IBM Tape Device Drivers

Programming Reference

GA32-0566-09

IBM Tape Device Drivers

Programming Reference

GA32-0566-09

Contents

Note!

Before using this information and the product that it supports, be sure to read the general information under "Notices" on page 379.

Ninth Edition (March 2016)

This edition replaces and makes obsolete GC35-0483-06, GC35-0346-10, GA32-0566-00, GA32-0566-01, GA32-0566-02, GA32-0566-03, GA32-0566-04, GA32-0566-05, GA32-0566-06, GA32-0566-07, and GA32-0566-08. Changes or additions are indicated by a vertical line in the left margin.

Preface

These publications and URLs provide user information and installation assistance for IBM® tape drive, medium changer, and library device drivers.

Special printing instructions

This Device Driver manual contains different sections for each type of operating platform, for example, AIX, HP-UX, Linux, Oracle Solaris, and Windows. The manual also contains a separate section on these operating systems for the 3494 Enterprise tape library.

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

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

1. Go to the beginning of the section and note the page number.
2. Go 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 that is entered prints.
4. Repeat these steps to print extra sections.

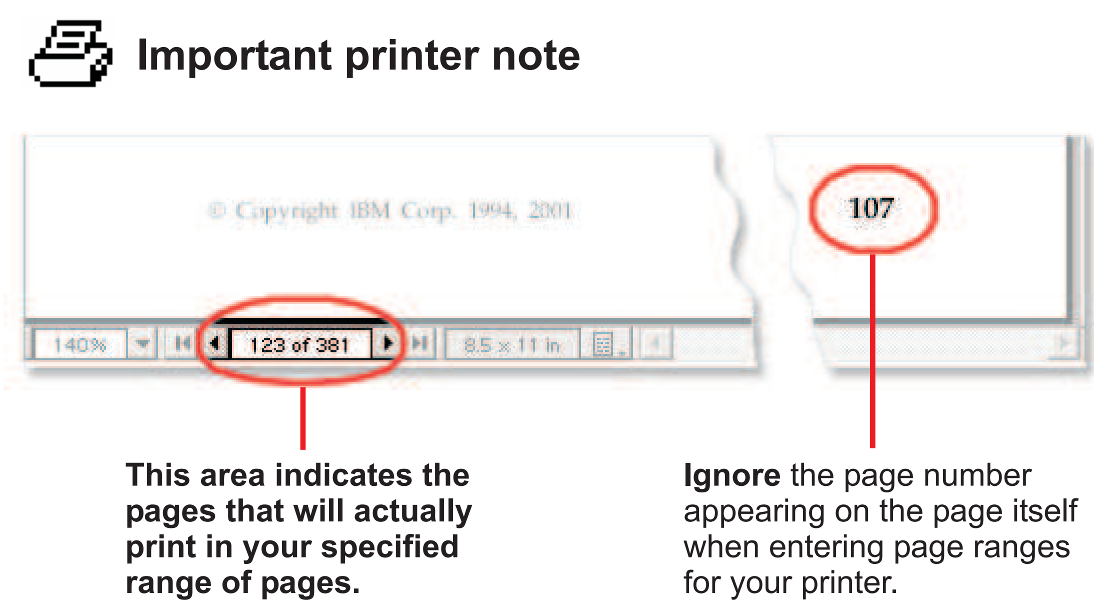

Figure 1. How to print Adobe pdf files

Attention: Only one Table of Contents and one Index is available for this entire book. If you want to print those items, you must repeat the process, by entering the page range of the Table of Contents and the Index page range.

Related information

Reference material, including the Adobe pdf version of this publication, is available at http://www-01.ibm.com/support/docview.wss?uid=ssg1S7003032.

A companion publication that covers installation and user aspects for the device drivers is *IBM Tape Device Drivers: Installation and Users Guide*, GC27-2130-00, at http://www-01.ibm.com/support/docview.wss?uid=ssg1S7002972.

AIX

The following URL points to information about IBM System p (also known as pSeries) servers: http://www-1.ibm.com/servers/eserver/pseries.

HP-UX

The following URL relates to HP HP-UX systems: http://www.hp.com.

Linux

The following URLs relate to Linux distributions: http://www.redhat.com and http://www.suse.com.

Solaris

The following URL relates to Oracle Solaris systems: http://www.oracle.com/us/sun/index.html .

Microsoft Windows

The following URL relates to Microsoft Windows systems: http://www.microsoft.com .

Additional information

The following publication contains information that is related to the IBM tape drive, medium changer, and library device drivers: *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.

Chapter 1. Common extended features

Tape drive functions and device driver ioctls

Beginning with the TS1140 (JAG 4), TS2250, and TS2350 (LTO 5) generation of tape drives, functions are supported that previous generations of LTO and JAG tape drives do not support. The device drivers provide *ioctls* that applications can use for these functions. Refer to the appropriate platform section for the specific ioctls and data structures that are not included in this section.

- Media Partitioning

  Supported tape drives: LTO 5 and JAG 4 and later models
- Data Safe (Append-Only) Mode

  Supported tape drives: LTO 5 and JAG 4 and later models
- Read Position SCSI Command for Long and Extended forms

  Supported tape drives: LTO 5 and JAG 4 and later models
- Locate(16) SCSI Command

  Supported tape drives: LTO 5 and JAG 4 and later models
- Logical Block Protection

  Supported tape drives: LTO 5 and JAG 2/3/4 and later models
- Programmable Early Warning (PEW)

  Supported tape drives: LTO 5 and JAG 2 and later models
- Log Sense Page and Subpage

  Supported tape drives: LTO 5 and JAG 3 and later models
- Mode Sense Page and Subpage

  Supported tape drives: LTO-4 and JAG 2 and later models
- Verify Tape

  Supported tape drives: LTO5 and JAG 2 and later models

Media partitioning

There are two types of partitioning: Wrap-wise partitioning and Longitudinal partitioning (maximum 2 partitions).

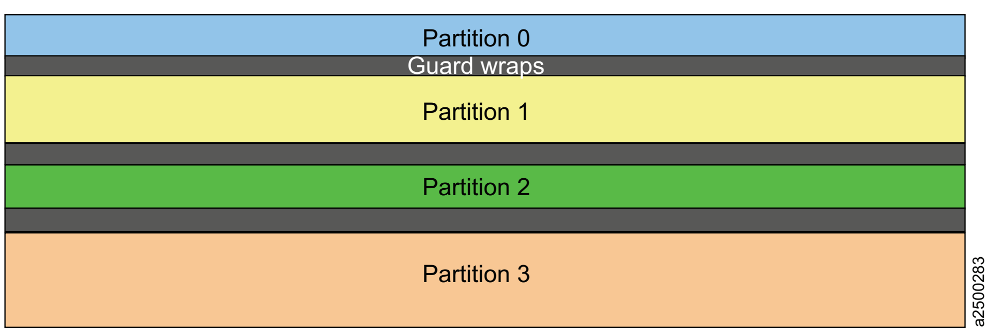

Figure 2. Wrap-wise partitioning

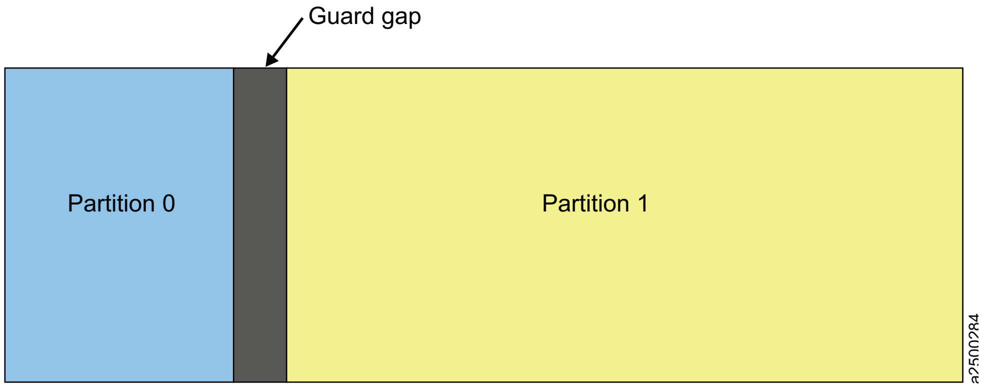

Figure 3. Longitudinal partitioning

In Wrap-wise partitioning, media can be partitioned into 1 or 2 partitions for LTO 5 and 1 - 4 partitions for TS1140. For later generations, see drive documentation for the number of partitions supported. The data partition (the default) for a single partition always exists as partition 0. WORM media cannot be partitioned.

The ioctls the device drivers provide for tape partitioning are

- **Query Partition**

The Query Partition ioctl returns the partition information for the current media in the tape drive. It also returns the current active partition the tape drive is using for the media.

Note: If the **Create Partition** ioctl fails, then the **Query Partition** ioctl does not return the correct partition information. To get the correct information, the application must unload and reload the tape again.

- **Create Partition**

The **Create Partition** ioctl is used to format the current media in the tape driver to either 1 or 2 partitions. When two partitions are created, the FDP, SDP, or IDP partition type is specified by the application. The tape must be positioned at the beginning of tape (partition 0 logical block id 0) before this ioctl is used or the ioctl fails.

If the **number_of_partitions** field to create in the ioctl structure is one partition, all other fields are ignored and not used. The tape drive formats the media by using its default partitioning type and size for a single partition.

