe drive status pages are:\n");

    for (i = 0; i  element_info.drives; i++)
    {
        dump_bytes ((uchar *)(inventory.drive_status+i),
                    sizeof (struct element_status));
        printf ("\n--- more ---");
        getchar();
    }
}
else
{
    perror ("The SMCIOC_INVENTORY ioctl failed");
    smcioc_request_sense();
}
```

SMCIOC_LOAD_MEDIUM

This *ioctl* command loads a tape from a specific slot into the drive or from the first full slot into the drive if the slot address is specified as zero.

An example of the SMCIOC_LOAD_MEDIUM command is:

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

/* load cartridge from slot 3 */
if (ioctl (tapefd, SMCIOC_LOAD_MEDIUM,3)<0)
  {
      printf ("IOCTL failure. errno=%d\n",errno)
      exit(1):
  }

/* load first cartridge from magazine */
if (ioctl (tapefd, SMCIOC_LOAD_MEDIUM,0)<0)
  {
      printf ("IOCTL failure. errno=%d\n",errno)
      exit(1):
  }
```

SMCIOC_UNLOAD_MEDIUM

This *ioctl* command moves a tape from the drive and returns it to a specific slot or to the first empty slot in the magazine if the slot address is specified as zero. If this command is issued to a tape special file, such as */dev/rmt0*, the tape is rewound and unloaded automatically from the drive first.

An example of the SMCIOC_UNLOAD_MEDIUM command is:

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

/* unload cartridge to slot 3 */
if (ioctl (tapefd, SMCIOC_UNLOAD_MEDIUM,3)<0)
  {
      printf ("IOCTL failure. errno=%d\n",errno)
      exit(1):
  }

/* unload cartridge to first empty slot in magazine */
if (ioctl (tapefd, SMCIOC_UNLOAD_MEDIUM,0)<0)
  {
      printf ("IOCTL failure. errno=%d\n",errno)
      exit(1):
  }
```

SMCIOC_PREVENT_MEDIUM_REMOVAL

This *ioctl* command prevents an operator from removing medium from the device until the SMCIOC_ALLOW_MEDIUM_REMOVAL command is issued or the device is reset.

There is no associated data structure.

An example of the SMCIOC_PREVENT_MEDIUM_REMOVAL command is:

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

if (!ioctl (smcfd, SMCIOC_PREVENT_MEDIUM_REMOVAL, NULL))
    printf ("The SMCIOC_PREVENT_MEDIUM_REMOVAL ioctl succeeded\n");
else
{
    perror ("The SMCIOC_PREVENT_MEDIUM_REMOVAL ioctl failed");
    smcioc_request_sense();
}
```

SMCIOC_ALLOW_MEDIUM_REMOVAL

This *ioctl* command allows an operator to remove medium from the device. This command is normally used after an SMCIOC_PREVENT_MEDIUM_REMOVAL command to restore the device to the default state.

There is no associated data structure.

An example of the SMCIOC_ALLOW_MEDIUM_REMOVAL command is:

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

if (!ioctl (smcfd, SMCIOC_ALLOW_MEDIUM_REMOVAL, NULL))
    printf ("The SMCIOC_ALLOW_MEDIUM_REMOVAL ioctl succeeded\n");
else
{
    perror ("The SMCIOC_ALLOW_MEDIUM_REMOVAL ioctl failed");
    smcioc_request_sense();
}
```

SMCIOC_READ_ELEMENT_DEVIDS

This *ioctl* command issues the SCSI Read Element Status command with the DVCID (device id) bit set and returns the element descriptors for the data transfer elements. The element_address field specifies the starting address of the first data transfer element. The number_elements field specifies the number of elements to return. The application must allocate a return buffer large enough for the number_elements specified in the input structure.

The input data structure is:

```
struct read_element_devids
{
   ushort element_address;            /* starting element address */
   ushort number_elements;            /* number of elements */
   struct element_devid *drive_devid; /* data transfer element pages */
};
```

The output data structure is:

```
struct element_devid
{
   ushort address;        /* element address */
   uint   :4,             /* reserved */
          access:1,       /* robot access allowed */
          except:1,       /* abnormal element state */
          :1,             /* reserved */
          full:1;         /* element contains medium */
   uchar  resvd1;         /* reserved */
   uchar  asc;            /* additional sense code */
   uchar  ascq;           /* additional sense code qualifier */
   uint   notbus:1,       /* element not on same bus as robot */
          :1,             /* reserved */
          idvalid:1,      /* element address valid */
          luvalid:1,      /* logical unit valid */
          :1,             /* reserved */
          lun:3;          /* logical unit number */
   uchar  scsi;           /* scsi bus address */
   uchar  resvd2;         /* reserved */
   uint   svalid:1,       /* element address valid */
          invert:1,       /* medium inverted */
          :6;             /* reserved */
   ushort source;         /* source storage element address */
   uint   :4,             /* reserved */
```

```
          code_set:4;     /* code set X'2' is all ASCII identifier */
   uint   :4,             /* reserved */
          ident_type:4;   /* identifier type */
   uchar  resvd3;         /* reserved */
   uchar  ident_len;      /* identifier length */
   uchar  identifier[36]; /* device identification */
};
```

An example of the SMCIOC_READ_ELEMENT_DEVIDS command is:

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

int smcioc_read_element_devids()
 {
 int i;
 struct element_devid *elem_devid, *elemp;
 struct read_element_devids devids;
 struct element_info element_info;

 if (ioctl(fd, SMCIOC_ELEMENT_INFO, &element_info))
   return errno;

 if (element_info.drives)
   {
   elem_devid = malloc(element_info.drives * sizeof(struct element_devid));
   if (elem_devid == NULL)
     {
     errno = ENOMEM;
     return errno;
     }
    bzero((caddr_t)elem_devid,element_info.drives * sizeof(struct element_devid));
    devids.drive_devid = elem_devid;
    devids.element_address = element_info.drive_addr;
    devids.number_elements = element_info.drives;

    printf("Reading element device ids...\n");

    if (ioctl (fd, SMCIOC_READ_ELEMENT_DEVIDS, &devids))
      {
      free(elem_devid);
      return errno;
      }

    elemp = elem_devid;
    for (i = 0; i  element_info.drives; i++, elemp++)
     {
     printf("\nDrive Address %d\n",elemp->address);
     if (elemp->except)
       printf("  Drive State ..................... Abnormal\n");
     else
       printf("  Drive State ..................... Normal\n");
     if (elemp->asc == 0x81 && elemp->ascq ==0x00)
       printf("  ASC/ASCQ ........................ %02X%02X (Drive Present)\n",
          elemp->asc,elemp->ascq);
     else if (elemp->asc == 0x82 && elemp->ascq ==0x00)
       printf("  ASC/ASCQ ........................ %02X%02X (Drive Not Present)\n",
          elemp->asc,elemp->ascq);
     else
       printf("  ASC/ASCQ ........................ %02X%02X\n",
          elemp->asc,elemp->ascq);
     if (elemp->full)
       printf("  Media Present ................... Yes\n");
     else
       printf("  Media Present ................... No\n");
     if (elemp->access)
       printf("  Robot Access Allowed ............ Yes\n");
     else
```

```
            printf("  Robot Access Allowed ........... No\n");
          if (elemp->svalid)
          printf("  Source Element Address ......... %d\n",elemp->source);
          else
           printf("  Source Element Address Valid ... No\n");
          if (elemp->invert)
            printf("  Media Inverted ................. Yes\n");
          else
            printf("  Media Inverted ................. No\n");
          if (elemp->notbus)
            printf("  Same Bus as Medium Changer ..... No\n");
          else
            printf("  Same Bus as Medium Changer ..... Yes\n");
          if (elemp->idvalid)
            printf("  SCSI Bus Address ............... %d\n",elemp->scsi);
          else
            printf("  SCSI Bus Address Valid ......... No\n");
          if (elemp->luvalid)
            printf("  Logical Unit Number ............ %d\n",elemp->lun);
          else
            printf("  Logical Unit Number Valid ...... No\n");
          printf("  Device ID ..................... %0.36s\n", elemp->identifier);
          }
         }
        else
         {
         printf("\nNo drives found in element information\n");
         }

    free(elem_devid);
    return errno;
    }
```

Chapter 7. Return Codes

This chapter describes the return codes that the device driver generates when an error occurs during an operation. The standard *errno* values are in the AIX */usr/include/sys/errno.h* header file.

If the return code is EIO, then the application can issue the STIOCQRYSENSE *ioctl* command with the LASTERROR option or the SIOC_REQSENSE *ioctl* command to analyze the sense data and determine why the error occurred.

The following codes and their descriptions apply to all operations:

[EBADF]	A bad file descriptor was passed to the device.
[EBUSY]	An excessive busy state was encountered in the device.
[EFAULT]	A memory failure occurred due to an invalid pointer or address.
[EMEDIA]	An unrecoverable media error was detected in the device.
[ENOMEM]	Insufficient memory was available for an internal memory operation.
[ENOTREADY]	The device was not ready for operation or a tape was not in the drive.
[ENXIO]	The device was not configured and is not receiving requests.
[EPERM]	The process does not have permission to perform the desired function.
[ETIMEDOUT]	A command timed out in the device.
[ENOCONNECT]	The device did not respond to selection.
[ECONNREFUSED]	The device driver has detected that the device vital product data (VPD) has changed. The device must be unconfigured in AIX and reconfigured to correct the condition.

Open Error Codes

The following codes and their descriptions apply to open operations:

[EAGAIN]	The device was opened before the open operation.
[EBADF]	A write operation was attempted on a device that was opened with the O_RDONLY flag.
[EBUSY]	The device was reserved by another initiator or an excessive busy state was encountered.
[EINVAL]	The operation requested has invalid parameters or an invalid combination of parameters, or the device is rejecting open commands.
[ENOTREADY]	If the device was not opened with the O_NONBLOCK or O_NDELAY flag, then the drive

	is not ready for operation or a tape is not in the drive. If a nonblocking flag was used, then the drive is not ready for operation.
[EWRPROTECT]	An open operation with the O_RDWR or O_WRONLY flag was attempted on a write-protected tape.
[EIO]	An I/O error occurred that indicates a failure to operate the device. Perform the failure analysis.

Write Error Codes

The following codes and their descriptions apply to write operations:

[EINVAL]	The operation requested has invalid parameters or an invalid combination of parameters. The number of bytes requested in the write operation was not a multiple of the block size for a fixed block transfer. The number of bytes requested in the write operation was greater than the maximum block size allowed by the device for variable block transfers.
[ENOSPC]	A write operation failed because it reached the early warning mark while it was in label-processing mode. This return code is returned only once when the early warning is reached.
[ENXIO]	A write operation was attempted after the device reached the logical end of the medium.
[EWRPROTECT]	A write operation was attempted on a write-protected tape.
[EIO]	The physical end of the medium was detected, or it is a general error that indicates a failure to write to the device. Perform the failure analysis.

Read Error Codes

The following codes and their descriptions apply to read operations:

[EBADF]	A read operation was attempted on a device that was opened with the O_WRONLY flag.
[EINVAL]	The operation requested has invalid parameters or an invalid combination of parameters. The number of bytes requested in the read operation was not a multiple of the block size for a fixed block transfer. The number of bytes requested in the read operation was greater than the maximum size allowed by the device for variable block transfers.
[ENOMEM]	The number of bytes requested in the read operation of a variable block record was less than the size of the block. This error is known as an overlength condition.

Close Error Code

The following code and its description apply to close operations:

[EIO] An I/O error occurred during the operation. Perform the failure analysis.

[ENOTREADY] A command issued during close, such as a rewind command, failed because the device was not ready.

IOCTL Error Codes

The following codes and their descriptions apply to *ioctl* operations:

[EINVAL] The operation requested has invalid parameters or an invalid combination of parameters.

This error code also results if the *ioctl* is not supported for the device. For example: the tape *ioctl* operations when the */dev/rmt*.smc* device is opened. Only the *ioctl* commands in Chapter 6, “Medium Changer IOCTL Operations”, on page 39 are valid in the media changer special file.

[EWRPROTECT] An operation that modifies the media was attempted on a write-protected tape or a device that was opened with the O_RDONLY flag.

[EIO] An I/O error occurred during the operation. Perform the failure analysis.

Part 2. HP-UX Tape and Medium Changer Device Driver

Chapter 8. HP-UX Programming Interface

The HP-UX programming interface to the ATDD software conforms to the standard HP-UX tape device driver interface. The following user-callable entry points are supported:

- *open()*
- *close()*
- *read()*
- *write()*
- *ioctl()*

open

The *open* entry point is called to make the driver and device ready for input/output (I/O). Only one *open* at a time is allowed for each tape device. Additional opens of the same device (whether from the same or a different client system) fail with an EBUSY error. ATDD supports multiple opens to the medium changer if the configuration parameter RESERVE is set to 0. To set the configuration parameter, see the *IBM Ultrium Device Drivers: Installation and User's Guide* for guidance .

The following code fragment illustrates a call to the *open* routine:

```
/*integer file handle */
int tape;
/*Open for reading/writing */
tape =open ("/dev/rmt/0mn",O_RDWR);
/*Print msg if open failed */
if (tape ==-1)
{
printf("open failed \n");
printf("errno =%d \n",errno);
exit (-1);
}
```

If the open system call fails, it returns -1, and the system *errno* value contains the error code as defined in the */usr/include/sys/errno.h* header file.

The oflags parameters are defined in the */usr/include/sys/fcntl.h* system header file. Use bitwise inclusive OR operations to aggregate individual values together. ATDD recognizes and supports the following oflags values:

O_RDONLY
: This flag only allows operations that do not alter the content of the tape. All special files support this flag.

O_RDWR
: This flag allows data on the tape to be read and written. An open call to any *tape drive* special file where the tape device has a write protected cartridge mounted fails.

O_WRONLY
: This flag does not allow the tape to be read. All other tape operations are allowed. An open call to any *tape drive* special file where the tape device has a write protected cartridge mounted fails.

O_NDELAY
: This option indicates to the driver not to wait until the tape drive is ready before opening the device and sending commands. If the flag is not set, an

open call requires a physical tape to be loaded and ready. The open without the flag will fail and an EIO is returned if the the tape drive isn't ready.

O_APPEND

If set, the file offset is set to the end of the file prior to each write. This flag is used in conjunction with the O_WRONLY flag to append data to the end of the current data on the tape. This flag is illegal in combination with the O_RDONLY or O_RDWR flag. An open call to any tape drive special file where the tape device has a write protected cartridge mounted fails. During an open for append operation, the tape is positioned after the last block or filemark that was written to the tape. This process can take several minutes to complete for a full tape.

close

The *close* entry point is called to terminate I/O to the driver and device.

The following code fragment illustrates a call to the close routine:

```
int rc;
rc =close (tape);
if (rc ==-1)
{
  printf("close failed \n");
  printf("errno =%d \n",errno);
  exit (-1);
}
```

where *tape* is the *open* file handle returned by the open call. The *close* routine normally would not return an error. The exception is related to the fact that any data buffered on the drive will be flushed out to tape before completion of the *close*. If any error occurs in flushing the data, an error code will be returned by the close routine.

An application should explicitly issue the close() call when the I/O resource is no longer necessary or in preparation for termination. The operating system will implicitly issue the close() call for an application that terminates without closing the resource itself. If an application terminates unexpectedly but leaves behind child processes that had inherited the file descriptor for the open resource, the operating system will not implicitly close the file descriptor because it believes it is still in use.

The close operation behavior depends on which special file was used during the open operation and which tape operation was last performed while it was opened. The commands are issued to the tape drive during the close operation according to the following logic:

```
if last operation was WRITE FILEMARK
  WRITE  FILEMARK
  BACKWARD  SPACE 1 FILEMARK

if last operation was WRITE
  WRITE  FILEMARK
  WRITE  FILEMARK
  BACKWARD  SPACE  1  FILEMARK

if last operation was READ
if special file is NOT BSD
if EOF was encountered
   FORWARD  SPACE  1  FILEMARK
```

```
SYNC BUFFER

if special file is REWIND ON CLOSE
   REWIND
```

read

The *read* entry point is called to read data from tape. The caller provides a buffer address and length, and the driver returns data from the tape to the buffer. The amount of data returned never exceeds the length parameter.

The following code fragment illustrates a *read* call to the driver:

```
actual = read(tape, buf_addr, bufsize);

if (actual > 0)
   printf("Read %d bytes\n", actual);
else if (actual == 0)
   printf("Read found file mark\n");
else
{
   printf("Error on read\n");
   printf("errno = %d\n",errno);
   exit (-1);
}
```

where *tape* is the open file handle, *buf_addr* is the address of a buffer in which to place the data, and *bufsize*, is the number of bytes to be read.

The returned value, *actual*, is the actual number of bytes read. Zero indicates a file mark.

variable block size

When in variable block size mode, the *bufsize* parameter can be any value that is valid to the drive. The amount of data returned will equal the size of the next record on the tape or the size requested *(bufsize)*, whichever is less. If *bufsize* is less than the actual record size on the tape, the remainder of the record will be lost, because the next read will start from the beginning of the next record.

fixed block size

If the tape drive is configured for fixed block size operation, the *bufsize* parameter must be a multiple of the device block size, or an error code (EINVAL) will be returned. If the *bufsize* parameter is valid, the *read* command will always return the amount of data requested unless a file mark is encountered. In that case, it will return all data that occurred before the filemark, and *actual* will equal the number of bytes returned.

write

The *write* entry point is called to write data to the tape. The caller provides the address and length of the buffer to be written to tape. Physical limitations of the drive can cause write to fail, for example, attempting to write past physical end of tape.

The following code fragment illustrates a call to the *write* routine:

```
actual = write(tape, buf_addr, bufsize);

if (actual < 0)
```

```
{
   printf("Error on write\n");
   printf("errno = %d\n",errno);
   exit (-1);
}
```

where *tape* is the open file handle, *buf_addr* is the buffer address, and *bufsize* is the size of the buffer in bytes.

The *bufsize* parameter must be a multiple of the block size, or an error will be returned (EINVAL). If the write size exceeds the device maximum block size or the configured buffer size of the tape drive, an error will be returned (EINVAL).

ioctl

The ATDD software supports all the *ioctl* commands supported by the HP-UX native drivers, tape2, stape. See the following HP-UX *man* pages for more information:

- *mt(7)*
- *scsi(7)*

Chapter 9. IOCTL Operations

The following sections describe the *ioctl*operations supported by the ATDD device driver. Usage and syntax is given, and examples are exhibited.

The *ioctl* operations supported by the IBM SCSI Tape and Medium Changer Device Driver for HP-UX are described in:

- "General SCSI IOCTL Operations"
- "SCSI Medium Changer IOCTL Operations" on page 65
- "SCSI Tape Drive IOCTL Operations" on page 74
- "Base Operating System Tape Drive IOCTL Operations" on page 86
- "Service Aid IOCTL Operations" on page 87

The following files should be included by user programs that issue the following *iotcls* to access the tape driver:

- #include <*sys/st.h*>
- #include <*sys/svc.h*>
- #include <*sys/smc.h*>
- #include <*sys/mtio.h*>

General SCSI IOCTL Operations

A set of general SCSI *ioctl* commands gives applications access to standard SCSI operations, such as device identification, access control, and problem determination for both tape drive and medium changer devices.

The following commands are supported:

IOC_TEST_UNIT_READY Determine if the device is ready for operation.

IOC_INQUIRY Collect the inquiry data from the device.

IOC_INQUIRY_PAGE Return the inquiry data for a special page from the device.

IOC_LOG_SENSE_PAGE Return a log sense page from the device.

IOC_MODE_SENSE Return the mode sense data from the device.

IOC_REQUEST_SENSE Return the device sense data.

IOC_RESERVE Reserve the device for exclusive use by the initiator.

IOC_RELEASE Release the device from exclusive use by the initiator.

IOC_PREVENT_ MEDIUM_REMOVAL
Prevent medium removal by an operator.

IOC_ALLOW_ MEDIUM_REMOVAL
Allow medium removal by an operator.

IOC_GET_DRIVER_INFO Return the driver information.

These commands and associated data structures are defined in the *st.h* and *smc.h* header files in the */usr/include/sys* directory, which is installed with the HP-UX ATDD package. Any application program that issues these commands must include one or both header files.

A sample program called *tapeutil.c* is provided with the HP-UX utility package and installed in the */opt/tapeutil* directory. This source code demonstrates the use of these *ioctl* commands.

IOC_TEST_UNIT_READY

This command determines if the device is ready for operation.

No data structure is required for this command.

An example of the IOC_TEST_UNIT_READY command is:

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

if (!(ioctl (dev_fd, IOC_TEST_UNIT_READY, 0))) {
  printf ("The IOC_TEST_UNIT_READY ioctl succeeded.\n");
}
else {
  perror ("The IOC_TEST_UNIT_READY ioctl failed");
  scsi_request_sense ();
}
```

IOC_INQUIRY

This command collects the inquiry data from the device.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  uchar qual            : 3,        /* peripheral qualifier */
        periph_type     : 5,        /* peripheral device type */
        type            : 5;        /* device type */
  uchar rm              : 1,        /* removable medium */
        mod             : 7;        /* device type modifier */
  uchar iso             : 2,        /* ISO version */
        ecma            : 3,        /* ECMA version */
        ansi            : 3;        /* ANSI version */
  uchar aen             : 1,        /* asynchronous even notification */
        trmiop          : 1,        /* terminate I/O process message */
                        : 2,        /* reserved */
        rdf             : 4;        /* response data format */
  uchar len;                        /* additional length */
  uchar                 : 8;        /* reserved */
  uchar                 : 4,        /* reserved */
       encsrv           : 1,        /* enclosure service */
       barcod           : 1,        /* bar code scanner attached  */
       multip           : 1,        /* multi-port */
       mchngr           : 1,        /* medium changer mode */
                        : 3;        /* reserved */
  uchar reladr          : 1,        /* relative addressing */
        wbus32          : 1,        /* 32-bit wide data transfers */
        wbus16          : 1,        /* 16-bit wide data transfers */
        sync            : 1,        /* synchronous data transfers */
       linked           : 1,        /* linked commands */
                        : 1,        /* reserved */
        cmdque          : 1,        /* command queueing */
        sftre           : 1;        /* soft reset */
  uchar vid[8];                     /* vendor ID */
```

```
  uchar pid[16];                    /* product ID */
  uchar rev[4];                     /* product revision level */
  uchar vendor[92];                 /* vendor specific (padded to 128) */
} inquiry_data_t;
```

An example of the IOC_INQUIRY command is:

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

inquiry_data_t inquiry_data;

if (!(ioctl (dev_fd, IOC_INQUIRY, &inquiry_data))) {
  printf ("The IOC_INQUIRY ioctl succeeded.\n");
  printf ("\nThe inquiry data is:\n");
  dump_bytes ((char *)&inquiry_data, sizeof (inquiry_data_t));
}
else {
  perror ("The IOC_INQUIRY ioctl failed");
  scsi_request_sense ();
}
```

IOC_INQUIRY_PAGE

This command returns the inquiry data when a non-zero page code is requested. For inquiry page 0x80, data mapped by structure *inq_pg_80_t* is returned in the data array; else, an array of data is returned in the data array.

The following data structures for inquiry page x80 is filled out and returned by the driver:

```
    typedef struct {
       uchar page_code;                 /* page code */
       uchar data[253];                 /* inquiry parameter List  */
} inquiry_page_t;

typedef struct {
        uchar periph_qual      : 3,     /* peripheral qualifier */
        periph_type            : 5;     /* peripheral device type */
        uchar page_code;                /* page code */
        uchar reserved_1;               /* reserved */
        uchar page_len;                 /* page length */
        uchar serial[12];               /* serial number */
} inq_pg_80_t;
```

An example of the IOC_INQUIRY_PAGE command is:

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

   inquiry_page_t inquiry_page;
   inquiry_page.page_code = (uchar) page;

   if (!(ioctl (dev_fd, IOC_INQUIRY_PAGE, &inquiry_page))){
      printf ("Inquiry Data (Page 0x%02x):\n", page);
      dump_bytes ((char *) &inquiry_page.data, inquiry_page.data[3]+4);
   }
  else {
      perror ("The IOC_INQUIRY_PAGE ioctl for page 0x%X failed.\n", page);
     scsi_request_sense ();
   }
```

IOC_REQUEST_SENSE

This command returns the device sense data. If the last command resulted in an error, then the sense data is returned for that error. Otherwise, a new (unsolicited) Request Sense command is issued to the device.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  uchar valid           : 1,        /* sense data is valid */
        code            : 7;        /* error code */
  uchar segnum;                     /* segment number */
  uchar fm              : 1,        /* filemark detected */
        eom             : 1,        /* end of media */
        ili             : 1,        /* incorrect length indicator */
                        : 1,        /* reserved */
        key             : 4;        /* sense key */
  uchar info[4];                    /* information bytes */
  uchar addlen;                     /* additional sense length */
  uchar cmdinfo[4];                 /* command-specific information */
  uchar asc;                        /* additional sense code */
  uchar ascq;                       /* additional sense code qualifier */
  uchar fru;                        /* field-replaceable unit code */
  uchar sksv            : 1,        /* sense key specific valid */
        cd              : 1,        /* control/data */
                        : 2,        /* reserved */
        bpv             : 1,        /* bit pointer valid */
        sim             : 3;        /* system information message */
  uchar field[2];                   /* field pointer */
  uchar vendor[110];                /* vendor specific (padded to 128) */
} sense_data_t;
```

An example of the IOC_REQUEST_SENSE command is:

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

sense_data_t sense_data;

if (!(ioctl (dev_fd, IOC_REQUEST_SENSE, &sense_data))) {
  printf ("The IOC_REQUEST_SENSE ioctl succeeded.\n");
  printf ("\nThe request sense data is:\n");
  dump_bytes ((char *)&sense_data, sizeof (sense_data_t));
}
else {
  perror ("The IOC_REQUEST_SENSE ioctl failed");
}
```

IOC_LOG_SENSE_PAGE

This *ioctl* command returns a log sense page from the device. The desired page is selected by specifying the page_code in the log_sense_page structure.

The structure of a log page consists of the following log page header and log parameter(s):

Log Page

- Log Page Header
  - Page Code
  - Page Length
- Log Parameters (one or more may exist)
  - Parameter Code
  - Control Byte
  - Parameter Length
  - Parameter Value

The following data structure is filled out and returned by the driver:

```
typedefstruct {
  uchar page code;                  /* page code */
  uchar data[MAX_LGPGDATA];         /* log data structure */
}log_sns_pg_t;
```

An example of the IOC_LOG_SENSE_PAGE command is:

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

  int i, j=0;
  int true;
  int len, parm_len;
  int parm_code;
  log_sns_pg_t   log_sns_page;

  memset ((char *) &log_sns_page, (char)0, sizeof(log_sns_pg_t));
  log_sns_page.page_code = (uchar) page;

  if (!(rc = ioctl (dev_fd, IOC_LOG_SENSE_PAGE, &log_sns_page))) {
     len =(int) ((log_sns_page.data[2] << 8) + log_sns_page.data[3]) + 4;
     if ( type != 1) {
        printf("Log Sense Data (Page 0x%02x):\n", page);
        dump_bytes ((char *) &log_sns_page.data, len);
     }
     else {
        for(i=4; i<=len; i=(parm_len+4)){
           j += i;
           parm_code = (int) ((log_sns_page.data[j] << 8) + log_sns_page.data[j+1]);
           parm_len = (int) (log_sns_page.data[j+3]);
           if (true = (parm_code == parmcode)) {
              printf("Log Sense Data (Page 0x%02x, Parameter Code 0x%04x):\n",
                 page, parmcode);
              dump_bytes ((char *) &log_sns_page.data[j], (parm_len+4));
              break;
           }
        }
    if (!true)
          printf("IOC_LOG_SENSE_PAGE for Page 0x%02x, Parameter Code 0x%04x failed.\n",
                page, parmcode);}
  }
  else {
    printf("IOC_LOG_SENSE_PAGE for page 0x%X failed.\n", page);
    scsi_request_sense ();
  }
```

IOC_MODE_SENSE

This command returns a mode sense page from the device. The desired page is selected by specifying the page_code in the mode_sns_t structure.

The following data structure is filled out and returned by the driver:

```
typedef struct {
    uchar page_code;                  /* page code  */
    uchar cmd_code;                   /* SCSI command code */
    uchar data[253];                  /* mode parameter list
 }mode_sns_t;
```

An example of the IOC_MODE_SENSE command is:

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

  int offset;
  mode_sns_t mode_data;
  mode_data.page_code = (uchar) page;

  memset ((char *) &mode_data, (char)0, sizeof(mode_sns_t));
```

```
if (!(rc = ioctl (dev_fd, IOC_MODE_SENSE, &mode_data))) {
  if ( mode_data.cmd_code == 0x1A )
    offset = (int) (mode_data.data[3]) + sizeof(mode_hdr6_t);
  if ( mode_data.cmd_code == 0x5A )
    offset = (int) ((mode_data.data[6] << 8) + mode_data.data[7]) + sizeof(mode_hdr10_t);
    printf("Mode Data (Page 0x%02x):\n", mode_data.page_code);
    dump_bytes ((char *) &mode_data.data[offset], (mode_data.data[offset+1] + 2));
}
else {
  printf("IOC_MODE_SENSE for page 0x%X failed.\n", mode_data.page_code);
  scsi_request_sense ();
}
```

IOC_RESERVE

This command persistently reserves the device for exclusive use by the initiator. The ATDD device driver normally reserves the device in the open operation and releases the device in the close operation. Issuing this command will prevent the driver from releasing the device during the close operation; hence the device reservation is maintained after the device is closed. This command is negated by issuing the IOC_RELEASE *ioctl* command.

No data structure is required for this command.

An example of the IOC_RESERVE command is:

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

if (!(ioctl (dev_fd, IOC_RESERVE, 0))) {
  printf ("The IOC_RESERVE ioctl succeeded.\n");
}
else {
  perror ("The IOC_RESERVE ioctl failed");
  scsi_request_sense ();
}
```

IOC_RELEASE

This command releases the persistent reservation of the device for exclusive use by the initiator. It negates the result of the IOC_RESERVE *ioctl* command issued either from the current or a previous open session.

No data structure is required for this command.

An example of the IOC_RELEASE command is:

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

if (!(ioctl (dev_fd, IOC_RELEASE, 0))) {
  printf ("The IOC_RELEASE ioctl succeeded.\n");
}
else {
  perror ("The IOC_RELEASE ioctl failed");
  scsi_request_sense ();
}
```

IOC_PREVENT_MEDIUM_REMOVAL

This command prevents an operator from removing media from the tape drive or the medium changer.

No data structure is required for this command.

An example of the IOC_PREVENT_MEDIUM_REMOVAL command is:

```
#include <sys/st.h>
if (!(ioctl (dev_fd,IOC_PREVENT_MEDIUM_REMOVAL,NULL))) {
 printf ("The IOC_PREVENT_MEDIUM_REMOVAL ioctl succeeded \n");
}
else {
 perror ("The IOC_PREVENT_MEDIUM_REMOVAL ioctl failed");
 scsi_request_sense();
}
```

IOC_ALLOW_MEDIUM_REMOVAL

This command allows an operator to remove media from the tape drive and the medium changer. This command is normally used after an IOC_PREVENT_MEDIUM_REMOVAL command to restore the device to the default state.

No data structure is required for this command.

An example of the IOC_ALLOW_MEDIUM_REMOVAL command is:

```
#include <sys/st.h>
if (!(ioctl (dev_fd,IOC_ALLOW_MEDIUM_REMOVAL,NULL))) {
 printf ("The IOC_ALLOW_MEDIUM_REMOVAL ioctl succeeded \n");
}
else {
 perror ("The IOC_ALLOW_MEDIUM_REMOVAL ioctl failed");
 scsi_request_sense();
}
```

IOC_GET_DRIVER_INFO

This command returns the information of the current installed ATDD.

The following data structure is filled out and returned by the driver.

```
typedef struct {
    char driver_id[64];               /* the name of the tape driver (ATDD) */
    char version[25];                 /* the version of the tape driver     */
} Get_driver_info_t;
```

An example of the IOC_GET_DRIVER_INFO command is:

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

get_driver_info_t get_driver_info;

  if (!(rc = ioctl (dev_fd, IOC_GET_DRIVER_INFO, &get_driver_info))) {
    strncpy (driver_level, get_driver_info.version, 7);
    PRINTF ("The version of %s(Advanced Tape Device Driver): %s\n", get_driver_info.driver_
id, driver_level);
  }
  else {
    PERROR ("Failure obtaining the version of ATDD");
    PRINTF ("\n");
    scsi_request_sense ();
  }
```

SCSI Medium Changer IOCTL Operations

A set of medium changer *ioctl* commands gives applications access to IBM medium changer devices.

The following commands are supported:

SMCIOC_MOVE_MEDIUM Transport a cartridge from one element to another element.

SMCIOC_POS_TO_ELEM Move the robot to an element.

SMCIOC_ELEMENT_INFO Return the information about the device elements.

SMCIOC_INVENTORY Return the information about the medium changer elements.

SMCIOC_AUDIT Perform an audit of the element status.

SMCIOC_LOCK_DOOR Lock and unlock the library access door.

SMCIOC_READ_ELEMENT_DEVIDS
Return the device ID element descriptors for drive elements.

SMCIOC_EXCHANGE_MEDIUM
Exchange a cartridge in an element with another cartridge.

SMCIOC_INIT_ELEM_STAT_RANGE
Issue the SCSI Initialize Element Status with Range command.

These commands and associated data structures are defined in the *smc.h* header file in the */usr/include/sys* director, which is installed with the ATDD package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the device driver and installed in the */opt/tapeutil* directory with the *tapeutil* package. This source code demonstrates the use of these *ioctl* commands.

SMCIOC_MOVE_MEDIUM

This command transports a cartridge from one element to another element.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  ushort robot;                    /* robot address */
  ushort source;                   /* move from location */
  ushort destination;              /* move to location */
  uchar invert;                    /* invert medium before insertion */
} move_medium_t;
```

An example of the SMCIOC_MOVE_MEDIUM command is:

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

move_medium_t move_medium;

move_medium.robot = 0;
move_medium.invert = NO_FLIP;
move_medium.source = src;
move_medium.destination = dst;

if (!(ioctl (dev_fd, SMCIOC_MOVE_MEDIUM, &move_medium))) {
  printf ("The SMCIOC_MOVE_MEDIUM ioctl succeeded.\n");
}
else {
  perror ("The SMCIOC_MOVE_MEDIUM ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_POS_TO_ELEM

This command moves the robot to an element.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  ushort robot;                       /* robot address */
  ushort destination;                 /* move to location */
  uchar invert;                       /* invert medium before insertion */
} pos_to_elem_t;
```

An example of the SMCIOC_POS_TO_ELEM command is:

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

  pos_to_elem_t pos_to_elem;

  pos_to_elem.robot = 0;
  pos_to_elem.invert = NO_FLIP;
  pos_to_elem.destination = dst;

  if (!(ioctl (dev_fd, SMCIOC_POS_TO_ELEM, &pos_to_elem))) {
    printf ("The SMCIOC_POS_TO_ELEM ioctl succeeded.\n");
  }
  else {
    perror ("The SMCIOC_POS_TO_ELEM ioctl failed");
    scsi_request_sense ();
  }
```

SMCIOC_ELEMENT_INFO

This command requests the information about the device elements.

There are four types of medium changer elements. (Not all medium changers support all four types.) The robot elements are associated with the cartridge transport devices. The cell elements are associated with the cartridge storage slots. The port elements are associated with the import/export mechanisms. The drive elements are associated with the data-transfer devices. The quantity of each element type and its starting address is returned by the driver.

The following data structure is filled out and returned by the driver:

```
  typedef struct {
    ushort robot_address;             /* medium transport element address */
    ushort robot_count;               /* number medium transport elements */
    ushort cell_address;              /* medium storage element address */
    ushort cell_count;                /* number medium storage elements */
    ushort port_address;              /* import/export element address */
    ushort port_count;                /* number import/export elements */
    ushort drive_address;             /* data-transfer element address */
    ushort drive_count;               /* number data-transfer elements */
  } element_info_t;
```

An example of the SMCIOC_ELEMENT_INFO command is:

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

  element_info_t element_info;

  if (!(ioctl (dev_fd, SMCIOC_ELEMENT_INFO, &element_info))) {
    printf ("The SMCIOC_ELEMENT_INFO ioctl succeeded.\n");
    printf ("\nThe element information data is:\n");
    dump_bytes ((char *)&element_info, sizeof (element_info_t));
  }
```

```
 else {
  perror ("The SMCIOC_ELEMENT_INFO ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_INVENTORY

This command returns the information about the medium changer elements (SCSI Read Element Status command).

There are four types of medium changer elements. (Not all medium changers support all four types.) The robot elements are associated with the cartridge transport devices. The cell elements are associated with the cartridge storage slots. The port elements are associated with the import/export mechanisms. The drive elements are associated with the data-transfer devices.

Note: The application must allocate buffers large enough to hold the returned element status data for each element type. The SMCIOC_ELEMENT_INFO *ioctl* command is generally called first to establish the criteria.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  element_status_t *robot_status;    /* medium transport element pages */
  element_status_t *cell_status;     /* medium storage element pages */
  element_status_t *port_status;     /* import/export element pages */
  element_status_t *drive_status;    /* data-transfer element pages */
} inventory_t;
```

One or more of the following data structures are filled out and returned to the user buffer by the driver:

```
typedef struct {
  ushort address;                    /* element address */
  uchar                 : 2,         /* reserved */
        inenab          : 1,         /* medium in robot scope */
        exenab          : 1,         /* medium not in robot scope */
        access          : 1,         /* robot access allowed */
        except          : 1,         /* abnormal element state */
        impexp          : 1,         /* medium imported or exported */
        full            : 1;         /* element contains medium */
  uchar                 : 8;         /* reserved */
  uchar asc;                         /* additional sense code */
  uchar ascq;                        /* additional sense code qualifier */
  uchar notbus          : 1,         /* element not on same bus as robot */
                        : 1,         /* reserved */
        idvalid         : 1,         /* element address valid */
        luvalid         : 1,         /* logical unit valid */
                        : 1,         /* reserved */
        lun             : 3;         /* logical unit number */
  uchar scsi;                        /* SCSI bus address */
  uchar                 : 8;         /* reserved */
  uchar svalid          : 1,         /* element address valid */
        invert          : 1,         /* medium inverted */
                        : 6;         /* reserved */
  ushort source;                     /* source storage element address */
  uchar volume[36];                  /* primary volume tag */
  uchar vendor[80];                  /* vendor specific (padded to 128) */
} element_status_t;
```

An example of the SMCIOC_INVENTORY command is:

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

ushort i;
```

```
element_info_t element_info;
inventory_t inventory;

smc_element_info ();  /* get element information first */

inventory.robot_status = (element_status_t *)malloc
   (sizeof (element_status_t) * element_info.robot_count);
inventory.cell_status  = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.cell_count );
inventory.port_status  = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.port_count );
inventory.drive_status = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.drive_count);

if (!inventory.robot_status || !inventory.cell_status ||
    !inventory.port_status  || !inventory.drive_status) {
  perror ("The SMCIOC_INVENTORY ioctl failed");
  return;
}

if (!(ioctl (dev_fd, SMCIOC_INVENTORY, &inventory))) {

  printf ("\nThe SMCIOC_INVENTORY ioctl succeeded.\n");

  printf ("\nThe robot status pages are:\n");

  for (i = 0; i < element_info.robot_count; i++) {
    dump_bytes ((char *)(&inventory.robot_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }

  printf ("\nThe cell status pages are:\n");

  for (i = 0; i < element_info.cell_count; i++) {
    dump_bytes ((char *)(&inventory.cell_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }
  printf ("\nThe port status pages are:\n");

  for (i = 0; i < element_info.port_count; i++) {
    dump_bytes ((char *)(&inventory.port_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }

  printf ("\nThe drive status pages are:\n");

  for (i = 0; i < element_info.drive_count; i++) {
    dump_bytes ((char *)(&inventory.drive_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }

}
else {
  perror ("The SMCIOC_INVENTORY ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_AUDIT

This command causes the medium changer device to perform an audit of the element status (SCSI Initialize Element Status command).

No data structure is required for this command.

An example of the SMCIOC_AUDIT command is:

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

if (!(ioctl (dev_fd, SMCIOC_AUDIT, 0))) {
  printf ("The SMCIOC_AUDIT ioctl succeeded.\n");
}
else {
  perror ("The SMCIOC_AUDIT ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_LOCK_DOOR

This command locks and unlocks the library access door. Not all IBM medium changer devices support this operation.

The following data structure is filled out and supplied by the caller:

```
typedef uchar lock_door_t;
```

An example of the SMCIOC_LOCK_DOOR command is:

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

lock_door_t lock_door;

lock_door = LOCK;

if (!(ioctl (dev_fd, SMCIOC_LOCK_DOOR, &lock_door))) {
  printf ("The SMCIOC_LOCK_DOOR ioctl succeeded.\n");
}
else {
  perror ("The SMCIOC_LOCK_DOOR ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_READ_ELEMENT_DEVIDS

This *ioctl* command issues the SCSI Read Element Status command with the device ID (DVCID) bit set and returns the element descriptors for the data transfer elements. The *element_address* field specifies the starting address of the first data transfer element. The *number_elements* field specifies the number of elements to return. The application must allocate a return buffer large enough for the *number_elements* specified in the input structure.

The input data structure is:

```
The input data structure is:
  typedef struct {
    ushort element_address;              /* starting element address */
    ushort number_elements;              /* number of elements       */
    element_devid_t *drive_devid;        /* data transfer element pages */
  } read_element_devids_t;
```

The output data structure is:

```
typedef struct {
   ushort address;            /* element address */
   uchar          :4,         /* reserved */
          access  :1,         /* robot access allowed */
          except  :1,         /* abnormal element state */
                  :1,         /* reserved */
          full    :1;         /* element contains medium */
   uchar  resvd1;             /* reserved */
   uchar  asc;                /* additional sense code */
   uchar  ascq;               /* additional sense code qualifier */
   uchar  notbus  :1,         /* element not on same bus as robot */
                  :1,         /* reserved */
          idvalid :1,         /* element address valid */
          luvalid :1,         /* logical unit valid */
                  :1,         /* reserved */
          lun     :3;         /* logical unit number */
   uchar  scsi;               /* scsi bus address */
   uchar  resvd2;             /* reserved */
   uchar  svalid  :1,         /* element address valid */
          invert  :1,         /* medium inverted */
                  :6;         /* reserved */
   ushort source;             /* source storage element address */
   uchar          :4,         /* reserved */
          code_set:4;         /* code set X'2' is all ASCII identifier */
   uchar          :4,         /* reserved */
          id_type :4;         /* identifier type */
   uchar  resvd3;             /* reserved */
   uchar  id_len;             /* identifier length */
   uchar  dev_id[36];         /* device identification with serial number */
} element_devid_t;
```

An example of the SMIOC_READ_ELEMENT_DEVIDS command is:

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

int rc;
int i;

element_devid_t *elem_devid, *elemp;
read_element_devids_t devids;
element_info_t element_info;

if (rc = ioctl (dev_fd, SMCIOC_ELEMENT_INFO, &element_info)) {
  PERROR ("The SMCIOC_READ_ELEMENT_DEVIDS ioctl failed: Get the element info failure.\n");
  PRINTF ("\n");
  scsi_request_sense ();
  return (rc);
}

if (element_info.drive_count) {
  elem_devid = malloc(element_info.drive_count * sizeof(element_devid_t));

  if (elem_devid == NULL) {
    PRINTF ("The SMCIOC_READ_ELEMENT_DEVIDS ioctl failed: Memory allocation failure.\n");
    return (ENOMEM);
  }
  bzero(elem_devid, element_info.drive_count * sizeof(element_devid_t));

  devids.drive_devid = elem_devid;
  devids.element_address = element_info.drive_address;
  devids.number_elements = element_info.drive_count;

  printf("Reading element device ids...\n");

  if (!(rc = ioctl (dev_fd, SMCIOC_READ_ELEMENT_DEVIDS, &devids))) {
    elemp = elem_devid;
```

```
PRINTF ("\nThe SMCIOC_READ_ELEMENT_DEVIDS ioctl succeeded.\n");
PRINTF ("\nThe drives status datas are:\n");

for (i = 0; i < element_info.drive_count; i++, elemp++) {
   printf("\n  Drive Address ................. %d\n",elemp->address);
   if (elemp->except)
      printf("  Drive State ................... Abnormal\n");
   else
      printf("  Drive State ................... Normal\n");
   if (elemp->asc == 0x81 && elemp->ascq ==0x00)
      printf("  ASC/ASCQ ...................... %02X%02X (Drive Present)\n",
        elemp->asc,elemp->ascq);
  else if (elemp->asc == 0x82 && elemp->ascq ==0x00)
      printf("  ASC/ASCQ ...................... %02X%02X (Drive Not Present)\n",
         elemp->asc,elemp->ascq);
  else
      printf("  ASC/ASCQ ...................... %02X%02X\n",
         elemp->asc,elemp->ascq);
  if (elemp->full)
      printf("  Media Present ................. Yes\n");
  else
      printf("  Media Present ................. No\n");
  if (elemp->access)
     printf("  Robot Access Allowed .......... Yes\n");
  else
     printf("  Robot Access Allowed .......... No\n");
  if (elemp->svalid)
     printf("  Source Element Address ........ %d\n",elemp->source);
  else
     printf("  Source Element Address Valid ... No\n");
  if (elemp->invert)
     printf("  Media Inverted ................ Yes\n");
  else
     printf("  Media Inverted ................ No\n");
  if (elemp->notbus)
     printf("  Same Bus as Medium Changer ..... No\n");
  else
     printf("  Same Bus as Medium Changer ..... Yes\n");
  if (elemp->idvalid)
     printf("  SCSI Bus Address ............... %d\n",elemp->scsi);
  else
     printf("  SCSI Bus Address Vaild ........ No\n");
  if (elemp->luvalid)
     printf("  Logical Unit Number ............ %d\n",elemp->lun);
  else
     printf("  Logical Unit Number Valid ...... No\n");
  if (elemp->dev_id[0] == '\0')
     printf("  Device ID ..................... No\n");
  else
     printf("  Device ID ..................... %0.36s\n", elemp->dev_id);


   PRINTF ("\n--- more ---");
   getchar();
  }
 }
 else {
    PERROR ("The SMCIOC_READ_ELEMENT_DEVIDS ioctl failed");
    PRINTF ("\n");
    scsi_request_sense ();
 }
}
else {
   printf("\nNo drives found in element information\n");
```

```
    }

  free (elem_devid);
  return (rc);
```

SMCIOC_EXCHANGE_MEDIUM

This *ioctl* command exchanges a cartridge in an element with another cartridge. This command is equivalent to two SCSI Move Medium commands. The first moves the cartridge from the source element to the destination1 element, and the second moves the cartridge that was previously in the destination1 element to the destination2 element. The destination2 element can be the same as the source element.

The input data structure is:

```
typedef struct {
 ushort robot;     /* robot address */
 ushort source;    /* move from location */
 ushort destination1;  /* move to location */
 ushort destination2;  /* move to location */
 uchar invert1;    /* invert before placement into destination 1 */
 uchar invert2;    /* invert before placement into destination 2 */ }
 exchange_medium_t;
```

An example of the SMCIOC_EXCHANGE_MEDIUM command is:

```
#include <sys/smc.h>
 int rc;
  exchange_medium_t exchange_medium;
  exchange_medium.robot = 0;
  exchange_medium.invert1 = NO_FLIP;
  exchange_medium.invert2 = NO_FLIP;
  exchange_medium.source = (short)src;
  exchange_medium.destination1 = (short)dst;
  exchange_medium.destination2 = (short)dst2;
  if (!(rc = ioctl (dev_fd, SMCIOC_EXCHANGE_MEDIUM,
   &exchange_medium))) {
   PRINTF ("The SMCIOC_EXCHANGE_MEDIUM ioctl succeeded.\n");
   }
  else {
   PERROR ("The SMCIOC_EXCHANGE_MEDIUM ioctl failed");
   PRINTF ("\n");
   scsi_request_sense ();
  }
return (rc);
```

SMCIOC_INIT_ELEM_STAT_RANGE

This *ioctl* command issues the SCSI Initialize Element Status with Range command and is used to audit specific elements in a library by specifying the starting element address and number of elements. Use the SMCIOC_INIT_ELEM_STAT ioctl to audit all elements.

The data structure is:

```
typedef struct {
 ushort element_address;  /* starting element address */
 ushort number_elements;  /* number of elements */
} element_range_t;
```

An example of the SMCIOC_INIT_ELEM_STAT_RANGE command is:

```
#include <sys/smc.h>
 int rc;
  element_range_t elem_range;
```

```
elem_range.element_address = (short)src;
elem_range.number_elements = (short)number;
if (!(rc = ioctl (dev_fd, SMCIOC_INIT_ELEM_STAT_RANGE, &elem_range))) {
 PRINTF ("The SMCIOC_INIT_ELEM_STAT_RANGE ioctl succeeded.\n"); }
else {
 PERROR ("The SMCIOC_INIT_ELEM_STAT_RANGE ioctl failed");
 PRINTF ("\n");
 scsi_request_sense ();
 }
return (rc);
```

SCSI Tape Drive IOCTL Operations

A set of enhanced *ioctl* commands gives applications access to additional features of IBM tape drives.

The following commands are supported:

1. **STIOC_TAPE_OP** Perform the tape drive operation.
2. **STIOC_GET_DEVICE_STATUS** Return the status information about the tape drive.
3. **STIOC_GET_DEVICE_INFO** Return the configuration information about the tape drive.
4. **STIOC_GET_MEDIA_INFO** Return the information about the currently mounted tape.
5. **STIOC_GET_POSITION** Return the information about the tape position.
6. **STIOC_SET_POSITION** Set the physical position of the tape.
7. **STIOC_GET_PARM** Return the current value of the working parameter for the tape drive.
8. **STIOC_SET_PARM** Set the current value of the working parameter for the tape drive.
9. **STIOC_DISPLAY_MSG** Display the messages on the tape drive console.
10. **STIOC_SYNC_BUFFER** Flush the drive buffers to the tape.
11. **STIOC_REPORT_DENSITY_SUPPORT** Return supported densities from the tape device.

These commands and associated data structures are defined in the *st.h* header file in the */usr/include/sys* directory, which is installed with the ATDD package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the *tapeutil* package and installed in the */opt/tapeutil* directory. This source code demonstrates the use of these *ioctl* commands.

STIOC_TAPE_OP

This command performs standard tape drive operations. It is similar to the MTIOCTOP *ioctl* command defined in the */usr/include/sys/mtio.h* system header file, but the STIOC_TAPE_OP command uses the ST_OP opcodes and the data structure defined in the */usr/include/sys/st.h* system header file. Most STIOC_TAPE_OP *ioctl* commands map to the MTIOCTOP *ioctl* command. See “MTIOCTOP” on page 86.

For all *space* operations, the resulting tape position is at the end-of-tape side of the record or filemark for forward movement and at the beginning-of-tape side of the record or filemark for backward movement.

The following data structure is filled out and supplied by the caller:

```
/*from st.h    */
typedef struct {
 short st_op;    /*st operations defined below */
 daddr_t st_count;  /*how many of them */
}tape_op_t;
```

The *st_op* field is set to one of the following:

ST_OP_WEOF
Write *st_count* filemarks.

ST_OP_FSF
Space forward *st_count* filemarks.

ST_OP_BSF
Space backward *st_count* filemarks. Upon completion, the tape is positioned at the beginning-of-tape side of the requested filemark.

ST_OP_FSR
Space forward the *st_count* number of records.

ST_OP_BSR
Space backward the *st_count* number of records.

ST_OP_REW
Rewind the tape. The *st_count* parameter does not apply.

ST_OP_OFFL
Rewind and unload the tape. The *st_count* parameter does not apply.

ST_OP_NOP
No tape operation is performed. The status is determined by issuing the Test Unit Ready command. The *st_count* parameter does not apply.

ST_OP_RETEN
Retension the tape. The *st_count* parameter does not apply.

ST_OP_ERASE
Erase the entire tape from the current position. The *st_count* parameter does not apply.

ST_OP_EOD
Space forward to the end of the data. The *st_count* parameter does not apply.

ST_OP_NBSF
Space backward *st_count* filemarks, then space backward before all data records in that tape file. For a given ST_OP_NBSF operation with *st_count=n*, the equivalent position can be achieved with ST_OP_BSF and ST_OP_FSF, as follows:

```
ST_OP_BSF with mst_count = n +1
ST_OP_FSF with st_count = 1
```

ST_OP_GRSZ
Return the current record (block) size. The *st_count* parameter contains the value.

ST_OP_SRSZ
Set the working record (block) size to *st_count*.

ST_OP_RES
Reserve the tape drive. The *st_count* parameter does not apply.

ST_OP_REL
Release the tape drive. The *st_count* parameter does not apply.

ST_OP_LOAD
Load the tape in the drive. The st_count parameter does not apply.

ST_OP_UNLOAD
Unload the tape from the drive. The *st_count* parameter does not apply.

An example of the STIOC_TAPE_OP command is:

```
#include &lt sys/st.h>

tape_op_t tape_op;

tape_op.st_op =st_op;
tape_op.st_count =st_count;

if (!(ioctl (dev_fd,STIOC_TAPE_OP, &TAPE_OP))){
 printf ("The STIOC_TAPE_OP ioctl succeeded.\n");
}
else {
  perror ("The STIOC_TAPE_OP ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_DEVICE_STATUS

This command returns the status information about the tape drive. It is similar to the MTIOCGET *ioctl* command defined in the */usr/include/sys/mtio.h* system header file. The STIOC_GET_DEVICE_STATUS and MTIOCGET commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The STIOC_GET_DEVICE_STATUS *ioctl* command maps to the MTIOCGET *ioctl* command. The two *ioctl* commands are interchangeable. See “MTIOCGET” on page 87.

The following data structure is returned by the driver:

```
  /* from mtio.h */
  struct mtget {
    long mt_type;              /* type of magnetic tape device */
    long mt_resid;             /* residual count */
  /* The following two registers are device dependent */
    long mt_dsreg1;            /* status register (msb) */
    long mt_dsreg2;            /* status register (lsb) */
  /* The following are device-independent status words */
    long mt_gstat;             /* generic status */
    long mt_erreg;             /* error register */
    daddr_t mt_fileno;         /* No longer used. Always set to -1. */
    daddr_t mt_blkno;          /* No longer used. Always set to -1. */
      };

  /* from st.h */
  typedef struct mtget device_status_t;
```

The *mt_flags* field, which returns the type of automatic cartridge stacker or loader installed on the tape drive, is set to one of the following values:

STF_ACL	Automatic Cartridge Loader
STF_RACL	Random Access Cartridge Facility

An example of the STIOC_GET_DEVICE_STATUS command is:

```
#include <sys/mtio.h>
#include <sys/st.h>

device_status_t device_status;

if (!(ioctl (dev_fd, STIOC_GET_DEVICE_STATUS, &device_status))) {
  printf ("The STIOC_GET_DEVICE_STATUS ioctl succeeded.\n");
  printf ("\nThe device status data is:\n");
  dump_bytes ((char *)&device_status, sizeof (device_status_t));
}
else {
  perror ("The STIOC_GET_DEVICE_STATUS ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_DEVICE_INFO

This command returns the configuration information about the tape drive. The STIOC_GET_DEVICE_INFO command uses the following data structure defined in the */usr/include/sys/st* system header file.

The following data structure is returned by the driver:

```
/* from st.h */
struct mtdrivetype {
  char name[64];                    /* name */
  char vid[25];                     /* vendor ID, product ID */
  char type;                        /* drive type */
  int bsize;                        /* block size */
  int options;                      /* drive options */
  int max_rretries;                 /* maximum read retries */
  int max_wretries;                 /* maximum write retries */
  uchar default_density;           /* default density chosen */
};

  typedef struct mtdrivetype device_info_t;
```

An example of the STIOC_GET_DEVICE_INFO command is:

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

device_info_t device_info;

if (!(ioctl (dev_fd, STIOC_GET_DEVICE_INFO, &device_info))) {
  printf ("The STIOC_GET_DEVICE_INFO ioctl succeeded.\n");
  printf ("\nThe device information is:\n");
  dump_bytes ((char *)&device_info, sizeof (device_info_t));
}
 else {
  perror ("The STIOC_GET_DEVICE_INFO ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_MEDIA_INFO

This command returns the information about the currently mounted tape.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  uint media_type;                  /* type of media loaded */
  uint media_format;                /* format of media loaded */
  uchar write_protect;              /* write protect (physical/logical) */
} media_info_t;
```

The *media_type* and *media_format* format fields are set to one of the values in st.h.

The *write_protect* field is set to 1 if the currently mounted tape is physically or logically write-protected.

An example of the STIOC_GET_MEDIA_INFO command is:

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

media_info_t media_info;

if (!(ioctl (dev_fd, STIOC_GET_MEDIA_INFO, &media_info))) {
  printf ("The STIOC_GET_MEDIA_INFO ioctl succeeded.\n");
  printf ("\nThe media information is:\n");
  dump_bytes ((char *)&media_info, sizeof (media_info_t));
}
else {
  perror ("The STIOC_GET_MEDIA_INFO ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_POSITION

This command returns the information about the tape position.

The tape position is defined as where the next read or write operation will occur. The STIOC_GET_POSITION and STIOC_SET_POSITION commands can be used independently or in conjunction with each other.

The following data structure is filled out and supplied by the caller (and also filled out and returned by the driver):

```
typedef struct {
  uchar block_type;                /* block type (logical or physical) */
  uchar bot;                       /* physical beginning of tape */
  uchar eot;                       /* logical end of tape */
  uint position;                   /* current or new block ID */
  uint last_block;                 /* last block written to tape */
  uint block_count;                /* blocks remaining in buffer */
  uint byte_count;                 /* bytes remaining in buffer */
} position_data_t;
```

The *block_type* field is set to LOGICAL_BLK for standard SCSI logical tape positions or PHYSICAL_BLK for composite tape positions used for high-speed locate operations implemented by the tape drive. The IBM Ultrium tape devices support the LOGICAL_BLK type.

The *block_type* is the only field that the caller must fill out. The other fields are ignored. Tape positions can be obtained with the STIOC_GET_POSITION command, saved, and used later with the STIOC_SET_POSITION command to return quickly to the same location on the tape.

The *position* field returns the current position of the tape (physical or logical).

The *last_block* field returns the last block of data that was transferred physically to the tape.

The *block_count* field returns the number of blocks of data remaining in the buffer.

The *byte_count* field returns the number of bytes of data remaining in the buffer.

The *bot* and *eot* fields indicate if the tape is positioned at the beginning of tape or the end of tape, respectively.

An example of the STIOC_GET_POSITION command is:

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

position_data_t position_data;
position_data.block_type = type;

if (!(ioctl (dev_fd, STIOC_GET_POSITION, &position_data))) {
  printf ("The STIOC_GET_POSITION ioctl succeeded.\n");
  printf ("\nThe tape position data is:\n");
  dump_bytes ((char *)&position_data, sizeof (position_data_t));
}
else {
  perror ("The STIOC_GET_POSITION ioctl failed");
  scsi_request_sense ();
}
```

STIOC_SET_POSITION

This command sets the physical position of the tape.

The tape position is defined as where the next read or write operation will occur. The STIOC_GET_POSITION and STIOC_SET_POSITION commands can be used independently or in conjunction with each other.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar block_type;                /* block type (logical or physical) */
  uchar bot;                       /* physical beginning of tape */
  uchar eot;                       /* logical end of tape */
  uint position;                   /* current or new block ID */
  uint last_block;                 /* last block written to tape */
  uint block_count;                /* blocks remaining in buffer */
  uint byte_count;                 /* bytes remaining in buffer */
} position_data_t;
```

The *block_type* field is set to LOGICAL_BLK for standard SCSI logical tape positions or PHYSICAL_BLK for composite tape positions used for high-speed locate operations implemented by the tape drive. The IBM Ultrium tape devices support the LOGICAL_BLK type.

The *block_type* and *position* fields must be filled out by the caller. The other fields are ignored. The type of position specified in the *position* field must correspond with the type specified in the *block_type* field. Tape positions can be obtained with the STIOC_GET_POSITION command, saved, and used later with the STIOC_SET_POSITION command to quickly return to the same location on the tape.

An example of the STIOC_SET_POSITION command is:

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

position_data_t position_data;
position_data.block_type = type;
position_data.position = value;

if (!(ioctl (dev_fd, STIOC_SET_POSITION, &position_data))) {
  printf ("The STIOC_SET_POSITION ioctl succeeded.\n");
}
```

```
else {
  perror ("The STIOC_SET_POSITION ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_PARM

This command returns the current value of the working parameter for the specified tape drive. This command is used in conjunction with the STIOC_SET_PARM command.

The following data structure is filled out and supplied by the caller (and also filled out and returned by the driver):

```
typedef struct {
  uchar type;                    /* type of parameter to get or set */
  uint value;                    /* current or new value of parameter */
} parm_data_t;
```

The *value* field returns the current value of the specified parameter within the following ranges for the specific *type*.

The *type* field, which the caller fills out, should be set to one of the following values:

BLOCKSIZE Block Size (0–16777215)

A value of zero indicates variable block size. The IBM Ultrium devices support about a 16 MB maximum block size.

COMPRESSION Compression Mode (0 or 1)

If this mode is enabled, data is compressed by the tape device before storing it on tape.

BUFFERING Buffering Mode (0 or 1)

If this mode is enabled, data is stored in hardware buffers in the tape device and not immediately committed to tape, thus increasing data throughput performance.

IMMEDIATE Immediate Mode (0 or 1)

If this mode is enabled, then a rewind command returns with the status before the completion of the physical rewind operation by the tape drive.

TRAILER Trailer Label Mode (0 or 1)

If this mode is enabled, then writing records past the early warning mark on the tape is allowed. The first write operation to detect EOM returns 0. This write operation will not complete successfully. All subsequent write operations are allowed to continue despite the check conditions that result from EOM. When the end of the physical volume is reached, EIO is returned.

An application using the trailer label processing options should stop normal data writing when LEOM (Logic End of Medium) is reached, and perform end of volume processing. Such processing typically consists of writing a final data record, a

filemark, a ″trailing″ tape label, and finally two more filemarks indicating the end of data (EOD).

WRITEPROTECT Write Protect Mode

This configuration parameter returns the current write-protection status of the mounted cartridge. The following values are recognized:

- NO_PROTECT

  The tape is not physically or logically write-protected. Operations that alter the contents of the media are permitted.
- PHYS_PROTECT

  The tape is physically write-protected. The write-protect switch on the tape cartridge is in the protect position. This mode can only be queried. It cannot be altered through device driver functions.

ACFMODE Automatic Cartridge Facility Mode. This mode is not supported for the Ultrium devices.

SCALING Capacity Scaling

This configuration parameter is not supported for Ultrium devices.

SILI Suppress Illegal Length Indication

If this mode is enabled, and a larger block of data is requested than is actually read from the tape block, the tape device will suppress raising a check condition. This eliminates error processing normally performed by the device driver and results in improved read performance for some situations.

An example of the STIOC_GET_PARM command is:

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

parm_data_t parm_data;
parm_data.type = type;

if (!(ioctl (dev_fd, STIOC_GET_PARM, &parm_data))) {
  printf ("The STIOC_GET_PARM ioctl succeeded.\n");
  printf ("\nThe parameter data is:\n");
  dump_bytes ((char *)&parm_data.value, sizeof (int));
}
else {
  perror ("The STIOC_GET_PARM ioctl failed");
  scsi_request_sense ();
}
```

STIOC_SET_PARM

This command sets the current value of the working parameter for the specified tape drive. This command is used in conjunction with the STIOC_GET_PARM command.

The ATDD driver ships with default settings for all configuration parameters. Changing the working parameters dynamically through this STIOC_SET_PARM

command affects the tape drive only during the current open session. The working parameters will revert back to the defaults when the tape drive is closed and reopened.

To change the default configuration settings, see Chapter 19 in the *IBM SCSI Tape Drive, Medium Changer, and Library Device Drivers: Installation and User's Guide*.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar type;                    /* type of parameter to get or set */
  uint value;                    /* current or new value of parameter */
} parm_data_t;
```

The *value* field specifies the new value of the specified parameter, within the ranges indicated below for the specific *type*.

The *type* field, which the caller fills out, should be set to one of the following values:

BLOCKSIZE Block Size (0–16777215)

A value of zero indicates variable block size. The IBM Ultrium devices support about a 16 MB maximum block size.

COMPRESSION Compression Mode(0 or 1)

If this mode is enabled, data is compressed by the tape device before storing it on tape.

BUFFERING Buffering Mode (0 or 1)

If this mode is enabled, data is stored in hardware buffers in the tape device and not immediately committed to tape, thus increasing data throughput performance.

IMMEDIATE Immediate Mode (0 or 1)

If this mode is enabled, then a rewind command returns with the status before the completion of the physical rewind operation by the tape drive.

TRAILER Trailer Label Mode (0 or 1)

If this mode is enabled, then writing records past the early warning mark on the tape is allowed. The first write operation to detect EOM returns ENOSPC. This write operation will not complete successfully. All subsequent write operations are allowed to continue despite the check conditions that result from EOM. When the end of the physical volume is reached, EIO is returned.

An application using the trailer label processing option should stop normal data writing when LEOM (Logic End of Medium) is reached, and perform end of volume processing. Such processing typically consists of writing a final data record, a filemark, a ″trailing″ tape label, and finally two more filemarks to indicate end of data (EOD).

WRITEPROTECT Write Protect Mode

This configuration parameter establishes the current write-protection status of the mounted cartridge. The following values are recognized:

- NO_PROTECT

  The tape is not physically or logically write-protected. Operations that alter the contents of the media are permitted.
- PHYS_PROTECT

  The tape is physically write-protected. The write-protect switch on the tape cartridge is in the protect position. This mode cannot be altered through device driver functions.

ACFMODE Automatic Cartridge Facility Mode

This configuration parameter is read-only. This mode is not supported for Ultrium devices.

SCALING Capacity Scaling

This configuration parameter is not supported for Ultrium devices.

SILI Suppress Illegal Length Indication

If this mode is enabled and a larger block of data is requested than is actually read from the tape block, the tape device will suppress raising a check condition. This eliminates error processing normally performed by the device driver and results in improved read performance for some situations.

An example of the STIOC_SET_PARM command is:

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

parm_data_t parm_data;
parm_data.type = type;
parm_data.value = value;

if (!(ioctl (dev_fd, STIOC_SET_PARM, &parm_data))) {
  printf ("The STIOC_SET_PARM ioctl succeeded.\n");
}
else {
  perror ("The STIOC_SET_PARM ioctl failed");
  scsi_request_sense ();
}
```

STIOC_DISPLAY_MSG

This command displays and manipulates one or two messages on the tape drive operator panel. This command is not supported for Ultrium devices.

STIOC_SYNC_BUFFER

This command immediately flushes the drive buffers to the tape (commits the data to the media).

No data structure is required for this command.

An example of the STIOC_SYNC_BUFFER command is:

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

if (!(ioctl (dev_fd, STIOC_SYNC_BUFFER, 0))) {
  printf ("The STIOC_SYNC_BUFFER ioctl succeeded.\n");
}
else {
  perror ("The STIOC_SYNC_BUFFER ioctl failed");
  scsi_request_sense ();
}
```

STIOC_ REPORT_ DENSITY_ SUPPORT

This *ioctl* command issues the SCSI Report Density Support command to the tape device and returns either all supported densities or supported densities for the currently mounted media. The media field specifies which type of report is requested. The count field in *rpt_dens_sup_t* is returned by the device driver and indicates how many density reports in the reports array field were returned.

The following data structures are filled out and returned by the driver:

```
#define ALL_MEDIA_DENSITY              0
#define CURRENT_MEDIA_DENSITY          1

typedef struct {
      uchar  pri_density_code;          /* primary density code  */
      uchar  sec_density_code;          /* secondary density code  */
      uchar  wrtok          : 1,        /* write ok, device can write this format */
             dup            : 1,        /* zero if density only reported once  */
             deflt          : 1,        /* current density is default format */
             res_1          : 5;        /* reserved  */
      uchar  reserved1[2];              /* reserved  */
      uchar  bits_per_mm[3];            /* bits per mm */
      uchar  media_width[2];            /* media width in millimeters  */
      uchar  tracks[2];                 /* tracks  */
      uchar  capacity[4];               /* capacity in megabytes */
      char   assigning_org[8];          /* assigning organization in ASCII */
      char   density_name[8];           /* density name in ASCII */
      char   description[20];           /* description in ASCII */
 } density_report_t;

typedef struct {
       uchar  media;                    /* report all or current media as defined above */
       uchar  count;                    /* number of density reports returned in array  */
       density_report_t reports[MAX_DENSITY_REPORTS];
}rpt_dens_sup_t;
```

An example of the STIOC_REPORT_DENSITY_SUPPORT command is:

```
 #include <sys/st.h>
int  i, width, media_tracks;
long bits, media_capacity;
rpt_dens_sup_t     density;
density_report_t   report;

density.media = ALL_MEDIA_DENSITY;
printf("\nIssuing Report Density Support for ALL media... \n");

  if (!(rc = ioctl(dev_fd, STIOC_REPORT_DENSITY_SUPPORT, &density))) {
     printf("The STIOC_REPORT_DENSITY_SUPPORT ioctl succeeded.\n");
     printf( "Total number of densities reported: %d \n", density.count);
     for (i=0; i<density.count;i++) {
        printf("\n");
        printf("Density Name.................. %0.8s \n", density.reports[i].density_name);
        printf ("Assigning Organization....... %0.8s \n", density.reports[i].assigning_org) ;
        printf ("Description.................. %0.20s \n", density.reports[i].description);
        printf ("Primary Density Code......... %02X \n", density.reports[i].pri_density_code);
        printf ("Secondary Density Code....... %02X \n", density.reports[i].sec_density_code);
```

```
      if (density.reports[i].wrtok)
            printf ("Write OK..................... Yes \n");
      else
            printf ("Write OK..................... No \n");
      if (density.reports[i].dup)
            printf ("Duplicate.................... Yes \n");
      else
            printf ("Duplicate.................... No \n");
      if (density.reports[i].deflt)
            printf ("Default...................... Yes \n");
      else
            printf ("Default...................... No \n");
      bits =(density.reports[i].bits_per_mm[0]<<16)+(density.reports[i].bits_per_mm[1]<<8)
              +density.reports[i].bits_per_mm[2];
      printf("Bits per MM.................. %ld \n", bits);
      width = ((density.reports[i].media_width[0] << 8) + density.reports[i].media_width[1]);
      printf ("Media Width (millimeters).... %d \n", width);
      media_tracks = ((density.reports[i].tracks[0] << 8) + density.reports[i].tracks[1]);
      printf ("Tracks....................... %d \n", media_tracks);
      media_capacity =(density.reports[i].capacity[0]<<24)+(density.reports[i].capacity[1]<<16)
             +(density.reports[i].capacity[2] << 8) + (density.reports[i].capacity[3]);
      printf ("Capacity (megabytes)......... %ld \n", media_capacity);
      printf ("\n--- more ---");
      getchar ();
 }
}
else {
    perror ("The STIOC_REPORT_DENSITY_SUPPORT ioctl for all media failed");
    scsi_request_sense ();
}

density.media = CURRENT_MEDIA_DENSITY;
printf ("\nIssuing Report Density Support for CURRENT media... \n");

if (!(rc = ioctl(dev_fd, STIOC_REPORT_DENSITY_SUPPORT, &density))) {
   printf ("The STIOC_REPORT_DENSITY_SUPPORT ioctl succeeded.\n");
   printf ( "Total number of densities reported: %d \n", density.count);
   for (i=0;i<density.count;i++) {
        printf ("Density Name................ %0.8s \n", density.reports[i].density_name);
        printf("Assigning Organization....... %0.8s \n", density.reports[i].assigning_org);
        printf("Description.................. %0.20s \n", density.reports[i].description);
        printf("Primary Density Code......... %02X \n", density.reports[i].pri_density_code);
        printf("Secondary Density Code....... %02X \n", density.reports[i].sec_density_code);
        if (density.reports[i].wrtok)
              printf("Write OK..................... Yes \n");
        else
              printf("Write OK..................... No \n");
        if (density.reports[i].dup)
              printf("Duplicate.................... Yes \n");
        else
              printf("Duplicate.................... No \n");
        if (density.reports[i].deflt)
              printf("Default...................... Yes \n");
        else
              printf("Default...................... No \n");
        bits =(density.reports[i].bits_per_mm[0]<<16)+(density.reports[i].bits_per_mm[1]<<8)
                +density.reports[i].bits_per_mm[2];
        printf("Bits per MM.................. %ld \n", bits);
        width = ((density.reports[i].media_width[0] << 8) + density.reports[i].media_width[1]);
        printf("Media Width (millimeters).... %d \n", width);
        media_tracks = ((density.reports[i].tracks[0] << 8) + density.reports[i].tracks[1]);
        printf("Tracks....................... %d \n", media_tracks);
        media_capacity =(density.reports[i].capacity[0]<<24)+(density.reports[i].capacity
              [1]<<16)+(density.reports[i].capacity[2] << 8) + (density.reports
              [i].capacity[3]);
        printf("Capacity (megabytes)......... %ld \n", media_capacity);
        printf("\n--- more ---");
```

```
        getchar ();
        }
    }
else {
        perror("The STIOC_REPORT_DENSITY_SUPPORT ioctl for current media failed");
        scsi_request_sense ();
}
```

Base Operating System Tape Drive IOCTL Operations

The set of native magnetic tape *ioctl* commands that is available through the HP-UX base operating system is provided for compatibility with existing applications.

The following commands are supported:

MTIOCTOP Perform the magnetic tape drive operations.

MTIOCGET Return the status information about the tape drive.

These commands and associated data structures are defined in the *mtio.h* system header file in the */usr/include/sys* directory. Any application program that issues these commands must include this header file.

MTIOCTOP

This command performs the magnetic tape drive operations. It is defined in the */usr/include/sys/mtio.h* header file. The MTIOCTOP commands use the MT opcodes and the data structure defined in the *mtio.h* system header file.

Note: To compile the application code with the *mtio.h* and *st.h* on HP-UX 10.20, the patch *PHKL_22286* or later is requested.

For all *space* operations, the resulting tape position is at the end-of-tape side of the record or filemark for forward movement and at the beginning-of-tape side of the record or filemark for backward movement.

The following data structure is filled out and supplied by the caller:

```
/*from mtio.h */
struct mtop {
short mt_op;    /*operations (defined below)*/
daddr_t mt_count;   /*how many to perform */
};
```

The mt_op field is set to one of the following:

MTWEOF
Write *mt_count* filemarks

MTFSF
Space forward *mt_count* filemarks.

MTBSF
Space backward *mt_count* filemarks. Upon completion, the tape is positioned at the beginning-of-tape side of the requested filemark.

MTFSR
Space forward the *mt_count* number of records.

MTBSR
Space backward the *mt_count* number of records.

MTREW
Rewind the tape. The *mt_count* parameter does not apply.

MTOFFL
Rewind and unload the tape. The *mt_count* parameter does not apply.

MTNOP
No tape operation is performed. The status is determined by issuing the Test Unit Ready command. The *mt_count* parameter does not apply.

MTEOD
Space forward to the end of the data. The *mt_count* parameter does not apply.

MTRES
Reserve the tape drive. The *mt_count* parameter does not apply.

MTREL
Release the tape drive. The *mt_count* parameter does not apply.

MTERASE
Erase the tape media. The *mt_count* parameter does not apply.

MTIOCGET

This command returns the status information about the tape drive. It is identical to the STIOC_GET_DEVICE_STATUS *ioctl* command defined in the */usr/include/sys/st.h* header file. The STIOC_GET_DEVICE_STATUS and MTIOCGET commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The two *ioctl* commands are interchangeable. See “STIOC_GET_DEVICE_STATUS” on page 76.

Service Aid IOCTL Operations

A set of service aid *ioctl* commands gives applications access to serviceability operations for IBM tape subsystems.

The following commands are supported:

STIOC_DEVICE_SN	Query the serial number of the device.
STIOC_FORCE_DUMP	Force the device to perform a diagnostic dump.
STIOC_STORE_DUMP	Force the device to write the diagnostic dump to the currently mounted tape cartridge.
STIOC_READ_BUFFER	Read data from the specified device buffer.
STIOC_WRITE_BUFFER	Write data to the specified device buffer.

These commands and associated data structures are defined in the *svc.h* header file in the */usr/include/sys* directory, which is installed with the ATDD package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the device driver and installed in the */opt/tapeutil* directory with the *tapeutil* package. This source code demonstrates the use of these *ioctl* commands.

STIOC_DEVICE_SN

This command queries the serial number of the device.

The following data structure is filled out and returned by the driver:

```
typedef uint device_sn_t;
```

An example of the STIOC_DEVICE_SN command is:

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

device_sn_t device_sn;

if (!(ioctl (dev_fd, STIOC_DEVICE_SN, &device_sn))) {
  printf ("Tape device %s serial number:%x\n", dev_name, device_sn);
}
else {
  perror ("Failure obtaining tape device serial number");
  scsi_request_sense ();
}
```

STIOC_FORCE_DUMP

This command forces the device to perform a diagnostic dump.