When the type field in the ioctl structure is set to either FDP or SDP, the **size_unit** and **size** fields in the ioctl structure are not used. When the type field in the ioctl structure is set to IDP, the **size_unit** and **size** fields are used to specify the size for each partition. One of the two partition sizes for either partition 0 or 1 must be specified as 0xFFFF to use the remaining capacity. The other partition is created by using the **size_unit** and **size** field for the partition.

- **Set Active Partition**

The **Set Active Partition** ioctl is used to position the tape drive to a specific partition. It becomes the current active partition for subsequent commands and a specific logical bock id in the partition. To position to the beginning of the partition, the **logical_block_id** field in the ioctl structure must be set to 0.

Data safe (append-only) mode

Data safe (append-only) mode sets the drive into a logical WORM mode so any non-WORM tape when loaded is handled similarly to a WORM tape. After data or filemarks are written to the tape, it cannot normally be overwritten. New data or filemarks can be appended only at the end of previously written data. Data safe mode applies only to drive operation. When a non-WORM tape is unloaded, it does not change and is still a non-WORM tape.

Conditions exist when the drive is in data safe mode an application might want to explicitly overwrite previously written data by issuing a **`write`**, **`write filemark`**, or **`erase`** command. These commands are referred to as write type commands. An application might also want to explicitly partition the tape with the **Create Partition** ioctl that issues a **`format`** command. The drive supports a new **`Allow Data Overwrite SCSI`** command for this purpose.

The ioctls that the device drivers provide for data safe mode are

- **Querying and setting data safe mode**

All platform device drivers except Windows added a data safe mode parameter to existing ioctls that are used to query or set tape drive parameters. The Windows device driver added two new ioctls to query or set data safe mode.

A query ioctl returns the current drive mode, either data safe mode off (normal mode) or data safe mode on. A set ioctl sets the drive to either data safe mode off (normal mode) or data safe mode on. Data safe mode can be set whether a tape is loaded in the drive or not. Data safe mode can be set back to normal mode only when a tape is not currently loaded in the drive.

- **Allow Data Overwrite**

The **Allow Data Overwrite** ioctl is used to allow previously written data on the tape to be overwritten when data safe mode is enabled on the drive, for a subsequent write type command, or to allow a format command with the **Create Partition** ioctl.

To allow a subsequent write type command, the tape position must be set to the correct partition and logical block address within the partition before the ioctl is used. The **partition_number** and **logical_block_id** fields in the ioctl structure must be set to that partition and logical block ID. The **allow_format_overwrite** field in the ioctl structure must be set to 0.

To allow a subsequent **Create Partition** ioctl to format the tape, the **allow_format_overwrite** field in the ioctl structure must be set to 1. The **partition_number** and **logical_block_id** fields are not used. But, the tape must be at the beginning of tape (partition 0 logical block id 0) before the **Create Partition** ioctl is issued.

Read Position long/extended form and Locate(16) commands

Because of the increased tape media capacity and depending on the block sizes and number of files an application can write on tape, the 4-byte fields such as the logical block id the current **`Read Position`** command (referred to as the short form) that returns 20 bytes might overflow. The same applies to the **`Locate(10)`** command for the logical block id.

LTO 5 and later supports new forms of the existing `Read Position` command in addition to the current short form. The short form continues to return 4-byte fields in 20 bytes of return data. The long form returns 8-byte fields in 32 bytes of return data with the current position information for the logical block id and logical filemark. The extended form returns 8-byte fields in 32 bytes of return data with the current position information for the logical block id and buffer status. The format of return data in the `Read Position` command is specified by using a service action field in the Read Position SCSI CDB.

LTO 5 and later also supports the `Locate(16)` command that uses 8-byte fields. This command can either position the tape to a logical block id or a logical filemark by setting the **dest_type** field in the Locate(16) SCSI CDB. After the locate command completes, the tape is positioned at the BOP side of the tape.

The ioctls the device drivers provide are

- **Read Tape Position**

The **Read Tape Position** ioctl returns the `Read Position` command data in either the short, long, or extended form. The form to be returned is specified by setting the **data_format** field in the ioctl structure.

- **Set Tape Position**

The **Set Tape Position** ioctl issues a `Locate(16)` command to position the tape in the current active partition to either a logical block id or logical filemark. The **logical_id_type** field in the ioctl structure specifies either a logical block or logical filemark.

Logical Block Protection

The ioctls the device drivers provide are

- **Query Logical Block Protection**

  This ioctl queries whether the drive can support this feature, what lbp method is used, and where the protection information is included.

  The **lbp_capable** field indicates that the drive has the logical block protection (LBP) capability or not. The **lbp_method** field is shown if LBP is enabled and what the protection method is. The LBP information length is shown in the **lbp_info_length** field. The fields of **lbp_w**, **lbp_r**, and **rbdp** present that the protection information is included in write, read, or recover buffer data. The **rbdp** field is not supported for the LTO drive.
- **Set Logical Block Protection**

  This ioctl enables or disables Logical Block Protection, sets up what method is used, and where the protection information is included.

  The **lbp_capable** field is ignored in this ioctl by the tape driver. If the **lbp_method** field is 0 (`LBP_DISABLE`), all other fields are ignored and not used. When the **lbp_method** field is set to a valid non-zero method, all other fields are used to specify the setup for LBP.

Programmable Early Warning (PEW)

With the tape parameter, the application is allowed to request the tape drive to create a zone that is called the programmable early warning zone (PEWZ) in front of Early Warning (EW).

Figure 4. Programmable Early Warning Zone (PEWZ)

This parameter establishes the programmable early warning zone size. It is a 2-byte numerical value that specifies how many MB before the standard end-of-medium early warning zone to place the programmable early warning indicator. If the value is set to a positive integer, a user application is warned that the tape is running out of space when the tape head reaches the PEW location. If pew is set to 0, then there is no early warning zone and the user is notified only at the standard early warning location.

Log Sense page and subpage

This ioctl of the **SIOC_LOG_SENSE10_PAGE** issues a **`Log Sense(10)`** command and returns log sense data for a specific page and subpage. This ioctl command is enhanced to add a subpage variable from the log sense page. It returns a log sense page or subpage from the device. The wanted page is selected by specifying the **page_code** or **subpage_code** in the structure. Optionally, a specific parm pointer, also known as a parm code, and the number of parameter bytes can be specified with the command.

Mode Sense page and subpage

This ioctl of the **SIOC_MODE_SENSE** issues a **`Mode Sense(10)`** or **`(6)`** command and returns the whole mode sense data. The data includes the header, block descriptor, and page code for a specific page or subpage from the device.

Verify Tape

The ioctl of **VERIFY_DATA_TAPE** issues the **`VERIFY`** command to cause data to be read from the tape and passed through the drive's error detection and correction hardware. This action determines whether data can be recovered from the tape. Also, whether the protection information is present and validates correctly on logical block on the medium. The driver returns a failure or success signal if the **`VERIFY SCSI`** command is completed in a **Good SCSI** status. The **`Verify`** command is supported on all LTO libraries. Verify to EOD (ETD) or verify by filemark (VBF) is supported on drives that support Logical Block Protection (LBP).

RAO - Recommended Access Order

The 3592 E07 implements a function that is called Recommended Access Order. This function provides the capability to improve multiple block recall and retrieval times. It provides an application with the optimized order in which a list of blocks must be recalled to minimize the required total time period.

An application uses the **`GRAO`** command to request that the drive generate a recommended access order for the User Data Segments that are sent in this command. After a **`GRAO`** command completes, use the **`RRAO`** command to receive the results.

`QUERY_RAO_INFO` IOCTL

The IOCTL queries the maximum number and size of User Data Segments (UDS) that are supported from tape drive and driver for the wanted `uds_type`: with or without geometry, the geometry can be used to build a representation of the physical layout of the UDS on tape.. The application calls this IOCTL before the **`GENERATE_RAO`** and **`RECEIVE_RAO`** IOCTLs are issued. The return in this IOCTL is to be used by the application to limit the number of UDS requested in calls to the **`GENERATE_RAO`** IOCTL.

GENERATE_RAO

The IOCTL is called to send a **GRAO** list (UDS's descriptors list) to request the drive to generate a Recommending Access Order list. The process method to create a RAO list is either 1 or 2. 1 does not reorder the UDS's list, but does calculate the estimated locate time for each UDS in the list, and 2 reorders the UDS's list and calculates the estimated locate time for each UDS in its resultant position. The type of UDS is either with or without the geometry. The **uds_number** must be not larger than **max_host_uds_number** returned in the **`QUERY_RAO_INFO`** IOCTL. A UDS descriptor has a name, partition number and beginning and ending logical object identifiers.

RECEIVE_RAO

After a **`GENERATE_RAO`** IOCTL is completed, the application calls the **`RECEIVE_RAO`** IOCTL to receive a recommended access order of UDS from the drive. The application must allocate to the buffer an accurate size to receive the list.

The **grao** list, for the **generate_rao** and **receive_rao** structures, is in the following format when lists are sent or received. The structures for the headers and UDS segments are not provided by the device driver but is the responsibility of the calling application. It is defined in the *IBM Enterprise Tape System 3592 SCSI Reference*.

```
-- List Header
-- UDS Segment Descriptor (first)
   ......
-- UDS Segment Descriptor (last)
```

Chapter 2. AIX tape and medium changer device driver

This chapter provides an introduction to the IBM AIX® Enhanced Tape and Medium Changer Device Driver (Atape) programming interface to IBM TotalStorage (formally Magstar®) and System Storage® tape and medium changer devices.