No data structure is required for this command.

An example of the STIOC_FORCE_DUMP command is:

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

if (!(ioctl (dev_fd, STIOC_FORCE_DUMP, 0))) {
  printf ("Dump completed successfully.\n");
}
else {
  perror ("Failure performing device dump");
  scsi_request_sense ();
}
```

STIOC_STORE_DUMP

This command forces the device to write the diagnostic dump to the currently mounted tape cartridge.

No data structure is required for this command.

An example of the STIOC_STORE_DUMP command is:

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

if (!(ioctl (dev_fd, STIOC_STORE_DUMP, 0))) {
  printf ("Dump store on tape successfully.\n");
}
else {
  perror ("Failure storing dump on tape");
  scsi_request_sense ();
}
```

STIOC_READ_BUFFER

This command reads data from the specified device buffer.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar mode;                          /* transfer mode */
  uchar id;                            /* device buffer id */
  uint offset;                         /* buffer offset */
  uint size;                           /* byte count */
  uchar *buffer;                       /* data buffer */
} buffer_io_t;
```

The *mode* field should be set to one of the following values:

VEND_MODE	Vendor specific mode
DSCR_MODE	Descriptor mode
DNLD_MODE	Download mode

The *id* field should be set to one of the following values:

ERROR_ID	Diagnostic dump buffer
UCODE_ID	Microcode buffer

An example of the STIOC_READ_BUFFER command is:

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

buffer_io_t buffer_io;

if (!(ioctl (dev_fd, STIOC_READ_BUFFER, &buffer_io))) {
  printf ("Buffer read successfully.\n");
}
else {
  perror ("Failure reading buffer");
  scsi_request_sense ();
}
```

STIOC_WRITE_BUFFER

This command writes data to the specified device buffer.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar mode;                          /* transfer mode */
  uchar id;                            /* device buffer id */
  uint offset;                         /* buffer offset */
  uint size;                           /* byte count */
  uchar *buffer;                       /* data buffer */
} buffer_io_t;
```

The *mode* field should be set to one of the following values:

VEND_MODE	Vendor specific mode
DSCR_MODE	Descriptor mode
DNLD_MODE	Download mode

The *id* field should be set to one of the following values:

ERROR_ID	Diagnostic dump buffer
UCODE_ID	Microcode buffer

An example of the STIOC_WRITE_BUFFER command is:

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

buffer_io_t buffer_io;

if (!(ioctl (dev_fd, STIOC_WRITE_BUFFER, &buffer_io))) {
  printf ("Buffer written successfully.\n");
}
else {
  perror ("Failure writing buffer");
  scsi_request_sense ();
}
```

Part 3. LinuxTape and Medium Changer Device Driver

Chapter 10. Software Interface

IBM supplies a SCSI tape drive and medium changer device driver for the Linux platform called *IBMtape*. The IBMtape driver operates the IBM Ultrium 3580, 3581, 3583 and 3584 devices. IBMtape supports the following Linux-defined entry points:

- *open*
- *close*
- *read*
- *write*
- *ioctl*

open

This entry point is driven by the *open* system call.

The programmer can access *IBMtape* devices with one of three access modes: *write only*, *read only*, or *read and write*.

IBMtape also supports the *append open* flag. When *open* function is called with the append flag set to TRUE, *IBMtape* will attempt to seek two consecutive filemarks and place the initial tape position between them. *Open append* fails [*errno* EIO] if no tape is loaded or there are not two consecutive filemarks on the loaded tape. *Open append* does not automatically imply write access; therefore, an access mode must accompany the *append* flag during the *open* operation.

The *open* function will attempt a SCSI *reserve* command to the target device. If the *reserve* command fails, then *open* fails and *errno* EBUSY is returned.

close

This entry point is driven explicitly by the *close* system call and implicitly by the operating system at application program termination.

For non-rewinding devices, such as */dev/IBMtape0n*, if the last command before the *close* function was a successful *write*, IBMtape writes two consecutive filemarks marking the end of data. It then sets the tape position between the two consecutive filemarks. If the last command before the *close* function successfully wrote one filemark, then one additional filemark is written marking the end of data and the tape position is set between the two consecutive filemarks.

For rewind devices, such as */dev/IBMtape0*, if the last command before the *close* function was a successful *write*, IBMtape writes two consecutive filemarks marking the end of data and issues a *rewind* command. If the last command before the *close* function successfully wrote one filemark, one additional filemark is written marking the end of data, and the *rewind* command is issued.

If an SIOC_RESERVE *ioctl* has been issued from an application before *close*, the close function does not release the device; otherwise, it issues the SCSI release command. In both situations, the *close* function attempts to deallocate all resources allocated for the device. If, for some reason, IBMtape is not able to *close*, an error code is returned.

Note: The return code for *close* should always be checked. If *close* is unsuccessful, retry is recommended.

read

This entry point is driven by the *read* system call. The *read* operation can be performed when there is a tape loaded in the device.

IBMtape supports two modes of *read* operation. If the *read_past_filemark* flag is set to TRUE (by using STIOCSETP *ioctl*), then when a *read* operation encounters a filemark, it returns the number of bytes read before encountering the filemark, and sets the tape position after the filemark. If the *read_past_filemark* flag is set to FALSE (by default or by using STIOCSETP *ioctl*), then when a *read* operation encounters a filemark, if data was read, the *read* function returns the number of bytes read, and positions the tape before the filemark. If no data was read, *read* returns 0 bytes and positions the tape after the filemark.

If *read* function reaches end of the data on the tape, then EIO is returned and ASC, ASCQ keys (obtained by request sense *ioctl*s) would indicate end of data. IBMtape also conforms to all SCSI standard read operation rules (such as fixed block versus variable block).

write

This entry point is driven by the *write* system call. The write operation can be performed when there is a tape loaded in the device.

IBMtape supports early warning processing. When the *trailer_labels* flag is set to TRUE (by default or by using the STIOCSETP *ioctl* call), IBMtape fails with *errno* ENOSPACE only when a *write* operation first encounters the early warning zone for end of tape. After the ENOSPACE error code is returned, IBMtape will suppress all warning messages from the device generated by subsequent *write* commands, effectively allowing *write* and *write filemark* commands in the early warning zone. When physical end of tape is reached, error code EIO will be returned and the ASC, ASCQ keys (obtained by the request sense IOCTL) will confirm the end of physical medium condition. When the *trail_labels* flag is set to FALSE (by using the STIOCSETP *ioctl* call), IBMtape will return ENOSPACE when attempting any *write* command in the early warning zone.

ioctl

IBMtape also conforms to all SCSI standard *ioctl* operation rules (such as fixed block versus variable block).

This entry point provides a set of tape and SCSI specific functions. It allows Linux applications to access and control the features and attributes of the tape device programmatically.

Medium Changer Devices

IBMtape supports the following Linux entry points for the medium changer devices:

- *open*
- *close*
- *ioctl*

open

This entry point is driven by the *open* system call. The *open* function will attempt a SCSI reserve command to the target device. If the *reserve* command fails, then open fails with *errno* EBUSY.

close

This entry point is driven explicitly by the *close* system call and implicitly by the operating system at program termination. If an SIOC_RESERVE *ioctl* has been issued from an application before *close*, the *close* function does not release the device; otherwise, it issues the SCSI release command. In both situations, the *close* function attempts to deallocate all resources allocated for the device. If, for some reason, IBMtape is not able to *close*, an error code is returned.

ioctl

This entry point provides a set of medium changer and SCSI-specific functions. It allows Linux applications to access and control the features and attributes of the tape system's robotic device programmatically.

Chapter 11. Special Files

A setup script called *IBMtapeconfig* (IBM special file utility) is supplied with IBMtape.

IBMtapeconfig generates special files for devices that are currently claimed by the IBMtape driver by reading the device information contained in */proc/scsi/IBMtape* and */proc/scsi/IBMchanger*. The following table shows these special files.

Table 3. Special Files Supplied with IBMtapeconfig

Special File Example	Description
/dev/IBMtape3 /dev/IBMtape3n	SCSI tape drive special files. Two special files are created for each device. The trailing ″n″ marks the ″No-Rewind-on-Close″ special file. A number between 0 and 63 will be assigned: 0 for the first tape drive encountered, 1 for the next, and so on.
/dev/IBMchanger6	SCSI Medium Changer special file. A number between 0 and 127 will be assigned: 0 for the first changer encountered, 1 for the next, and so on.

Chapter 12. General IOCTL Operations

This chapter describes the set of *ioctl* commands which provide control and access to the SCSI tape and medium changer devices. These commands are available for all of the tape and medium changer devices. They can be issued to any one of IBMtape special files.

Overview

The following *ioctl* commands are supported:

SIOC_INQUIRY	Return the inquiry data.
SIOC_REQSENSE	Return the sense data.
SIOC_RESERVE	Reserve the device.
SIOC_RELEASE	Release the device.
SIOC_TEST_UNIT_READY	Issue the SCSI Test Unit Ready command.
SIOC_LOG_SENSE_PAGE	Return the log sense data.
SIOC_MODE_SENSE_PAGE	Return the mode sense data.
SIOC_INQUIRY_PAGE	Return the inquiry data for a specific page.
SIOC_PASS_THROUGH	Pass through custom built SCSI commands

These *ioctl* commands and their associated structures are defined in the *IBM_tape.h* header file, which can usually be found in */usr/include/system* after installing IBMtape. The *IBM_tape.h* header file should be included in the corresponding C programs that call functions provided by IBMtape.

A sample source program, *IBMtapeutil.c*, demonstrating the use of these *ioctl* commands, is provided with the IBMtapeutil.x.x.x.tar package (x.x.x is the release version of *IBMtapeutil*).

All *ioctl* commands require a file descriptor of an open file. Use the *open* command to open a device and obtain a valid file descriptor.

SIOC_INQUIRY

This *ioctl* command collects the inquiry data from the device. The data structure is:

```
struct inquiry_data {
   uint  qual   :3,   /* peripheral qualifier              */
         type   :5;   /* device type                       */
   uint  rm     :1,   /* removable medium                  */
         mod    :7;   /* device type modifier              */
   uint  iso    :2,   /* ISO version                       */
         ecma   :3,   /* EMCA version                      */
         ansi   :3;   /* ANSI version                      */
   uint  aenc   :1,   /* asynchronous event notification   */
         trmiop :1,   /* terminate I/O process message     */
                :2,   /* reserved                          */
         rdf    :4;   /* response data format              */
   unchar len;        /* additional length                 */
   unchar resvd1;     /* reserved                          */
         uint   :4,   /* reserved                          */
         mchngr :1,   /* medium changer mode (SCSI-3 only) */
                :3;   /* reserved                          */
   uint  reladr :1,   /* relative addressing               */
```

```
        wbus32 :1,   /* 32-bit wide data transfers        */
        wbus16 :1,   /* 16-bit wide data transfers        */
        sync   :1,   /* synchronous data transfers        */
        linked :1,   /* linked commands                   */
               :1,   /* reserved                          */
        cmdque :1,   /* command queueing                  */
        sftre  :1;   /* soft reset                        */
   unchar vid[8];     /* vendor ID                         */
   unchar pid[16];    /* product ID                        */
   unchar revision[4];/* product revision level           */
   unchar vendor1[20];/* vendor specific                   */
   unchar resvd2[40]; /* reserve                           */
   unchar vendor2[31];/* vendor specific (padded to 127)   */
};
```

An example of the SIOC_INQUIRY command is:

```
#include <sys/IBM_tape.h>
char vid[9];
char pid[17];
char revision[5];
struct inquiry_data inqdata;
printf("Issuing inquiry...\n");
memset(&inquiry_data, 0, sizeof(struct inquiry_data));
if (!ioctl (fd, SIOC_INQUIRY, &inqdata)) {
   printf ("The SIOC_INQUIRY ioctl succeeded\n");
   printf ("\nThe inquiry data is:\n");
   /*-
    * Just a dump byte won't work because of the compiler
    * bit field mapping
   -*/
   /* print out structure data field */
 printf("\nInquiry Data:\n");
 printf("Peripheral Qualifer-----------------0x%02x\n", inqdata.qual);
 printf("Peripheral Device Type--------------0x%02x\n", inqdata.type);
 printf("Removal Medium Bit------------------%d\n", inqdata.rm);
 printf("Device Type Modifier----------------0x%02x\n", inqdata.mod);
 printf("ISO version-------------------------0x%02x\n", inqdata.iso);
 printf("ECMA version------------------------0x%02x\n", inqdata.ecma);
 printf("ANSI version------------------------0x%02x\n", inqdata.ansi);
 printf("Asynchronous Event Notification Bit-%d\n", inqdata.aenc);
 printf("Terminate I/O Process Message Bit---%d\n", inqdata.trmiop);
 printf("Response Data Format----------------0x%02x\n", inqdata.rdf);
 printf("Additional Length-------------------0x%02x\n", inqdata.len);
 printf("Medium Changer Mode-----------------0x%02x\n", inqdata.mchngr);
 printf("Relative Addressing Bit-------------%d\n", inqdata.reladr);
 printf("32 Bit Wide Data Transfers Bit------%d\n", inqdata.wbus32);
 printf("16 Bit Wide Data Transfers Bit------%d\n", inqdata.wbus16);
 printf("Synchronous Data Transfers Bit------%d\n", inqdata.sync);
 printf("Linked Commands Bit-----------------%d\n", inqdata.linked);
 printf("Command Queueing Bit----------------%d\n", inqdata.cmdque);
 printf("Soft Reset Bit----------------------%d\n", inqdata.sftre);

 strncpy(vid, inqdata.vid, 8);
 vid[8] = '\0';
 strncpy(pid, inqdata.pid, 16);
 pid[16] = '\0';
 strncpy(revision, inqdata.revision, 4);
 revision[4] = '\0';

 printf("Vendor ID----------------------------%s\n", vid);
 printf("Product ID---------------------------%s\n", pid);
 printf("Product Revision Level---------------%s\n", revision);

 dump_bytes(inqdata.vendor1, 20, "vendor1");
 dump_bytes(inqdata.vendor2, 31, "vendor2");
}
```

```
else {
   perror ("The SIOC_INQUIRY ioctl failed");
   sioc_request_sense();
}
```

SIOC_REQSENSE

This *ioctl* command returns the device sense data. If the last command resulted in an error, then the sense data is returned for the error. Otherwise, a new sense command is issued to the device. The data structure is:

```
struct request_sense {
   uint  valid    :1, /* sense data is valid             */
         err_code :7; /* error code                      */
   uchar segnum;      /* segment number                  */
   uint  fm       :1, /* filemark detected               */
         eom      :1, /* end of medium                   */
         ili      :1, /* incorrect length indicator      */
         resvd1   :1, /* reserved                        */
         key      :4; /* sense key                       */
   int info;          /* information bytes               */
   unchar addlen;     /* additional sense length         */
   uint  cmdinfo;     /* command specific information    */
   uchar asc;         /* additional sense code           */
   uchar ascq;        /* additional sense code qualifier */
   uchar fru;         /* field replaceable unit code     */
   uint  sksv     :1, /* sense key specific valid        */
         cd       :1, /* control/data                    */
         resvd2   :2, /* reserved                        */
         bpv      :1, /* bit pointer valid               */
         sim      :3; /* system information message      */
   uchar field[2];    /* field pointer                   */
   uchar vendor[109]; /* vendor specific (padded to 127) */
};
```

An example of the SIOC_REQSENSE command is:

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

struct request_sense sense_data;
int rc;
printf("Issuing request sense...\n");
memset(&sense_data, 0, sizeof(struct request_sense));
rc = ioctl(fd, SIOC_REQSENSE, &sense_data);
if (rc == 0)
{
 if(!sense_data.err_code)
  printf("No valid sense data returned.\n");
 else
 {
  /*print out data fields */
  printf("Information Field Valid Bit-----%d \n",sense_data.valid);
  printf("Error Code----------------------0x%02x \n",sense_data.err_code);
  printf("Segment Number------------------0x%02x \n",sense_data.segnum);
  printf("filemark Detected Bit----------%d \n",sense_data.fm);
  printf("End Of Medium Bit--------------%d \n",sense_data.eom);
  printf("Illegal Length Indicator Bit----%d \n",sense_data.ili);
  printf("Sense Key-----------------------0x%02x \n",sense_data.key);
  if(sense_data.valid)
   printf("Information Bytes-------------0x%02x 0x%02x 0x%02x 0x%02x \n",
    sense_data.info >>24,sense_data.info >>16,
    sense_data.info >>8,sense_data.info &0xFF);
  printf("Additional Sense Length---------0x%02x \n",sense_data.addlen);
  printf("Command Specific Information----0x%02x 0x%02x 0x%02x 0x%02x \n",
    sense_data.cmdinfo >>24,sense_data.cmdinfo >>16,
    sense_data.cmdinfo >>8,sense_data.cmdinfo &0xFF);
  printf("Additional Sense Code-----------0x%02x \n",sense_data.asc);
```

```
  printf("Additional Sense Code Qualifier-0x%02x \n",sense_data.ascq);
  printf("Field Replaceable Unit Code-----0x%02x \n",sense_data.fru);
  printf("Sense Key Specific Valid Bit----%d \n",sense_data.sksv);
  if(sense_data.sksv)
   {
    printf("Command Data Block Bit--%d \n",sense_data.cd);
    printf("Bit Pointer Valid Bit---%d \n",sense_data.bpv);
    if(sense_data.bpv)
     printf("System Information Message-0x%02x \n",sense_data.sim);
    printf("Field Pointer----------------0x%02x%02x \n",
      sense_data.field [0 ],sense_data.field [1 ]);
   }
  dump_bytes(sense_data.vendor,109,"Vendor");
 }
}
return rc;
```

SIOC_RESERVE

This *ioctl* command explicitly reserves the device and prevents it from being released after a *close* operation. The device is not released until an SIOC_RELEASE command is issued. The *ioctl* command can be used for applications that require multiple open and close processing in a host-sharing environment.

There are no arguments for the SIOC_RESERVE command.

An example of the SIOC_RESERVE command is:

```
#include  <sys/IBM_tape.h>
if (!ioctl (fd, SIOC_RESERVE, NULL)) {
   printf ("The SIOC_RESERVE ioctl succeeded\n");
}
else {
   perror ("The SIOC_RESERVE ioctl failed");
   sioc_request_sense(); /* see IBMtapeutil for the source */
}
```

SIOC_RELEASE

This *ioctl* command explicitly releases the device and allows other hosts to access it. The *ioctl* command is used with the SIOC_RESERVE *ioctl* command for applications that require multiple open and close processing in a host-sharing environment.

There are no arguments for this *ioctl* command.

An example of the SIOC_RELEASE command is:

```
#include  <sys/IBM_tape.h>
if (!ioctl (fd, SIOC_RELEASE, NULL)) {
   printf ("The SIOC_RELEASE ioctl succeeded\n");
}
else {
   perror ("The SIOC_RELEASE ioctl failed");
   sioc_request_sense(); /* see IBMtapeutil for the source */
}
```

SIOC_TEST_UNIT_READY

This *ioctl* command issues the SCSI Test Unit Ready command to the device.

There are no arguments for this *ioctl* command.

An example of the SIOC_TEST_UNIT_READY command is:

```
#include <sys/IBM_tape.h>
if (!ioctl (fd, SIOC_TEST_UNIT_READY, NULL)) {
   printf ("The SIOC_TEST_UNIT_READY ioctl succeeded\n");
}
else {
   perror ("The SIOC_TEST_UNIT_READY ioctl failed");
   sioc_request_sense();
}
```

SIOC_LOG_SENSE_PAGE

This *ioctl* command returns a log page from the device. The desired page is selected by specifying the p*age_code* in the *log_sense_page* structure. Optionally, a specific *parm pointer*, also known as a *parm code*, and the number of parameter bytes can be specified with the command.

To obtain the entire log page, the *len* and *parm_pointer* fields should be set to zero. To obtain the entire log page starting at a specific parameter code, set the *parm_pointer* field to the desired code and the *len* field to zero. To obtain a specific number of parameter bytes, set the *parm_pointer* field to the desired code and set the *len* field to the number of parameter bytes plus the size of the log page header (4 bytes). The first 4 bytes of returned data will always be the log page header. See the appropriate device manual to determine the supported log pages and content.

The data structure is:

```
struct log_sense_page {
   char          page_code;
   unsigned short len;
   unsigned short parm_pointer;
   char          data[LOGSENSEPAGE];
};
```

An example of the SIOC_LOG_SENSE_PAGE command is:

```
#include <sys/IBM_tape.h>
struct log_sense_page log_page;
int temp;
/* get log page 0, list of log pages */
log_page.page_code = 0x00;
log_page.len = 0;
log_page.parm_pointer = 0;
if (!ioctl (fd, SIOC_LOG_SENSE_PAGE, &log_page)) {
   printf ("The SIOC_LOG_SENSE_PAGE ioctl succeeded\n");
   dump_bytes(log_page.data, LOGSENSEPAGE);
}
else {
   perror ("The SIOC_LOG_SENSE_PAGE ioctl failed");
   sioc_request_sense();
}
/* get fraction of volume traversed */
log_page.page_code = 0x38;
log_page.len = 0;
log_page.parm_pointer = 0x000F;
if (!ioctl (fd, SIOC_LOG_SENSE_PAGE, &log_page)) {
   temp = log_page.data[sizeof(log_page_header) + 4)];
```

```
   printf ("The SIOC_LOG_SENSE_PAGE ioctl succeeded\n");
   printf ("Fractional Part of Volume Traversed %x\n",temp);
}
else {
   perror ("The SIOC_LOG_SENSE_PAGE ioctl failed");
   sioc_request_sense();
}
```

SIOC_MODE_SENSE_PAGE

This *ioctl* command returns a mode sense page from the device. The desired page is selected by specifying the page_code in the mode_sense_page structure. See the appropriate device manual to determine the supported mode pages and content.

The data structure is:

```
struct mode_sense_page {
   char page_code;
   char data[MODESENSEPAGE];
};
```

An example of the SIOC_MODE_SENSE_PAGE command is:

```
#include <sys/IBM_tape.h>
struct mode_sense_page mode_page;
/* get medium changer mode */
mode_page.page_code = 0x20;
if (!ioctl (fd, SIOC_MODE_SENSE_PAGE, &mode_page)) {
   printf ("The SIOC_MODE_SENSE_PAGE ioctl succeeded\n");
   if (mode_page.data[2] == 0x02)
      printf ("The library is in Random mode.\n");
   else if (mode_page.data[2] == 0x05)
      printf ("The library is in Automatic (Sequential) mode.\n");
}
else {
   perror ("The SIOC_MODE_SENSE_PAGE ioctl failed");
   sioc_request_sense();
}
```

SIOC_INQUIRY_PAGE

This *ioctl* command returns an inquiry page from the device. The desired page is selected by specifying the page_code in the inquiry_page structure. See the appropriate device manual to determine the supported inquiry pages and content.

The data structure is:

```
struct inquiry_page {
   char page_code;
   char data[INQUIRYPAGE];
};
```

An example of the SIOC_INQUIRY_PAGE command is:

```
#include <sys/IBM_tape.h>
struct inquiry_page inq_page;
/* get inquiry page x83 */
inq_page.page_code = 0x83;
if (!ioctl (fd, SIOC_INQUIRY_PAGE, &inq_page)) {
printf ("The SIOC_INQUIRY_PAGE ioctl succeeded\n");
 dump_bytes (inq_data, INQUIRYPAGE);
}
```

```
else {
perror ("The SIOC_INQUIRY_PAGE ioctl failed");
 sioc_request_sense();
}
```

SCSI_PASS_THROUGH

This *ioctl* command passes the built command data block structure along with I/O buffer pointers to the lower SCSI layer. Status codes returned from the lower SCSI layer are returned to the caller, along with sense key, ASC, ASCQ keys. The ASC, ASCQ and sense key are only valid when the *SenseDataValid* field is true.

```
#define SCSI_PASS_THROUGH _IOWR('P',0x01,SCSIPassThrough) /* Pass Through  */
  typedef struct _SCSIPassThrough
  {
    unchar   CDB[12];                   /* Command Data Block */
    unchar   CommandLength;             /* Command Length     */
    unchar * Buffer ;                   /* Command Buffer     */
    ulong    BufferLength;              /* Buffer Length             */
    unchar   DataDirection;             /* Data Transfer Direction   */
    ushort   TimeOut;                   /* Time Out Value            */
    unchar   TargetStatus;              /* Target Status             */
    unchar   MessageStatus;             /* Message from host adapter */
    unchar   HostStatus;                /* Host status               */
    unchar   DriverStatus;              /* Driver status             */
    unchar   SenseDataValid;            /* Sense Data Valid          */
    unchar   ASC;            /* ASC key if the SenseDataValid is True */
    unchar   ASCQ;          /* ASCQ key if the SenseDataValid is True */


     unchar   SenseKey;     /* Sense key if the SenseDataValid is True */
  } SCSIPassThrough, *PSCSIPassThrough;
  #define SCSI_DATA_OUT  1
  #define SCSI_DATA_IN   2
  #define SCSI_DATA_NONE 3
```

SCSI_DATA_OUT indicates sending data out of the initiator (host bus adapter), also known as write.

SCSI_DATA_IN indicates receiving data into the initiator (host bus adapter), also known as read.

SCSI_DATA_NONE indicates no data are transferred.

An example of the SIOC_PASS_THROUGH command is:

```
#include <sys/IBM_tape.h>
SCSIPassThrough PassThrough;
memset(&PassThrough, 0, sizeof(SCSIPassThrough);
/*Issue test unit ready command */
PassThrough.CDB[0] = 0x00;
PassThrough.CommandLength = 6;
PassThrough.DataDirection = SCSI_DATA_NONE;
if (!ioctl (fd, SIOC_PASS_THROUGH, &PassThrough)){
 printf ("The SIOC_PASS_THROUGH ioctl succeeded \n");
 if((PassThrough.TargetStatus == STATUS_SUCCESS)&&
  (PassThrough.MessageStatus == STATUS_SUCCESS)&&
  (PassThrough.HostStatus == STATUS_SUCCESS)&&
  (PassThrough.DriverStatus == STATUS_SUCCESS))
 printf("Test Unit Ready returns success \n");

 else {
  printf("Test Unit Ready failed \n");
  if(PassThrough.SenseDataValid)
   printf("Sense Key %02x,ASC %02x,ASCQ %02x \n",
```

```
        PassThrough.SenseKey,PassThrough.ASC,
        PassThrough.ASCQ);
 }
}
else {
 perror ("The SIOC SIOC_PASS_THROUGH ioctl failed");
 sioc_request_sense();
}
```

Chapter 13. Tape Drive IOCTL Operations

The device driver supports the set of tape *ioctl* commands that is available with the base Linux operating system, as well as a set of expanded tape *ioctl* commands that give applications access to additional features and functions of the tape drives.

Overview

The following *ioctl* commands are supported:

STIOCTOP Perform basic tape operations.

STIOCQRYP Query the tape device, device driver, and media parameters.

STIOCSETP Change the tape device, device driver, and media parameters.

STIOCSYNC Synchronize the tape buffers with the tape.

STIOCQRYPOS Query the tape position and the buffered data.

STIOCSETPOS Set the tape position.

STIOCQRYSENSE Query the sense data from the tape device.

STIOCQRYINQUIRY Return the inquiry data.

STIOC_LOCATE Locate to a certain tape position.

STIOC_READ_POSITION Read the current tape position.

STIOC_RESET_DRIVE Issue a SCSI Send Diagnostic command to reset the tape drive

STIOC_PREVENT_MEDIUM_REMOVAL
Prevent medium removal by an operator.

STIOC_ALLOW_MEDIUM_REMOVAL
Allow medium removal by an operator.

STIOC_REPORT_DENSITY_SUPPORT
Return supported densities from the tape device.

These *ioctl* commands and their associated structures are defined in the *IBM_tape.h* header file, which can usually be found in */usr/include/sys/* directory. This header should be included in the corresponding C program using the *ioctl* commands. A sample *IBMtapeutil.c* program that demonstrates the use of these *ioctl* commands is provided with the device driver.

STIOCTOP

This *ioctl* command performs basic tape operations. The *st_count* variable is used for many of its operations. Normal error recovery applies to these operations. The device driver can issue several tries to complete them. For all forward movement space operations, the tape position finishes on the end-of-tape side of the record or filemark, and on the beginning-of-tape side of the record or filemark for backward movement.

The input data structure is:

```
struct stop {
   short st_op;       /* operations defined below */
   daddr_t st_count;  /* how many of them to do (if applicable) */
};
```

The *st_op* variable is set to one of the following operations:

STOFFL Unload the tape. The *st_count* parameter does not apply.

STREW Rewind the tape. The *st_count* parameter does not apply.

STERASE Erase the entire tape. The *st_count* parameter does not apply.

STRETEN Perform the rewind operation. The tape devices perform the retension operation automatically when needed.

STWEOF Write the *st_count* number of filemarks.

STFSF Space forward the *st_count* number of filemarks.

STRSF Space backward the *st_count* number of filemarks.

STFSR Space forward the *st_count* number of records.

STRSR Space backward the *st_count* number of records.

STTUR Issue the Test Unit Ready command. The *st_count* parameter does not apply.

STLOAD Issue the SCSI Load command. The *st_count* parameter does not apply. The operation of the SCSI Load command varies, depending on the type of device. See the appropriate hardware reference manual.

STSEOD Space forward to the end of the data. The *st_count* parameter does not apply.

STEJECT Unload the tape. The *st_count* parameter does not apply.

STINSRT Issue the SCSI Load command. The *st_count* parameter does not apply.

Note: If zero is used for operations that require the count parameter, then the command is not issued to the device, and the device driver will return a successful completion.

An example of the STIOCTOP command is:

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

struct stop stop;
stop.st_op=STWEOF;

stop.st_count=3;
if (ioctl(tapefd,STIOCTOP,&stop)<0) {
   printf("ioctl failure. errno=%d",errno);
   exit(errno);
}
```

STIOCQRYP or STIOCSETP

The STIOCQRYP *ioctl* command allows the program to query the tape device, the device driver, and the media parameters. The STIOCSETP *ioctl* command allows the program to change the tape device, the device driver, and the media parameters. Before issuing the STIOCSETP command, use the STIOCQRYP command to query and fill the fields of the data structure that you do not want to

change. Then issue the STIOCSETP command to change the selected fields. Changing certain fields (such as *buffered_mode*) impacts performance. If the *buffered_mode* field is FALSE, then each record written to the tape is immediately transferred to the tape. This operation guarantees that each record is on the tape, but impacts performance.

The following parameters returned by the STIOCQRYP *ioctl* command cannot be changed by the STIOCSETP *ioctl* command:

- *hkwrd*

  This parameter is accepted, but ignored.
- *logical_write_protect*

  This parameter sets the type of logical write protection for the tape loaded in the drive.
- *write_protect*

  If the currently mounted tape is write protected, then this field is set to TRUE. Otherwise, it is set to FALSE.
- *min_blksize*

  This parameter is the minimum block size for the device. The driver gets this field by issuing the SCSI Read Block Limits command to the device.
- *max_blksize*

  This parameter is the maximum block size for the device. The driver gets this field by issuing the SCSI Read Block Limits command to the device.
- *retain_reservation*

  This parameter is accepted, but ignored.
- *medium_type*

  This parameter is the media type of the currently loaded tape. Some tape devices support multiple media types and will report different values in this field. See the hardware reference guide for the specific tape device to determine the possible values.
- *capacity_scaling*

  This parameter sets the capacity or logical length of the current tape. By reducing the capacity of the tape, the tape drive can access data faster at the expense of data capacity. Capacity Scaling is not currently supported in IBMtape.
- *density_code*

  This parameter is the density setting for the current loaded tape. Some tape devices support multiple densities and will report the current setting in this field. See the hardware reference guide for the specific tape device to determine the possible values.
- *volid*

  This field is always set to zero.
- *emulate_autoloader*

  This parameter is accepted, but ignored.
- *record_space_mode*

  Only *SCSI_SPACE_MODE* is supported.

The following parameters can be changed using the STIOCSETP *ioctl* command:

- *blksize*

  This parameter specifies the new effective block size for the tape device. Use 0 for variable block mode.

- *compression*

  This parameter turns the hardware compression On or Off.

- *max_scsi_xfer*

  This parameter is the maximum transfer size allowed per SCSI command. It is usually limited by the amount of kernel DMA memory that the system can allocate.

- *trailer_labels*

  If this parameter is set to On, then writing a record past the early warning mark on the tape is allowed. Only the first write operation to detect the early warning mark returns the ENOSPC error code. All subsequent write operations are allowed to continue, despite the check conditions that result from writing in the early warning zone. When the end of the physical volume is reached, EIO is returned.

  If this parameter is set to Off, the first write in the early warning zone fails, the ENOSPC error code is returned, and subsequent write operations fail.

- *rewind_immediate*

  This parameter turns the immediate bit On or Off for subsequent rewind commands. If it is set to On, then the STREW tape operation executes faster, but the next command takes longer to finish unless the physical rewind operation is complete.

- *logging*

  This parameter turns the volume logging for the tape device On or Off.

- *read_sili_bit*

  This parameter is accepted, but ignored. SILI bit is currently not supported, due to Linux system environment limitations.

- *read_past_file_mark*

  This parameter changes the behavior of the *read* function when it encounters a filemark. If the read_past_filemark flag is true, then when a *read* operation encounters a filemark, IBMtape returns the number of bytes read before encountering the filemark and sets the tape position at the end of tape (EOT) side of the filemark. If the read_past_filemark flag is false (by default), then when a *read* operation counters a filemark, if data was read, then read function returns the number of bytes read and positions the tape at the beginning of tape (BOT) side of the filemark. If no data was read, then *read* returns 0 bytes and positions the tape at the EOT side of the filemark.

- *trace*

  This parameter turns the trace for the tape devices On or Off.

- *disable_sim_logging*

  This parameter is accepted but ignored.

- *disable_auto_drive_dump*

  If this parameter is Off, the drive dump will be automatically retrieved by the IBMtape device driver whenever there is a drive dump in the tape device.

The input or output data structure is:

```
struct stchgp_s {
   int     blksize;                /* new block size                */
   boolean trace;                  /* TRUE=trace on                 */
   ulong   hkwrd;                  /* trace hookword                */
   int     sync_count;             /* obsolete (not used)           */
   boolean autoload;               /* on/off autoload feature       */
   boolean buffered_mode;          /* on/off buffered mode          */
   boolean compression;            /* on/off compression            */
```

```
   boolean trailer_labels;          /* on/off allow writing after EOM  */
   boolean rewind_immediate;        /* on/off immediate rewinds        */
   boolean bus_domination;          /* obsolete (not used)             */
   boolean logging;                 /* volume logging                  */
   boolean write_protect;           /* write-protected media           */
   uint    min_blksize;             /* minimum block size              */
   uint    max_blksize;             /* maximum block size              */
   uint    max_scsi_xfer;           /* maximum SCSI transfer length    */
   char    volid[16];               /* volume ID                       */
   uchar   acf_mode;                /* auto cartridge facility mode    */
      #define ACF_NONE       0
      #define ACF_MANUAL     1
      #define ACF_SYSTEM     2
      #define ACF_AUTOMATIC  3
      #define ACF_ACCUMULATE 4
      #define ACF_RANDOM     5
   uchar   record_space_mode;       /* fsr/bsr space mode              */
      #define SCSI_SPACE_MODE  1
      #define Linux_SPACE_MODE 2
   uchar   logical_write_protect;   /* logical write protect           */
      #define NO_PROTECT0
      #define ASSOCIATED_PROTECT 1
      #define PERSISTENT_PROTECT 2
      #define WORM_PROTECT 3
   uchar   capacity_scaling;        /* capacity scaling                */
      #define SCALE_100 0
      #define SCALE_75  1
      #define SCALE_50  2
      #define SCALE_25  3
   uchar   retain_reservation;      /* retain reservation              */
   uchar   alt_pathing;             /* alternate pathing active        */
   uchar   emulate_autoloader;      /* mimic autoloader in random mode */
   uchar   medium_type;             /* medium type                     */
   uchar   density_code;            /* density code                    */
   boolean disable_sim_logging;     /*disable sim/mim error logging    */
   read_past_filemark
   boolean read_sili_bit;           /*SILI bit setting for reads       */
   uchar   reserved [21];
};
```

An example of the STIOCQRYP and STIOCSETP commands is:

```
#include <sys/IBM_tape.h>
struct stchgp_s stchgp;
/* get current parameters */
if (ioctl(tapefd,STIOCQRYP,&stchgp)<0){
   printf("ioctl failure.  errno=%d",errno);
   exit(errno);
}
/* set new parameters */
stchgp.rewind_immediate=1;
stchgp.trailer_labels=1;
if (ioctl(tapefd,STIOCSETP,&stchgp)<0){
   printf("ioctl failure. errno=%d",errno);
   exit(errno);
}
```

STIOCSYNC

This *ioctl* command immediately flushes the tape buffers to the tape. There are no arguments for this *ioctl* command.

An example of the STIOCSYNC command is:

```
if (ioctl(tapefd,STIOCSYNC,NULL)<0){
   printf("ioctl failure. errno=%d",errno);
   exit(errno);
}
```

STIOCQRYPOS

STIOCQRYPOS queries the tape position. Tape position is defined as the location where the next read or write operation will occur. The query function can be used independently of, or in conjunction with, the STIOCSETPOS *ioctl* command.

A write filemark of count 0 is always issued to the drive which flushes all data from the buffer to the tape media. After the write filemark completes, the query is issued.

After a *query* operation, the *curpos* field is set to an unsigned integer that represents the current position.

The *eot* field is set to TRUE if the tape is positioned between the early warning and the physical end of the tape. Otherwise, it is set to FALSE.

The *lbot* field is valid only if the last command was a *write* command. If a query is issued and the last command was not a write command, the *lbot* field will contain the value **LBOT_UNKNOWN**. Note that *lbot* indicates the last block of data that was transferred to the tape, not the number of application write operations, since, depending on the block size, a single write operation may write multiple blocks. For example, if an application writes data equivalent to 12 blocks then issues STIOCQRYPOS and finds that the *lbot* field equals 8, this indicates that three blocks are in the tape buffer.

The number of blocks and number of bytes currently in the tape device buffers are returned in the *num_blocks* and *num_bytes* fields, respectively.

The *bot* field is set to TRUE if the tape position is at the beginning of the tape. Otherwise, it is set to FALSE.

The *partition-number* field returned is always 0.

The block ID of the next block of data that will be transferred to or from the physical tape is returned in the *tapepos* field.

The position data structure is:

```
typedef unsigned int blockid_t;
struct stpos_s {
   char      block_type;        /* Format of block ID information  */
      #define QP_LOGICAL 0      /* SCSI logical block ID format    */
      #define QP_PHYSICAL 1     /* Vendor-specific block ID format */
   boolean   eot;               /* Position is after early warning,*/
                                /* before physical end of tape.    */
   blockid_t curpos;            /* For query pos, current position.*/
                                /* For set pos, position to go to. */
   blockid_t lbot;              /* Last block written to tape.     */
      #define LBOT_NONE 0xFFFFFFFF    /* No blocks written to tape.*/
      #define LBOT_UNKNOWN 0xFFFFFFFE /* Unable to determine info. */
   uint      num_blocks;        /* Number of blocks in buffer.     */
   uint      num_bytes;         /* Number of bytes in buffer.      */
   boolean   bot;               /* Position is at beginning of tape*/
   uchar     partition_number;  /* Current partition number on tape*/
```

```
   uchar     reserved1[2];
   blockid_t tapepos;           /* Next block to be transferred.   */
   uchar      reserved2[48];
};
```

An example of the STIOCQRYPOS command is:

```
#include <sys/IBM_tape.h>
struct stpos_s stpos;
stpos.block_type=QP_PHYSICAL;
if (ioctl(tapefd,STIOCQRYPOS,&stpos)<0){
   printf("ioctl failure.errno=%d",errno);
   exit(errno);
}
oldposition=stpos.curpos;
```

STIOCSETPOS

STIOCSETPOS issues a high speed *locate* operation to the position specified on the tape. It uses the same position data structure described for STIOCQRYPOS, however, only the *block_type* and *curpos* fields are used during a *set* operation. STIOCSETPOS can be used independently of, or in conjunction with, the STIOCQRYPOS *ioctl*.

The *block_type* must be set to either QP_PHYSICAL or QP_LOGICAL; however, there is no difference in how IBMtape processes the request.

An example of the STIOCQRYPOS and STIOCSETPOS commands is:

```
#include <sys/IBM_tape.h>
struct stpos_s stpos;
stpos.block_type=QP_PHYSICAL;
if (ioctl(tapefd,STIOCQRYPOS,&stpos)<0){
   printf("ioctl failure.errno=%d",errno);
   exit(errno);
}
oldposition=stpos.curpos;

stpos.curpos=oldposition;
stpos.block_type=QP_PHYSICAL;
if (ioctl(tapefd,STIOCSETPOS,&stpos)<0) {
   printf("ioctl failure.errno=%d",errno);
   exit(errno);
}
```

STIOCQRYSENSE

This *ioctl* command returns the last sense data collected from the tape device, or it issues a new Request Sense command and returns the collected data. If *sense_type* = LASTERROR is requested, then the sense data is valid only if the last tape operation had an error which caused a sense command to be issued to the device. If the sense data is valid, then the command completes successfully, and the *len* field is set to a value greater than zero. The *residual_count* field contains the residual count from the last operation.

The input or output data structure is:

```
#define MAXSENSE 255
struct stsense_s {
   /* input */
   char sense_type;        /* fresh (new sense) or sense from last error */
      #define FRESH     1  /* Initiate a new sense command */
      #define LASTERROR 2  /* Return sense gathered from   */
```

```
                              /* the last SCSI sense command. */
   /* output */
   uchar sense[MAXSENSE];  /* actual sense data */
   int   len;              /* length of valid sense data returned */
   int   residual_count;   /* residual count from last operation */
   uchar reserved[60];
};
```

An example of the STIOCQRYSENSE command is:

```
#include <sys/IBM_tape.h>
struct stsense_s stsense;
stsense.sense_type=LASTERROR;
#define MEDIUM_ERROR 0x03
if (ioctl(tapefd,STIOCQRYSENSE,&stsense)<0){
   printf("ioctl failure.errno=%d",errno);
   exit(errno);
}
if ((stsense.sense[2]&0x0F)==MEDIUM_ERROR) {
   printf("We're in trouble now!");
   exit(SENSE_KEY(&stsense.sense));
}
```

STIOCQRYINQUIRY

This *ioctl* command returns the inquiry data from the device. The data is divided into standard and vendor-specific portions.

The output data structure is:

```
/*inquiry data info */
struct inq_data_s {
   BYTE b0;
   /*macros for accessing fields of byte 1 */
      #define PERIPHERAL_QUALIFIER(x) ((x->b0 &0xE0)>>5)
      #define PERIPHERAL_CONNECTED 0x00
      #define PERIPHERAL_NOT_CONNECTED 0x01
      #define LUN_NOT_SUPPORTED 0x03
      #define PERIPHERAL_DEVICE__TYPE(x) (x->b0 &0x1F)
      #define DIRECT_ACCESS 0x00
      #define SEQUENTIAL_DEVICE 0x01
      #define PRINTER_DEVICE 0x02
      #define PROCESSOR_DEVICE 0x03
      #define CD_ROM_DEVICE 0x05
      #define OPTICAL_MEMORY_DEVICE 0x07
      #define MEDIUM_CHANGER_DEVICE 0x08
      #define UNKNOWN 0x1F
   BYTE b1;
   /*macros for accessing fields of byte 2 */
      #define RMB(x) ((x->b1 &0x80)>>7)          /*removable media bit */
      #define FIXED 0
      #define REMOVABLE 1
      #define device_type_qualifier(x) (x->b1 &0x7F) /*vendor specific */
   BYTE b2;
   /*macros for accessing fields of byte 3 */
      #define ISO_Version(x) ((x->b2 &0xC0)>>6)
      #define ECMA_Version(x) ((x->b2 &0x38)>>3)
      #define ANSI_Version(x) (x->b2 &0x07)
      #define NONSTANDARD 0
      #define SCSI1 1
      #define SCSI2 2
      #define SCSI3 3
   /*macros for accessing fields of byte 4 */
      /* asynchronous event notification */
      #define AENC(x) ((x->b3 &0x80)>>7)
      /* support terminate I/O process message? */
```

```
      #define TrmIOP(x) ((x->b3 &0x40)>>6)
      #define Response_Data_Format(x) (x->b3 &0x0F)
      #define SCSI1INQ 0        /* SCSI-1 standard inquiry data format */
      #define CCSINQ 1          /* CCS standard inquiry data format    */
      #define SCSI2INQ 2        /* SCSI-2 standard inquiry data format */
   BYT Eadditional_length;       /* bytes following this field minus 4  */
   BYTE res5;
   BYTE b6
      #define MChngr(x) ((x->b6 & 0x08)>>3)
   BYTE b7;
   /*macros for accessing fields of byte 7 */
      #define RelAdr(x) ((x->b7 &0x80)>>7)
      /* the following fields are true or false */
      #define WBus32(x) ((x->b7 &0x40)>>6)
      #define WBus16(x) ((x->b7 &0x20)>>5)
      #define Sync(x) ((x->b7 &0x10)>>4)
      #define Linked(x) ((x->b7 &0x08)>>3)
      #define CmdQue(x) ((x->b7 &0x02)>>1)
      #define SftRe(x) (x->b7 &0x01)
   char vendor_identification [8 ];
   char product_identification [16 ];
   char product_revision_level [4 ];
};
struct st_inquiry
{
   struct inq_data_s standard;
   BYTE vendor_specific [255-sizeof(struct inq_data_s)];
};
```

An example of the STIOCQRYINQUIRY command is:

```
struct st_inquiry inqd;
if (ioctl(tapefd,STIOCQRYINQUIRY,&inqd)<0){
   printf("ioctl failure.errno=%d",errno);
   exit(errno);
}
if (ANSI_Version(((struct inq_data_s *)&(inqd.standard)))==SCSI2)
 printf("Hey!  We have a SCSI-2 device\n");
```

STIOC_LOCATE

This *ioctl* command causes the tape to be positioned at the specified block ID. The block ID used for the command must be obtained using the STIOC_READ_POSITION command.

An example of the STIOC_LOCATE command is:

```
#include <sys/IBM_tape.h>
unsigned int current_blockid;

/* read current tape position */
if (ioctl(tapefd,STIOC_READ_POSITION,&current_blockid)<0){
   printf("ioctl failure. errno=%d"n,errno);
   exit(1);
}

/* restore current tape position */
if (ioctl(tapefd,STIOC_LOCATE,current_blockid)<0){
   printf("ioctl failure.errno=%d"n,errno);
   exit(1);
}
```

STIOC_READ_POSITION

This *ioctl* command returns the block ID of the current position of the tape. The block ID returned from this command can be used with the STIOC_LOCATE command to set the position of the tape.

An example of the STIOC_READ_POSITION command is:

```
#include <sys/IBM_tape.h>
unsigned int current_blockid;
/* read current tape position */
if (ioctl(tapefd,STIOC_READ_POSITION,&current_block)<0){
    printf("ioctl failure.errno=%d"n,errno);
    exit(1);
}
/* restore current tape position */
if (ioctl(tapefd,STIOC_LOCATE,current_blockid)<0){
    printf("ioctl failure.errno=%d"n,errno);
    exit(1);
}
```

STIOC_RESET_DRIVE

This *ioctl* command issues a SCSI Send Diagnostic command to reset the tape drive. There are no arguments for this command.

An example of the STIOC_RESET_DRIVE command is:

```
/* reset the tape drive */
if (ioctl(tapefd,STIOC_RESET_DRIVE,NULL)<0){
   printf("ioctl failure.errno=%d",errno);
   exit(errno);
}
```

STIOC_PREVENT_MEDIUM_REMOVAL

This *ioctl* command prevents an operator from removing medium from the device until the STIOC_ALLOW_MEDIUM_REMOVAL command is issued or the device is reset. There is no associated data structure.

An example of the STIOC_PREVENT_MEDIUM_REMOVAL command is:

```
#include <sys/IBM_tape.h>
if (!ioctl (tapefd, STIOC_PREVENT_MEDIUM_REMOVAL, NULL))
   printf ("The STIOC_PREVENT_MEDIUM_REMOVAL ioctl succeeded\n");
else {
   perror ("The STIOC_PREVENT_MEDIUM_REMOVAL ioctl failed");
   smcioc_request_sense();
}
```

STIOC_ALLOW_MEDIUM_REMOVAL

This *ioctl* command allows an operator to remove medium from the device. This command is normally used after an STIOC_PREVENT_MEDIUM_REMOVAL command to restore the device to the default state. There is no associated data structure.

An example of the STIOC_ALLOW_MEDIUM_REMOVAL command is:

```
#include <sys/IBM_tape.h>
if (!ioctl (tapefd, STIOC_ALLOW_MEDIUM_REMOVAL, NULL)) printf ("The
   STIOC_ALLOW_MEDIUM_REMOVAL ioctl succeeded\n");
```

```
else {
   perror ("The STIOC_ALLOW_MEDIUM_REMOVAL ioctl failed");
   smcioc_request_sense();
}
```

STIOC_REPORT_DENSITY_SUPPORT

This *ioctl* command issues the SCSI Report Density Support command to the tape device and returns all supported densities or supported densities for the currently mounted media. The *media* field specifies which type of report is requested. The *number_reports* field is returned by the device driver and indicates how many density reports in the *reports array* field were returned.

The data structures used with this *ioctl* are:

```
struct density_report {
   uchar primary_density_code;   /* primary density code */
   uchar secondary_density_code; /* secondary density code */
   uint  wrtok  :1,              /* write ok, device can write
                                    this format */
         dup    :1,              /* zero if density only reported once */
         deflt  :1,              /* current density is default format */
                :5;              /* reserved */
   char reserved[2];             /* reserved */
   uint bits_per_mm :24;         /* bits per mm */
   ushort media_width;           /* media width in millimeters */
   ushort tracks;                /* tracks */
   uint capacity;                /* capacity in megabytes */
   char assigning_org[8];        /* assigning organization in ASCII */
   char density_name[8];         /* density name in ASCII */
   char description[20];         /* description in ASCII */
};

struct report_density_support {
   uchar media;                  /* report all or current media as defined above */
   ushort number_reports;        /* number of density reports returned in array */
   struct density_report reports[MAX_DENSITY_REPORTS];
};
```

Examples of the STIOC_REPORT_DENSITY_SUPPORT command are:

```
#include <sys/IBM_tape.h>
int stioc_report_density_support(void)
{
   int i;
   struct report_density_support density;
   printf("Issuing Report Density Support for ALL supported media\n");
   density.media = ALL_MEDIA_DENSITY;
   if (ioctl(fd, STIOC_REPORT_DENSITY_SUPPORT, &density) != 0)
 return errno;
   printf("Total number of densities reported:
      %d\n",density.number_reports);
   for (i = 0; i<density.number_reports; i++) {
      printf("\n");
      printf(" Density Name.................  %0.8s\n",
         density.reports[i].density_name);
      printf(" Assigning Organization.......  %0.8s\n",
         density.reports[i].assigning_org);
      printf(" Density Name.................  %0.8s\n",
         density.reports[i].density_name);
      printf(" Description..................  %0.20s\n",
         density.reports[i].description);
      printf(" Primary Density Code.........  %02X\n",
         density.reports[i].primary_density_code);
      printf(" Secondary Density Code.......  %02X\n",
         density.reports[i].secondary_density_code);
```

```
            if (density.reports[i].wrtok)
               printf(" Write OK...................Yes\n");
            else
               printf(" Write OK...................No\n");
            if (density.reports[i].dup)
               printf(" Duplicate..................Yes\n");
            else
               printf(" Duplicate..................No\n");
            if (density.reports[i].deflt)
               printf(" Default....................Yes\n");
            else
               printf(" Default....................No\n");
            printf(" Bits per MM.................  %d\n",
               density.reports[i].bits_per_mm);
            printf(" Media Width (millimeters)....  %d\n",
               density.reports[i].media_width);
            printf(" Tracks.....................  %d\n",
               density.reports[i].tracks);
            printf(" Capacity (megabytes).........  %d\n",
               density.reports[i].capacity);
            if (opcode) {
               printf ("\nHit enter> to continue?");
               getchar();
            }
         }
         printf("\nIssuing Report Density Support for CURRENT media\n");
         density.media = CURRENT_MEDIA_DENSITY;
         if (ioctl(fd, STIOC_REPORT_DENSITY_SUPPORT, &density) != 0) return
         errno;
         for (i = 0; i<density.number_reports; i++) {
            printf("\n");
            printf(" Density Name.................  %0.8s\n",
               density.reports[i].density_name);
            printf(" Assigning Organization.......  %0.8s\n",
               density.reports[i].assigning_org);
            printf(" Description..................  %0.20s\n",
               density.reports[i].description);
            printf(" Primary Density Code.........  %02X\n",
               density.reports[i].primary_density_code);
            printf(" Secondary Density Code.......  %02X\n",
               density.reports[i].secondary_density_code);
            if (density.reports[i].wrtok)
               printf(" Write OK...................Yes\n");
            else
               printf(" Write OK...................No\n");
            if (density.reports[i].dup)
               printf(" Duplicate..................Yes\n");
            else
               printf(" Duplicate..................No\n");
            if (density.reports[i].deflt)
               printf(" Default....................Yes\n");
            else
               printf(" Default....................No\n");
            printf(" Bits per MM.................  %d\n",
               density.reports[i].bits_per_mm);
            printf(" Media Width (millimeters)....  %d\n",
               density.reports[i].media_width);
            printf(" Tracks.....................  %d\n",
               density.reports[i].tracks);
            printf(" Capacity (megabytes).........  %d\n",
               density.reports[i].capacity);
         }
         return errno;
      }
```

Chapter 14. Tape Drive Compatibility IOCTL Operations

The following *ioctl* commands are supported to help provide compatibility for some previously compiled programs. Where practical, such programs should be updated to use the suggested *ioctl* commands and be recompiled.

MTIOCTOP

This *ioctl* command is similar in function to the *st* MTIOCTOP command. It is provided as a convenience for precompiled programs which call that *ioctl* command. Refer to */usr/include/sys/mtio.h* or */usr/include/linux/mtio.h* for information on the MTIOCTOP command.

MTIOCGET

This *ioctl* command is similar in function to the *st* MTIOCGET command. It is provided as a convenience for precompiled programs which call that *ioctl* command. Refer to */usr/include/sys/mtio.h* or */usr/include/linux/mtio.h* for information on the MTIOCGET command.

MTIOCPOS

This *ioctl* command is similar in function to the *st* MTIOCPOS command. It is provided as a convenience for precompiled programs which call that *ioctl* command. Refer to */usr/include/sys/mtio.h* or */usr/include/linux/mtio.h* for information on the MTIOCPOS command.

Chapter 15. Medium Changer IOCTL Operations

This chapter describes the set of *ioctl* commands that provides control and access to the SCSI medium changer functions. These *ioctl* operations can be issued to the medium changer special file, such as *IBMchanger0*, to access the medium changer.

The following *ioctl* commands are supported:

SMCIOC_ELEMENT_INFO Obtain the device element information.

SMCIOC_MOVE_MEDIUM Move a cartridge from one element to another element.

SMCIOC_EXCHANGE_MEDIUM
Exchange a cartridge in an element with another cartridge.

SMCIOC_POS_TO_ELEM Move the robot to an element.

SMCIOC_INIT_ELEM_STAT Issue the SCSI Initialize Element Status command.

SMCIOC_INIT_ELEM_STAT_RANGE
Issue the SCSI Initialize Element Status with Range command.

SMCIOC_INVENTORY Return the information about the four element types.

SMCIOC_LOAD_MEDIUM Load a cartridge from a slot into the drive.

SMCIOC_UNLOAD_MEDIUM Unload a cartridge from the drive and return it to a slot.

SMCIOC_PREVENT_MEDIUM_REMOVAL
Prevent medium removal by the operator.

SMCIOC_ALLOW_MEDIUM_REMOVAL
Allow medium removal by the operator.

SMCIOC_READ_ELEMENT_DEVIDS
Return the device id element descriptors for drive elements.

These *ioctl* commands and their associated structures are defined in the *IBM_tape.h* header file, which can usually be found in */usr/include/sys* after installing IBMtape. The *IBM_tape.h* header file should be included in the corresponding C program using the functions.

A sample *IBMtapeutil.c* program is provided with the device driver, demonstrating the use of these *ioctl* commands. *IBMtapeutil.c* can be found in the IBMtapeutil.x.x.x.tar package (x.x.x is the IBMtapeutil version).

SMCIOC_ELEMENT_INFO

This *ioctl* command obtains the device element information.

The data structure is:

```
struct element_info {
   ushort robot_addr;  /* first robot address */
   ushort robots;      /* number of medium transport elements */
   ushort slot_addr;   /* first medium storage element address */
```

```
   ushort slots;       /* number of medium storage elements */
   ushort ie_addr;     /* first import/export element address */
   ushort ie_stations; /* number of import/export elements */
   ushort drive_addr;  /* first data-transfer element address */
   ushort drives;      /* number of data-transfer elements */
};
```

An example of the SMCIOC_ELEMENT_INFO command is:

```
#include <sys/IBM_tape.h>
struct element_info element_info;
if (!ioctl (smcfd, SMCIOC_ELEMENT_INFO, &element_info)) {
   printf ("The SMCIOC_ELEMENT_INFO ioctl succeeded\n");
   printf ("\nThe element information data is:\n");
   dump_bytes ((uchar *)&element_info, sizeof (struct element_info));
}
else {
   perror ("The SMCIOC_ELEMENT_INFO ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_MOVE_MEDIUM

This *ioctl* command moves a cartridge from one element to another element.

The data structure is:

```
struct move_medium {
   ushort robot;       /* robot address */
   ushort source;      /* move from location */
   ushort destination; /* move to location */
   char invert;        /* invert before placement bit */
};
```

An example of the SMCIOC_MOVE_MEDIUM command is:

```
#include <sys/IBM_tape.h>
struct move_medium move_medium;
move_medium.robot = 0;
move_medium.invert = 0;
move_medium.source = source;
move_medium.destination = dest;
if (!ioctl (smcfd, SMCIOC_MOVE_MEDIUM, &move_medium))
 printf ("The SMCIOC_MOVE_MEDIUM ioctl succeeded\n");
else {
   perror ("The SMCIOC_MOVE_MEDIUM ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_EXCHANGE_MEDIUM

This *ioctl* command exchanges a cartridge in an element with another cartridge. This command is equivalent to two SCSI Move Medium commands. The first moves the cartridge from the source element to the *destination1* element, and the second moves the cartridge that was previously in the *destination1* element to the *destination 2* element. This function is only available in the IBM 3584 UltraScalable Tape Library. The *destination2* element can be the same as the source element.

The input data structure is:

```
struct exchange_medium {
   ushort robot;        /* robot address */
   ushort source;       /* source address for exchange */
   ushort destination1; /* first destination address for exchange */
```

```
   ushort destination2; /* second destination address for exchange */
   char   invert1;      /* invert before placement into destination1 */
   char   invert2;      /* invert before placement into destination2 */
};
```

An example of the SMCIOC_EXCHANGE_MEDIUM command is:

```
#include <sys/IBM_tape.h>
struct exchange_medium exchange_medium;
exchange_medium.robot = 0;
exchange_medium.invert1 = 0;
exchange_medium.invert2 = 0;
exchange_medium.source = 32;       /* slot 32 */
exchange_medium.destination1 = 16; /* drive address 16 */
exchange_medium.destination2 = 35; /* slot 35 */

/* exchange cartridge in drive address 16 with cartridge from */
/* slot 32 and return the cartridge currently in the drive to */
/* slot 35 */
if (!ioctl (smcfd, SMCIOC_EXCHANGE_MEDIUM, &exchange_medium))
   printf("The SMCIOC_EXCHANGE_MEDIUM ioctl succeeded\n");
else {
 perror ("The SMCIOC_EXCHANGE_MEDIUM ioctl failed");
  smcioc_request_sense();
}
```

SMCIOC_POS_TO_ELEM

This *ioctl* command moves the robot to an element.

The input data structure is:

```
struct pos_to_elem {
   ushort robot;       /* robot address */
   ushort destination; /* move to location */
   char invert;        /* invert before placement bit */
};
```

An example of the SMCIOC_POS_TO_ELEM command is:

```
#include <sys/IBM_tape.h>
char buf[10];
struct pos_to_elem pos_to_elem;
pos_to_elem.robot = 0;
pos_to_elem.invert = 0;
pos_to_elem.destination = dest;
if (!ioctl (smcfd, SMCIOC_POS_TO_ELEM, &pos_to_elem))
 printf ("The SMCIOC_POS_TO_ELEM ioctl succeeded\n");
else {
   perror ("The SMCIOC_POS_TO_ELEM ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_INIT_ELEM_STAT

This *ioctl* command instructs the medium changer robotic device to issue the SCSI Initialize Element Status command. There is no associated data structure.

An example of the SMCIOC_INIT_ELEM_STAT command is:

```
#include <sys/IBM_tape.h>
if (!ioctl (smcfd, SMCIOC_INIT_ELEM_STAT, NULL))
 printf ("The SMCIOC_INIT_ELEM_STAT ioctl succeeded\n");
```

```
else {
   perror ("The SMCIOC_INIT_ELEM_STAT ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_INIT_ELEM_STAT_RANGE

This *ioctl* command issues the SCSI Initialize Element Status with Range command and audits specific elements in a library by specifying the starting element address and number of elements. Use the SMCIOC_INIT_ELEM_STAT *ioctl* to audit all elements.

The data structure is:

```
struct element_range {
   ushort element_address; /* starting element address */
   ushort number_elements; /* number of elements */
}
```

An example of the SMCIOC_INIT_ELEM_STAT_RANGE command is:

```
#include <sys/IBM_tape.h>
struct element_range elements;
/* audit slots 32 to 36 */
elements.element_address = 32;
elements.number_elements = 5;
if (!ioctl (smcfd, SMCIOC_INIT_ELEM_STAT_RANGE, &elements))
 printf ("The SMCIOC_INIT_ELEM_STAT_RANGE ioctl succeeded\n");
else {
   perror ("The SMCIOC_INIT_ELEM_STAT_RANGE ioctl failed");
   smcioc_request_sense();
}
```

Note: Use the SMCIOG_INVENTORY *ioctl* command to obtain the current version after issuing this *ioctl* command.

SMCIOC_INVENTORY

This *ioctl* command returns the information about the four element types. The software application processes the input data (the number of elements about which it requires information) and allocates a buffer large enough to hold the output for each element type.

The input data structure is:

```
struct element_status {
   ushort address;    /* element address                  */
   uint          :2,  /* reserved                         */
         inenab  :1,  /* media into changer's scope       */
         exenab  :1,  /* media out of changer's scope     */
         access  :1,  /* robot access allowed             */
         except  :1,  /* abnormal element state           */
                 :1,  /* reserved                         */
         full    :1;  /* element contains medium          */
   uchar resvd1;      /* reserved                         */
   uchar asc;         /* additional sense code            */
   uchar ascq;        /* additional sense code qualifier  */
   uint  notbus   :1, /* element not on same bus as robot */
                  :1, /* reserved                         */
         idvalid  :1, /* element address valid            */
         luvalid  :1, /* logical unit valid               */
                  :1, /* reserved                         */
         lun      :3; /* logical unit number              */
   uchar scsi;        /* SCSI bus address                 */
```

```
   uchar resvd2;      /* reserved                          */
   uint  svalid  :1,  /* element address valid             */
         invert  :1,  /* medium inverted                   */
                 :6;  /* reserved                          */
   ushort source;     /* source storage element address    */
   uchar volume[36];  /* primary volume tag                */
   uchar resvd3[4];   /* reserved                          */
};
struct inventory {
   struct element_status *robot_status; /* medium transport elem pgs */
   struct element_status *slot_status;  /* medium storage elem pgs */
   struct element_status *ie_status;    /* import/export elem pgs */
   struct element_status *drive_status; /* data-transfer elem pgs */
};
```

An example of the SMCIOC_INVENTORY command is:

```
#include <sys/IBM_tape.h>
ushort i;
struct element_status robot_status[1];
struct element_status slot_status[20];
struct element_status ie_status[1];
struct element_status drive_status[1];
struct inventory inventory;
bzero((caddr_t)robot_status,sizeof(struct element_status));
for (i=0;i <20;i++)
   bzero((caddr_t)(&slot_status[i]),sizeof(struct element_status));
bzero((caddr_t)ie_status,sizeof(struct element_status));
bzero((caddr_t)drive_status,sizeof(struct element_status));
smcioc_element_info();
inventory.robot_status = robot_status;
inventory.slot_status = slot_status;
inventory.ie_status = ie_status;
inventory.drive_status = drive_status;
if (!ioctl (smcfd, SMCIOC_INVENTORY, &inventory)) {
   printf ("\nThe SMCIOC_INVENTORY ioctl succeeded\n");
   printf ("\nThe robot status pages are:\n");
   for (i = 0; i<element_info.robots; i++) {
      dump_bytes ((uchar *)(inventory.robot_status+i), sizeof (struct
      element_status));
      printf ("\n--- more ---");
      getchar();
   }
   printf ("\nThe slot status pages are:\n");
   for (i = 0; i<element_info.slots; i++) {
      dump_bytes ((uchar *)(inventory.slot_status+i), sizeof (struct
      element_status));
      printf ("\n--- more ---");
      getchar();
   }
   printf ("\nThe ie status pages are:\n");
   for (i = 0; <i element_info.ie_stations; i++) {
      dump_bytes ((uchar *)(inventory.ie_status+i), sizeof (struct
      element_status));
      printf ("\n--- more ---");
      getchar();
   }
   printf ("\nThe drive status pages are:\n");
   for (i = 0; <i element_info.drives; i++) {
      dump_bytes ((uchar *)(inventory.drive_status+i), sizeof (struct element_status));
      printf ("\n--- more ---");
      getchar();
   }
}
```

```
else {
   perror ("The SMCIOC_INVENTORY ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_LOAD_MEDIUM

This *ioctl* command loads a tape from a specific slot into the drive or from the first full slot into the drive, if the slot address is specified as zero. Only the IBM 3581 Ultrium Tape Autoloader supports this command.

An example of the SMCIOC_LOAD_MEDIUM command is:

```
#include <sys/IBM_tape.h>
/* load cartridge from slot 3 */
if (ioctl (tapefd, SMCIOC_LOAD_MEDIUM,3)<0){
   printf ("IOCTL failure.errno=%d\n",errno);
 exit(1);
}
/* load first cartridge from magazine */
if (ioctl (tapefd, SMCIOC_LOAD_MEDIUM,0)<0){
 printf ("IOCTL failure.errno=%d\n",errno);
 exit(1);
}
```

SMCIOC_UNLOAD_MEDIUM

This *ioctl* command moves a tape from the drive and returns it to a specific slot or to the first empty slot in the magazine if the slot address is specified as zero. An *unload/offline* command must be sent to the tape first, otherwise, this *ioctl* command will fail with EIO. If this command is issued to a tape special file, such as */dev/IBMtape0*, the tape is rewound and unloaded automatically from the drive first. Only the IBM 3581 Ultrium Tape Autoloader supports this command.

An example of the SMCIOC_UNLOAD_MEDIUM command is:

```
#include <sys/IBM_tape.h>
/* unload cartridge to slot 3 */
if (ioctl (tapefd, SMCIOC_UNLOAD_MEDIUM,3)<0){
   printf ("IOCTL failure.errno=%d\n",errno);
  exit(1);
}
/* unload cartridge to first empty slot in magazine */
if (ioctl (tapefd, SMCIOC_UNLOAD_MEDIUM,0)<0){
   printf ("IOCTL failure.errno=%d\n",errno);
  exit(1);
}
```

SMCIOC_PREVENT_MEDIUM_REMOVAL

This *ioctl* command prevents an operator from removing medium from the device until the SMCIOC_ALLOW_MEDIUM_REMOVAL command is issued or the device is reset. There is no associated data structure.

An example of the SMCIOC_PREVENT_MEDIUM_REMOVAL command is:

```
#include <sys/IBM_tape.h>
if (!ioctl (smcfd, SMCIOC_PREVENT_MEDIUM_REMOVAL, NULL))
 printf ("The SMCIOC_PREVENT_MEDIUM_REMOVAL ioctl succeeded\n");
else {
   perror ("The SMCIOC_PREVENT_MEDIUM_REMOVAL ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_ALLOW_MEDIUM_REMOVAL

This *ioctl* command allows an operator to remove medium from the device. This command is normally used after an SMCIOC_PREVENT_MEDIUM_REMOVAL command to restore the device to the default state. There is no associated data structure.

An example of the SMCIOC_ALLOW_MEDIUM_REMOVAL command is:

```
#include <sys/IBM_tape.h>
if (!ioctl (smcfd, SMCIOC_ALLOW_MEDIUM_REMOVAL, NULL))
 printf ("The SMCIOC_ALLOW_MEDIUM_REMOVAL ioctl succeeded\n");
else {
   perror ("The SMCIOC_ALLOW_MEDIUM_REMOVAL ioctl failed");
   smcioc_request_sense();
}
```

SMCIOC_READ_ELEMENT_DEVIDS

This *ioctl* command issues the SCSI Read Element Status command with the DVCID (device id) bit set and returns the element descriptors for the data transfer elements. The *element_address* field specifies the starting address of the first data transfer element. The *number_elements* field specifies the number of elements to return. The application must allocate a return buffer large enough for the number of elements specified in the input structure.