Software interface for tape devices

The AIX tape and medium changer device driver provides the following 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 extra 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.

Software interface for medium changer devices

The AIX tape and medium changer device driver provides 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.

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 (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. In addition to the tape special files, a special file is provided to tape devices that allow access to the medium changer as a separate device. See Table 1. The asterisk (*) represents a number that is assigned to a particular device (such as **rmt0**).

Table 1. Special files for tape devices

Special File Name	Rewind on Close1	Retension on Open2	Bytes per Inch3	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[4]	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[5]	Yes	No	N/A	No	No
/dev/rmt*.smc[6]	N/A	N/A	N/A	N/A	N/A

Note:

1. The **Rewind on Close** special files for the Ultrium tape drives write filemarks under certain conditions before rewinding. See "Opening the special file for I/O" on page 10.
2. The **Retension on Open** special files rewind the tape on open only. Retensioning is not completed because these tape products run the retension operation automatically when needed.
3. The **Bytes per Inch** options are ignored for the tape devices that this driver supports. The density selection is automatic.
4. The **rmt*.10** file bypasses normal close processing, and the tape is left at the current position.
5. 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 that is attached to it, and the device driver returns a successful completion. Read and write

system calls return the requested number of bytes. This file can be used for application development or debugging problems.

6. The **rmt*.smc** file can be opened independently of the other tape special files.

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.

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 that are designed for a tape device

If a tape drive has an attached SCSI medium changer device, 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 that is 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 that is attached to it, and the device driver returns 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 that are 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, which are 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 complete the same function. The driver does not wait until the device is ready before it opens and allows commands to be sent. If the device is not ready, subsequent commands (which require that the device is ready or a physical tape is loaded) fail with ENOTREADY. Other commands, such as gathering the inquiry data, complete successfully.
- **O_APPEND**

  When the tape drive is opened with this flag, the driver rewinds the tape. Then, it seeks to the first two consecutive filemarks, and places the initial tape position between them. This status is the same if the tape was previously opened 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 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 "Return codes" on page 92 for a description of the *errno* values.

The extended open operation

An extended **open** operation is also supported on the device. This operation allows special types of processing 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 that are described in "Opening the special file for I/O." 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.

- **SC_FORCED_OPEN**

  The flag forces the release of any current reservation on the device by an initiator. The reservation can either be a **SCSI Reserve** or **SCSI Persistent Reserve**.

- **SC_KILL_OPEN**

  This flag kills all currently open processes and then exits the open with *errno* EINPROGRESS returned.

- **SC_PR_SHARED_REGISTER**

  This flag overrides the configuration reservation type attribute whether it was set to **reserve_6** or persistent. It sets the device driver to use **Persistent Reserve** while the device is open until closed. The configuration reservation type attribute is not changed and the next open without using this flag uses the configuration reservation type. The device driver also registers the host reservation key on the device. This flag can be used with the other extended flags.

- **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.

- **SC_NO_RESERVE**

  This flag prevents the **SCSI Reserve** command from being sent during an **open** operation.

- **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. This flag is the same as the SC_DIAGNOSTIC flag. The exception is that a **SCSI Test Unit Unit Ready** command is issued to the device during an open operation to clear any unit attentions.

- **SC_FEL**

  This flag turns on the forced error logging 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 up to eight processes to concurrently open a device when the device is already open by another process. There is no restriction for medium changer IOCTL commands that can be issued when this flag is used. However, for tape devices only a limited set of IOCTL commands can be issued. If an IOCTL command cannot be used with this flag, then *errno* EINVAL is returned.

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

If the **open** system call fails, the *errno* value contains the error code. See “Return codes” on page 92 for a description of the *errno* values.

Writing to the special file

Several subroutines allow writing 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), the **numbytes** value must be a multiple of the block size. If the block size is variable, the value that is specified in **numbytes** is written. If the **count** is less than zero, the *errno* value contains the error code that is returned from the driver.

See "Return codes" on page 92 for a description of the *errno* values.

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

Reading from the special file

Several subroutines allow reading 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), the **numbytes** value must be a multiple of the block size. If the **count** is less than zero, the *errno* value contains the error code that is returned from the driver.

See "Return codes" on page 92 for a description of the *errno* values.

If the block size is variable, then the value that is specified in **numbytes** is read. If the blocks read are smaller than requested, the block is returned up to the maximum size of one block. If the blocks read are greater than requested, 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. 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.

Reading with the TAPE_READ_REVERSE extended parameter

The **TAPE_READ_REVERSE** extended read parameter reads data from the tape in the reverse direction. The order of the data that is returned in the buffer for each block that is read from the tape is the same as if it were read in the forward direction. However, the last block that is written is the first block in the buffer. This parameter can be used with both fixed and variable block sizes. The **TAPE_SHORT_READ** extended parameter can be used with this parameter, if the block size is set to variable (0).

Use this parameter with the **readx** or **readvx** subroutine that specifies the **TAPE_READ_REVERSE** extended parameter.

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

The **TAPE_READ_REVERSE** parameter is defined in the **/usr/include/sys/Atape.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 must 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 the special file was closing. The **close** subroutine tries to run 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, it tries to close the connection. If the **close** operation fails, consider the device closed and try another **open** operation to continue processing the tape. After a **close** failure, assume that either the data or the tape is inconsistent.

For tape drives, the result of a **close** operation depends on the special file that was used during the **open** operation and the tape operation that was run 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
```

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 separate 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 TotalStorage and System Storage Tape 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 so that 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 so 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 the pages that are supported by the device. Page 0x00 is followed by all the pages that are 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 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 */
    };
```

Persistent reservation support and IOCTL operations

ODM attributes and configuring persistent reserve support

Two new ODM attributes are added for PR (Persistent Reservation) support:

- reserve_type
- reserve_key

The reserve type attribute determines the type of reservation that the device driver uses for the device. The values can be **reserve_6**, which is the default for the device driver or persistent. This attribute can be set by either using the **AIX SMIT** menu to **Change/Show Characteristics of a Tape Drive** or from a command line with the AIX command

```
chdev –l rmtx –a reserve_type=persistent or –a reserve_type=reserve_6
```

The **reserve_key** attribute is used to optionally set a user-defined host reservation key for the device when the **reserve_type** is set to persistent. The default for this attribute is blank (NULL). The default uses a device driver unique host reservation key that is generated for the device. This attribute can be set by either using the **AIX SMIT** menu to **Change/Show Characteristics of a Tape Drive** or from a command line with the AIX command

```
chdev –l rmtx –a reserve_key=key
```

The key value can be specified as a 1-8 character ASCII alphanumeric key or a 1-16 hexadecimal key that has the format 0xkey. If fewer than 8 characters are used for an ASCII key such as hostA, the remaining characters are set to 0x00 (NULL).

Note: If the Data Path Failover (DPF) feature is enabled for a logical device by setting the **alternate_pathing** attribute to yes, the configuration **reserve_type** attribute is not used and the device driver uses persistent reservation. Either the user-defined **reserve_key** value or if not defined the default device driver host reservation key is used.

Default device driver host reservation key

If a user-defined host reservation key is not specified, then the device driver uses a unique static host reservation key for the device. This key is generated when the first device is configured and the device driver is initially loaded into kernel memory. The key is 16 hexadecimal digits in the format `0xApppppppssssssss`, where `ppppppp` is the configuration process id that loaded the device driver. Also, `ssssssss` is the 32-bit value of the TOD clock when the device driver was loaded. When any device is configured and the **reserve_key** value is NULL, then the device driver sets the **reserve_key** value to this default internally for the device.

Preempting and clearing another host reservation

When another host initiator is no longer using the device but has left either an **SCSI-2 Reserve 6** or a **Persistent Reserve** active preventing by using the device, either type of reservation can be cleared by using the openx() extended parameter **`SC_FORCED_OPEN`**.

Note: This parameter must be used only when the application or user is sure that the reservation must be cleared.

Openx() extended parameters

The following openx( ) extended parameters are provided for managing device driver reserve during open processing and release during close processing. These parameters apply to either **SCSI-2 Reserve 6** or **Persistent Reserve**. The **`SC_PASSTHRU`** parameter applies only to the Atape device driver and is defined in **/usr/include/sys/Atape.h**. All other parameters are AIX system parameters that are defined in **/usr/include/sys/scsi.h**. AIX base tape device drivers might not support all of these parameters.

- **`SC_PASSTHRU`**
- **`SC_DIAGNOSTIC`**
- **`SC_NO_RESERVE`**
- **`SC_RETAIN_RESERVATION`**
- **`SC_PR_SHARED_REGISTER`**

- **SC_FORCED_OPEN**

The **SC_PASSTHRU** parameter bypasses all commands that are normally issued on open and close by the device driver. In addition to bypassing the device driver that reserves on open and releases the device on close, all other open commands except test unit ready such as mode selects and rewind on close (if applicable) are also bypassed. A test unit ready is still issued on open to clear any pending unit attentions from the device. This is the only difference in using the **SC_DIAGNOSTIC** parameter.