The input data structure is:

```
struct read_element_devids {
   ushort element_address;            /* starting element address */
   ushort number_elements;            /* number of elements */
   struct element_devid *drive_devid; /* data transfer element pages */
};
```

The output data structure is:

```
struct element_devid {
   ushort address;       /* element address                       */
   uint             :4,  /* reserved                              */
         access     :1,  /* robot access allowed                  */
         except     :1,  /* abnormal element state                */
                    :1,  /* reserved                              */
         full       :1;  /* element contains medium               */
   uchar resvd1;         /* reserved                             */
   uchar asc;            /* additional sense code                */
   uchar ascq;           /* additional sense code qualifier      */
   uint  notbus     :1,  /* element not on same bus as robot      */
                    :1,  /* reserved                              */
         idvalid    :1,  /* element address valid                 */
         luvalid    :1,  /* logical unit valid                    */
                    :1,  /* reserved                              */
         lun        :3;  /* logical unit number                   */
   uchar scsi;           /* scsi bus address                     */
   uchar resvd2;         /* reserved                             */
   uint  svalid     :1,  /* element address valid                 */
         invert     :1,  /* medium inverted                       */
                    :6;  /* reserved                              */
   ushort source;        /* source storage element address        */
   uint             :4,  /* reserved                              */
         code_set   :4;  /* code set X'2' is all ASCII identifier */
   uint             :4,  /* reserved                              */
         ident_type :4;  /* identifier type                       */
```

```
   uchar resvd3;         /* reserved                              */
   uchar ident_len;      /* identifier length                     */
   uchar identifier[36]; /* device identification                 */
};
```

An example of the SMCIOC_READ_ELEMENT_DEVIDS command is:

```
#include <sys/IBM_tape.h>
int smcioc_read_element_devids() {
int i;
struct element_devid *elem_devid, *elemp;
struct read_element_devids devids;
struct element_info element_info;
if (ioctl(fd, SMCIOC_ELEMENT_INFO, &element_info)) return errno;
if (element_info.drives) {
   elem_devid = malloc(element_info.drives
      * sizeof(struct element_devid));
   if (elem_devid == NULL) {
      errno = ENOMEM;
      return errno;
   }
   bzero((caddr_t)elem_devid,element_info.drives
      * sizeof(struct element_devid));
   devids.drive_devid = elem_devid;
   devids.element_address = element_info.drive_addr;
   devids.number_elements = element_info.drives;
   printf("Reading element device ids?\n");
   if (ioctl (fd, SMCIOC_READ_ELEMENT_DEVIDS, &devids)) {
      free(elem_devid);
      return errno;
   }
   elemp = elem_devid;
   for (i = 0; <i element_info.drives; i++, elemp++) {
      printf("\nDrive Address %d\n",elemp->address);
      if (elemp->except)
         printf(" Drive State ..................... Abnormal\n");
      else
         printf(" Drive State ..................... Normal\n");
      if (elemp->asc == 0x81 && elemp->ascq ==0x00)
         printf(" ASC/ASCQ ........................ %02X%02X (Drive Present)\n",
         elemp->asc,elemp->ascq);
      else if (elemp->asc == 0x82 && elemp->ascq ==0x00)
         printf(" ASC/ASCQ ........................ %02X%02X (Drive Not Present)\n",
    elemp->asc,elemp->ascq);
      else
         printf(" ASC/ASCQ ........................ %02X%02X\n",
            elemp->asc,elemp->ascq);
      if (elemp->full)
         printf(" Media Present ................... Yes\n");
      else
         printf(" Media Present ................... No\n");
      if (elemp->access)
         printf(" Robot Access Allowed ............ Yes\n");
      else
         printf(" Robot Access Allowed ............ No\n");
      if (elemp->svalid)
         printf(" Source Element Address ......... %d\n",
            elemp->source);
      else
         printf(" Source Element Address Valid ...... No\n");
      if (elemp->invert)
         printf(" Media Inverted .................. Yes\n");
      else
         printf(" Media Inverted .................. No\n");
      if (elemp->notbus)
         printf(" Same Bus as Medium Changer ..... No\n");
      else
```

```
            printf(" Same Bus as Medium Changer ..... Yes\n");
         if (elemp->idvalid)
            printf(" SCSI Bus Address ............... %d\n",elemp->scsi);
         else
            printf(" SCSI Bus Address Valid ......... No\n");
         if (elemp->luvalid)
            printf(" Logical Unit Number ............ %d\n",elemp->lun);
         else
            printf(" Logical Unit Number Valid ...... No\n");
         printf(" Device ID ......................    %0.36s\n",
            elemp->identifier);
      }
      else {
         printf("\nNo drives found in element information\n");
      }
      free(elem_devid);
      return errno;
   }
```

Chapter 16. Return Codes

This chapter describes error codes that IBMtape generates when an error occurs during an operation. On error, the operation will return negative one (-1), and the external variable *errno* will be set to one of the listed error codes. *Errno* values are defined in */usr/include/errno.h* (and other files which it includes). Application programs must include *errno.h* to be able to interpret the return codes.

Note: For error code EIO, an application can retrieve more information from the device itself. Issue the **STIOCQRYSENSE** *ioctl* command when the *sense_type* equals **LASTERROR**,or the **SIOC_REQSENSE** *ioctl* command, to retrieve sense data. Then analyze the sense data using the appropriate hardware or SCSI reference for that device.

General Error Codes

The following codes and their descriptions apply to all operations:

[EBUSY]	An excessive busy state was encountered in the device.
[EFAULT]	A memory failure occurred due to an invalid pointer or address.
[EIO]	Any of the following conditions: • An unrecoverable media error was detected in the device. • The device was not ready for operation or a tape was not in the drive. • The device did not respond to selection. • A bad file descriptor was passed to the device.
[ENOMEM]	Insufficient memory was available for an internal memory operation.
[ENXIO]	The device was not configured and is not receiving requests.
[EPERM]	The process does not have permission to perform the desired function.
[ETIMEDOUT]	A command timed out in the device.

Open Error Codes

The following codes and their descriptions apply to *open* operations:

[EAGAIN]	The device was already open when an open was attempted
[EBUSY]	The device was reserved by another initiator or an excessive busy state was encountered.
[EINVAL]	The operation requested has invalid parameters or an invalid combination of parameters, or the device is rejecting open commands.
[EIO]	An I/O error occurred that indicates a failure to operate the device. Perform failure analysis.
[ENOMEM]	Insufficient memory was available for an internal memory operation.
[EPERM]	Either: • An open operation with the O_RDWR or O_WRONLY flag was attempted on a write-protected tape. • A write operation was attempted on a device that was opened with the O_RDONLY flag.

[EBUSY]	The SCSI subsystem was busy.
[EFAULT]	Memory reallocation failed.
[EIO]	A command issued during close, such as a rewind command, failed because the device was not ready. An I/O error occurred during the operation. Perform failure analysis.

Read Error Codes

The following codes and their descriptions apply to *read* operations:

[EFAULT]	Failure copying from user to kernel space or vice versa.
[EINVAL]	Any of the following conditions: • The operation requested has invalid parameters or an invalid combination of parameters. • The number of bytes requested in the read operation was not a multiple of the block size for a fixed block transfer. • The number of bytes requested in the read operation was greater than the maximum size allowed by the device for variable block transfers. • A read for multiple fixed odd-byte-count blocks was issued.
[ENOMEM]	One of the following situations occurred:: • The number of bytes requested in the read operation of a variable block record was less than the size of the block. This error is known as an overlength condition. • Insufficient memory was available for an internal memory operation.
[EPERM]	A read operation was attempted on a device that was opened with the O_WRONLY flag.

Write Error Codes

The following codes and their descriptions apply to *write* operations:

[EFAULT]	Failure copying from user to kernel space or vice versa.
[EINVAL]	Any of the following conditions: • The operation requested has invalid parameters or an invalid combination of parameters. • The number of bytes requested in the write operation was not a multiple of the block size for a fixed block transfer. • The number of bytes requested in the write operation was greater than the maximum block size allowed by the device for variable block transfers.
[EIO]	The physical end of the medium was detected, or it is a general error that indicates a failure to write to the device. Perform failure analysis.
[ENOMEM]	Insufficient memory was available for an internal memory operation.

[ENOSPC]	A write operation failed because it reached the early warning mark. This error code is returned only once when the early warning is reached and *trailer_labels* is set to true. A write operation was attempted after the device reached the logical end of the medium and *trailer_labels* were set to false.
[EPERM]	A write operation was attempted on a write-protected tape.

IOCTL Error Codes

The following codes and their descriptions apply to *ioctl* operations:

[EBUSY]	SCSI subsystem was busy.
[EFAULT]	Failure copying from user to kernel space or vice versa.
[EINVAL]	The operation requested has invalid parameters or an invalid combination of parameters. This error code also results if the *ioctl* command is not supported by the device. (For example, attempting to issue tape drive *ioctl* commands to a SCSI medium changer). An invalid or nonexistent *ioctl* command was specified.
[EIO]	An I/O error occurred during the operation. Perform failure analysis.
[ENOMEM]	Insufficient memory was available for an internal memory operation.
[ENOSYS]	The underlying function for this *ioctl* command does not exist on this device. (Other devices may support the function.)
[EPERM]	An operation that modifies the media was attempted on a write-protected tape or a device that was opened with the O_RDONLY flag.

Part 4. Solaris Tape and Medium Changer Device Driver

Chapter 17. IOCTL Operations

The following sections describe the *ioctl* operations supported by the IBMtape device driver. Usage and syntax are given, and examples are shown.

The *ioctl* operations supported by the IBM SCSI Tape and Medium Changer Device Driver for Solaris are described in:

General SCSI IOCTL Operations

A set of general SCSI *ioctl* commands gives applications access to standard SCSI operations, such as device identification, access control, and problem determination for both tape drive and medium changer devices.

The following commands are supported:

Name	Description
IOC_TEST_UNIT_READY	Determine if the device is ready for operation.
IOC_INQUIRY	Collect the inquiry data from the device.
IOC_INQUIRY_PAGE	Return the inquiry data for a special page from the device.
IOC_LOG_SENSE_PAGE	Return a log sense page from the device.
IOC_MODE_SENSE	Return the mode sense data from the device.
IOC_DRIVER_INFO	Return the driver information.
IOC_REQUEST_SENSE	Return the device sense data.
IOC_RESERVE	Reserve the device for exclusive use by the initiator.
IOC_RELEASE	Release the device from exclusive use by the initiator.

These commands and associated data structures are defined in the *st.h* and *smc.h* header files in the */usr/include/sys* directory, which is installed with the IBMtape package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the device driver and installed in the */opt/IBMtape* directory with the IBMtape package. This source code demonstrates the use of these *ioctl* commands.

IOC_TEST_UNIT_READY

This command determines if the device is ready for operation.

No data structure is required for this command.

An example of the IOC_TEST_UNIT_READY command is:

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

if (!(ioctl (dev_fd, IOC_TEST_UNIT_READY, 0))) {
  printf ("The IOC_TEST_UNIT_READY ioctl succeeded.\n");
}

else {
  perror ("The IOC_TEST_UNIT_READY ioctl failed");
  scsi_request_sense ();
}
```

IOC_INQUIRY

This command collects the inquiry data from the device.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  uchar qual            : 3,        /* peripheral qualifier */
        type            : 5;        /* device type */
  uchar rm              : 1,        /* removable medium */
        mod             : 7;        /* device type modifier */
  uchar iso             : 2,        /* ISO version */
        ecma            : 3,        /* ECMA version */
        ansi            : 3;        /* ANSI version */
  uchar aen             : 1,        /* asynchronous even notification */
        trmiop          : 1,        /* terminate I/O process message */
                        : 2,        /* reserved */
        rdf             : 4;        /* response data format */
  uchar len;                        /* additional length */
  uchar                 : 8;        /* reserved */
  uchar                 : 4,        /* reserved */
  uchar        : 1,     /* reserved */
        encsrv          : 1,        /* enclosure service */
        barcod          : 1,        /* bar code scanner attached  */
        multip          : 1,        /* multi-port */
    mchngr          : 1,        /* medium changer mode */
                        : 3;        /* reserved */
  uchar reladr          : 1,        /* relative addressing */
        wbus32          : 1,        /* 32-bit wide data transfers */
        wbus16          : 1,        /* 16-bit wide data transfers */
        sync            : 1,        /* synchronous data transfers */
        linked          : 1,        /* linked commands */
                        : 1,        /* reserved */
        cmdque          : 1,        /* command queueing */
        sftre           : 1;        /* soft reset */
  uchar vid[8];                     /* vendor ID */
  uchar pid[16];                    /* product ID */
  uchar rev[4];                     /* product revision level */
  uchar vendor[92];                 /* vendor specific (padded to 128) */
} inquiry_data_t;
```

An example of the IOC_INQUIRY command is:

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

inquiry_data_t inquiry_data;

if (!(ioctl (dev_fd, IOC_INQUIRY, &inquiry_data))) {
  printf ("The IOC_INQUIRY ioctl succeeded.\n");
  printf ("\nThe inquiry data is:\n");
  dump_bytes ((char *)&inquiry_data, sizeof (inquiry_data_t));
}
```

```
else {
  perror ("The IOC_INQUIRY ioctl failed");
  scsi_request_sense ();
}
```

IOC_INQUIRY_PAGE

This command returns the inquiry data for a special page from the device.

The following data structures for inquiry page, inquiry page x80 and x83 are filled out and returned by the driver:

```
typedef struct {
     uchar page_code;                          /* page code */
     uchar data[253];              /* inquiry parameter List  */
 } inquiry_page_t;

 typedef struct {
     uchar page_code;                          /* page code */
     uchar data[253];              /* inquiry parameter List  */
 } inquiry_page_t;

 typedef struct {
      uchar periph_qual : 3,                 /* peripheral qualifier */
                periph_type : 5;                 /* peripheral device type */
      uchar page_code;                          /* page code */
      uchar reserved_1;                          /* reserved */
      uchar page_len;                            /* page length */
      uchar serial[12];                          /* serial number */
 } inq_pg_80_t;
```

An example of the IOC_INQUIRY_PAGE command is:

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

  inquiry_page_t inquiry_page;
  inquiry_page.page_code = (uchar) page;

  if (!(ioctl (dev_fd, IOC_INQUIRY_PAGE, &inquiry_page))){
     printf ("Inquiry Data (Page 0x%02x):\n", page);
     dump_bytes ((char *)&inquiry_page.data, inquiry_page.data[3]+4);
  }
 else {
     perror ("The IOC_INQUIRY_PAGE ioctl for page 0x%X failed.\n", page);
     scsi_request_sense ();
  }
```

IOC_REQUEST_SENSE

This command returns the device sense data. If the last command resulted in an error, then the sense data is returned for that error. Otherwise, a new (unsolicited) Request Sense command is issued to the device.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  uchar valid           : 1,         /* sense data is valid */
        code            : 7,         /* error code */
  uchar segnum;                      /* segment number */
  uchar fm              : 1,         /* filemark detected */
        eom             : 1,         /* end of media */
        ili             : 1,         /* incorrect length indicator */
                        : 1,         /* reserved */
        key             : 4;         /* sense key */
  uchar info[4];                     /* information bytes */
  uchar addlen;                      /* additional sense length */
```

```
  uchar cmdinfo[4];                /* command-specific information */
  uchar asc;                       /* additional sense code */
  uchar ascq;                      /* additional sense code qualifier */
  uchar fru;                       /* field-replaceable unit code */
  uchar sksv            : 1,       /* sense key specific valid */
        cd              : 1,       /* control/data */
                        : 2,       /* reserved */
        bpv             : 1,       /* bit pointer valid */
        sim             : 3;       /* system information message */
  uchar field[2];                  /* field pointer */
  uchar vendor[110];               /* vendor specific (padded to 128) */
} sense_data_t;
```

An example of the IOC_REQUEST_SENSE command is:

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

sense_data_t sense_data;

if (!(ioctl (dev_fd, IOC_REQUEST_SENSE, &sense_data))) {
  printf ("The IOC_REQUEST_SENSE ioctl succeeded.\n");
  printf ("\nThe request sense data is:\n");
  dump_bytes ((char *)&sense_data, sizeof (sense_data_t));
}

else {
  perror ("The IOC_REQUEST_SENSE ioctl failed");
}
```

IOC_LOG_SENSE_PAGE

This *ioctl* command returns a log sense page from the device. The desired page is selected by specifying the page_code in the log_sense_page structure.

The structure of a log page consists of the following log page header and log parameter(s):

Log Page

- Log Page Header
- Page Code
- Page Length
- Log Parameter(s) (One or more may exist)
- Parameter Code
- Control Byte
- Parameter Length
- Parameter Value

The following data structure is filled out and returned by the driver.

```
 typedef struct {
        uchar  page_code;                                    /* page code */
        uchar  data[MAX_LGPGDATA];           /* log data structure  */
  }log_sns_pg_t;
```

An example of the IOC_LOG_SENSE_PAGE command is:

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

static int scsi_log_sense_page (int page, int type, int parmcode)
{
  int i, j=0;
  int true;
```

```
int len, parm_len;
int parm_code;
log_sns_pg_t   log_sns_page;
log_page_hdr_t page_header;

memset ((char *) &log_sns_page, (char)0, sizeof(log_sns_pg_t));
log_sns_page.page_code = (uchar) page;

if (!(rc = ioctl (dev_fd, IOC_LOG_SENSE_PAGE, &log_sns_page))) {
   len =(int) ((log_sns_page.data[2] << 8) + log_sns_page.data[3]) + 4;
   if ( type != 1) {
      printf("Log Sense Data (Page 0x%02x):\n", page);
      dump_bytes ((char *) &log_sns_page.data, len);
   }
   else {
      for(i=4; i<=len; i=(parm_len+4)){
         j += i;
         parm_code = (int) ((log_sns_page.data[j] << 8) + log_sns_page.data[j+1]);
         parm_len = (int) (log_sns_page.data[j+3]);
         if (true = (parm_code == parmcode)) {
            printf("Log Sense Data (Page 0x%02x, Parameter Code 0x%04x):\n",page, parmcode);
            dump_bytes ((char *) &log_sns_page.data[j], (parm_len+4));
            break;
         }
      }
      if (!true)
        printf("IOC_LOG_SENSE_PAGE for Page 0x%02x, Parameter Code 0x%04x failed.\n",
              page, parmcode);
   }
 }
 else {
   printf("IOC_LOG_SENSE_PAGE for page 0x%X failed.\n", page);
   scsi_request_sense ();
 }
```

IOC_MODE_SENSE

This command returns a mode sense page from the device. The desired page is selected by specifying the page_code in the mode_sns_t structure.

The following data structure is filled out and returned by the driver.

```
#define MAX_MSDATA 253              /* The maximum data length which this */
                                    /* ioctl can return, including        */
                                    /* headers and block descriptors.     */

#define MODESNS_10_CMD 0x5A         /* SCSI cmd code for 10-byte version  */
                                    /* of the command                     */
#define MODESNS_6_CMD  0x1A         /* SCSI cmd code for 6-byte version   */
                                    /* of the command                     */

typedef struct {
   uchar      page_code;            /* Page Code:  Set this field with    */
                                    /*    the desired mode page number    */
                                    /*    before issuing the ioctl.       */
   uchar      cmd_code;             /* SCSI Command Code:  Upon return,   */
                                    /*    this field is set with the      */
                                    /*    SCSI command code to which      */
                                    /*    the device responded.           */
                                    /*    x'5A' = Mode Sense (10)         */
                                    /*    x'1A' = Mode Sense (6)          */
   uchar      data[MAX_MSDATA];     /* Mode Parameter List:  Upon return, */
                                    /*    this field contains the mode    */
                                    /*    parameters list, up to the max  */
                                    /*    length supported by the ioctl.  */
} mode_sns_t;
```

An example of the IOC_MODE_SENSE command is:

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

mode_sns_t mode_data;
mode_data.page_code =(uchar)page;

memset ((char *), (char)0, sizeof(mode_sns_t));

if (!(rc =ioctl (dev_fd, IOC_MODE_SENSE, ))){
   if (mode_data.cmd_code ==0x1A )
       offset =(int)(mode_data.data [3] ) + sizeof(mode_hdr6_t);
   if (mode_data.cmd_code ==0x5A )
      offset =(int)((mode_data.data [6 ]<<8) + mode_data.data [7 ] ) + sizeof(mode_hdr10_t);
   printf("Mode Data (Page 0x%02x):\n", mode_data.page_code);
   dump_bytes ((char *) [offset ], (mode_data.data [offset+1] + 2));
}
else {
   printf("IOC_MODE_SENSE for page 0x%X failed.\n",mode_data.page_code);
   scsi_request_sense ();
}
```

IOC_DRIVER_INFO

This command returns the information about the currently installed IBMtape driver.

The following data structure is filled out and returned by the driver:

```
typedef struct {
   uchar reserved_1[4];      /* Reserved for IBM Development Use             */
   uchar reserved_2[4];      /* Reserved for IBM Development Use             */
   uchar reserved_3[4];      /* Reserved for IBM Development Use             */
   uchar reserved_4[4];      /* Reserved for IBM Development Use             */
   uchar name[16];           /* IBMtape device driver name                   */
   uchar version[16];        /* IBMtape device driver version                */
   uchar sver[16];           /* Short version string (less '.' & '_' chars)  */
   uchar seq[16];            /* Sequence number                              */
   uchar os[16];             /* Operating System                             */
   uchar reserved_5[159];    /* Reserved for IBM Development Use             */
} IBMtape_info_t;
```

An example of the IOC_DRIVER_INFO command is:

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

IBMtape_info_t IBMtape_info;

 if (!(rc = ioctl (dev_fd, IOC_DRIVER_INFO, ))) {
    printf ("IBMtape tape device driver information:\n");
    printf("Name: %s\n", IBMtape_info.name);
    printf("Version: %s\n", IBMtape_info.version);
    printf("Short version string: %s\n", IBMtape_info.sver);
    printf("Operating System: %s\n", IBMtape_info.os);
 }
 else {
    perror("Failure obtaining the information of IBMtape");
    printf("\n");
    scsi_request_sense ();
 }
```

IOC_RESERVE

This command persistently reserves the device for exclusive use by the initiator. The IBMtape device driver normally reserves the device in the open operation and releases the device in the close operation. Issuing this command prevents the driver from releasing the device during the close operation; hence the device

reservation is maintained after the device is closed. This command is negated by issuing the IOC_RELEASE *ioctl* command.

No data structure is required for this command.

An example of the IOC_RESERVE command is:

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

if (!(ioctl (dev_fd, IOC_RESERVE, 0))) {
  printf ("The IOC_RESERVE ioctl succeeded.\n");
}

else {
  perror ("The IOC_RESERVE ioctl failed");
  scsi_request_sense ();
}
```

IOC_RELEASE

This command releases the persistent reservation of the device for exclusive use by the initiator. It negates the result of the IOC_RESERVE *ioctl* command issued either from the current or a previous open session.

No data structure is required for this command.

An example of the IOC_RELEASE command is:

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

if (!(ioctl (dev_fd, IOC_RELEASE, 0))) {
  printf ("The IOC_RELEASE ioctl succeeded.\n");
}

else {
  perror ("The IOC_RELEASE ioctl failed");
  scsi_request_sense ();
}
```

SCSI Medium Changer IOCTL Operations

A set of medium changer *ioctl* commands gives applications access to IBM medium changer devices.

The following commands are supported:

Name	Description
SMCIOC_MOVE_MEDIUM	Transport a cartridge from one element to another element.
SMCIOC_POS_TO_ELEM	Move the robot to an element.
SMCIOC_ELEMENT_INFO	Return the information about the device elements.
SMCIOC_INVENTORY	Return the information about the medium changer elements.
SMCIOC_AUDIT	Perform an audit of the element status.
SMCIOC_LOCK_DOOR	Lock and unlock the library access door.
SMCIOC_READ_ELEMENT_DEVIDS	Return the device ID element descriptors for drive elements.

These commands and associated data structures are defined in the *smc.h* header file in the */usr/include/sys* directory, which is installed with the IBMtape package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the device driver and installed in the */opt/IBMtape* directory with the IBMtape package. This source code demonstrates the use of these *ioctl* commands.

SMCIOC_MOVE_MEDIUM

This command transports a cartridge from one element to another element.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  ushort robot;                    /* robot address */
  ushort source;                   /* move from location */
  ushort destination;              /* move to location */
  uchar invert;                    /* invert medium before insertion */
} move_medium_t;
```

An example of the SMCIOC_MOVE_MEDIUM command is:

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

move_medium_t move_medium;

move_medium.robot = 0;
move_medium.invert = NO_FLIP;
move_medium.source = src;
move_medium.destination = dst;

if (!(ioctl (dev_fd, SMCIOC_MOVE_MEDIUM, &move_medium))) {
  printf ("The SMCIOC_MOVE_MEDIUM ioctl succeeded.\n");
}

else {
  perror ("The SMCIOC_MOVE_MEDIUM ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_POS_TO_ELEM

This command moves the robot to an element.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  ushort robot;                    /* robot address */
  ushort destination;              /* move to location */
  uchar invert;                    /* invert medium before insertion */
} pos_to_elem_t;
```

An example of the SMCIOC_POS_TO_ELEM command is:

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

pos_to_elem_t pos_to_elem;

pos_to_elem.robot = 0;
pos_to_elem.invert = NO_FLIP;
pos_to_elem.destination = dst;

if (!(ioctl (dev_fd, SMCIOC_POS_TO_ELEM, &pos_to_elem))) {
  printf ("The SMCIOC_POS_TO_ELEM ioctl succeeded.\n");
}
```

```
else {
  perror ("The SMCIOC_POS_TO_ELEM ioctl failed");
   scsi_request_sense ();
}
```

SMCIOC_ELEMENT_INFO

This command requests the information about the device elements.

There are four types of medium changer elements. (Not all medium changers support all four types.) The robot elements are associated with the cartridge transport devices. The cell elements are associated with the cartridge storage slots. The port elements are associated with the import/export mechanisms. The drive elements are associated with the data-transfer devices. The quantity of each element type and its starting address is returned by the driver.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  ushort robot_address;              /* medium transport element address */
  ushort robot_count;                /* number medium transport elements */
  ushort cell_address;               /* medium storage element address */
  ushort cell_count;                 /* number medium storage elements */
  ushort port_address;               /* import/export element address */
  ushort port_count;                 /* number import/export elements */
  ushort drive_address;              /* data-transfer element address */
  ushort drive_count;                /* number data-transfer elements */
} element_info_t;
```

An example of the SMCIOC_ELEMENT_INFO command is:

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

element_info_t element_info;

if (!(ioctl (dev_fd, SMCIOC_ELEMENT_INFO, &element_info))) {
  printf ("The SMCIOC_ELEMENT_INFO ioctl succeeded.\n");
  printf ("\nThe element information data is:\n");
  dump_bytes ((char *)&element_info, sizeof (element_info_t));
}

else {
  perror ("The SMCIOC_ELEMENT_INFO ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_INVENTORY

This command returns the information about the medium changer elements (SCSI Read Element Status command).

There are four types of medium changer elements. (Not all medium changers support all four types.) The robot elements are associated with the cartridge transport devices. The cell elements are associated with the cartridge storage slots. The port elements are associated with the import/export mechanisms. The drive elements are associated with the data-transfer devices.

Note: The application must allocate buffers large enough to hold the returned element status data for each element type. The SMCIOC_ELEMENT_INFO *ioctl* is generally called first to establish the criteria.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  element_status_t *robot_status;    /* medium transport element pages */
  element_status_t *cell_status;     /* medium storage element pages */
  element_status_t *port_status;     /* import/export element pages */
  element_status_t *drive_status;    /* data-transfer element pages */
} inventory_t;
```

One or more of the following data structures are filled out and returned to the user buffer by the driver:

```
typedef struct {
  ushort address;                    /* element address */
  uchar                 : 2,         /* reserved */
        inenab          : 1,         /* medium in robot scope */
        exenab          : 1,         /* medium not in robot scope */
        access          : 1,         /* robot access allowed */
        except          : 1,         /* abnormal element state */
                        : 1,         /* reserved */
        full            : 1;         /* element contains medium */
  uchar                 : 8;         /* reserved */
  uchar asc;                         /* additional sense code */
  uchar ascq;                        /* additional sense code qualifier */
  uchar notbus          : 1,         /* element not on same bus as robot */
                        : 1,         /* reserved */
        idvalid         : 1,         /* element address valid */
        luvalid         : 1,         /* logical unit valid */
                        : 1,         /* reserved */
        lun             : 3;         /* logical unit number */
  uchar scsi;                        /* SCSI bus address */
  uchar                 : 8;         /* reserved */
  uchar svalid          : 1,         /* element address valid */
        invert          : 1,         /* medium inverted */
                        : 6;         /* reserved */
  ushort source;                     /* source storage element address */
  uchar volume[36];                  /* primary volume tag */
  uchar vendor[80];                  /* vendor specific (padded to 128) */
} element_status_t;
```

An example of the SMCIOC_INVENTORY command is:

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

ushort i;
element_info_t element_info;
inventory_t inventory;

smc_element_info ();  /* get element information first */

inventory.robot_status = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.robot_count);
inventory.cell_status  = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.cell_count );
inventory.port_status  = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.port_count );
inventory.drive_status = (element_status_t *)malloc
    (sizeof (element_status_t) * element_info.drive_count);

if (!inventory.robot_status || !inventory.cell_status ||
    !inventory.port_status  || !inventory.drive_status) {
  perror ("The SMCIOC_INVENTORY ioctl failed");
   eturn;
}

if (!(ioctl (dev_fd, SMCIOC_INVENTORY, &inventory))) {

  printf ("\nThe SMCIOC_INVENTORY ioctl succeeded.\n");
```

```
  printf ("\nThe robot status pages are:\n");

  for (i = 0; i < element_info.robot_count; i++) {
    dump_bytes ((char *)(&inventory.robot_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }

  printf ("\nThe cell status pages are:\n");

  for (i = 0; i < element_info.cell_count; i++) {
    dump_bytes ((char *)(&inventory.cell_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }
  printf ("\nThe port status pages are:\n");

  for (i = 0; i < element_info.port_count; i++) {
    dump_bytes ((char *)(&inventory.port_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }

  printf ("\nThe drive status pages are:\n");

  for (i = 0; i < element_info.drive_count; i++) {
    dump_bytes ((char *)(&inventory.drive_status[i]),
        sizeof (element_status_t));
    printf ("\n--- more ---");
    getchar ();
  }

}

else {
  perror ("The SMCIOC_INVENTORY ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_AUDIT

This command causes the medium changer device to perform an audit of the element status (SCSI Initialize Element Status command).

No data structure is required for this command.

An example of the SMCIOC_AUDIT command is:

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

if (!(ioctl (dev_fd, SMCIOC_AUDIT, 0))) {
  printf ("The SMCIOC_AUDIT ioctl succeeded.\n");
}

else {
  perror ("The SMCIOC_AUDIT ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_LOCK_DOOR

This command locks and unlocks the library access door. Not all IBM medium changer devices support this operation.

The following data structure is filled out and supplied by the caller:

```
typedef uchar lock_door_t;
```

An example of the SMCIOC_LOCK_DOOR command is:

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

lock_door_t lock_door;

lock_door = LOCK;

if (!(ioctl (dev_fd, SMCIOC_LOCK_DOOR, &lock_door))) {
  printf ("The SMCIOC_LOCK_DOOR ioctl succeeded.\n");
}

else {
  perror ("The SMCIOC_LOCK_DOOR ioctl failed");
  scsi_request_sense ();
}
```

SMCIOC_READ_ELEMENT_DEVIDS

This *ioctl* command issues the SCSI Read Element Status command with the DVCID (device ID) bit set and returns the element descriptors for the data transfer elements. The *element_address* field is used to specify the starting address of the first data transfer element and the *number_elements* field specifies the number of elements to return. The application must allocate a return buffer large enough for the *number_elements* specified in the input structure.

The input data structure is:

```
typedef struct read_element_devids_s {
  ushort element_address;               /*starting element address */
  ushort number_elements;               /*number of elements */
  element_devids_t *drive_devid;        /*data transfer element pages */
}read_element_devids_t;
```

The output data structure is:

```
typedef struct {
   ushort address;   /*element address */
   uchar         :2,    /*reserved */
          inenab :1,    /*medium in robot scope */
          exenab :1,    /*medium not in robot scope */
          access :1,    /*robot access allowed */
          except :1,    /*abnormal element state */
          impexp :1,    /*medium imported or exported */
          full :1;      /*element contains medium */
  uchar :8;             /*reserved */
  uchar asc;            /*additional sense code */
  uchar ascq;           /*additional sense code qualifier */
  uchar   notbus :1,    /*element not on same bus as robot */
                 :1,    /*reserved */
         idvalid :1,    /*scsi bus id valid */
         luvalid :1,    /*logical unit valid */
                 :1,    /*reserved */
         Lun     :3;    /*logical unit */
  uchar scsi;           /*scsi bus id */
  uchar          :8;    /*reserved */
  uchar   svalid :1,    /*element address valid */
          invert :1,    /*medium inverted */
                 :6;    /*reserved */
  ushort source;        /*source storage element address */
  uchar          :4,    /*reserved */
         codeset :4;    /*code set */
  uchar          :2,    /*reserved */
```

```
          assoc :2,    /*Association */
         idtype :4;    /*Identifier Type */
  uchar :8;            /*reserved */
  uchar idlength;      /*Length of Device Identifier */
  uchar vendorid [8 ];  /*Vendor ID */
  uchar devtype [16 ];  /*Device type and Model Numer */
  uchar serialnum [12 ];/*Serial Number of device (ASCII)*/
}element_devids_t;
```

An example of the SMCIOC_READ_ELEMENT_DEVIDS command is:

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

int rc;
int i;
element_devids_t *elem_devid,*elemp;
read_element_devids_t devids;
element_info_t element_info;

if (ioctl(dev_fd,SMCIOC_ELEMENT_INFO,&element_info))
return errno;

if (element_info.drive_count)
{
  elem_devid =malloc(element_info.drive_count *sizeof(element_devids_t));
  if (elem_devid ==NULL) {
     errno =ENOMEM;
     return errno;
  }
   bzero((caddr_t)elem_devid,element_info.drive_count *sizeof(element_devids_t));
  devids.drive_devid =elem_devid;
  devids.element_address =element_info.drive_address;
  devids.number_elements =element_info.drive_count;
  printf("Reading element device ids...\n");
  if (rc =ioctl (dev_fd,SMCIOC_READ_ELEMENT_DEVIDS,&devids)) {
     free(elem_devid);
     perror ("SMCIOC_READ_ELEMENT_DEVIDS failed");
     printf ("\n");
     scsi_request_sense ();
     return rc;
  }

  elemp =elem_devid;
  for (i =0;i <element_info.drive_count;i++,elemp++) {
  printf("\nDrive Address %d \n",elemp->address);
  if (elemp->except)
     printf("Drive State ....................Abnormal \n");
  else
     printf("Drive State ....................Normal \n");
  if (elemp->asc ==0x81 &&elemp->ascq ==0x00)
    printf("ASC/ASCQ .......................%02X%02X (Drive Present)\n",
             elemp->asc,elemp->ascq);
  else if (elemp->asc ==0x82 &&elemp->ascq ==0x00)
     printf("ASC/ASCQ .......................%02X%02X (Drive Not Present)\n",
              elemp->asc,elemp->ascq);
  else
     printf("ASC/ASCQ .......................%02X%02X \n",
              elemp->asc,elemp->ascq);
  if (elemp->full)
    printf("Media Present ..................Yes \n");
  else
    printf("Media Present ..................No \n");
  if (elemp->access)
    printf("Robot Access Allowed ...........Yes \n");
  else
   printf("Robot Access Allowed ...........No \n");
   if (elemp->svalid)
```

```
            printf("Source Element Address .........%d \n",elemp->source);
          else
            printf("Source Element Address Valid ...No \n");
           if (elemp->invert)
            printf("Media Inverted .................Yes \n");
           else
            printf("Media Inverted .................No \n");
           if (elemp->notbus)
            printf("Same Bus as Medium Changer .....No \n");
           else
            printf("Same Bus as Medium Changer .....Yes \n");
           if (elemp->idvalid)
            printf("SCSI Bus Address ...............%d \n",elemp->scsi);
           else
            printf("SCSI Bus Address Vaild .........No \n");
           if (elemp->luvalid)
            printf("Logical Unit Number ............%d \n",elemp->lun);
          else
            printf("Logical Unit Number Valid ......No \n");
            printf("Device ID Info \n");
             printf("Vendor .....................%0.8s \n",elemp->vendorid);
             printf("Model ......................%0.16s \n",elemp->devtype);
             printf("Serial Number ..............%0.12s \n",elemp->serialnum);
           }
        }
        else {
          printf("\nNo drives found in element information \n");
        }

        free(elem_devid);
        return errno;}
```

SCSI Tape Drive IOCTL Operations

A set of enhanced *ioctl* commands gives applications access to additional features of IBM tape drives.

The following commands are supported:

Name	Description
STIOC_TAPE_OP	Perform the tape drive operation.
STIOC_GET_DEVICE_STATUS	Return the status information about the tape drive.
STIOC_GET_DEVICE_INFO	Return the configuration information about the tape drive.
STIOC_GET_MEDIA_INFO	Return the information about the currently mounted tape.
STIOC_GET_POSITION	Return the information about the tape position.
STIOC_SET_POSITION	Set the physical position of the tape.
STIOC_GET_PARM	Return the current value of the working parameter for the tape drive.
STIOC_SET_PARM	Set the current value of the working parameter for the tape drive.
STIOC_DISPLAY_MSG	Display the messages on the tape drive console.
STIOC_SYNC_BUFFER	Flush the drive buffers to the tape.