The **SC_DIAGNOSTIC** parameter bypasses all commands that are normally issued on open and close by the device driver. In addition to bypassing the device driver that reserves on open and releases the device on close, all other open commands such test unit ready, mode selects, and rewind on close (if applicable) are also bypassed.

The **SC_NO_RESERVE** parameter bypasses the device driver that issues a reserve on open only. All other normal open device driver commands are still issued such as test unit ready and mode selects.

The **SC_RETAIN_RESERVATION** parameter bypasses the device driver that issues a release on close only. All other normal close device driver commands are still issued such as rewind (if applicable).

The **SC_PR_SHARED_REGISTER** parameter sets the device driver **reserve_type** to persistent and overrides the configuration **reserve_type** attribute whether it was set to **reserve_6** or **persistent**. A subsequent reserve on the current open by the device driver (if applicable) uses **Persistent Reserve**. The **reserve_type** is only changed for the current open. The next open without using this parameter uses the configuration **reserve_type**. In addition to setting the **reserve_type** to persistent, the device driver registers the host reservation key on the device. This parameter can also be used with the extended parameters.

The **SC_FORCED_OPEN** parameter first clears either a **SCSI-2 Reserve 6** or a **Persistent Reservation** if one currently exists on the device from another host. The device driver open processing then continues according to the type of open. This parameter can also be used with the extended parameters.

AIX tape persistent reserve IOCTLs

The Atape device driver supports the AIX common tape **Persistent Reserve** IOCTLs for application programs to manage their own **Persistent Reserve** support. The IOCTLs are defined in the header file **/usr/include/sys/tape.h**.

The following two IOCTLs return **Persistent Reserve** information by using the **SCSI Persistent Reserve In** command.

- **STPRES_READKEYS**
- **STPRES_READRES**

The following four IOCTLs complete **Persistent Reserve** functions by using the **SCSI Persistent Reserve Out** command.

- **STPRES_CLEAR**
- **STPRES_PREEMPT**
- **STPRES_PREEMPT_ABORT**
- **STPRES_REGISTER**

Except for the **STPRES_REGISTER** IOCTL, the other three IOCTLs require that the host reservation key is registered on the device first. This action can be done by either issuing the **STPRES_REGISTER** IOCTL before the IOCTLs are issued or by opening the device with the **SC_PR_SHARED_REGISTER** parameter.

The **STPRES_READKEYS** IOCTL issues the **persistent reserve in** command with the read keys service action. The following structure is the argument for this IOCTL.

```
struct st_pres_in {
  ushort      version;
  ushort      allocation_length;
  uint        generation;
  ushort      returned_length;
  uchar       scsi_status;
  uchar       sense_key;
  uchar       scsi_asc;
  uchar       scsi_ascq;
  uchar       *reservation_info;
}
```

The `allocation_length` is the maximum number of bytes of key values that are returned in the **reservation_info** buffer. The `returned_length` value indicates how many bytes of key values that device reported in the parameter data. Also, it shows the list of key values that are returned by the device up to `allocation_length` bytes. If the `returned_length` is greater than the `allocation_length`, then the application did not provide an `allocation_length` large enough for all of the keys the device registered. The device driver does not consider it an error.

The **SYPRES_READRES** IOCTL issues the **persistent reserve in** command with the read reservations service action. The **STPRES_READRES** IOCTL uses the same following IOCTL structure as the **STPRES_READKEYS** IOCTL.

```
struct st_pres_in {
  ushort      version;
  ushort      allocation_length;
  uint        generation;
  ushort      returned_length;
  uchar       scsi_status;
  uchar       sense_key;
  uchar       scsi_asc;
  uchar       scsi_ascq;
  uchar       *reservation_info;
}
```

The allocation length is the maximum number of bytes of reservation descriptors that are returned in the **reservation info** buffer. The `returned_length` value indicates how many bytes of reservation descriptor values that device reported in the parameter data. Also, it shows the list of reservation descriptor values that are returned by the device up to `allocation_length` bytes. If the `returned_length` is greater than the `allocation_length`, then the application did not provide an `allocation_length` large enough for all of the reservation descriptors the device registered. The device driver does not consider it an error.

The **STPRES_CLEAR** IOCTL issues the **persistent reserve out** command with the clear service action. The following structure is the argument for this IOCTL.

```
struct st_pres_clear {
    ushort            version;
    uchar             scsi_status;
```

```
    uchar              sense_key;
    uchar              scsi_asc;
    uchar              scsi_ascq;
}
```

The **STPRES_CLEAR** IOCTL clears a persistent reservation and all persistent reservation registrations on the device.

The **STPRES_PREEMPT** IOCTL issues the **persistent reserve out** command with the preempt service action. The following structure is the argument for this IOCTL.

```
struct st_pres_preempt {
    ushort             version;
    unsigned long long preempt_key;
    uchar              scsi_status;
    uchar              sense_key;
    uchar              scsi_asc;
    uchar              scsi_ascq;
}
```

The **STPRES_PREEMPT** IOCTL preempts a persistent reservation or registration. The `preempt_key` contains the value of the registration key of the initiator that is to be preempted. The determination of whether it is the persistent reservation or registration that is preempted is made by the device. If the initiator corresponding to the `preempt_key` is associated with the reservation that is preempted, then the reservation is preempted and any matching registrations are removed. If the initiator corresponding to the `preempt_key` is not associated with the reservation that is preempted, then any matching registrations are removed. The SPC2 standard states that if a valid request for a preempt service action fails, it can be because of the condition in which another initiator has left the device. The suggested recourse in this case is for the preempting initiator to issue a logical unit reset and retry the preempting service action.

The **STPRES_PREEMPT_ABORT** IOCTL issues the **persistent reserve out** command with the preempt and abort service action. The **STPRES_PREEMPT_ABORT** IOCTL uses the same argument structure as the **STPRES_PREEMPT** IOCTL.

```
struct st_pres_preempt {
    ushort             version;
    unsigned long long preempt_key;
    uchar              scsi_status;
    uchar              sense_key;
    uchar              scsi_asc;
    uchar              scsi_ascq;
}
```

The **STPRES_PREEMPT_ABORT** IOCTL preempts a persistent reservation or registration and abort all outstanding commands from the initiators corresponding to the `preempt_key` registration key value. The `preempt_key` contains the value of the registration key of the initiator for which the preempt and abort is to apply. The determination of whether it is the persistent reservation or registration that is to be preempted is made by the device. If the initiator corresponding to the `preempt_key` is associated with the reservation that is preempted, then the reservation is preempted and any matching registrations are removed. If the initiator corresponding to the `preempt_key` is not associated with the reservation that is preempted, then any matching registrations are removed. Regardless of whether the preempted initiator holds the reservation, all outstanding commands from all initiators corresponding to the `preempt_key` are aborted.

The **STPRES_REGISTER** IOCTL issues the **persistent reserve out** command with the register service action. The following structure is the argument for this IOCTL.

```
struct st_pres_register {
   ushort            version;
   uchar             scsi_status;
   uchar             sense_key;
   uchar             scsi_asc;
   uchar             scsi_ascq;
}
```

The **STPRES_REGISTER** IOCTL registers the current host persistent reserve registration key value with the device. The **STPRES_REGISTER** IOCTL is only supported if the device is opened with a **reserve_type** set to persistent, otherwise an error of EACCESS is returned. The intended use of this IOCTL is to allow a preempted host to regain access to a shared device without requiring that the device is closed and reopened.

If a persistent reserve IOCTL fails, the return code is set to **-1** and the *errno* value is set to one of the following.

- **ENOMEM** Device driver cannot obtain memory to run the command.
- **EFAULT** An error occurred while the caller's data buffer was manipulated
- **EACCES** The device is opened with a **reserve_type** set to **reserve_6**
- **EINVAL** The requested IOCTL is not supported by this version of the device driver or invalid parameter that is provided in the argument structure
- **ENXIO** The device indicated that the persistent reserve command is not supported
- **EBUSY** The device returned a SCSI status byte of **RESERVATION CONFLICT** or **BUSY**. Or, the reservation for the device was preempted by another host and the device driver does not issue further commands.
- **EIO** Unknown I/O failure occurred on the command

Atape persistent reserve IOCTLs

The Atape device driver provides **Persistent Reserve** IOCTLs for application programs to manage their own Persistent Reserve support. These IOCTLs are defined in the header file **/usr/include/sys/Atape_pr.h**.

The following IOCTLs return Persistent Reserve information by using the **SCSI Persistent Reserve In** command.

- **STIOC_READ_RESERVEKEYS**
- **STIOC_READ_RESERVATIONS**
- **STIOC_READ_RESERVE_FULL_STATUS**

The following IOCTLs complete Persistent Reserve functions by using the **SCSI Persistent Reserve Out** command.

- **STIOC_REGISTER_KEY**
- **STIOC_REMOVE_REGISTRATION**
- **STIOC_CLEAR_ALL_REGISTRATIONS**
- **STIOC_PREEMPT_RESERVATION**
- **STIOC_PREEMPT_ABORT**
- **STIOC_CREATE_PERSISTENT_RESERVE**

The following IOCTLs are modified to handle both **SCSI-2 Reserve 6** and **Persistent Reserve** based on the current **reserve_type** setting.

- **SIOC_RESERVE**
- **SIOC_RELEASE**

The **STIOC_READ_RESERVEKEYS** IOCTL returns the reservation keys from the device. The argument for this IOCTL is the address of a **read_keys** structure. If the **reserve_key_list** pointer is NULL, then only the generation and length fields are returned. This action allows an application to first obtain the length of the **reserve_key_list** and malloc a return buffer before the IOCTL is issued with a **reserve_key_list** pointer to that buffer. If the return length is 0, then no reservation keys are registered with the device.

The following structure is used for this IOCTL.

```
struct read_keys
{
    uint  generation;           /* counter for PERSISTENT RESERVE OUT requests */
    uint  length;               /* number of bytes in the Reservation Key list */
    ullong *reserve_key_list;   /* list of reservation keys                    */
};
```

The **STIOC_READ_RESERVATIONS** IOCTL returns the current reservations from the device if any exist. The argument for this IOCTL is the address of a **read_reserves** structure. If the **reserve_list** pointer is NULL, then only the generation and length fields are returned. This action allows an application to first obtain the length of the **reserve_list** and malloc a return buffer before the IOCTL is issued with a **reserve_list** pointer to that buffer. If the return length is 0, then no reservations currently exist on the device.

The following structures are used for this IOCTL.

```
struct reserve_descriptor
{
    ullong      key;                /* reservation key                      */
    uint        scope_spec_addr;    /* scope-specific address               */
    uchar       reserved;
    uint        scope:4,            /* persistent reservation scope         */
                type:4;             /* reservation type                     */
    ushort      ext_length;         /* extent length                        */
};

struct read_reserves
{
    uint            generation;    /* counter for PERSISTENT RESERVE OUT requests   */
    uint            length;        /* number of bytes in the Reservation list       */
    struct reserve_descriptor* reserve_list; /* list of reservation key descriptors */
};
```

The **STIOC_READ_RESERVE_FULL_STATUS** IOCTL returns extended information for all reservation keys and reservations from the device if any exist. The argument for this IOCTL is the address of a **read_full_status** structure. If the **status_list** pointer is NULL, then only the generation and length fields are returned. This action allows an application to first obtain the length of the **status_list** and malloc a return buffer before the IOCTL is issued with a **status_list** pointer to that buffer. If the return length is 0, then no reservation keys or reservations currently exist on the device.

The following structures are used for this IOCTL.

```
struct transport_id
{
    uint format_code:2,
         rsvd:2,
```

```
        protocol_id:4;
};

struct fcp_transport_id
{
    uint format_code:2,
         rsvd:2,
         protocol_id:4;
    char    reserved1[7];
    ullong  n_port_name;
    char    reserved2[8];
};

struct scsi_transport_id
{
    uint format_code:2,
         rsvd:2,
         protocol_id:4;
    char    reserved1[1];
    ushort scsi_address;
    ushort obsolete;
    ushort target_port_id;
    char    reserved2[16];
};

struct sas_transport_id
{
    uint format_code:2,
         rsvd:2,
         protocol_id:4;
    char    reserved1[3];
    ullong  sas_address;
    char    reserved2[12];
};

struct status_descriptor
{
    ullong      key;                 /* reservation key               */
    char        reserved1[4];
    uint        rsvd:5,
                spc2_r:1,            /* future use for SCSI-2 reserve */
                all_tg_pt:1,         /* all target ports              */
                r_holder:1;          /* reservation holder            */
    uint        scope:4,             /* persistent reservation scope  */
                type:4;              /* reservation type              */
    char        reserved2[4];
    ushort      target_port_id;      /* relative target port id       */
    uint        descriptor_length;   /* additional descriptor length  */
    union {
      struct transport_id transport_id;  /* transport ID                 */
      struct fcp_transport_id fcp_id;    /* FCP transport ID             */
      struct sas_transport_id sas_id;    /* SAS transport ID             */
      struct scsi_transport_id scsi_id;  /* SCSI transport ID            */
      };
};

struct read_full_status
{
    uint            generation;     /* counter for PERSISTENT RESERVE OUT requests  */
    uint            length;         /* number of bytes for total status descriptors */
    struct status_descriptor *status_list; /* list of reserve status descriptors    */
};
```

The **STIOC_REGISTER_KEY** IOCTL registers a host reservation key on the device. The argument for this IOCTL is the address of an unsigned long key that can be 1 - 16 hexadecimal digits. If the key value is 0, then the device driver registers the configuration reserve key on the device. This key is either a user-specified host key or the device driver default host key.

If the host has a current persistent reservation on the device and the key is different from the current reservation key, the reservation is retained and the host reservation key is changed to the new key.

The **STIOC_REMOVE_REGISTRATION** IOCTL removes the host reservation key and reservation if one exists from the device. There is no argument for this IOCTL. The **SIOC_RELEASE** IOCTL can also be used to complete the same function.

The **STIOC_CLEAR_ALL_REGISTRATIONS** IOCTL clears all reservation keys and reservations on the device (if any exist) for the same host and any other host. There is no argument for this IOCTL.

The **STIOC_PREEMPT_RESERVATION** IOCTL registers a host reservation key on the device and then preempts the reservation that is held by another host if one exists. Or, it creates a new persistent reservation by using the host reservation key. The argument for this IOCTL is the address of an unsigned long key that can be 1 - 16 hexadecimal digits. If the key value is 0, then the device driver registers the configuration reserve key on the device. This key is either a user-specified host key or the device driver default host key.

The **STIOC_PREEMPT_ABORT** IOCTL registers a host reservation key on the device, preempts the reservation that is held by another host, and clears the task that is set for the preempted initiator if one exists. Or, it creates a new persistent reservation by using the host reservation key. The argument for this IOCTL is the address of an unsigned long key that can be 1 - 16 hexadecimal digits. If the key value is 0, then the device driver registers the configuration reserve key on the device. This key is either a user-specified host key or the device driver default host key.

The **STIOC_CREATE_PERSISTENT_RESERVE** IOCTL creates a persistent reservation on the device by using the host reservation key that was registered with the **STIOC_REGISTER_KEY** IOCTL. There is no argument for this IOCTL. The **SIOC_RESERVE** IOCTL can also be used to complete the same function.

The **SIOC_RESERVE** IOCTL reserves the device. If the **reserve_type** is set to **reserve_6**, the device driver issues a **SCSI Reserve 6** command. If the **reserve_type** is set to persistent, the device driver first registers the current host reservation key and then creates a persistent reservation. The current host reservation key can be either the configuration key for the device or a key that was registered previously with the **STIOC_REGISTER_KEY** IOCTL.

The **SIOC_RELEASE** IOCTL releases the device. If the **reserve_type** is set to **reserve_6**, the device driver issues a **SCSI Release 6** command. If the **reserve_type** is set to persistent, the device driver removes the host reservation key and reservation if one exists from the device.

If a persistent reserve IOCTL fails, the return code is set to **-1** and the *errno* value is set to one of the following.
- **ENOMEM** Device driver cannot obtain memory to complete the command.
- **EFAULT** An error occurred while the caller's data buffer was manipulated
- **EACCES** The current open is using a **reserve_type** set to **reserve_6**

- **EINVAL** Device does not support either the **SCSI Persistent Reserve In/Out** command, the service action for the command, or the sequence of the command such as issuing the **STIOC_REMOVE_REGISTRATION** IOCTL when no reservation key was registered for the host.
- **EBUSY** Device failed the command with reservation conflict. Either a **SCSI-2 Reserve 6** reservation is active, the sequence of the command such as issuing the **STIOC_CREATE_PERSISTENT_RESERVE** IOCTL when no reservation key was registered for the host, or the reservation for the device was preempted by another host and the device driver does not issue further commands.
- **EIO** Unknown I/O failure occurred on the command.

General IOCTL operations

This chapter describes the IOCTL commands that provide control and access to the tape and medium changer devices. These commands are available for all 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
Return device information.

STIOCMD
Issue the **AIX Pass-through** command.

STPASSTHRU
Issue the **AIX Pass-through** command.

SIOC_PASSTHRU_COMMAND
Issue the **Atape Pass-through** command.

SIOC_INQUIRY
Return inquiry data.

SIOC_REQSENSE
Return sense data.

SIOC_RESERVE
Reserve the device.

SIOC_RELEASE
Release the device.

SIOC_TEST_UNIT_READY
Issue a **SCSI Test Unit Ready** command.

SIOC_LOG_SENSE_PAGE
Return log sense data for a specific page.

SIOC_LOG_SENSE10_PAGE
Return log sense data for a specific page and Subpage.

SIOC_MODE_SENSE_PAGE
Return mode sense data for a specific page.

SIOC_MODE_SENSE_SUBPAGE
Return mode sense data for a specific page and subpage.

SIOC_MODE_SENSE
Return whole mode sense data include header, block descriptor, and page for a specific page.

SIOC_MODE_SELECT_PAGE
Set mode sense data for a specific page.

SIOC_MODE_SELECT_SUBPAGE
Set mode sense data for a specific page and subpage.

SIOC_INQUIRY_PAGE
Return inquiry data for a specific page.

SIOC_DISABLE_PATH
Manually disable (fence) a SCSI path for a device.

SIOC_ENABLE_PATH
Enable a manually disabled (fenced) SCSI path for a device.

SIOC_SET_PATH
Explicitly set the current path that is used by the device driver.