These commands and associated data structures are defined in the *st.h* header file in the */usr/include/sys* directory, which is installed with the IBMtape package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the device driver and installed in the */opt/IBMtape* directory with the IBMtape package. This source code demonstrates the use of these *ioctl* commands.

STIOC_TAPE_OP

This command performs the standard tape drive operations. It is identical to the MTIOCTOP *ioctl* command defined in the */usr/include/sys/mtio.h* system header file. The STIOC_TAPE_OP and MTIOCTOP commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The STIOC_TAPE_OP *ioctl* command maps to the MTIOCTOP *ioctl* command. The two *ioctl* commands are interchangeable. See “MTIOCTOP” on page 160.

For all space operations, the resulting tape position is at the end-of-tape side of the record or filemark for forward movement and at the beginning-of-tape side of the record or filemark for backward movement.

The following data structure is filled out and supplied by the caller:

```
/* from mtio.h */
struct mtop {
  short mt_op;                      /* operations (defined below) */
  daddr_t mt_count;                 /* how many to perform */
};

/* from st.h */
typedef struct mtop tape_op_t;
```

The *mt_op* field is set to one of the following:

Name	Description
MTWEOF	Write the *mt_count* number of filemarks.
MTFSF	Space forward the *mt_count* number of filemarks.
MTBSF	Space backward the *mt_count* number of filemarks.
MTFSR	Space forward the *mt_count* number of records.
MTBSR	Space backward the *mt_count* number of records.
MTREW	Rewind the tape. The *mt_count* parameter does not apply.
MTOFFL	Rewind and unload the tape. The *mt_count* parameter does not apply.
MTNOP	No tape operation is performed. The status is determined by issuing the Test Unit Ready command. The *mt_count* parameter does not apply.
MTRETEN	Retension the tape. The *mt_count* parameter does not apply.
MTERASE	Erase the entire tape from the current position. The *mt_count* parameter does not apply.
MTEOM	Space forward to the end of the data. The *mt_count* parameter does not apply.

MTNBSF	Position the tape to the start of the previous file. MTNBSF(mt_count)=MTBSF(mt_count+1) + MTFSF(1)
MTGRSZ	Return the current record (block) size. The *mt_count* parameter contains the value.
MTSRSZ	Set the working record (block) size to *mt_count*.
STLOAD	Load the tape in the drive. The *mt_count* parameter does not apply.
STUNLOAD	Unload the tape from the drive. The *mt_count* parameter does not apply.

An example of the STIOC_TAPE_OP command is:

```
#include <sys/mtio.h>
#include <sys/st.h>

 tape_op_t tape_op;

tape_op.mt_op = mt_op;
tape_op.mt_count = mt_count;

if (!(ioctl (dev_fd, STIOC_TAPE_OP, &tape_op))) {
  printf ("The STIOC_TAPE_OP ioctl succeeded.\n");
}

else {
  perror ("The STIOC_TAPE_OP ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_DEVICE_STATUS

This command returns the status information about the tape drive. It is identical to the MTIOCGET *ioctl* command defined in the */usr/include/sys/mtio.h* system header file. The STIOC_GET_DEVICE_STATUS and MTIOCGET commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The STIOC_GET_DEVICE_STATUS *ioctl* command maps to the MTIOCGET *ioctl* command. The two *ioctl* commands are interchangeable. See “MTIOCGET” on page 160.

The following data structure is returned by the driver:

```
/* from mtio.h */
struct mtget {
  short mt_type;                    /* type of tape device */
  short mt_dsreg;                   /* drive status register */
  short mt_erreg;                   /* error register */
  daddr_t mt_resid;                 /* residual count */
  daddr_t mt_fileno;                /* current file number */
  daddr_t mt_blkno;                 /* current block number */
  u_short mt_flags;                 /* device flags */
  short mt_bf;                      /* optimum blocking factor */
};

/* from st.h */
typedef struct mtget device_status_t;
```

The *mt_flags* field, which returns the type of automatic cartridge stacker or loader installed on the tape drive, is set to one of the following values:

Value	Description

STF_ACL	Automatic Cartridge Loader
STF_RACL	Random Access Cartridge Facility

An example of the STIOC_GET_DEVICE_STATUS command is:

```
#include <sys/mtio.h>
#include <sys/st.h>

device_status_t device_status;

if (!(ioctl (dev_fd, STIOC_GET_DEVICE_STATUS, &device_status))) {
  printf ("The STIOC_GET_DEVICE_STATUS ioctl succeeded.\n");
  printf ("\nThe device status data is:\n");
   ump_bytes ((char *)&device_status, sizeof (device_status_t));
}

else {
  perror ("The STIOC_GET_DEVICE_STATUS ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_DEVICE_INFO

This command returns the configuration information about the tape drive. It is identical to the MTIOCGETDRIVETYPE *ioctl* command defined in the */usr/include/sys/mtio.h* system header file. The STIOC_GET_DEVICE_INFO and MTIOCGETDRIVETYPE commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The STIOC_GET_DEVICE_STATUS *ioctl* command maps to the MTIOCGETDRIVETYPE *ioctl* command. The two *ioctl* commands are interchangeable. See “MTIOCGETDRIVETYPE” on page 160.

The following data structure is returned by the driver:

```
/* from mtio.h */
struct mtdrivetype {
        char    name[64];             /* Name, for debug */
        char    vid[25];                   /* Vendor id and model (product) id */
        char    type;                     /* Drive type for driver */
        int     bsize;                    /* Block size */
        int     options;                  /* Drive options */
        int     max_rretries;             /* Max read retries */
        int     max_wretries;             /* Max write retries */
        uchar_t densities[MT_NDENSITIES];  /* density codes, low->hi */
        uchar_t default_density;          /* Default density chosen */
        uchar_t speeds[MT_NSPEEDS];      /* speed codes, low->hi */
        ushort_t non_motion_timeout;     /* Inquiry type commands */
        ushort_t io_timeout;              /* io timeout. seconds */
        ushort_t rewind_timeout;          /* rewind timeout. seconds */
        ushort_t space_timeout;           /* space cmd timeout. seconds */
        ushort_t load_timeout;            /* load tape time in seconds */
        ushort_t unload_timeout;          /* Unload tape time in scounds */
        ushort_t erase_timeout;           /* erase timeout. seconds */
};

/* from st.h */
typedef struct mtdrivetype device_info_t;
```

An example of the STIOC_GET_DEVICE_INFO command is:

```
#include <sys/mtio.h>
#include <sys/st.h>

device_info_t device_info;

if (!(ioctl (dev_fd, STIOC_GET_DEVICE_INFO, &device_info))) {
```

```
  printf ("The STIOC_GET_DEVICE_INFO ioctl succeeded.\n");
  printf ("\nThe device information is:\n");
  dump_bytes ((char *)&device_info, sizeof (device_info_t));
}

else {
  perror ("The STIOC_GET_DEVICE_INFO ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_MEDIA_INFO

This command returns the information about the currently mounted tape.

The following data structure is filled out and returned by the driver:

```
typedef struct {
  uint media_type;  /*type of media loaded */
  uint media_format; /*format of media loaded */
  uchar write_protect; /*write protect (physical/logical)*/
}media_info_t;
```

The *media_type* field is set to one of the values in st.h.

The *media_format* field, which returns the current recording format, is set to one of the values in st.h.

The *write_protect* field is set to 1 if the currently mounted tape is physically or logically write protected.

An example of the STIOC_GET_MEDIA_INFO command is:

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

media_info_t media_info;

if (!(ioctl (dev_fd, STIOC_GET_MEDIA_INFO, &media_info))){
  printf ("The STIOC_GET_MEDIA_INFO ioctl succeeded.\n");
  printf ("\nThe media information is:\n");
  dump_bytes ((char *)&media_info, sizeof (media_info_t));
}
else {
  perror ("The STIOC_GET_MEDIA_INFO ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_POSITION

This command returns the information about the tape position.

The tape position is defined as where the next read or write operation will occur. The STIOC_GET_POSITION and STIOC_SET_POSITION commands can be used independently or in conjunction with each other.

The following data structure is filled out and supplied by the caller (and also filled out and returned by the driver):

```
typedef struct {
  uchar block_type;                /* block type (logical or physical) */
  uchar bot;                       /* physical beginning of tape */
  uchar eot;                       /* logical end of tape */
  uint position;                   /* current or new block ID */
```

```
  uint last_block;                  /* last block written to tape */
  uint block_count;                 /* blocks remaining in buffer */
  uint byte_count;                  /* bytes remaining in buffer */
} position_data_t;
```

The *block_type* field may be set to LOGICAL_BLK or PHYSICAL_BLK. Ultrium™ tape drives support only the LOGICAL_BLK type.

The *block_type* is the only field that the caller must fill out. The other fields are ignored. Tape positions can be obtained with the STIOC_GET_POSITION command, saved, and used later with the STIOC_SET_POSITION command to return to the same location on the tape quickly.

The *position* field returns the current position of the tape (physical or logical).

The *last_block* field returns the last block of data that was transferred physically to the tape.

The *block_count* field returns the number of blocks of data remaining in the buffer.

The *byte_count* field returns the number of bytes of data remaining in the buffer.

The *bot* and *eot* fields indicate if the tape is positioned at the beginning of tape or the end of tape, respectively.

An example of the STIOC_GET_POSITION command is:

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

position_data_t position_data;
position_data.block_type = type;

if (!(ioctl (dev_fd, STIOC_GET_POSITION, &position_data))) {
  printf ("The STIOC_GET_POSITION ioctl succeeded.\n");
  printf ("\nThe tape position data is:\n");
  dump_bytes ((char *)&position_data, sizeof (position_data_t));
}

else {
  perror ("The STIOC_GET_POSITION ioctl failed");
  scsi_request_sense ();
}
```

STIOC_SET_POSITION

This command sets the physical position of the tape.

The tape position is defined as where the next read or write operation will occur. The STIOC_GET_POSITION and STIOC_SET_POSITION commands can be used independently or in conjunction with each other.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar block_type;                 /* block type (logical or physical) */
  uchar bot;                        /* physical beginning of tape */
  uchar eot;                        /* logical end of tape */
  uint position;                    /* current or new block ID */
  uint last_block;                  /* last block written to tape */
   uint block_count;                 /* blocks remaining in buffer */
  uint byte_count;                  /* bytes remaining in buffer */
} position_data_t;
```

The *block_type* field is set to LOGICAL_BLK or PHYSICAL_BLK. Ultrium tape drives support only the LOGICAL_BLK type.

The *block_type* and *position* fields must be filled out by the caller. The other fields are ignored. The type of position specified in the *position* field must correspond with the type specified in the *block_type* field. Tape positions can be obtained with the STIOC_GET_POSITION command, saved, and used later with the STIOC_SET_POSITION command to return to the same location on the tape quickly.

An example of the STIOC_SET_POSITION command is:

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

position_data_t position_data;
position_data.block_type = type;
position_data.position = value;

if (!(ioctl (dev_fd, STIOC_SET_POSITION, &position_data))) {
  printf ("The STIOC_SET_POSITION ioctl succeeded.\n");
}

else {
   perror ("The STIOC_SET_POSITION ioctl failed");
  scsi_request_sense ();
}
```

STIOC_GET_PARM

This command returns the current value of the working parameter for the specified tape drive. This command is used in conjunction with the STIOC_SET_PARM command.

The following data structure is filled out and supplied by the caller (and also filled out and returned by the driver):

```
typedef struct {
  uchar type;                     /* type of parameter to get or set */
  uint value;                     /* current or new value of parameter */
} parm_data_t;
```

The *value* field returns the current value of the specified parameter, within the ranges indicated below for the specific *type*.

The *type* field, which is filled out by the caller, should be set to one of the following values:

Value	Description
BLOCKSIZE	Block Size (0–16777215) A value of zero indicates variable block size.
COMPRESSION	Compression Mode (0 or 1) If this mode is enabled, data is compressed by the tape device before storing it on tape.
BUFFERING	Buffering Mode (0 or 1) If this mode is enabled, data is stored in hardware buffers in the tape device and not committed to tape immediately, thus increasing data throughput performance.

IMMEDIATE Immediate Mode (0 or 1)

If this mode is enabled, then a rewind command returns with the status before the completion of the physical rewind operation by the tape drive.

TRAILER Trailer Label Mode (0 or 1)

If this mode is enabled, then writing records past the early warning mark on the tape is allowed. The first write operation to detect EOM returns 0. This write operation will not complete successfully. All subsequent write operations are allowed to continue despite the check conditions that result from EOM. When the end of the physical volume is reached, EIO is returned.

WRITEPROTECT Write-Protect Mode

This configuration parameter returns the current write-protection status of the mounted cartridge. The following values are recognized:

- NO_PROTECT

  The tape is not physically write-protected. Operations that alter the contents of the media are permitted.
- PHYS_PROTECT

  The tape is physically write-protected. The write-protect switch on the tape cartridge is in the protect position. This mode is queryable only.

SILI Suppress Illegal Length Indication

If this mode is enabled and a larger block of data is requested than is actually read from the tape block, the tape device will suppress raising a check condition. This eliminates error processing normally performed by the device driver and results in improved read performance for some situations.

An example of the STIOC_GET_PARM command is:

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

parm_data_t parm_data;
parm_data.type = type;

if (!(ioctl (dev_fd, STIOC_GET_PARM, &parm_data))) {
  printf ("The STIOC_GET_PARM ioctl succeeded.\n");
  printf ("\nThe parameter data is:\n");
  dump_bytes ((char *)&parm_data.value, sizeof (int));
}

else {
   perror ("The STIOC_GET_PARM ioctl failed");
  scsi_request_sense ();
}
```

STIOC_SET_PARM

This command sets the current value of the working parameter for the specified tape drive. This command is used in conjunction with the STIOC_GET_PARM command.

The default values of most of these parameters, in effect when a tape drive is first opened, are determined by the values in the *IBMtape.conf* configuration file located in the */usr/kernel/drv* directory. Changing the working parameters dynamically through this STIOC_SET_PARM command affects the tape drive only during the current open session. The working parameters will revert back to the defaults when the tape drive is closed and reopened.

Note: The COMPRESSION, ACFMODE, and SCALING parameters are not supported in the *IBMtape.conf* configuration file. The default value for compression mode is established through the specific special file used to open the device. The default value of the ACF mode is established by the mode that the ACF is in at the time the device is opened. Write protection cannot be set with set parameters.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar type;                    /* type of parameter to get or set */
  uint value;                    /* current or new value of parameter */
} parm_data_t;
```

The *value* field specifies the new value of the specified parameter, within the ranges indicated below for the specific *type*.

The *type* field, which is filled out by the caller, should be set to one of the following values:

Value	Description
BLOCKSIZE	Block Size (0–16777215) A value of zero indicates variable block size.
COMPRESSION	Compression Mode (0 or 1 If this mode is enabled, data is compressed by the tape device before storing it on tape.
BUFFERING	Buffering Mode (0 or 1) If this mode is enabled, data is stored in hardware buffers in the tape device and not immediately committed to tape, thus increasing data throughput performance.
IMMEDIATE	Immediate Mode (0 or 1) If this mode is enabled, then a rewind command returns with the status before the completion of the physical rewind operation by the tape drive.
TRAILER	Trailer Label Mode (0 or 1) If this mode is enabled, then writing records past the early warning mark on the tape is allowed. The first write operation to detect EOM returns ENOSPC. This write operation will not complete

successfully. All subsequent write operations are allowed to continue despite the check conditions that result from EOM. When the end of the physical volume is reached, EIO is returned.

SILI Suppress Illegal Length Indication

If this mode is enabled and a larger block of data is requested than is actually read from the tape block, the tape device will suppress raising a check condition. This eliminates error processing normally performed by the device driver and results in improved read performance for some situations.

An example of the STIOC_SET_PARM command is:

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

parm_data_t parm_data;
parm_data.type = type;
parm_data.value = value;

if (!(ioctl (dev_fd, STIOC_SET_PARM, &parm_data))) {
  printf ("The STIOC_SET_PARM ioctl succeeded.\n");
}

else {
  perror ("The STIOC_SET_PARM ioctl failed");
  scsi_request_sense ();
}
```

STIOC_SYNC_BUFFER

This command immediately flushes the drive buffers to the tape (commits the data to the media).

No data structure is required for this command.

An example of the STIOC_SYNC_BUFFER command is:

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

if (!(ioctl (dev_fd, STIOC_SYNC_BUFFER, 0))) {
  printf ("The STIOC_SYNC_BUFFER ioctl succeeded.\n");
}

else {
  perror ("The STIOC_SYNC_BUFFER ioctl failed");
  scsi_request_sense ();
}
```

Base Operating System Tape Drive IOCTL Operations

The set of native magnetic tape *ioctl* commands that is available through the Solaris base operating system is provided for compatibility with existing applications.

The following commands are supported:

Name	Description
MTIOCTOP	Perform the magnetic tape drive operations.
MTIOCGET	Return the status information about the tape drive.

MTIOCGETDRIVETYPE	Return the configuration information about the tape drive.
USCSICMD	User SCSI command interface

These commands and associated data structures are defined in the *mtio.h* system header file in the */usr/include/sys* directory and in the uscsi.h system header file in /usr/include/sys/scsi/impl directory. Any application program that issues these commands must include this header file.

MTIOCTOP

This command performs the magnetic tape drive operations. It is identical to the STIOC_TAPE_OP *ioctl* command that is defined in the */usr/include/sys/st.h* header file. The STIOC_TAPE_OP and MTIOCTOP commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The two *ioctl* commands are interchangeable. See “STIOC_TAPE_OP” on page 151.

MTIOCGET

This command returns the status information about the tape drive. It is identical to the STIOC_GET_DEVICE_STATUS *ioctl* command defined in the */usr/include/sys/st.h* header file. The STIOC_GET_DEVICE_STATUS and MTIOCGET commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The two *ioctl* commands are interchangeable. See “STIOC_GET_DEVICE_STATUS” on page 152.

MTIOCGETDRIVETYPE

This command returns the configuration information about the tape drive. It is identical to the STIOC_GET_DEVICE_INFO *ioctl* command defined in the */usr/include/sys/st.h* header file. The STIOC_GET_DEVICE_INFO and MTIOCTOP commands both use the same data structure defined in the */usr/include/sys/mtio.h* system header file. The two *ioctl* commands are interchangeable. See “STIOC_GET_DEVICE_INFO” on page 153.

USCSICMD

This command provides the user a SCSI command interface.

Attention: The uscsi command is very powerful, but somewhat dangerous, and so its use is restricted to processes running as root, regardless of the file permissions on the device node. The device driver code expects to own the device state, and uscsi commands can change the state of the device and confuse the device driver. It is best to use uscsi commands only with no side effects, and avoid commands such as Mode Select, as they may cause damage to data stored on the drive or system panics. Also, as the commands are not checked in any way by the device driver, any block may be overwritten, and the block numbers are absolute block numbers on the drive regardless of which slice number is used to send the command.

The following data structure is returned by the driver:

```
/* from uscsi.h */
struct uscsi_cmd {
        int              uscsi_flags;     /* read, write, etc. see below */
        short           uscsi_status;    /* resulting status  */
        short           uscsi_timeout;   /* Command Timeout */
        caddr_t       uscsi_cdb;       /* cdb to send to target */
        caddr_t       uscsi_bufaddr;   /* i/o source/destination */
```

```
        size_t          uscsi_buflen;    /* size of i/o to take place */
        size_t          uscsi_resid;     /* resid from i/o operation */
        uchar_t       uscsi_cdblen;    /* # of valid cdb bytes */
        uchar_t       uscsi_rqlen;     /* size of uscsi_rqbuf */
        uchar_t       uscsi_rqstatus;  /* status of request sense cmd */
        uchar_t       uscsi_rqresid;   /* resid of request sense cmd */
        caddr_t       uscsi_rqbuf;     /* request sense buffer */
        void            *uscsi_reserved_5;      /* Reserved for Future Use */
};
```

An example of the USCSICMD command is:

```
   #include <sys/scsi/impl/uscsi.h>

   int rc, i, j, cdb_len, option, ubuf_fg, rq_fg;
   struct uscsi_cmd uscsi_cmd;
   uchar cdb[64] = "";
   char cdb_byte[3] = "";
   char buf[64] = "";
   char rq_buf[255];
   char uscsi_buf[255];

   memset ((char *), (char)0, sizeof(uscsi_cmd));
   memset ((char *), (char)0, sizeof(rq_buf));
   memset ((char *), (char)0, sizeof(uscsi_buf));

   printf("Enter the SCSI cdb in hex (e.g.: INQUIRY 12 00 00 00 80 00) ");
   gets (buf);
   cdb_len = j = 0;
   for (i=0;i<64;i++) {
      if (buf[i] != ' ') {
         cdb_byte[j] = buf[i];
         j += 1;
      }
      else {
         if (j != 2) {
            printf ("Usage Error: Command byte must be two digits.\n");
            return (0);
         }
         cdb_byte[2] = '\0';
         cdb[cdb_len] = strtol(cdb_byte,NULL,16);
         cdb_len += 1;
         j = 0;
      }
      if (buf[i] == '\0') {
         cdb[cdb_len] = strtol(cdb_byte,NULL,16);
         break;
      }
   }
   uscsi_cmd.uscsi_cdblen = cdb_len + 1;
   uscsi_cmd.uscsi_cdb = (char *)cdb;

   printf("Set the uscsi_flagsg:  \n");
   printf("  1. no read and no write                  \n");
   printf("  2. read (USCSI_READ)                     \n");
   printf("  3. write (USCSI_WRITE)                   \n");
   printf("  4. read/write (USCSI_READ | USCSI_WRITE) \n");
   printf("                                           \n");
   printf("Select operation or <enter> q to quit: ");
   gets (buf);
   if (buf[0]=='q') return(0);
   option = atoi(buf);
   switch(option) {
     case 1:
        uscsi_cmd.uscsi_flags = 0;
        break;
     case 2:
```

```
                        uscsi_cmd.uscsi_flags = USCSI_READ;
                        break;
                    case 3:
                        uscsi_cmd.uscsi_flags = USCSI_WRITE;
                        break;
                    case 4:
                        uscsi_cmd.uscsi_flags = USCSI_READ | USCSI_WRITE;
                        break;
                }

                printf("Set the USCSI_RQENABLE flag on ? (y/n) ");
                gets (buf);
                if (buf[0]=='y') {
                   uscsi_cmd.uscsi_flags = uscsi_cmd.uscsi_flags | USCSI_RQENABLE;
                   rq_fg = TRUE;
                }

                printf("Enter the value of the command timeout: ");
                gets (buf);
                uscsi_cmd.uscsi_timeout = atoi(buf);

                printf("Any data to be read from or written to the device? (y/n) ");
                gets (buf);
                if (buf[0]=='y') {
                   uscsi_cmd.uscsi_bufaddr = (char *)&uscsi_buf;
                   uscsi_cmd.uscsi_buflen = sizeof(uscsi_buf);
                   ubuf_fg = TRUE;
                }
                else {
                   uscsi_cmd.uscsi_bufaddr = NULL;
                   uscsi_cmd.uscsi_buflen = 0;
                   ubuf_fg = FALSE;
                }

                if (device.ultrium)
                   uscsi_cmd.uscsi_rqlen = 36;
                else if (device.t3590 || device.t3570)
                   uscsi_cmd.uscsi_rqlen = 96;
                else if (device.t3490)
                   uscsi_cmd.uscsi_rqlen = 54;
                uscsi_cmd.uscsi_rqbuf = (char *)&rq_buf;

                PRINTF ("\nData in struct uscsi_cmd before to issue the cmd:");
                DUMP_BYTES ((char *)&uscsi_cmd, sizeof(uscsi_cmd));

                if (!(rc = ioctl (dev_fd, USCSICMD, &uscsi_cmd))) {
                   PRINTF ("\nUSCSICMD command succeeded.\n");
                   if (ubuf_fg)
                      DUMP_BYTES ((char *)&uscsi_buf, (uscsi_cmd.uscsi_buflen - uscsi_cmd.uscsi_resid));
                   PRINTF ("\nData in struct uscsi_cmd after to issue the cmd:");
                   DUMP_BYTES ((char *)&uscsi_cmd, sizeof(uscsi_cmd));
                }
                else {
                   PRINTF ("\n");
                   PERROR ("USCSICMD command failed");
                   PRINTF ("SCSI statuss returned by the device is %d\n", uscsi_cmd.uscsi_status);
                   PRINTF ("Untransferred data length of the uscsi_cmd data is %d\n", uscsi_cmd.uscsi_resid);
                   PRINTF ("Data in struct uscsi_cmd after to issue the cmd:");
                   DUMP_BYTES ((char *)&uscsi_cmd, sizeof(uscsi_cmd));
                   if (rq_fg) {
                      PRINTF ("\nUntransferred data length of the sense data is %d\n", uscsi_cmd.uscsi_rqresid);
                      PRINTF ("Sense data from the struct uscsi_cmd:\n");
                      DUMP_BYTES ((char *)&rq_buf, uscsi_cmd.uscsi_rqlen);
                  }
                }

                return (rc);
```

Service Aid IOCTL Operations

A set of service aid *ioctl* commands gives applications access to serviceability operations for IBM tape subsystems.

The following commands are supported:

Name	Description
STIOC_DEVICE_SN	Query the device subsystem number. This command is not applicable to Ultrium devices.
IOC_FORCE_DUMP	Force the device to perform a diagnostic dump.
IOC_STORE_DUMP	Force the device to write the diagnostic dump to the currently mounted tape cartridge.
IOC_READ_BUFFER	Read data from the specified device buffer.
IOC_WRITE_BUFFER	Write data to the specified device buffer.

These commands and associated data structures are defined in the *svc.h* header file in the */usr/include/sys* directory, which is installed with the IBMtape package. Any application program that issues these commands must include this header file.

A sample program called *tapeutil.c* is provided with the device driver and installed in the */opt/IBMtape* directory with the IBMtape package. This source code demonstrates the use of these *ioctl* commands.

IOC_FORCE_DUMP

This command forces the device to perform a diagnostic dump.

No data structure is required for this command.

An example of the IOC_FORCE_DUMP command is:

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

if (!(ioctl (dev_fd, IOC_FORCE_DUMP, 0))) {
  printf ("Dump completed successfully.\n");
}

else {
  perror ("Failure performing device dump");
  scsi_request_sense ();
}
```

IOC_STORE_DUMP

This command forces the device to write the diagnostic dump to the currently mounted tape cartridge.

No data structure is required for this command.

An example of the IOC_STORE_DUMP command is:

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

if (!(ioctl (dev_fd, IOC_STORE_DUMP, 0))) {
  printf ("Dump store on tape successfully.\n");
}
```

```
else {
  perror ("Failure storing dump on tape");
  scsi_request_sense ();
}
```

IOC_READ_BUFFER

This command reads data from the specified device buffer.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar mode;                          /* transfer mode */
  uchar id;                            /* device buffer id */
  uint offset;                         /* buffer offset */
  uint size;                           /* byte count */
  uchar *buffer;                       /* data buffer */
} buffer_io_t;
```

The *mode* field should be set to one of the following values:

Value	Description
VEND_MODE	Vendor specific mode
DSCR_MODE	Descriptor mode
DNLD_MODE	Download mode

The *id* field should be set to one of the following values:

Value	Description
ERROR_ID	Diagnostic dump buffer
UCODE_ID	Microcode buffer

An example of the IOC_READ_BUFFER command is:

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

buffer_io_t buffer_io;

if (!(ioctl (dev_fd, IOC_READ_BUFFER, &buffer_io))) {
  printf ("Buffer read successfully.\n");
}

else {
  perror ("Failure reading buffer");
  scsi_request_sense ();
}
```

IOC_WRITE_BUFFER

This command writes data to the specified device buffer.

The following data structure is filled out and supplied by the caller:

```
typedef struct {
  uchar mode;                          /* transfer mode */
  uchar id;                            /* device buffer id */
  uint offset;                         /* buffer offset */
  uint size;                           /* byte count */
  uchar *buffer;                       /* data buffer */
} buffer_io_t;
```

The *mode* field should be set to one of the following values:

Value	Description
VEND_MODE	Vendor specific mode
DSCR_MODE	Descriptor mode
DNLD_MODE	Download mode

The *id* field should be set to one of the following values:

Value	Description
ERROR_ID	Diagnostic dump buffer
UCODE_ID	Microcode buffer

An example of the IOC_WRITE_BUFFER command is:

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

buffer_io_t buffer_io;

if (!(ioctl (dev_fd, IOC_WRITE_BUFFER, &buffer_io))) {
  printf ("Buffer written successfully.\n");
}

else {
  perror ("Failure writing buffer");
  scsi_request_sense ();
}
```

Chapter 18. Return Codes

The calls to the IBMtape device driver will return error codes that describe the outcome of the call. The error codes returned are defined in the *errno.h* system header file in the */usr/include/sys* directory.

For the *open*, *close*, and *ioctl* calls, the return code of the function call will be either 0 for success, or -1 for failure, in which case the system global variable *errno* will contain the error value. For the *read* and *write* calls, the return code of the function call will contain the actual number of bytes read or written if the operation was successful, or 0 if no data was transferred due to encountering end of file (EOF) or end of tape (EOT). If the read or write operation completely failed, the return code will be -1, and the error value will be stored in the system global variable *errno*.

The error codes returned from IBMtape are described in the following section.

Note: The EIO return code is overloaded. To provide the application with additional information about why this return code occurred, use the IOC_REQUEST_SENSE *ioctl* command to obtain the sense data for the error. The sense data can be used to further evaluate why the error occurred.

General Error Codes

The following codes and their descriptions apply in general to all operations:

Name	Description
[EACCES]	An operation to modify the media was attempted illegally.
[EBADF]	A bad file descriptor was specified for the device.
[EBUSY]	An excessive busy state was encountered for the device.
[ECONNRESET]	A SCSI bus reset was detected by the device.
[EFAULT]	A memory failure occurred due to an invalid pointer or address.
[EINVAL]	The requested operation or specified parameter was invalid.
[EIO]	A general I/O failure occurred for the device.
[ENOMEM]	Insufficient memory was available for an internal operation.
[ENOSPC]	The write operation will exceed the remaining available space.
[ENXIO]	The device was not configured or it is not receiving requests.
[EPROTO]	A SCSI command or data transfer protocol error has occurred.
[ETIMEDOUT]	A SCSI command timed out waiting for the device.

Open Error Codes

The following codes and their descriptions apply to the *open* operation:

Name	Description
[EACCES]	An attempt to open the device for write or append mode failed because the currently mounted tape is write-protected.
[EBUSY]	The device is reserved by another initiator or already opened by another process.
[EINVAL]	The requested operation is not supported, or the specified parameter or flag was invalid.
[EIO]	A general failure occurred during the open operation for the device. (If it was opened with the O_APPEND flag, then the tape is full.)
[ENXIO]	The device was not configured or it is not receiving requests.

Close Error Codes

The following codes and their descriptions apply to the *close* operation:

Name	Description
[EBADF]	A bad file descriptor was specified for the device.
[EIO]	A general failure occurred during the close operation for the device.
[ENXIO]	The device was not configured or it is not receiving requests.

Read Error Codes

The following codes and their descriptions apply to the *read* operation:

Name	Description
[EBADF]	A bad file descriptor was specified for the device.
[EFAULT]	A memory failure occurred due to an invalid pointer or address.
[EINVAL]	The requested operation is not supported, or the specified parameter or flag was invalid.
	The number of bytes requested was not a multiple of the block size for a fixed block transfer.
	The number of bytes requested was greater than the maximum size allowed by the device for variable block transfers.
[EIO]	A SCSI or device failure occurred.
	The physical end of the media was detected.
[ENOMEM]	Insufficient memory was available for an internal operation.

	The number of bytes requested for a variable block transfer was less than the size of the block (overlength condition).
[ENXIO]	The device was not configured or it is not receiving requests.
	A read operation was attempted after the device reached the logical end of the media.

Write Error Codes

The following codes and their descriptions apply to the *write* operation:

Name	Description
[EACCES]	An operation to modify the media was attempted on a write-protected tape.
[EBADF]	A bad file descriptor was specified for the device.
[EFAULT]	A memory failure occurred due to an invalid pointer or address.
[EINVAL]	The requested operation is not supported, or the specified parameter or flag was invalid.
	The number of bytes requested was not a multiple of the block size for a fixed block transfer.
	The number of bytes requested was greater than the maximum size allowed by the device for variable block transfers.
	A write operation was attempted on a device that has been opened for O_RDONLY.
[EIO]	A SCSI or device failure occurred.
	The physical end of the media was detected.
[ENOMEM]	Insufficient memory was available for an internal operation.
[ENOSPC]	The write operation failed because the logical end of the media was encountered while trailer label mode was not enabled and early warning (0 return code) was already provided.
[ENXIO]	The device was not configured or it is not receiving requests.
	A write operation was attempted after the device reached the logical end of the media.

IOCTL Error Codes

The following codes and their descriptions apply to the *ioctl* operations:

Name	Description
[EACCES]	An operation to modify the media was attempted on a write-protected tape or on a device opened for read only.

[EBADF]	A bad file descriptor was specified for the device.
[EFAULT]	A memory failure occurred due to an invalid pointer or address.
[EINVAL]	The requested operation is not supported, or the specified parameter or combination of parameters was invalid.
[EIO]	A general failure occurred for the device.
[ENXIO]	The device was not configured or it is not receiving requests.

Opening a Special File

The *open()* system call provides the mechanism for starting an I/O session with a tape drive or medium changer. An example of opening the *0st* tape drive using the *fd* file descriptor is:

```
fd = open ("/dev/rmt/0st", O_FLAGS);
```

If the open() system call fails, it will return -1, and the system *errno* value will contain the error code as defined in the */usr/include/sys/errno.h* header file.

The O_FLAGS parameters specify the options that affect the characteristics of the opened device or the results of the open operation. The O_FLAGS parameters (which can be specified in combinations) are defined in the *fcntl.h* system header file in the */usr/include/sys* directory. The IBMtape device driver special files recognize and support the following O_FLAG values:

- O_RDONLY

  This flag allows only operations that do not alter the content of the tape. All special files support this flag.
- O_RDWR

  This flag allows the tape to be accessed and altered completely. The *smc* special file does not support this flag. An open() call to the *smc* special file or to any *st* special file where the tape device has a write-protected cartridge mounted will fail.
- O_WRONLY

  This flag does not allow the tape to be read. All other tape operations are allowed. The *smc* special file does not support this flag. An open() call to the *smc* special file or to any *st* special file where the tape device has a write-protected cartridge mounted will fail.
- O_NDELAY or O_NONBLOCK

  These two flags perform the same function. This option indicates to the driver not to wait until the tape drive is ready before opening the device and sending commands. If the drive is not ready, then subsequent commands that require a physical tape to be loaded and ready will fail. Other commands that do not require a tape to be loaded, such as inquiry or move medium commands, will complete successfully. All special files support these flags.
- O_APPEND

  This flag is used in conjunction with the O_WRONLY flag to append data to the end of the current data on the tape. This flag is illegal in combination with the O_RDONLY or O_RDWR flag. The *smc* special file does not support this flag. An open() call to the *smc* special file or to any *st* special file where the tape device has a write-protected cartridge mounted will fail.

During an open for append operation, the tape is positioned after the last block or filemark that was written to the tape. This process can take several minutes to complete for a full tape.

Writing to a Special File

The *write()* system call provides the mechanism for writing data to a tape. This call is not applicable to the *smc* special file and will fail. An example of writing to a tape drive is:

```
count = write (fd, buffer, numbytes);
```

The write operation returns the actual *count* of bytes written during the operation. It can be less than the value in *numbytes*.

If *count* is zero, then the application should assume that logical end of tape (EOT) has been encountered.

If *count* is less than zero, then the *errno* system variable contains the error code returned from the driver.

If the device block size is fixed (block_size≠0), then the *numbytes* value must be a multiple of this block size.

If the block size is variable (block_size>0), then the value specified in *numbytes* is the size of the block written.

The *writev()* system call is also supported.

Reading from a Special File

The *read()* system call provides the mechanism for reading data from a tape. This call is not applicable to the *smc* special file and will fail. An example of reading from a tape drive is:

```
count = read (fd, buffer, numbytes);
```

The read operation returns the actual *count* of bytes read during the operation. It can be less than the value in *numbytes*.

If *count* is zero, then the application should assume that logical end of tape (EOT) or the end of the current file (EOF) has been encountered. When a filemark is encountered, the tape will be left positioned to the EOT side (following) the file mark.

If *count* is less than zero, then the *errno* system variable contains the error code returned from the driver.

If the device block size is fixed (block_size≠0), then the *numbytes* value must be a multiple of the block size.

If the block size is variable (block_size>0), then the value specified in *numbytes* is the number of bytes the IBMtape device driver will attempt to read. If the blocks read from the tape are smaller than the requested value, then the maximum number of bytes returned is the size of one block. In this scenario, the device raises a check condition for an underlength read condition. The device driver must respond to this with error recovery procedures that can potentially degrade read

performance. Enabling the SILI mode (suppress illegal length indication) prevents the tape device from raising a check condition for underlength reads. This reduces the device driver overhead and in some cases improves the read performance. The IBMtape device driver always returns the actual number of bytes read through *count*, regardless of the SILI mode. If the blocks read from the tape are greater than the requested value, then the device will raise a check condition for an overlength error, and the device driver will return -1 with *errno* set to ENOMEM.

The *readv()* system call is also supported.

Closing a Special File

The *close()* system call provides the mechanism for ending an I/O session with a tape drive or medium changer. Closing a device special file is a simple process. The file descriptor that is returned from the *open()* system call is supplied to the *close()* system call as in the following example:

```
rc = close (fd);
```

An application should explicitly issue the close() call when the I/O resource is no longer necessary or in preparation for termination. The operating system will implicitly issue the close() call for an application that terminates without closing the resource itself. If an application terminates unexpectedly but leaves behind child processes that had inherited the file descriptor for the open resource, the operating system will not implicitly close the file descriptor because it believes it is still in use.

If the close() system call fails, it will return -1, and the system *errno* value will contain the error code as defined in the */usr/include/sys/errno.h* header file. The close() operation attempts to perform as many of the necessary tasks as possible even if there are failures during portions of the close operation. The IBMtape device driver is guaranteed to leave the device instance in the closed mode providing that the close() system call is in fact called either explicitly or implicitly. If the close() system call returns with a -1, assume that the device is indeed closed and that another open is required to continue processing the tape. After a close() failure, assume that the tape position may be inconsistent.

The close operation behavior depends on which special file was used during the open operation and which tape operation was last performed while it was opened. The commands are issued to the tape drive during the close operation according to the following logic:

```
if last operation was WRITE FILEMARK
  WRITE FILEMARK
  BACKWARD SPACE 1 FILEMARK

if last operation was WRITE
  WRITE FILEMARK
  WRITE FILEMARK
  BACKWARD SPACE 1 FILEMARK

if last operation was READ
  if special file is NOT BSD
    if EOF was encountered
      FORWARD SPACE 1 FILEMARK

SYNC BUFFER

if special file is REWIND ON CLOSE
  REWIND
```

Issuing IOCTL Operations to a Special File

The *ioctl()* system call provides the mechanism for performing special I/O control operations to the tape drive or medium changer device. An example of issuing an *ioctl* to a tape drive or medium changer device is:

```
rc = ioctl (fd, command, buffer);
```

The *fd* is the file descriptor returned from the *open()* system call. The *command* is the value of the IOCTL operation defined in the appropriate header file, and *buffer* is the address of the user memory where data is passed to the device driver and returned to the application.

The *rc* indicates the outcome of the operation upon return. An *rc* of 0 indicates success. Any other value indicates a failure as defined in the */usr/include/sys/errno.h* header file.

The *ioctl* operations supported by the IBM SCSI Tape and Medium Changer Device Driver for Solaris are defined in the following header files included with the IBMtape package and installed in the */usr/include/sys* subdirectory. These header files should be included by any application source files requiring to access the *ioctl* functions supported by the IBMtape device driver. (Existing applications that make use of the standard Solaris tape drive *ioctl*operations defined in the native *mtio.h* header file in the */usr/include/sys* are fully supported by the IBMtape device driver.)

- st.h (tape drive operations)
- smc.h (medium changer operations)
- svc.h (service aid operations)

Part 5. Windows Tape Device Drivers

Chapter 19. Windows NT® Programming Interface

Two device drivers support the IBM family of Ultrium devices:

- *Ibmtape.sys* supports the IBM 3580 Ultrium Tape Drive.
- *ibmchgr.sys* provides changer support for the IBM 3581 Ultrium Tape Autoloader, the IBM 3583 Scalable Ultrium Tape Library, and the IBM 3584 UltraScalable Tape Library.

The programming interface conforms to the standard Microsoft® Windows NT tape device driver interface. It is detailed in the Microsoft Developer Network (MSDN) Software Development Kit (SDK) and Driver Development Kit (DDK).

The following user-callable tape driver entry points are supported under *Ibmtape.sys*:

- CreateFile()
- CloseHandle()
- ReadFile()
- WriteFile()
- WriteTapemark()
- SetTapePosition()
- GetTapePosition()
- SetTapeParameters()
- GetTapeParameters()
- PrepareTape()
- EraseTape()
- DeviceIoControl()

The following user-callable tape driver entry points are supported under *Ibmchgr.sys*:

- CreateFile()
- CloseHandle()
- DeviceIoControl()

Users who want to write application programs to issue commands to IBM Ultrium device drivers should obtain a license to the Microsoft Developer Network and the Microsoft Visual C++ Compiler. Users also need access to IBM hardware reference manuals for IBM Ultrium devices.

User programs that access the IBM Magstar® device driver should do the following:

1. Add the following lines to *ntddtape.h*:

```
#define IOCTL_TAPE_OBTAIN_SENSE                \
        CTL_CODE(IOCTL_TAPE_BASE, 0x0819,      \
        METHOD_BUFFERED, FILE_READ_ACCESS)
#define IOCTL_TAPE_OBTAIN_VERSION              \
        CTL_CODE(IOCTL_TAPE_BASE, 0x081a,      \
        METHOD_BUFFERED, FILE_READ_ACCESS)
#define IOCTL_TAPE_LOG_SELECT                  \
        CTL_CODE(IOCTL_TAPE_BASE, 0x081c,      \
        METHOD_BUFFERED, FILE_READ_ACCESS) FILE_WRITE_ACCESS
#define IOCTL_TAPE_LOG_SENSE                   \
        CTL_CODE(IOCTL_TAPE_BASE, 0x081d,      \
        METHOD_BUFFERED, FILE_READ_ACCESS)
```

```
#define IOCTL_TAPE_REPORT_MEDIA_DENSITY        \
        CTL_CODE(IOCTL_TAPE_BASE, 0x081e,      \
        METHOD_BUFFERED, FILE_READ_ACCESS)
```

2. Include the following files in your application:

```
#include <ntddscsi.h>
#include <ntddtape.h> /* Modified as indicated above */
```

CreateFile

The *CreateFile* entry point is called to make the driver and device ready for I/O. Only one *CreateFile* at a time is allowed on an IBM 3580 device. Additional opens of the same LUN on a device will fail.

The following code fragment illustrates a call to the *CreateFile* routine:

```
HANDLE ddHandle0; //file handle for the tape device

/*
** Open for Reading/Writing,
**The device special file name is in the form of tapex and
** x is an integer from 0 to n - can be determined from Registry
*/
ddHandle0 = CreateFile("\\\\.\\tape0",
                       DWORD dwDesiredAccess,
                       DWORD dwShareMode,
                       LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                       DWORD dwCreationDistribution,
                       DWORD dwFlagsAndAttributes,
                       HANDLE hTemplateFile
                      );
  if(ddHandle0 == INVALID_HANDLE_VALUE)
  /* Error Handling routine */
```

CloseHandle

The *CloseHandle* entry point is called to terminate I/O to the driver and device.

The following code fragment illustrates a call to the *CloseHandle* routine:

```
BOOL rc;

rc = CloseHandle(
                 ddHandle0
                );
if (!rc)
{
   printf("close failed\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *ddHandle0* is the open file handle returned by the *CreateFile* call.

Variable and Fixed Block Read Write Processing

In Windows NT, tape APIs can be configured to manipulate tapes that use either fixed block size or variable block size.

If variable block size is desired, the block size must be set for variable size. The SetTapeParameters function must be called specifying the SET_TAPE_MEDIA_INFORMATION operation. The function requires the use of a TAPE_SET_MEDIA_PARAMETERS structure. The BlockSize member of the

structure must be set to the desired block size. A block size of 0 indicates variable block size. Any other value sets the media parameters to fixed block size equal to the BlockSize member.

In fixed block mode, the size of all data buffers used for reading and writing must be a multiple of the block size. To determine the fixed block size, the GetTapeParameters function must be used. Specifying the GET_TAPE_MEDIA_INFORMATION operation will yield a TAPE_GET_MEDIA_PARAMETERS structure. The BlockSize member of this structure reports the block size of the tape. The size of buffers used in read and write operations must be a multiple of the block size. This mode allows multiple blocks to be transferred in a single operation. In fixed block mode, transfer of odd block sizes (for example, 999 bytes) is not supported.

When reading or writing variable-sized blocks, the operation may not exceed the transfer length of the Host Bus Adapter. This length is the length of each transfer page (typically 4K) times the number of transfer pages (the scatter-gather variable, typically 16-17). Thus, the typical maximum transfer length for variable-sized transfers is 64K. This may be modified by changing the scatter-gather variable in the system registry, but this is not recommended because it uses up scarce system resources.

Reading a tape containing variable sized blocks can be accomplished even without knowing what size the blocks are. If a buffer is large enough to read the data in a block, then the data will be read without any errors. If the buffer is larger than a block, then only data in a single block will be read, and the tape will be advanced to the next block. The size of the block is returned by the read operation in the *pBytesRead parameter. If, on the other hand, a data buffer is too small to contain all of the data in a block, then two things occur. First, the data buffer will contain data from the tape, but the read operation will fail and GetLastError will return ERROR_MORE_DATA. This error value indicates that there is more data in the block to be read. Second, the tape is advanced to the next block. To reread the previous block, the tape must be repositioned to the desired block, and a larger buffer must be specified. It is best to specify as large a buffer as possible so that this does not occur.

If a tape has fixed size blocks, but the tape media parameters are set to variable block size, then no assumptions are made regarding the size of the blocks on the tape. Each read operation behaves as described above. The size of the blocks on the tape are treated as variable, but happen to be the same size.

If a tape has variable size blocks, but the tape media parameters are set to fixed block size, then the size of all blocks on the tape are expected to be the same fixed size. Reading a block of a tape in this situation will fail and GetLastError will return ERROR_INVALID_BLOCK_LENGTH. The only exception to this is if the block size in the media parameters is the same as the size of the variable block and the size of the read buffer happens to be a multiple of the size of the variable block.

If ReadFile encounters a tapemark, the data up to the tapemark is read, and the function fails. (The GetLastError function returns an error code indicating that a tapemark was encountered.) The tape will be positioned past the tapemark, and an application can call ReadFile again to continue reading.

ReadFile

The *ReadFile* entry point is called to read data from tape. The caller provides a buffer address and length, and the driver returns data from the tape to the buffer. The amount of data returned never exceeds the length parameter.

The following code fragment illustrates a *ReadFile* call to the driver:

```
BOOL rc;

rc = ReadFile(
              HANDLE hFile,
              LPVOID lpBuffer,
              DWORD  nBufferSize,
              LPDWORD lpBytesRead,
              LPOVERLAPPED lpOverlapped
             );
 if(rc)
{
  if (*lpBytesRead > 0)
    printf("Read %d bytes\n", *lpBytesRead);
  else
    printf("Read found file mark\n");
}
else
{
   printf("Error on read\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hFile* is the open file handle, *lpBuffer* is the address of a buffer in which to place the data, *nBufferSize* is the number of bytes to be read, and *lpBytesRead* is the number of bytes read.

If the function succeeds, the return value *rc* is nonzero.

WriteFile

The *WriteFile* entry point is called to write data to the tape. The caller provides the address and length of the buffer to be written to tape. The physical limitations of the drive can cause the write to fail. An example of this is attempting to write past the physical end of the tape.

The following code fragment illustrates a call to the *WriteFile* routine:

```
BOOL rc;

rc = WriteFile(
               HANDLE hFile,
               LPCVOID lpBuffer,
               DWORD nBufferSize,
               LPDWORD lpNumberOfBytesWritten,
               LPOVERLAPPED lpOverlapped
              );
if (!rc)
{
   printf("Error on write\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hFile* is the open file handle, *lpBuffer* is the buffer address, and *nBufferSize* is the size of the buffer in bytes.

If the function succeeds, the return value *rc* is nonzero. The application should also verify that all the requested data was written by examining the *lpNumberOfBytesWritten* parameter. See “WriteTapemark” for details on committing data on the media.

WriteTapemark

The *WriteTapemark* entry point is called to write a tape mark on the media. The *WriteTapemark* entry point can also be used to ensure that data is flushed from the IBM 3580 device buffer onto the media.

The following code fragment illustrates a call to the *WriteTapemark* routine:

```
DWORD rc;

rc = WriteTapemark(
                    HANDLE hDevice,
                    DWORD dwTapemarkType,
                    DWORD dwTapemarkCount,
                    BOOL bImmediate
                   );
if (rc)
{
   printf("Error on WriteTapemark\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle, *dwTapemarkType* is the type of operation requested (IBM 3580 supports only TAPE_FILEMARKS), and *dwTapemarkCount* is the number of tapemarks to write.

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

Application writers who are using the *WriteFile* entry point to write data to tape should understand that the IBM 3580 device buffers data in memory and writes that data to the media as those device buffers fill. Thus, a *WriteFile* call may return a successful return code, but the data may not be on the media yet. Calling the *WriteTapemark* entry point and receiving a good return code, however, will ensure that data has been committed properly to tape media (if all previous *WriteFile* calls were successful).

However, applications writing large amounts of data to tape may not want to wait until writing a tapemark to know whether previous data was written properly to the media. The *WriteTapemark* entry point may also be called with dwTapemarkCount parameter set to 0 and bImmediate parameter set to FALSE; this will have the effect of committed any uncommitted data written by previous *WriteFile* calls (since the last call to *WriteTapemark*) to the media. If no error has been returned by the *WriteFile* calls and the *WriteTapemark* call, the application can assume that all data is committed successfully to the media.

SetTapePosition

The *SetTapePosition* entry point is called to set the tape position.

The following code fragment illustrates a call to the *SetTapePosition* routine:

```
DWORD rc;

rc = SetTapePosition(
                     HANDLE hDevice,
                     DWORD dwPositionMethod,
                     DWORD dwPartition,
                     DWORD dwOffsetLow,
                     DWORD dwOffsetHigh,
                     BOOL bImmediate
                    );
if (rc)
{
   printf("Error on SetTapePosition\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle, *dwPositionMethod* is the type of positioning (only TAPE_LOGICAL_BLOCK, TAPE_REWIND, TAPE_SPACE_FILEMARKS, TAPE_SPACE_RELATIVE_BLOCKS, and TAPE_SPACE_END_OF_DATA are supported), and *dwOffsetLow/High* is the block address or count for the position method.

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

GetTapePosition

The *GetTapePosition* entry point is called to get the tape position.

The following code fragment illustrates a call to the *GetTapePosition* routine:

```
DWORD rc;

rc = GetTapePosition(
                     HANDLE hDevice,
                     DWORD dwPositionType,
                     LPDWORD lpdwPartition,
                     LPDWORD lpdwOffsetLow,
                     LPDWORD lpdwOffsetHigh
                    );
if (rc)
{
   printf("Error on GetTapePosition\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle, *dwPositionType* is the type of positioning (only TAPE_ABSOLUTE_POSITION is supported), and *lpdwOffsetLow/High* is the address of the variable that receives the current position.

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

SetTapeParameters

The *SetTapeParameters* entry point is called to either specify the block size of a tape or set tape device data compression.

The data structures are:

```
struct{ // structure used by operation SET_TAPE_MEDIA_INFORMATION
        ULONG BlockSize;
      }TAPE_SET_MEDIA_PARAMETERS;

struct{ // structure used by operation SET_TAPE_DRIVE_INFORMATION
        BOOLEAN ECC;                  // Ignored
        BOOLEAN Compression;          // Only compression can be set for Ultrium
        BOOLEAN DataPadding;          // Ignored
        BOOLEAN ReportSetmarks;       // Ignored
        ULONG   EOTWarningZoneSize;   // Ignored
      }TAPE_SET_DRIVE_PARAMETERS;
```

The following code fragment illustrates a call to the *SetTapeParameters* routine:

```
DWORD rc;

rc = SetTapeParameters(
                        HANDLE hDevice,
                        DWORD dwOperation,
                        LPVOID lpParameters
                       );
if (rc)
{
   printf("Error on SetTapeParameters\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle, *dwOperation* is the type of information to set (SET_TAPE_MEDIA_INFORMATION or SET_TAPE_DRIVE_INFORMATION), and *lpParameters* is the address of the data structure that contains the parameters (for SET_TAPE_DRIVE_INFORMATION, only compression can be changed).

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

GetTapeParameters

The *GetTapeParameters* entry point is called to get information that describes the tape or the tape driver.

The data structures are:

```
struct{ // structure used by GET_TAPE_MEDIA_INFORMATION
        LARGE_INTEGER   Capacity;
        LARGE_INTEGER   Remaining;
        DWORD           BlockSize;
        DWORD           PartitionCount;
        BOOLEAN         WriteProtected;
      }TAPE_GET_MEDIA_PARAMETERS;

struct{ // structure used by GET_TAPE_DRIVE_INFORMATION
        BOOLEAN ECC;
        BOOLEAN Compression;
        BOOLEAN DataPadding;
        BOOLEAN ReportSetmarks;
        ULONG   DefaultBlockSize;
        ULONG   MaximumBlockSize;
        ULONG   MinimumBlockSize;
```

```
        ULONG   MaximumPartitionCount;
        ULONG   FeaturesLow;
        ULONG   FeaturesHigh;
        ULONG   EOTWarningZoneSize;
      }TAPE_GET_DRIVE_PARAMETERS;
```

The following code fragment illustrates a call to the *GetTapeParameters* routine:

```
DWORD rc;

rc = GetTapeParameters(
                       HANDLE hDevice,
                       DWORD dwOperation,
                       LPDWORD lpdwSize,
                       LPVOID lpParameters
                      );
if (rc)
{
   printf("Error on GetTapeParameters\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle, *dwOperation* is the type of information requested (GET_TAPE_MEDIA_INFORMATION or GET_TAPE_DRIVE_INFORMATION), and *lpParameters* is the address of the returned data parameter structure.

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

PrepareTape

The *PrepareTape* entry point is called to either prepare the tape for access or removal.

The following code fragment illustrates a call to the *PrepareTape* routine:

```
DWORD rc;

rc = PrepareTape(
                 HANDLE hDevice,
                 DWORD dwOperation,
                 BOOL bImmediate
                );
if (rc)
{
   printf("Error on PrepareTape\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle and *dwOperation* is the type of operation requested.

EraseTape

The *EraseTape* entry point is called to erase all or a part of a tape. The erase is performed from the current location to the end of media, then the media is positioned to the location where the erase was started.

The following code fragment illustrates a call to the *EraseTape* routine:

```
DWORD rc;

rc = EraseTape(
                 HANDLE hDevice,
                 DWORD dwEraseType,
                 BOOL bImmediate
                );
if (rc)
{
   printf("Error on EraseTape\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle and *dwEraseType* is always TAPE_ERASE_LONG.

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

DeviceIoControl

The Windows NT software supports all the *DeviceIoControl* commands supported by the Windows NT native driver. The function is described in the Microsoft Windows NT Microsoft Developer Network (MSDN) Software Developer Kit (SDK) and Driver Development Kit (DDK).

The DeviceIoControl function sends a control code directly to a specified device driver, causing the corresponding device to perform the specified operation.

An example of SCSI Pass Through is contained in SPTI.C in the DDK.

The function call is described in the SDK, and examples of its use are shown in the DDK.

```
BOOL DeviceIoControl(
    HANDLE hDevice,           // handle to device of interest
    DWORD dwIoControlCode,    // control code of operation to perform
    LPVOID lpInBuffer,        // pointer to buffer to supply input data
    DWORD nInBufferSize,      // size of input buffer
    LPVOID lpOutBuffer,       // pointer to buffer to receive output data
    DWORD nOutBufferSize,     // size of output buffer
    LPDWORD lpBytesReturned,  // pointer to variable to receive output byte count
    LPOVERLAPPED lpOverlapped // pointer to overlapped structure
                              // for asynchronous operation
   );
```

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError().

Following is a list of some of the dwIoControlCode codes that are described in the DDK:

- IOCTL_SCSI_GET_ADDRESS
- IOCTL_SCSI_GET_CAPABILITIES
- IOCTL_SCSI_PASS_THROUGH
- IOCTL_SCSI_PASS_THROUGH_DIRECT

Device IOCTLs for DeviceIoControl

IOCTL_TAPE_OBTAIN_SENSE

Issue this command after an error occurs to obtain sense information associated with the error that occurred most recently. To guarantee that the application can obtain sense information associated with an error, the application should issue this command before issuing any other commands to the device.

The following output structure is filled in by the IOCTL_TAPE_OBTAIN_SENSE command passed by the caller:

```
#define SENSE_BUFFER_SIZE    36     /* Default request sense buffer size */

typedef struct _TAPE_OBTAIN_SENSE {
    ULONG   SenseDataLength;        /* Output is the number of bytes  */
                                    /* of valid sense data.           */
                                    /* Will be zero if no error with  /*
                                    /* sense data has occurred.       */
                                    /* The only sense data available  */
                                    /* is that of the last error.     */

    CHAR    SenseData[SENSE_BUFFER_SIZE];
} TAPE_OBTAIN_SENSE, *PTAPE_OBTAIN_SENSE;
```

An example of the IOCTL_TAPE_OBTAIN_SENSE command is:

```
int   *rc_ptr;
DWORD cb;
TAPE_OBTAIN_SENSE sense_data;

    *rc_ptr = DeviceIoControl(hDevice,
                          IOCTL_TAPE_OBTAIN_SENSE,
                          NULL,
```

IOCTL_TAPE_OBTAIN_VERSION

Issue this command to obtain the version of the device driver. It is in the form of a null-terminated string.

The following output structure is filled in by the IOCTL_TAPE_OBTAIN_VERSION command:

```
#define MAX_DRIVER_VERSIONID_LENGTH     12

typedef struct _TAPE_OBTAIN_VERSION {
    CHAR    VersionId[MAX_DRIVER_VERSIONID_LENGTH];
} TAPE_OBTAIN_VERSION, *PTAPE_OBTAIN_VERSION;
```

An example of the IOCTL_TAPE_OBTAIN_VERSION command is:

```
int   *rc_ptr
DWORD cb;
TAPE_OBTAIN_VERSION code_version;

    *rc_ptr = DeviceIoControl(hDevice,
                          IOCTL_TAPE_OBTAIN_VERSION,
                          NULL,
                          0,
                          &code_version,
                          (long)sizeof(TAPE_OBTAIN_VERSION),
                          &cb,
                          (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_LOG_SELECT

Issue this command to reset the log pages.

```
int *rc_ptr:
DWORD cb;

  *rc_ptr = DeviceIoControl ( hDevice,
                              IOCTL_TAPE_LOG_SELECT,
                              NULL,
                              0,
                              NULL,
                              0,
                              &CB,
                              (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_LOG_SENSE

Issue this command to obtain the log data of the requested log page from the IBM 3580 tape device. The data returned is formatted according to the IBM 3580 hardware reference manual. Table 4 lists the IBM 3580 devices supporting log pages.

Table 4. 3580 Devices Log Pages

Log Page	Content
00h	List of the supported log pages
02h	Write Error Counters Log
03h	Read Error Counters Log
0Ch	Sequential Access Dvice Log
2Eh	TapeAlert Log
30h	Tape Usage Log
31h	Tape Capacity Log
32h	Data Compression Log

The following input/output structure is used by the IOCTL_TAPE_LOG_SENSE command:

```
#define MAX_LOG_SENSE 1024  /* Maximum number of bytes the command */
                            /* will return */

typedef struct _TAPE_LOG_SENSE_PARAMETERS{
 UCHAR PageCode;         // The requested log page code
 UCHAR PC;               // PC = 0 for maximum values, 1 for current value, 3 for power-on values
 UCHAR PageLength[2];          /* Length of returned data, filled in by the command */
 UCHAR LogData[MAX_LOG_SENSE]; /* Log data, filled in by the command          */
 } TAPE_LOG_SENSE_PARAMETERS, *PTAPE_LOG_SENSE_PARAMETERS;
```

An example of the IOCTL_TAPE_LOG_SENSE COMMAND is:

```
int   *rc_ptr;
DWORD cb;
TAPE_LOG_SENSE_PARAMETERS logsense;

logsense.PageCode=0;
logsense.PC = 1;

    *rc_ptr = DeviceIoControl ( hDevice,
                                IOCTL_TAPE_LOG_SENSE,
                                &logsense,
                                (long)sizeof(TAPE_LOG_SENSE_PARAMETERS),
                                &logsense,
                                (long)sizeof(TAPE_LOG_SENSE_PARAMETERS),
                                &CB,
                                (LPOVERLAPPED) NULL);
```

```
if(*rc_ptr)
{
printf("Success!\n");

/* Call a function to display argument 2 bytes of argument 1 */
dump_data(logsense.LogData, (logsense.PageLength[0]<<8+logsense.PageLength[1]) );
```

IOCTL_TAPE_REPORT_MEDIA_DENSITY

Issue this command to obtain the media density information on the loaded media in the drive. If there is no media load, the command will fail.

The following output structure is filled in by the IOCTL_TAPE_REPORT_MEDIA_DENSITY command:

```
typedef struct_TAPE_REPORT_DENSITY{
 ULONG PrimaryDensityCode;    /* Primary Density Code */
 ULONG SecondaryDensityCode;  /* Secondary Density Code */
 BOOLEAN WriteOk;             /* 0 = does not support writing in this format */
                              /* 1 = support writing in this format */
 ULONG BitsPerMM;             /* Bits Per mm */
 ULONG MediaWidth;            /* Media Width */
 ULONG Tracks;                /* Tracks */
 ULONG Capacity;              /* Capacity in MegaBytes */
} TAPE_REPORT_DENSITY, *PTAPE_REPORT_DENSITY;
```

An example of the IOCTL_TAPE_REPORT_MEDIA_DENSITY command is:

```
int      *rc_ptr;
DWORD cb;
TAPE_REPORT_DENSITY tape_reportden;

*rc_ptr = DeviceIoControl (hDevice,
                           IOCTL_TAPE_REPORT_MEDIA_DENSITY,
                           NULL,
                           0,
                           &tape_reportden,
                           (long)sizeof(TAPE_REPORT_DENSITY),
                           &cb,
                           (LPOVERLAPPED) NULL);

if(*rc_ptr)
{
 printf("Report Density:\n");
 printf(" PrimaryDensityCode: %d, SecondaryDensityCode: %d \n"),
          tape_reportden.PrimaryDensityCode, tape_reportden.PrimaryDensityCode);
 printf(" Bits pre mm: %d, Media Width: %d, Tracks: %d, Capacity: %d\n"),
          tape_reportden.BidsPerMM, tape_reportden.MediaWidth,
          Tape_reportden.Tracks, Tape_reportden.Capacity);
 printf(" Drive supports writing in this format? (1 = yes, 0 = no) %d\n")
          tape_reportden.WriteOk;
}
else
/* Error Handling Code */
```

User programs that access the IBM Ultrium changer driver should perform the *CreateFile* and *CloseHandle* functions, which follow.

CreateFile

The *CreateFile* entry point is called to make the driver and device ready for I/O. Only one *CreateFile* at a time is allowed on an IBM Ultrium changer device. Additional opens of the same device will fail.

The following code fragment illustrates a call to the *CreateFile* routine:

```
HANDLE ddHandle 1 ; //file handle for the changer device

/*
** Open for media mover operations,
** where the device special file name is in the form of
**    1ba.b.c.d
**            a is the Target address (3 in the following example)
**            b is the LUM (1 in the following example)
**            c is the SCSI Bus (0 in the following example)
**            d is the SCSI Port (2 in the following example)
*/
ddHandle1 = CreateFile ( "\\\\.\\1b3.1.0.2",
                         DWORD dwDesiredAccess
                         DWORD dwShareMode,
                         LPSECURITY_ATTRIBUTES 1pSecurityAttributes,
                         DWORD dwCreationDistribution,
                         DWORD dwFlagsAndAttributes,
                         HANDLE hTemplateFile
                         );
if(ddHandle0) == INVALID_HANDLE_VALUE)
                 /* Error Handling routine*/
```

CloseHandle

The *CloseHandle* entry point is called to terminate I/O to the driver and device.

The following code fragment illustrates a call to the *CloseHandle* routine:

```
BOOL re;
rc = CloseHandle( ddHandle1);
if(!rc)
/* Error Handling routine */
```

where ddHandle1 is the open file handle returned by the *CreatFile* call.

Medium Changer IOCTLs for DeviceIoControl

The following section describes the set of *ioctl*commands (used with the DeviceIoControl()) that provide control and access to the IBM Ultrium medium changer functions of the IBM 3581, 3583, and 3584 devices. If the device is an IBM 3584, the *ioctl*operation can be accessed through LUN1 (see “CreateFile” function description to open the device).

The following *ioctl* commands are supported (through DeviceIoControl()) and the Ibmchgr.sys driver:

```
/*
This macro is defined in ntddk.h and devioctl.h
#define CTL_CODE(DeviceType, Function, Method, Access) \
     (((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
*/

#define IOCTL_BASE   33792
#define LB_ACCESS    FILE_READ_ACCESS | FILE_WRITE_ACCESS
#define M_LBI(x)     CTL_CODE(IOCTL_BASE + 1, x, METHOD_BUFFERED, LB_ACCESS)
```

```
#define M_RETURN_ELEMENT_COUNT (M_LBI(20))     // obtain the library element counts
#define M_LIBRARY_INVENTORY    (M_LBI(21))     // Read Element Status
#define M_MOVE_MEDIUM          (M_LBI(22))     // Move Medium
#define M_LIBRARY_AUDIT        (M_LBI(23))     // Initialize Element Status
```

M_MOVE_MEDIUM

This command transports a cartridge from one element to another.

The following structure is filled out and supplied by the caller:

```
typedef struct
{
  long   compcode;             /* O - request completion code */
  USHORT transport_element;    /* I - transport element address */
  USHORT source_address;       /* I - where we are moving from */
  USHORT destination_address;  /* I - where we are moving to */
  long   reserved;             /* always 0 */
} move_medium_t;
```

An example of the M_MOVE_MEDIUM command is:

```
int    *rc_ptr;
DWORD         cb;
move_medium_t lb_move_medium_data;

memset(&lb_move_medium_data,0x00,sizeof(move_medium_t));

lb_move_medium_data.source_address = 0x32;        // magazine
lb_move_medium_data.destination_address = 0x10;   // drive
lb_move_medium_data.transport_element = 0;        // transport

*rc_ptr = DeviceIoControl(hDevice,
                          M_MOVE_MEDIUM,
                          &lb_move_medium_data,
                          (long)sizeof(move_medium_t),
                          &lb_move_medium_data,
                          (long)sizeof(move_medium_t),
                          &cb,
                          (LPOVERLAPPED) NULL);

if(*rc_ptr)
{
  printf("The M_MOVE_MEDIUM ioctl succeeded.\n");
}
else
{
  printf("The M_MOVE_MEDIUM ioctl failed.
          Last error was %ld\n", GetLastError());
}
```

M_LIBRARY_AUDIT

This command generates an Initialize Element Status command.

The following structure is filled out and supplied by the caller:

```
typedef struct
{
  long compcode;  /* O - request completion code */
  long reserved;  /* must be set to 0 */
}lb_audit_t;
```

An example of the M_LIBRARY_AUDIT command is:

```
int    *rc_ptr;
DWORD      cb;
lb_audit_t lb_audit_data;
```

```
memset(&lb_audit_data,0x00, sizeof(lb_audit_t));

*rc_ptr = DeviceIoControl(hDevice,
                          M_LIBRARY_AUDIT,
                          &lb_audit_data,
                          (long)sizeof(lb_audit_t),
                          &lb_audit_data,
                          (long)sizeof(lb_audit_t),
                          &cb,
                          (LPOVERLAPPED) NULL);

if(*rc_ptr)
{
  printf("The M_LIBRARY_AUDIT ioctl succeeded.\n");
}
else
{
  printf("The M_LIBRARY_AUDIT ioctl failed.
         Last error was %ld\n", GetLastError());
}
```

M_RETURN_ELEMENT_COUNT

This command returns a count of the various elements.

The following structure is filled out and supplied by the caller:

```
typedef struct
{
  long compcode;        /* 0 - request completion code */
  long drives;          /* 0 - number of drives in the device */
  long slots;           /* 0 - number of storage slots in the device */
  long transp_elems;    /* 0 - number of transport elems in the device */
  long eeports;         /* 0 - number of entry exit ports in the device */
}element_count_t;
```

An example of the M_LIBRARY_AUDIT command is:

```
int    *rc_ptr;
DWORD            cb;
element_count_t  lb_element_count_data;

memset(&lb_element_count_data,'\0',sizeof(element_count_t));

*rc_ptr = DeviceIoControl(hDevice,
                          M_RETURN_ELEMENT_COUNT,
                          &lb_element_count_data,
                          (long)sizeof(element_count_t),
                          &lb_element_count_data,
                          (long)sizeof(element_count_t),
                          &cb,
                          (LPOVERLAPPED) NULL);

if(*rc_ptr)
{
  printf("The M_RETURN_ELEMENT_COUNT ioctl succeeded.\n");
}
else
{
  printf("The M_RETURN_ELEMENT_COUNT ioctl failed.
         Last error was %ld\n", GetLastError());
}
```

M_LIBRARY_INVENTORY

This command generates a Read Element Status command.

The following structures are filled out and supplied by the caller:

```
/*
* Data Transfer Element Descriptor (Drives)
*/
typedef struct
{
  /* This entire structure is OUTPUT INFORMATION */
  USHORT address;                  /* Element address of drive */
  long   abnormal_state;           /* 0 if drive is in normal state. */
  UCHAR  accessible;               /* TRUE if accessible, FALSE otherwise */
  UCHAR  full;                     /* TRUE if it contains media. FALSE otherwise */
  UCHAR  reserved;
  UCHAR  reserved;
  UCHAR  reserved;
  UCHAR  reserved;
  UCHAR  reserved;
  UCHAR  reserved;
  USHORT source_elem_addr;         /* source storage element address */
  UCHAR  reserved;
  UCHAR  barcode_flag;             /* TRUE if volume tag(barcode) is valid */
  long   barcode_len;              /* Length in bytes of valid barcode information */
  UCHAR  barcode[LB_BARCODE_LEN];  /* barcode label info */
} drive_state_t;


/*
* Storage Element(cartridge slot) Descriptor
*/
typedef struct
{
  /* This entire structure is OUTPUT INFORMATION */
  USHORT address;                  /* Element address of storage slot */
  long   abnormal_state;           /* 0 if slot is in normal state. */
  UCHAR  accessible;               /* TRUE if accessible, FALSE otherwise */
  UCHAR  full;                     /* TRUE if it contains media. FALSE otherwise */
  UCHAR  reserved;
  USHORT source_elem_addr;         /* source storage element address */
  UCHAR  reserved;
  UCHAR  barcode_flag;             /* TRUE if volume tag(barcode) is valid */
  long   barcode_len;              /* Length in bytes of valid barcode information */
  UCHAR  barcode[LB_BARCODE_LE];  /* barcode label info */
} storage_element_state_t;
/*
* Import/Export Element Descriptor
*/
typedef struct
{
  /* This entire structure is OUTPUT INFORMATION */
  USHORT address;             /* element address of entry exit port */
  UCHAR  import_enable;       /* TRUE if data cartridge can move INTO the device */
  UCHAR  export_enable;       /* TRUE if data cartridge can move OUT from device */
  long   abnormal_state;      /* 0 if I/E is in normal state. */
  UCHAR  accessible;          /* TRUE if accessible */
  UCHAR  full;                /* TRUE if it contains media. FALSE otherwise */
  UCHAR  oper_placed_media;   /* TRUE if operator placed the media in the report */
  UCHAR  reserved;
  USHORT source_elem_addr;    /* source storage element address */
  UCHAR  reserved;
  UCHAR  barcode_flag;        /* TRUE if volume tag(barcode) is valid */
  long   barcode_len;         /* Length in bytes of valid barcode information */
  UCHAR  barcode[LB_BARCODE_LEN];  /* barcode label info */
} import_export_state_t;
```