SIOC_QUERY_PATH
Query device and path information for the primary and first alternate SCSI path for a device. This IOCTL is obsolete but still supported. The **SIOC_DEVICE_PATHS** IOCTL can be used instead of this IOCTL.

SIOC_DEVICE_PATHS
Query device and path information for the primary and all alternate SCSI paths for the device.

SIOC_RESET_PATH
Issue an **Inquiry** command on each SCSI path that is not manually disabled (fenced) and enable the path if the **Inquiry** command succeeds.

SIOC_CHECK_PATH
Completes the same function as the **SIOC_RESET_PATH** IOCTL.

SIOC_QUERY_OPEN
Returns the process ID that currently has the device opened.

SIOC_RESET_DEVICE
Issues a SCSI target reset or SCSI lun reset (for FCP or SAS attached) to the device.

SIOC_DRIVER_INFO
Query the device driver information.

These IOCTL commands and their associated structures are defined by including the **/usr/include/sys/Atape.h** header file in the C program by using the functions.

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.devsubtype=ATAPE_3590
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;
info.devsubtype=ATAPE_3590;
```

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

```
info.devtype=DD_MEDIUM_CHANGER;
info.devsubtype=ATAPE_7337;
```

See the **Atape.h** header file for the defined **devsubtype** 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 returned **adapter_status** field is converted from the FCP codes that are defined in **/usr/include/sys/scsi_buf.h** to the SCSI codes defined in **/usr/include/sys/scsi.h**, if possible. This action 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, 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");
    }
```

STPASSTHRU

This IOCTL command issues the **AIX Pass-through** command that is supported by base AIX tape device drivers. The IOCTL command and structure are defined in the header files **/usr/include/sys/scsi.h** and **/usr/include/sys/tape.h**. Refer to AIX documentation for information about using the command.

SIOC_PASSTHRU_COMMAND

This IOCTL command issues the Atape device driver **Pass-through** command. The data structure that is used on this IOCTL is

```
struct scsi_passthru_cmd {
  uchar  command_length;      /* Length of SCSI command 6, 10, 12 or 16     */
  uchar  scsi_cdb[16];        /* SCSI command descriptor block              */
  uint   timeout_value;       /* Timeout in seconds or 0 for command default */
  uint   buffer_length;       /* Length of data buffer or 0                 */
  char   *buffer;             /* Pointer to data buffer or NULL             */
  uint   number_bytes;        /* Number of bytes transfered to/from buffer  */
  uchar  sense_length;        /* Number of valid sense bytes                */
  uchar  sense[MAXSENSE];     /* Sense data when sense length > 0           */
  uint   trace_length;        /* Number bytes in buffer to trace, 0 for none */
  char   read_data_command;   /* Input flag, set it to 1 for read type cmds  */
  char   reserved[27];
};
```

The **arg** parameter for the IOCTL is the address of a **scsi_passthru_cmd** structure.

The device driver issues the SCSI command by using the **command_length** and **scsi_cdb** fields. If the command receives data from the device (such as SCSI Inquiry), then the application must also set the **buffer_length** and **buffer pointer** for the return data along with the **read_data_command** set to **1**. For commands that send data to the device (such as **SCSI Mode Select**), the **buffer_length** and pointer is set for the send data and the **read_data_command** set to **0**. If the command has no data transfer, the buffer length is set to 0 and buffer pointer that is set to NULL.

The specified **timeout_value** field is used if not 0. If 0, then the device driver assigns its internal timeout value that is based on the SCSI command.

The **trace_length** field is normally used only for debug. It specifies the number of bytes on a data transfer type command that is traced when the AIX Atape device driver trace is running.

If the SCSI command fails, then the IOCTL returns **-1** and *errno* value is set for the failing command. If the device returned sense data for the failure, then the **sense_length** is set to the number of sense bytes returned in the sense field. If there was no sense data for the failure, the **sense_length** is 0.

If the SCSI command transfers data either to or from the device, then the **number_bytes** fields indicate how many bytes were transferred.

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,                 /* ECMA 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 input/output error (EIO), 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 reserves the device to the device driver. The specific SCSI command that is issued to the device depends on the current reservation type that is used by the device driver, either a **SCSI Reserve** or **Persistent Reserve**.

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 releases the current device driver reservation on the device. The specific SCSI command that is issued to the device depends on the current reservation type that is used by the device driver, either a **SCSI Reserve** or **Persistent Reserve**.

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 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 are set to zero. To obtain the entire log page that starts at a specific parameter code, set the **parm_pointer** field to the wanted code and the **len** field to zero. To obtain a specific number of parameter bytes, set the **parm_pointer** field to the wanted code. Then, 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 are always 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 3590 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_LOG_SENSE10_PAGE

This IOCTL command is enhanced to add a subpage variable from **SIOC_LOG_SENSE_PAGE**. It returns a log sense page or subpage from the device. The page is selected by specifying the **page_code** or **subpage_code** in the **log_sense10_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 are set to zero. To obtain the entire log page that starts at a specific parameter code, set the **parm_pointer** field to the wanted code and the **len** field to zero. To obtain a specific number of parameter bytes, set the **parm_pointer** field to the wanted code. Then, 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 are always the log page header. See the appropriate device manual to determine the supported log pages and content.

The data structure is

```
/* log sense page and subpage structure */
struct log_sense10_page
{
  uchar page_code;        /* [IN] log sense page code                      */
  uchar subpage_code;     /* [IN] log sense Subpage code                   */
  uchar reserved[2];
  unsigned short len;     /* [IN] specific allocation length for the data */
                          /* [OUT] number of valid bytes in
                             data(log_page_header_size+page_length)       */
  unsigned short parm_pointer;
                          /* [IN] specific parameter number at which
                             the data begins                              */
  char data[LOGSENSEPAGE]; /* [OUT] log sense page and Subpage data       */
};
```

An example of the **SIOC_LOG_SENSE10_PAGE** command is

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

  struct log_sense10_page logdata10;
  struct log_page_header *page_header;
  char text[80];

  logdata10.page_code = page;
  logdata10.subpage_code = subpage;
  logdata10.len = len;
  logdata10.parm_pointer = parm;
  page_header = (struct log_page_header *)logdata10.data;

  printf("Issuing log sense for page 0x%02X and subpage 0x%02X...\n",page,subpage);

  if (!ioctl (fd, SIOC_LOG_SENSE10_PAGE, &logdata10))
  {
  sprintf(text,"Log Sense Page 0x%02X, Subpage 0x%02X, Page Length %d
 Data",page,subpage,logdata10.len);
  dump_bytes(logdata10.data,logdata10.len,text);
  }
  else
  {
 perror ("The SIOC_LOG_SENSE10_PAGE ioctl failed");
 sioc_request_sense();
 }
```

SIOC_MODE_SENSE_PAGE

This IOCTL command returns a mode sense page from the device. The 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_MODE_SENSE_SUBPAGE

This IOCTL command returns the whole mode sense data, including header, block descriptor, and page code for a specific page or subpage from the device. The wanted page or subpage is inputted by specifying the **page_code** and **subpage_code** in the **mode_sense** structure.

The data structure is

```
struct mode_sense
{
uchar page_code;    /* [IN] mode sense page code */
uchar subpage_code; /* [IN] mode sense subpage code */
uchar reserved[6];
uchar cmd_code;     /* [OUT] SCSI Command Code: this field is set with */
                    /* SCSI command code which the device responded. */
                    /* x'5A' = Mode Sense (10) */
                    /* x'1A' = Mode Sense (6)  */
char data[MODESENSEPAGE]; /* [OUT] whole mode sense data include header,
block descriptor and page */
};
```

An example of the **SIOC_MODE_SENSE** command is

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

struct mode_sense modedata;
char text[80];
bzero(&modedata, sizeof(struct mode_sense));
modedata.page_code = page;
modedata.subpage_code = subpage;

printf("Issuing mode sense subpage for page 0x%02X subpage 0x%02X...\n",
page,subpage);

if (!ioctl (fd, SIOC_MODE_SENSE, &modedata))
 {
 sprintf(text,"Mode Sense 0x%02X Subpage 0x%02X cmd_code 0x%02X",
     modedata.page_code,modedata.subpage_code,modedata.cmd_code);
 dump_bytes((char *)&modedata, sizeof(struct mode_sense), text);
 }
else
```

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

SIOC_MODE_SELECT_PAGE

This IOCTL command sets device parameters in a specific mode page. The wanted 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 parameters that can be modified. The **arg** parameter for the IOCTL is the address of a **mode_sense_page** structure.

The data structure is

```
struct mode_sense_page
    {
    uchar page_code;            /* mode sense page code      */
    char data[MODESENSEPAGE];
    };
```

This data structure is also used for the **SIOC_MODE_SENSE_PAGE** IOCTL. The application must issue the **SIOC_MODE_SENSE_PAGE** IOCTL, and modify the wanted bytes in the returned **mode_sense_page** structure data field. Then, it issues this IOCTL with the modified fields in the structure.

SIOC_MODE_SELECT_SUBPAGE

This IOCTL command sets device parameters in a specific mode page and subpage. The wanted page and subpage are selected by specifying the **page_code** and **subpage_page** in the **mode_sense_subpage** structure. See the appropriate device manual to determine the supported mode pages, subpages, and parameters that can be modified. The **arg** parameter for the IOCTL is the address of a **mode_sense_subpage** structure.

The data structure is

```
  struct mode_sense_subpage
    {
    uchar page_code;            /* mode sense page code      */
    uchar subpage_code;         /* mode sense subpage code   */
    uint reserved:7,
         sp_bit:1;              /* mode select save page bit */
    char data[MODESENSEPAGE];
    };
```

This data structure is also used for the **SIOC_MODE_SENSE_SUBPAGE** IOCTL. The application must issue the **SIOC_MODE_SENSE_SUBPAGE** IOCTL, and modify the wanted bytes in the returned **mode_sense_subpage** structure data field. Then, it issues this IOCTL with the modified fields in the structure. If the device supports setting the **sp** bit for the mode page to **1**, then the **sp_bit** field can be set to **0** or **1**. If the device does not support the **sp** bit, then the **sp_bit** field must be set to **0**.

SIOC_QUERY_OPEN

This IOCTL command returns the ID of the process that currently has a device open. 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 by using the extended **open** parameter **SC_TMCP**, the process ID is returned for any other process that has the device open currently. Or, zero is returned if the device is not currently open. If the application opened the device without 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 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_DISABLE_PATH

This IOCTL command manually disables (fences) the device driver from using either the primary or an alternate SCSI path to a device until the **SIOC_ENABLE_PATH** command is issued for the same path that is manually disabled. The **arg** parameter on the IOCTL command specifies the path to be disabled. The primary path is path

1, the first alternate path 2, the second alternate path 3, and so on. This command can be used concurrently when the device is already open by another process by using the openx() extended parameter **SC_TMCP**.

This IOCTL command is valid only if the device has one or more alternate paths configured. Otherwise, the IOCTL command fails with *errno* set to EINVAL. The **SIOC_DEVICE_PATHS** IOCTL command can be used to determine the paths that are enabled or manually disabled.

An example of the **SIOC_DISABLE_PATH** command is

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

/* Disable primary SCSI path */
ioctl(fd, SIOC_DISABLE_PATH, PRIMARY_SCSI_PATH);

/* Disable alternate SCSI path */
ioctl(fd, SIOC_DISABLE_PATH, ALTERNATE_SCSI_PATH);
```

SIOC_ENABLE_PATH

This IOCTL command enables a manually disabled (fenced) path to a device that is disabled by **SIOC_DISABLE_PATH** IOCTL. The **arg** parameter on the IOCTL command specifies the path to be enabled. The primary path is path 1, the first alternate path 2, the second alternate path 3, and so on. This command can be used concurrently when the device is already open by another process by using the openx() extended parameter **SC_TMCP**.

The **SIOC_DEVICE_PATHS** IOCTL command can be used to determine the paths that are enabled or manually disabled.

SIOC_SET_PATH

This IOCTL command explicitly sets the current path to a device that the device driver uses. The **arg** parameter on the IOCTL command specifies the path to be set to the current path. The primary path is path 1, the first alternate path 2, the second alternate path 3, and so on. This command can be used concurrently when the device is already open by another process by using the openx() extended parameter **SC_TMCP**.

The **SIOC_DEVICE_PATHS** IOCTL command can be used to determine the current path the device driver is using for the device.

SIOC_DEVICE_PATHS

This IOCTL command returns a **device_paths** structure. The number of paths are configured to a device and a **device_path_t** path structure for each configured path. The device, HBA, and path information for the primary path are configured along with all alternate SCSI paths. This IOCTL command must be used instead of the **SIOC_QUERY_PATH** IOCTL that is obsolete. This command can be used concurrently when the device is already open by another process by using the openx() extended parameter **SC_TMCP**.

The data structures are

```
struct device_path_t {
  char  name[15];                /* logical device name             */
  char  parent[15];              /* logical parent name             */
  uchar id_valid;                /* obsolete and not set            */
  uchar id;                      /* SCSI target address of device   */
```

```
  uchar lun;                        /* SCSI logical unit of device       */
  uchar bus;                        /* SCSI bus for device               */
  uchar fcp_id_valid;               /* FCP scsi/lun id fields vaild      */
  unsigned long long fcp_scsi_id;   /* FCP SCSI id of device             */
  unsigned long long fcp_lun_id;    /* FCP logical unit of device        */
  unsigned long long fcp_ww_name;   /* FCP world wide name               */
  uchar enabled;                    /* path enabled                      */
  uchar drive_port_valid;           /* drive port field valid            */
  uchar drive_port;                 /* drive port number                 */
  uchar fenced;                     /* path fenced by disable ioctl      */
  uchar current_path;               /* Current path assignment           */
  uchar dynamic_tracking;           /* FCP Dynamic tracking enabled      */
  unsigned long long fcp_node_name; /* FCP node name                     */
  char type[16];                    /* Device type and model             */
  char serial[16];                  /* Device serial number              */
  uchar sas_id_valid;               /* FCP scsi/lun id fields vaild      */
  char  cpname[15];                 /* logical name of control path drive */
  uchar last_path;                  /* Last failure path */
  char reserved[4];
  };

struct device_paths {
  int number_paths;                 /* number of paths configured        */
  struct device_path_t path[MAX_SCSI_PATH];
  };
```

The **arg** parameter for the IOCTL is the address of a **device_paths** structure.

The **current_path** in the return structures is set to the current path the device uses for the device. If this IOCTL is issued to a medium changer smc logical driver, the **cpname** has the logical rmt name that is the control path drive for each smc logical path.

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.

Note: This IOCTL is obsolete but still supported. The **SIOC_DEVICE_PATHS** IOCTL must be used instead.

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     */
  uchar primary_fenced;                     /* Primary fenced by disable ioctl */
  uchar alternate_fenced;                   /* Alternate fenced by disable ioctl */
  uchar current_path;                       /* Current path assignment         */
  uchar primary_sas_id_valid;               /* Primary FCP scsi/lun id fields
                                               valid                           */
  uchar alternate_sas_id_valid;             /* Alternate FCP scsi/lun id fields
                                               valid                           */
  char reserved[55];  };
```

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->primary_fenced)
  printf("  Path Manually Disabled......... Yes\n");
else
  printf("  Path Manually Disabled......... 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");
  if (path->alternate_fenced)
    printf("  Path Manually Disabled......... Yes\n");
  else
    printf("  Path Manually Disabled......... No \n");

  }
}
```

SIOC_RESET_PATH and SIOC_CHECK_PATH

Both of these IOCTL commands check all SCSI paths to a device that are not manually disabled by the **SIOC_DISABLE_PATH** IOCTL. It is done by issuing a **SCSI Inquiry** command on each path to verify communication. If the command succeeds, then the path is enabled. If it fails, the path is disabled and is not used by the device driver. This command can be used concurrently when the device is already open by another process by using the openx() extended parameter **SC_TMCP**.

This IOCTL command returns the same data structure as the **SIOC_QUERY_PATH** IOCTL command with the updated path information for the primary and first alternate path. See the **SIOC_QUERY_PATH** IOCTL command for a description of the data structure and output information. If more than one alternate path is configured for the device, then the **SIOC_DEVICE_PATHS** IOCTL must be used to determine the paths that are enabled.

An example of the **SIOC_RESET_PATH** command is

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

int sioc_reset_path(void)
  {
  struct scsi_path path;

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

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

  return errno;
  }
```

SIOC_RESET_DEVICE

This IOCTL command issues a SCSI target reset to the device if parallel SCSI is attached or a SCSI lun reset if FCP/SAS is attached to the device. This IOCTL command can be used to clear a SCSI Reservation that is active on the device. This command can be used concurrently when the device is already open by another process by using the openx() extended parameter **SC_TMCP**.

There is no argument for this IOCTL and the **arg** parameter is ignored.

SIOC_DRIVER_INFO

This command returns the information about the currently installed Atape driver.

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

```
struct driver_info {
   uchar dd_name[16];              /* Atape driver name (Atape)          */
   uchar dd_version[16];           /* Atape driver version e.g. 12.0.8.0 */
   uchar os[16];                   /* Operating System (AIX)             */
   uchar os_version[32];           /* Running OS Version e.g. 6.1        */
   uchar sys_arch[16];             /* Sys Architecture (POWER or others) */
   uchar reserved[32];             /* Reserved for IBM Development Use   */
  };
```