```
 /*
** Medium changer (transport) Element Descriptor
*/

typedef struct
{
  /* This entire structure is OUTPUT INFORMATION */
  USHORT address;            /* Element address of the medium changer  */
  long   abnormal_state;     /* 0 if medium changer is in normal state. */
  UCHAR  accessible;         /* TRUE if accessible. */
  UCHAR  full;               /* TRUE if it contains media. FALSE otherwise */
  UCHAR  reserved;
  USHORT source_elem_addr;   /* source storage element address */
  UCHAR  reserved;
  UCHAR  barcode_flag;       /* TRUE if volume tag(barcode) is valid */
  long   barcode_len;        /* Length in bytes of valid barcode information */
  UCHAR  barcode[LB_BARCODE_LEN]; /* barcode label info */
} medium_changer_state_t;

 typedef struct
{
  long                    compcode;              /* O - completion code */
  long                    drives;                /* I - number of drives */
  long                    slots;                 /* I - number of storage slots */
  long                    eeports;               /* I - number of entry exit ports */
  long                    transp_elems;          /* I - number of transport elems */
  drive_state_t           *drive_state;          /* O - Pointer to user buffer */
  storage_element_state_t *storage_element_state; /* O - Pointer to user buffer */
  import_export_state_t   *import_export_state;  /* O - Pointer to user buffer */
  medium_changer_state_t  *medium_changer_state; /* O - Pointer to user buffer */
} lb_inventory_t;
```

An example of the M_LIBRARY_INVENTORY command is:

```
int    *rc_ptr;
DWORD            cb;
lb_inventory_t              lb_inventory_data;
lb_storage_element_state_t  *sp = NULL;
lb_drive_state_t            *dp = NULL;
lb_import_export_state_t    *ip = NULL;
lb_medium_changer_state_t   *tp = NULL;

 memset(&lb_inventory_data, '\0', sizeof(lb_inventory_t));

dp = (lb_drive_state_t *)malloc(sizeof(lb_drive_state_t)
     * lb_inventory_data.drives);
sp = (lb_storage_element_state_t *)malloc(sizeof(lb_storage_element_state_t)
     * lb_inventory_data.slots);
ip = (lb_import_export_state_t *)malloc(sizeof(lb_import_export_state_t)
     * lb_inventory_data.eeports);
tp = (lb_medium_changer_state_t *)malloc(sizeof(lb_medium_changer_state_t)
     * lb_inventory_data.transp_elems);

/* fill in drives buffer ptr */
if(dp != NULL)
{
  lb_inventory_data.drive_state = dp;
  memset(dp, '\0', sizeof(lb_drive_state_t)
     * lb_inventory_data.drives);
}
else
{
  /* handle the error */
  return 0;
}

 /* fill in slot buffer ptr    */
if(sp != NULL)
{
```

```
  lb_inventory_data.storage_element_state = sp;
  memset(sp, '\0', sizeof(lb_storage_element_state_t)
     * lb_inventory_data.slots);
}
else
{
  /* handle the error */
  free(dp);
  return 0;
}

/* fill in import export buffer ptr */
if(ip != NULL)
{
  lb_inventory_data.import_export_state = ip;
  memset(ip, '\0', sizeof(lb_import_export_state_t)
     * lb_inventory_data.eeports );
}
else
{
  /* handle the error */
  free(dp);
  free(sp);
  return 0;
}
/* fill in transport buffer ptr */
if(tp != NULL)
{
  lb_inventory_data.medium_changer_state = tp;
  memset(tp, '\0', sizeof(lb_medium_changer_state_t)
     * lb_inventory_data.transp_elems );
}
else
{
  /* handle the error */
  free(dp);
  free(sp);
  free(ip);
  return 0;
}
/* element counts retrieved via previous IOCTL M_RETURN_ELEMENT_COUNT */
lb_inventory_data.slots        = lb_element_count_data.slots;
lb_inventory_data.drives       = lb_element_count_data.drives;
lb_inventory_data.eeports      = lb_element_count_data.eeports;
lb_inventory_data.transp_elems = lb_element_count_data.transp_elems;
 *rc_ptr = DeviceIoControl(hDevice,
                           M_LIBRARY_INVENTORY,
                           &lb_inventory_data,
                           (long)sizeof(lb_inventory_t),
                           &lb_inventory_data,
                           (long)sizeof(lb_inventory_t),
                           &cb,
                           (LPOVERLAPPED) NULL);

if(*rc_ptr)
{
  printf("The M_LIBRARY_INVENTORY ioctl succeeded.\n");
}
else
{
  printf("The M_LIBRARY_INVENTORY ioctl failed. Last error was %ld\n",
         GetLastError());
}
```

Chapter 20. Windows 2000® Programming Interface

Two device drivers support the IBM family of Ultrium devices. They are:

- *ibmtape.sys*, which supports the IBM Ultrium tape drives.
- *ibmchgr.sys*, which provides changer support for the IBM Ultrium tape libraries.

The programming interface conforms to the standard Microsoft Windows 2000 tape device driver interface. It is detailed in the Microsoft Developer Network (MSDN) Software Development Kit (SDK) and Device Development Kit (DDK).

The following user-callable tape driver entry points are supported under *ibmtape.sys*:

- CreateFile()
- CloseHandle()
- DeviceIoControl()
- EraseTape()
- GetTapeParameters()
- GetTapePosition()
- GetTapeStatus()
- PrepareTape()
- ReadFile()
- SetTapeParameters()
- SetTapePosition()
- WriteFile()
- WriteTapemark()

If the Removable Storage Manager is stopped, then the following user-callable tape media changer driver entry points are supported under *ibmchgr.sys*:

- CreateFile()
- CloseHandle()
- DeviceIoControl()

Users who want to write application programs to issue commands to IBM Ultrium device drivers should obtain a license to the Microsoft Developer Network and the Microsoft Visual C++ Compiler. Users also need access to IBM hardware reference manuals for IBM Ultrium devices.

User programs that access the Ultrium device driver should do the following:

1. Include the following files in your application:

```
#include <ntddscsi.h>
#include <ntddchgr.h>
#include <ntddtape.h> /* Modified as indicated below */
```

2. Add the following lines to ntddtape.h:

```
#define IOCTL_TAPE_OBTAIN_SENSE CTL_CODE(IOCTL_TAPE_BASE, 0x0819,           \
                    METHOD_BUFFERED, FILE_READ_ACCESS)
#define IOCTL_TAPE_OBTAIN_VERSION CTL_CODE(IOCTL_TAPE_BASE, 0x081a,         \
                    METHOD_BUFFERED, FILE_READ_ACCESS)
#define IOCTL_TAPE_LOG_SELECT CTL_CODE(IOCTL_TAPE_BASE, 0x081c,             \
                    METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS)
#define IOCTL_TAPE_LOG_SENSE CTL_CODE(IOCTL_TAPE_BASE, 0x081d,              \
```

```
                        METHOD_BUFFERED, FILE_READ_ACCESS)
#define IOCTL_TAPE_REPORT_MEDIA_DENSITY CTL_CODE(IOCTL_TAPE_BASE, 0x081e,    \
                        METHOD_BUFFERED, FILE_READ_ACCESS)
```

Variable and Fixed Block Read Write Processing

In Windows 2000, tape APIs can be configured to manipulate tapes that use either fixed block size or variable block size.

If variable block size is desired, the block size must be set for variable size. The SetTapeParameters function must be called specifying the SET_TAPE_MEDIA_INFORMATION operation. The function requires the use of a TAPE_SET_MEDIA_PARAMETERS structure. The BlockSize member of the structure must be set to the desired block size. A block size of 0 indicates variable block size. Any other value sets the media parameters to fixed block size equal to the BlockSize member.

In fixed block mode, the size of all data buffers used for reading and writing must be a multiple of the block size. To determine the fixed block size, the GetTapeParameters function must be used. Specifying the GET_TAPE_MEDIA_INFORMATION operation yields a TAPE_GET_MEDIA_PARAMETERS structure. The BlockSize member of this structure reports the block size of the tape. The size of buffers used in read and write operations must be a multiple of the block size. This mode allows multiple blocks to be transferred in a single operation. In fixed block mode, transfer of odd block sizes (for example, 999 bytes) are not supported.

When reading or writing variable-sized blocks, the operation may not exceed the transfer length of the Host Bus Adapter. This length is the length of each transfer page (typically 4K) times the number of transfer pages (the scatter-gather variable, typically 16-17). Thus, the typical maximum transfer length for variable-sized transfers is 64K. This may be modified by changing the scatter-gather variable in the system registry, but this is not recommended because it uses up scarce system resources.

Reading a tape containing variable sized blocks can be accomplished even without knowing what size the blocks are. If a buffer is large enough to read the data in a block, then the data will be read without any errors. If the buffer is larger than a block, then only data in a single block will be read, and the tape will be advanced to the next block.

The size of the block is returned by the read operation in the *pBytesRead parameter. If, on the other hand, a data buffer is too small to contain all of the data in a block, then two things occur. First, the data buffer will contain data from the tape, but the read operation will fail and GetLastError will return ERROR_MORE_DATA. This error value indicates that there is more data in the block to be read. Second, the tape is advanced to the next block. To reread the previous block, the tape must be repositioned to the desired block, and a larger buffer must be specified. It is best to specify as large a buffer as possible so that this does not occur.

If a tape has fixed size blocks, but the tape media parameters are set to variable block size, then no assumptions are made regarding the size of the blocks on the tape. Each read operation behaves as described above. The size of the blocks on the tape are treated as variable but happen to be the same size. If a tape has variable size blocks, but the tape media parameters are set to fixed block size, then

the size of all blocks on the tape are expected to be the same fixed size. Reading a block of a tape in this situation will fail and GetLastError will return ERROR_INVALID_BLOCK_LENGTH. The only exception to this is if the block size in the media parameters is the same as the size of the variable block and the size of the read buffer happens to be a multiple of the size of the variable block.

If ReadFile encounters a tapemark, the data up to the tapemark is read and the function fails. (The GetLastError function returns an error code indicating that a tapemark was encountered.) The tape will be positioned past the tapemark, and an application can call ReadFile again to continue reading.

Write Tapemark

```
WriteTapemark(
 HANDLE hDevice,
 DWORD dwTapemarkType,
 DWORD dwTapemarkCount,
 BOOL bImmediate
);
```

dwTapemarkType is the type of operation requested.

The only type supported is:

TAPE_FILEMARKS

Application writers who are using the *WriteFile* entry point to write data to tape should understand that the tape device buffers data in its memory and writes that data to the media as those device buffers fill. Thus, a WriteFile call may return a successful return code, but the data may not be on the media yet. Calling the *WriteTapemark* entry point and receiving a good return code, however, ensures that data has been committed properly to tape media (if all previous *WriteFile* calls were successful). However, applications writing large amounts of data to tape may not want to wait until writing a tapemark to know whether or not previous data was written properly to the media.

The *WriteTapemark* entry point may also be called with the *dwTapemarkCount* parameter set to 0 and the bImmediate parameter set to FALSE; this has the effect of committing any uncommitted data written by previous WriteFile calls (since the last call to *WriteTapemark*) to the media. If no error has been returned by the *WriteFile* calls and the *WriteTapemark* call, the application can assume that all data is committed successfully to the media.

SetTapePosition

The *SetTapePosition* entry point is called to seek to a particular block of media data.

```
SetTapePosition(
 HANDLE hDevice,
 DWORD dwPositionMethod,
 DWORD dwPartition,      // ONLY partition 1 is supported
 DWORD dwOffsetLow,
 DWORD dwOffsetHigh,
 BOOL bImmediate
);
```

dwPositionMethod is the type of positioning.

For IBM Magstar devices the following types of tape marks and immediate values are supported:

TAPE_ABSOLUTE_BLOCK	bImmediate TRUE or FALSE
TAPE_LOGICAL_BLOCK	bImmediate TRUE or FALSE

For IBM Magstar devices, there is no difference between the absolute and logical block addresses.

TAPE_REWIND	bImmediate TRUE or FALSE
TAPE_SPACE_END_OF_DATA	bImmediate FALSE
TAPE_SPACE_FILEMARKS	bImmediate FALSE
TAPE_SPACE_RELATIVE_BLOCKS	bImmediate FALSE
TAPE_SPACE_SEQUENTIAL_FMKS	

GetTapePosition

The *GetTapePosition* entry point is called to retrieve the current tape position.

```
GetTapePosition(
 HANDLE hDevice,
 DWORD dwPositionType,
 LPDWORD lpdwPartition,
 LPDWORD lpdwOffsetLow,
 LPDWORD lpdwOffsetHigh
);
```

dwPositionType is the type of positioning.

TAPE_ABSOLUTE_POSITION or **TAPE_LOGICAL_POSITION** may be specified, but only the absolute position is returned.

SetTapeParameters

The *SetTapeParameters* entry point is called to either specify the block size of a tape or set tape device data compression. The data structures are:

```
struct{ // structure used by operation SET_TAPE_MEDIA_INFORMATION
 ULONG BlockSize;
}TAPE_SET_MEDIA_PARAMETERS;

struct{ // structure used by operation SET_TAPE_DRIVE_INFORMATION
 BOOLEAN ECC;              // Not Supported
 BOOLEAN Compression;      // Only compression can be set
 BOOLEAN DataPadding;      // Not Supported
 BOOLEAN ReportSetmarks;   // Not Supported
 ULONG EOTWarningZoneSize; // Not Supported
}TAPE_SET_DRIVE_PARAMETERS;

SetTapeParameters(
 HANDLE hDevice,
 DWORD dwOperation,
 LPVOID lpParameters
);
```

dwOperation is the type of information to set (SET_TAPE_MEDIA_INFORMATION or SET_TAPE_DRIVE_INFORMATION). For SET_TAPE_DRIVE_INFORMATION, only compression is changeable.

lpParameters is the address of either a TAPE_SET_MEDIA_PARAMETERS or a TAPE_SET_DRIVE_PARAMETERS data structure that contains the parameters.

GetTapeParameters

The *GetTapeParameters* entry point is called to get information that describes the tape or the tape driver.

The data structures are:

```
struct{ // structure used by GET_TAPE_MEDIA_INFORMATION
        LARGE_INTEGER   Capacity;
        LARGE_INTEGER   Remaining;
        DWORD           BlockSize;
        DWORD           PartitionCount;
        BOOLEAN         WriteProtected;
      }TAPE_GET_MEDIA_PARAMETERS;
struct{ // structure used by GET_TAPE_DRIVE_INFORMATION
        BOOLEAN ECC;
        BOOLEAN Compression;
        BOOLEAN DataPadding;
        BOOLEAN ReportSetmarks;
        ULONG   DefaultBlockSize;
        ULONG   MaximumBlockSize;
        ULONG   MinimumBlockSize;
        ULONG   MaximumPartitionCount;
        ULONG   FeaturesLow;
        ULONG   FeaturesHigh;
        ULONG   EOTWarningZoneSize;
      }TAPE_GET_DRIVE_PARAMETERS;
```

The following code fragment illustrates a call to the *GetTapeParameters* routine:

```
DWORD rc;

rc = GetTapeParameters(
                       HANDLE hDevice,
                       DWORD dwOperation,
                       LPDWORD lpdwSize,
                       LPVOID lpParameters
                      );
if (rc)
{
   printf("Error on GetTapeParameters\n");
   printf("System Error = %d\n",GetLastError());
   exit (-1);
}
```

where *hDevice* is the open file handle, *dwOperation* is the type of information requested (GET_TAPE_MEDIA_INFORMATION or GET_TAPE_DRIVE_INFORMATION), and *lpParameters* is the address of the returned data parameter structure.

If the function succeeds, the return value *rc* is “ERROR_SUCCESS”.

PrepareTape

The *PrepareTape* entry point is called to either prepare the tape for access or removal.

```
PrepareTape(
 HANDLE hDevice,
 DWORD dwOperation,
 BOOL bImmediate
);
```

dwOperation is the type of operation requested.

The following types of operations and immediate values are supported:

TAPE_LOAD	bImmediate TRUE or FALSE
TAPE_LOCK	bImmediate FALSE
TAPE_UNLOAD	bImmediate TRUE or FALSE
TAPE_UNLOCK	bImmediate FALSE

EraseTape

The *EraseTape* entry point is called to erase all or a part of a tape. The erase is performed from the current location.

```
EraseTape(
 HANDLE hDevice,
 DWORD dwEraseType,
 BOOL bImmediate
);
```

dwEraseType is the type of operation requested.

The following types of operations and immediate values are supported:

TAPE_ERASE_LONG	bImmediate TRUE or FALSE

DeviceIoControl()

The *DeviceIoControl()* function is described in the Microsoft Windows 2000 Microsoft Developer Network (MSDN) Software Developer Kit (SDK) and Device Development Kit (DDK).

The *DeviceIoControl()* function sends a control code directly to a specified device driver, causing the corresponding device to perform the specified operation.

```
BOOL DeviceIoControl(
 HANDLE hDevice, // handle to device of interest
 DWORD dwIoControlCode, // control code of operation to perform
 LPVOID lpInBuffer, // pointer to buffer to supply input data
 DWORD nInBufferSize, // size of input buffer
 LPVOID lpOutBuffer, // pointer to buffer to receive output data
 DWORD nOutBufferSize, // size of output buffer
 LPDWORD lpBytesReturned, // pointer to variable to receive output byte count
 LPOVERLAPPED lpOverlapped // pointer to overlapped structure for \
                                        asynchronous operation
);
```

Following is a list of the supported dwIoControlCode codes that are described in the MSDN DDK and used through the DeviceIoControl() API:

1. **IOCTL_SCSI_PASS_THROUGH** - tape and medium changer
2. **IOCTL_SCSI_PASS_THROUGH_DIRECT** - tape and medium changer
3. **IOCTL_STORAGE_RESERVE** - tape and medium changer
4. **IOCTL_STORAGE_RELEAS** - tape and medium changer
5. **IOCTL_CHANGER_EXCHANGE_MEDIUM** - medium changer not all changers
6. **IOCTL_CHANGER_GET_ELEMENT_STATUS** - medium changer if Bar Code Reader, then VolTags supported
7. **IOCTL_CHANGER_GET_PARAMETERS** - medium changer
8. **IOCTL_CHANGER_GET_PRODUCT_DATA** - medium changer
9. **IOCTL_CHANGER_GET_STATUS** - medium change

10. **IOCTL_CHANGER_INITIALIZE_ELEMENT_STATUS** - medium changer with range not supported by all changers
11. **IOCTL_CHANGER_MOVE_MEDIUM** - medium changer
12. **IOCTL_CHANGER_SET_ACCESS** - medium changer for IE port only and not for all changers
13. **IOCTL_CHANGER_SET_POSITION** - medium changer only some devices support the transport object

An example of the use of SCSI Pass Through (see 1 on page 200 and 2 on page 200) is contained in the example code SPTI.C contained in the DDK.

The function call *DeviceIoControl()* is described in the SDK, and examples of its use are shown in the DDK.

Medium Changer IOCTLs

Notes:

1. The Removable Storage Manager (RSM) must be stopped to use these IOCTLs. The RSM can be stopped from Computer Management(Local)->Services and Applications->Services->Removable Storage.
2. Because not all source or destination addresses, exchanges, moves, or operations are allowed for a particular IBM medium changer, the user must issue a IOCTL_CHANGER_GET_PARAMETER to determine the type of operations that are allowed by a changer device. Further information on allowable commands for a particular changer may be found in the IBM hardware reference manual for that device. It is strongly recommended that the user have copies of the hardware reference manual before constructing any applications for the changer devices.

IOCTL Commands

IOCTL_CHANGER_EXCHANGE_MEDIUM

The media from the source element is moved to the first destination element, and the medium that occupied the first destination element previously is moved to the second destination element (the second destination element may be the same as the source) by sending a ExchangeMedium (0xA65) SCSI command to the device. The input data is a structure of CHANGER_EXCHANGE_MEDIUM. This command is not supported by all devices.

IOCTL_CHANGER_GET_ELEMENT_STATUS

Returns the status of all elements or of a specified number of elements of a particular type by sending a ReadElementStatus (0xB8) SCSI command to the device. The input and output data is a structure of CHANGER_ELEMENT_STATUS.

IOCTL_CHANGER_GET_PARAMETERS

Returns the capabilities of the changer. The output data is in a structure of GET_CHANGER_PARAMETERS.

IOCTL_CHANGER_GET_PRODUCT_DATA

Returns the product data for the changer. The output data is in a structure of CHANGER_PRODUCT_DATA.

IOCTL_CHANGER_GET_STATUS

Returns the current status of the changer by sending a TestUnitReady (0x00) SCSI command to the device.

IOCTL_CHANGER_INITIALIZE_ELEMENT_STATUS

Initializes the status of all elements or a range of a particular element by sending a InitializeElementStatus (0x07) or IntializeElementStatusWithRange (0xE7) SCSI command to the device. The input data is a structure of CHANGER_INITIALIZE_ELEMENT_STATUS.

IOCTL_CHANGER_MOVE_MEDIUM

Moves a piece of media from a source to a destination by sending a MoveMedia (0xA5) SCSI command to the device. The input data is a structure of CHANGER_MOVE_MEDIUM.

IOCTL_CHANGER_REINITIALIZE_TRANSPORT

Physically re-calibrates a transport element by sending a RezeroUnit (0x01) SCSI command to the device. The input data is a structure of CHANGER_ELEMENT. This command is not supported by all devices.

IOCTL_CHANGER_SET_ACCESS

Sets the access state of the changers IE port by sending a PrevenAllowMediumRemoval (0x1E) SCSI command to the device. The input data is a structure of CHANGER_SET_ACCESS.

IOCTL_CHANGER_SET_POSITION

Sets the changers robotic transport to a specified address by sending a PositionToElement (0x2B) SCSI command to the device. The input data is a structure of CHANGER_SET_POSITION.

Vendor Specific (IBM) Device IOCTLs for DeviceIoControl

IOCTL_TAPE_OBTAIN_SENSE

Issue this command after an error occurs to obtain sense information associated with the error that occurred most recently. To guarantee that the application can obtain sense information associated with an error, the application should issue this command before issuing any other commands to the device. Subsequent operations (other that IOCTL_TAPE_OBTAIN_SENSE) will reset the sense data field before executing the operation.

This IOCTL is only for the tape path.

The following output structure is filled in by the IOCTL_TAPE_OBTAIN_SENSE command passed by the caller:

```
#define MAG_SENSE_BUFFER_SIZE 96 /* Default request sense buffer size for \
                                                    Windows 2000 */

typedef struct _TAPE_OBTAIN_SENSE {
 ULONG SenseDataLength;
 // The number of bytes of valid sense data.
 // Will be zero if no error with sense data has occurred.
 // The only sense data available is that of the last error.
 CHAR SenseData[MAG_SENSE_BUFFER_SIZE];
} TAPE_OBTAIN_SENSE, *PTAPE_OBTAIN_SENSE;
```

An example of the IOCTL_TAPE_OBTAIN_SENSE command is:

```
DWORD cb;
TAPE_OBTAIN_SENSE sense_data;
DeviceIoControl(hDevice,
        IOCTL_TAPE_OBTAIN_SENSE,
        NULL,
        0,
```

```
        &sense_data,
        (long)sizeof(TAPE_OBTAIN_SENSE),
        &cb,
        (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_OBTAIN_VERSION

Issue this command to obtain the version of the device driver. It is in the form of a null-terminated string.

This *ioctl* is only for the tape path.

The following output structure is filled in by the IOCTL_TAPE_OBTAIN_VERSION command:

```
#define MAX_DRIVER_VERSIONID_LENGTH 12

typedef struct _TAPE_OBTAIN_VERSION {
 CHAR VersionId[MAX_DRIVER_VERSIONID_LENGTH];
} TAPE_OBTAIN_VERSION, *PTAPE_OBTAIN_VERSION;
```

An example of the IOCTL_TAPE_OBTAIN_VERSION command is:

```
DWORD cb;
TAPE_OBTAIN_VERSION code_version;
DeviceIoControl(hDevice,
    IOCTL_TAPE_OBTAIN_VERSION,
    NULL,
    0,
    &code_version,
    (long)sizeof(TAPE_OBTAIN_VERSION),
    &cb,
    (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_LOG_SELECT

This command resets all log pages that can be reset on the device to their default values. This *ioctl* is only for the tape path.

An example of the IOCTL_TAPE_LOG_SELECT command is:

```
DWORD cb;
DeviceIoControl(hDevice,
        IOCTL_TAPE_LOG_SELECT,
        NULL,
        0,
        NULL,
        0,
        &cb,
        (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_LOG_SENSE

Issue this command to obtain the log data of the requested log page from IBM Magstar tape device. The data returned is formatted according to the IBM Magstar hardware reference manual.

This *ioctl* is only for the tape path.

The following input/output structure is used by the IOCTL_TAPE_LOG_SENSE command:

```
#define MAX_LOG_SENSE 1024 // Maximum number of bytes the command will return
typedef struct _TAPE_LOG_SENSE_PARAMETERS{
  UCHAR PageCode; // The requested log page code
```

```
 UCHAR PC; // PC = 0 for maximum values, 1 for current value, 3 for power-on values
 UCHAR PageLength[2]; /* Length of returned data, filled in by the command */
  UCHAR LogData[MAX_LOG_SENSE]; /* Log data, filled in by the command */
} TAPE_LOG_SENSE_PARAMETERS, *PTAPE_LOG_SENSE_PARAMETERS;
```

An example of the IOCTL_TAPE_LOG_SENSE command is:

```
DWORD cb;
TAPE_LOG_SENSE_PARAMETERS logsense;
logsense.PageCode=0;
logsense.PC = 1;

DeviceIoControl(hDevice,
        IOCTL_TAPE_LOG_SENSE,
        &logsense,
        (long)sizeof(TAPE_LOG_SENSE_PARAMETERS),
        &logsense,
        (long)sizeof(TAPE_LOG_SENSE_PARAMETERS),
        &cb,
        (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_REPORT_MEDIA_DENSITY

Issue this command to obtain the media density information on the loaded media in the drive. If there is no media load, the command will fail. This *ioctl* is only for the tape path.

The following output structure is filled in by the IOCTL_TAPE_REPORT_MEDIA_DENSITY command:

```
typedef struct_TAPE_REPORT_DENSITY{
 ULONG PrimaryDensityCode; /* Primary Density Code */
 ULONG SecondaryDensityCode; /* Secondary Density Code */
 BOOLEAN WriteOk; /* 0 = does not support writing in this format */
       /* 1 = support writing in this format */
 ULONG BitsPerMM; /* Bits Per mm */
 ULONG MediaWidth; /* Media Width */
 ULONG Tracks; /* Tracks */
 ULONG Capacity; /* Capacity in MegaBytes */
} TAPE_REPORT_DENSITY, *PTAPE_REPORT_DENSITY;
```

An example of the IOCTL_TAPE_REPORT_MEDIA_DENSITY command is:

```
DWORD cb;
TAPE_REPORT_DENSITY tape_reportden;

DeviceIoControl (hDevice,
         IOCTL_TAPE_REPORT_MEDIA_DENSITY,
         NULL,
         0,
         &tape_reportden,
         (long)sizeof(TAPE_REPORT_DENSITY),
         &cb,
         (LPOVERLAPPED) NULL);
```

Chapter 21. Event Log

The IBM 3580 and IBM Ultrium changers (IBM 3581, 3583, and 3584) device drivers (ibmtape.sys and ibmchgr.sys) log certain exception data to the Event Log when errors are encountered.

In order to interpret this event data, the user needs to be familiar with the following components:

- Microsoft Event Viewer
- The SDK and DDK components from the Microsoft Developer Network (MSDN)
- IBM 3580, 3581, 3583, and 3584 hardware terminology
- SCSI terminology

Several bytes of Event Detail data are logged under Source=Ibmlto or Ibmchgr (for Windows 2000) or Source=Ultrium or UltrChgr (for Windows NT).

The following description texts are expected:

- The description for Event ID (0) in Source (Ultrium) could not be found. It contains the following insertion string: \Device\Tapex.
- The description for Event ID(x) in Source (UltrChgr) could not be found. It contains the following insertion string: \Device\Ibx.x.x.x.

The user must view the event data in “Words” format to decode the data properly. References to names of Microsoft constants and files were correct at the publication time of this documentation.

Table 5 and Table 6 on page 206 indicate the hexadecimal offsets, names, and definitions for ibmlto, ibmchgr, and ultrium event data. UltrChgr event data has a unique format that will appear later in this chapter.

Table 5. ibmlto, ibmchgr, and ultrium Event Data

Offset	Name	Definition
0x00 to 0x01	DumpDataSize	Indicates the size in bytes required for any DumpData that the driver will place in the packet.
0x02	RetryCount	Indicates how many times the driver has retried the operation and encountered this error.
0x03	MajorFunctionCode	Indicates the IRP_MJ_XXX from the driver’s I/O stack location in the current IRP (from NTDDK.H).
0x0C to 0x0F	ErrorCode	For the IBM 3580 device driver (Ultrium), it is 0. For the IBM Ultrium changer device driver, it is always 0xC004000B (IO_ERR_CONTROLLER_ERROR, from NTIOLOGC.H).
0x10 to 0x13	UniqueErrorValue	Reserved
0x14 to 0x17	FinalStatus	Indicates the value set in the I/O status block of the IRP when it was completed or the STATUS_XXX returned by a support routine the driver called (from NTSTATUS.H).

Table 5. ibmlto, ibmchgr, and ultrium Event Data (continued)

Offset	Name	Definition
0x1C to 0x1F	IoControlCode	For the IBM 3580 device driver (Ultrium), it indicates the I/O control code from the driver's I/O stack location in the current IRP if the MajorFunctionCode is IRP_MJ_DEVICE_CONTROL; otherwise, this value will be 0. For the IBM Ultrium changer device driver, it indicates the I/O control code from the driver's I/O stack location in the current IRP.
0x28	Beginning of Dump Data	The following items are variable in length. See the DDK and SCSI documentation for details.
0x38	Beginning of SRB structure	The SCSI Request Block (from NTDDK.H).
0x68	Beginning of CDB structure	The Command Descriptor Block (from SCSI.H).
0x78	Beginning of SCSI Sense Data	(from SCSI.H). If the first word in this field is 0x00DF0000 (SCSI error marker) or 0x00EF0000 (Non_SCSI error marker), no valid sense information was available for this error.

For example, the following error is logged when a move medium is attempted and the destination element is full. Explanations of selected fields follow:

```
0000: 006c000f 00c40001 00000000 c004000b
0010: bcde7f48 c0000284 00000000 00000000
0020: 00000000 00000000 00000300 000052f4
0030: 00000000 00000000 004000c4 02000003
0040: 600c00ff 00000028 00000000 00000258
0050: 00000000 814dac28 00000000 bcde7f48
0060: 81841000 00000000 a5600000 00200010
0070: 00000000 00000000 70000500 00000058
0080: 00000000 3b0dff02 00790000 0000093e
0090: 00000000
```

Table 6. ibmlto, ibmchgr, and ultrium Event Data

Field	Value	Definition
DumpDataSize	0x006C	6C hex (108 dec) bytes of dump data, beginning at byte 28 hex.
RetryCount	0x00	This is the first time the operation has been attempted (no retries).
MajorFunctionCode	0x0F	IRP_MJ_INTERNAL_DEVICE_CONTROL
FinalStatus	0xC0000284	STATUS_DESTINATION_ELEMENT_FULL
IoControlCode	0x00000000	– –
SRB	0x004000C4...	From NTDDK.H, the first word of the SRB indicates the length of the SRB (40 hex bytes, 64 dec bytes), the function code (0x00), and the SrbStatus (from SRB.H, 0xC4 = SRB_STATUS_AUTOSENSE_VALID, SRB_STATUS_QUEUE_FROZEN, SRB_STATUS_ERROR)

Table 6. ibmlto, ibmchgr, and ultrium Event Data (continued)

Field	Value	Definition
CDB	0xA5...	From SCSI.H, the first byte of the CDB is the operation code. 0xA5 = SCSIOP_MOVE_MEDIUM.
Sense Data	0x70000500...	From SCSI.H, the first word of the sense data indicates the error code (0x70), the segment number (0x00), and the sense key (0x05, corresponding to an illegal SCSI request).

Table 7 and Table 8 on page 208 contain definitions for event data logged under UltrChgr. Note that expressions “Sense byte n” are zero-based.

Table 7. UltrChgr Event Data

Offset	Name	Definition
0x00 to 0x01	DumpDataSize	Indicates the size in bytes required for any DumpData that the driver will place in the packet.
0x02	RetryCount	Indicates how many times the driver has retried the operation and encountered this error.
0x03	MajorFunctionCode	Indicates the IRP_MJ_XXX from the driver’s I/O stack location in the current IRP (from NTDDK.H).
0x0C to 0x0F	ErrorCode	For the IBM 3580 device driver (Ultrium), it is zero. For the IBM Ultrium Changer device driver, it is always 0xC00400B (IO_ERR_CONTROLLER_ERR) (from NTIOLOGC.H).
0x10 to 0x13	UniqueErrorValue	Reserved
0x14 to 0x17	FinalStatus	Indicates the value set in the I/O status block of the IRP when it was completed or the STATUS_XXX returned by a support routine the driver called (from NTSTATUS.H).
0x1C to 0x1F	IoControlCode	For the IBM 3580 device driver (Ultrium), it indicates the I/O control code from the driver’s I/O stack location in the current IRP if the MajorFunctionCode is IRP_MJ_DEVICE_CONTROL; otherwise, this value will be zero. For the IBM Ultrium Changer device driver (UltrChgr), it indicates the I/O control code from the driver’s I/O stack location in the current IRP.
0x29	PathId	SCSI Path ID
0x2A	TargetId	SCSI Target ID
0x2B	LUN	SCSI Logical Unit Number
0x2D	CDB[0]	Command Operation Code
0x2E	SRB_STATUS	See MINITAPE.H or SRB.H.
0x2F	SCSI_STATUS	See SCSI.H or a SCSI specification.

Table 7. UltrChgr Event Data (continued)

Offset	Name	Definition
0x30 to 0x33	Timeout Value	For the IBM 3580 device driver (Ultrium), this value is always zero. For the IBM Ultrium Changer device driver (UltrChgr), this value is the command timeout value in seconds.
0x38	FRU or Sense Byte 14	For the IBM 3580 device driver (Ultrium), this value is the Field Replaceable Unit code. For the IBM Ultrium Changer device driver (UltrChgr), this value is Sense Byte 14.
0x39	SenseKeySpecific[0]	Indicates Sense Key Specific byte (Sense Byte 15).
0x3A	SenseKeySpecific[1] or CDB length	If valid sense data was returned, SenseKeySpecific[1] (Sense Byte 16) will be displayed; otherwise, the CDB length will be displayed. See offset 0x3D to determine whether valid sense data has been returned.
0x3B	SenseKeySpecific[2] or CDB[0]	If valid sense data was returned, SenseKeySpecific[2] (Sense Byte 17) will be displayed; otherwise, the CDB operation code will be displayed. See offset 0x3D to determine whether valid sense data has been returned.
0x3C	Sense Byte 0	Indicates the first byte of returned sense data.
0x3D	Sense Byte 2	Indicates the second byte of returned sense data. This byte contains the Sense Key and other flags. If this is set to 0xDF (SCSI Error Marker) or 0xEF (Non-SCSI Error Marker), no valid sense information was available for the error.
0x3E	ASC or SRB_STATUS	Indicates Sense Byte 12, if there was valid sense information; otherwise, the SRB status value will be given here. See offset 0x3D to determine whether valid sense data has been returned.
0x3F	ASCQ or SCSI_STATUS	Indicates Sense Byte 13, if there was valid sense information; otherwise, the SCSI status value will be given here. See offset 0x3D to determine whether valid sense data has been returned.

For example:

```
0000: 0018000f 006c0001 00000000 00000000
0010: 00000000 c0000185 00000000 00000000
0020: 00000000 00000000 00000300 0015c402
0030: 00000000 00000000 f50ac607 700b4b00
```

Table 8. UltrChgr Event Data

Field	Value	Definition
DumpDataSize	0x0018	–

Table 8. UltrChgr Event Data (continued)

Field	Value	Definition
RetryCount	0x00	–
MajorFunctionCode	0x0f	IRP_MJ_INTERNAL_DEVICE_CONTROL
FinalStatus	0xc0000185	STATUS_IO_DEVICE_ERROR
IoControlCode	0x00000000	–
PathId	0x00	–
TargetId	0x03	–
LUN	0x00	–
CDB[0]	0x15	Mode Select, Byte 6
SRB_STATUS	0xc4	SRB_STATUS_AUTOSENSE_VALID, SRB_STATUS_QUEUE_FROZEN, SRB_STATUS_ERROR
SCSI_STATUS	0x02	Check condition
FRU	0xF5	–
Sense Key Specific Sense Bytes 15 to 17	0x0ac607	–
Sense Byte 0	0x70	–
Sense Key Sense Byte 2	0xb4	–
ASC	0x4b	–
ACSQ	0x00	–

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries/regions in which IBM operates.

Any references to an IBM program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designed by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states/regions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

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

AIX	IBM	StorageSmart
ESCON	Magstar	

The following terms are trademarks of Hewlett-Packard, IBM, and Seagate in the United States:

Linear Tape-Open LTO Ultrium

Microsoft, Windows, Windows NT, Windows 2000, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries/regions, or both.

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

Index

M

O

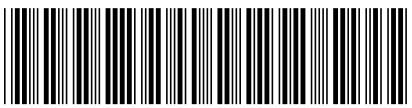
R

S

W

Printed in U.S.A.

GC35-0483-01



Spine information:

IBM Ultrium Device Drivers

IBM Ultrium Tape Device Drivers: Programming Reference