An example of the **SIOC_DRIVER_INFO** command is

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

int sioc_driver_info()
  {
  struct driver_info dd_info;

  printf("Issuing driver info...\n");

  if (!ioctl (fd, SIOC_DRIVER_INFO, &dd_info))
     {
     printf("Driver Name:      %s\n",dd_info.dd_name);
     printf("Driver Version:   %s\n",dd_info.dd_version);
     printf("Operating System: %s\n",dd_info.os);
     printf("OS Version:       %s\n",dd_info.os_version);
     printf("System Arch:      %s\n",dd_info.sys_arch);
     }
     return errno;
     }
```

Tape IOCTL operations

The device driver supports the tape IOCTL commands available with the base AIX operating system. In addition, it supports a set of expanded tape IOCTL commands that give applications access to extra features and functions of the tape drives.

Overview

The following IOCTL commands are supported.

STIOCHGP
Set the block size.

STIOCTOP
Complete 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.

STIOCDM
Display the message on the display panel.

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_RECOVER_BUFFER
Recover the buffered data from the tape device.

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_FMR_TAPE
Create an FMR tape.

MTDEVICE
Obtain the device number of a drive in an IBM Enterprise Tape Library 3494.

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.

STIOC_GET_DENSITY
Get the current write density settings from the tape device.

STIOC_SET_DENSITY
Set the write density settings on the tape device.

STIOC_CANCEL_ERASE
Cancel an **erase immediate** command that is in progress.

GET_ENCRYPTION_STATE
This IOCTL can be used for application, system, and library-managed encryption. It allows a query only of the encryption status.

SET_ENCRYPTION_STATE
This IOCTL can be used only for application-managed encryption. It sets encryption state for application-managed encryption.

SET_DATA_KEY
This IOCTL can be used only for application-managed encryption. It sets the data key for application-managed encryption.

READ_TAPE_POSITION
Read current tape position in either short, long, or extended form.

SET_TAPE_POSITION
Set the current tape position to either a logical object or logical file position.

CREATE_PARTITION
Create one or more tape partitions and format the media.

QUERY_PARTITION
Query tape partitioning information and current active partition.

SET_ACTIVE_PARTITION
Set the current active tape partition.

ALLOW_DATA_OVERWRITE
: Set the drive to allow a subsequent data overwrite type command at the current position or allow a **CREATE_PARTITION** IOCTL when data safe (append-only) mode is enabled.

QUERY_LOGICAL_BLOCK_PROTECTION
: Query Logical Block Protection (LBP) support and its setup.

SET_LOGICAL_BLOCK_PROTECTION
: Enable or disable Logical Block Protection (LBP), set the protection method, and how the protection information is transferred.

STIOC_READ_ATTRIBUTE
: Read attribute values from medium auxiliary memory.

STIOC_WRITE_ATTRIBUTE
: Write attribute values to medium auxiliary memory.

VERIFY_TAPE_DATA
: Read the data from tape and verify its correction.

QUERY_RAO_INFO
: Query the maximum number and size of User Data Segments (UDS).

GENERATE_RAO
: Send a GRAO list to request the drive to generate a **Recommended Access Order** list.

RECEIVE_RAO
: Receive a **Recommended Access Order** list of UDS from the drive.

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 that uses the functions.

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 runs 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.

STERASE_IMM
: Erase the entire tape with the immediate bit set. The **st_count** parameter does not apply.

  This action issues the **erase** command to the device with the immediate bit set in the SCSI CDB. When this command is used, another process can cancel the erase operation by issuing the **STIOC_CANCEL_ERASE** IOCTL. The application that issued the STERASE_IMM still waits for the **erase** command to complete like the **STERASE st_op** if the **STIOC_CANCEL_ERASE** IOCTL is not issued. Refer to for a description of the **STIOC_CANCEL_ERASE** IOCTL.

STERASEGAP
: Erase the gap that was written to the tape. The **st_count** parameter does not apply.

STRETEN
: Start the rewind operation. The tape devices run the retension operation automatically when needed.

STWEOF
: Write the **st_count** number of filemarks.

STWEOF_IMM
: Write the **st_count** number of filemarks with the immediate bit set.

  This action issues a **write filemark** command to the device with the immediate bit set in the SCSI CDB. The device returns immediate status and the IOCTL also returns immediately. Unlike the **STWEOF st_op**, any buffered write data are not flushed to tape before the filemarks are written. This action can improve the time that it takes for a **write filemark** command to complete.

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. This operation is supported except on the IBM 3490E tape devices.

STFSSF
: Space forward to the first **st_count** number of contiguous filemarks.

STRSSF
: Space backward to the first **st_count** number of contiguous filemarks.

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, the command is not issued to the device, and the device driver returns 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 the **STIOCSETP** IOCTL command is issued, 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 that is written to the tape is transferred to the tape immediately. This operation guarantees that each record is on the tape, but it impacts performance.

STIOCQRYP parameters that cannot be changed with the **STIOCSETP** IOCTL command

The following parameters that are 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 set to On, device driver tracing is active.
- **hkwrd**

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

  If the currently mounted tape is write-protected, 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.
- **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.
- **max_scsi_xfer**

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

  If the tape device has the ACF installed, this parameter returns the current mode of the ACF. Otherwise, the value of ACF_NONE is returned. The ACF mode can be set from the operator panel on the tape device.
- **alt_pathing**

  This parameter is the configuration setting for path failover support. If the path failover support is enabled, this parameter is set to TRUE.
- **medium_type**

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

  This parameter is the configuration setting for the reservation type that the device driver uses when the device is reserved, either a **SCSI Reserve 6** command or a **SCSI Persistent Reserve** command.
- **reserve_key**

  This parameter is the reservation key the device driver uses with SCSI Persistent Reserve. If a configuration reservation key was specified, then this key can be either a 1-8 ASCII character key or a 1-16 hexadecimal key. If a configuration key was not specified, then the reservation key is a 16 hexadecimal key that the device driver generates.

Parameters that can be changed with `STIOCSETP` IOCTL command

The following parameters can be changed with the **`STIOCSETP`** IOCTL command.

- **`blksize`**

  This parameter specifies the effective block size for the tape device.
- **`autoload`**

  This parameter turns the autoload feature On and Off in the device driver. If set to On, the cartridge loader is treated as a large virtual tape.
- **`buffered_mode`**

  This parameter turns the buffered mode write On and Off.
- **`compression`**

  This parameter turns the hardware compression On and Off.
- **`trailer_labels`**

  If this parameter is set to On, 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 does 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 EOM or after EOM is reached.
- **`rewind_immediate`**

  This parameter turns the immediate bit On and Off in rewind commands. If set to On, the STREW tape operation runs faster. However, 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 set to On, 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, the device driver initializes the **`volid`** to UNKNOWN. If logging is active, the parameter is used to identify the volume in the tape log file entry. It is reset to UNKNOWN when the tape is unloaded.
- **`emulate_autoloader`**

  This parameter turns the emulate autoloader feature On and Off.
- **`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.
- **`logical_write_protect`**

  This parameter sets or resets the logical write protect of the current tape.

  Note: The tape position must be at the beginning of the tape to change this parameter from its current value.
- **`capacity_scaling and capacity_scaling_value`**

  The **`capacity_scaling`** parameter queries the capacity or logical length of the current tape or on a set operation changes the current tape capacity. On a query operation, this parameter returns the current capacity for the tape. It is one of the defined values such as SCALE_100, SCALE_75, SCALE_VALUE If the query returns SCALE_VALUE, then the **capacity_scaling_value** parameter is the current capacity. Otherwise, the **capacity_scaling** parameter is the current capacity.

On a set operation, if the **capacity_scaling** parameter is set to SCALE_VALUE then the **capacity_scaling_value** parameter is used to set the tape capacity. Otherwise, one of the other defined values for the **capacity_scaling** parameter is used.

Note:

1. The tape position must be at the beginning of the tape to change this parameter from its current value.
2. Changing this parameter destroys any existing data on the tape.

- **retain_reservation**

  When this parameter if set to 1, the device driver does not release the device reservation when the device is closed for the current open and any subsequent opens and closes until the **STIOCSETP** IOCTL is issued with **retain_reservation** parameter set to 0. The device driver still reserves the device on open to make sure that the previous reservation is still valid.

- **data_safe_mode**

  This parameter queries the current drive setting for data safe (append-only) mode. Also, on a set operation it changes the current data safe mode setting on the drive. On a set operation, a parameter value of zero sets the drive to normal (non-data safe) mode and a value of 1 sets the drive to data safe mode.

- **disable_sim_logging**

  This parameter turns the automatic logging of tape SIM/MIM data On and Off. By default, the device driver reads **Log Sense Page X'31'** automatically when device sense data indicates that data is available. The data is saved in the AIX error log. Reading **Log Sense Page X'31'** clears the current SIM/MIM data.

  Setting this bit disables the device driver from reading the Log Sense Page so an application can read and manage its own SIM/MIM data. The SIM/MIM data is saved in the AIX error log if an application reads the data with the **SIOC_LOG_SENSE_PAGE** or **STIOC_LOG_SENSE** IOCTLs.

- **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 results in a SCSI check condition if less data is read than the read system call requested. This action 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                */
      uint hkwrd;              /* trace hook word              */
      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 tranfer len       */
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
  #define SCALE_VALUE         4  /* use capacity_scaling_value below  */
uchar retain_reservation;         /* retain reservation                 */
uchar alt_pathing;                /* alternate pathing active           */
boolean emulate_autoloader;       /* emulate autoloader in random mode */
uchar medium_type;                /* tape medium type                   */
uchar density_code;               /* tape density code                  */
boolean disable_sim_logging;      /* disable sim/mim error logging      */
boolean read_sili_bit;            /* SILI bit setting for read commands*/
uchar capacity_scaling_value;     /* capacity scaling provided value    */
uchar reserve_type;               /* reservation type                   */
  #define RESERVE6_RESERVE    0  /* SCSI Reserve 6 type                */
  #define PERSISTENT_RESERVE  1  /* persistent reservation type        */
uchar reserve_key[8];             /* persistent reservation key         */
uchar data_safe_mode;             /* data safe mode                     */
ushort pew_size;         /* programmable early warning size   */
uchar reserved[9];
};
```

- **pew_size**

  With the tape parameter, the application is allowed to request the tape drive to create a zone that is called the programmable early warning zone (PEWZ) in the front of Early Warning (EW).

Figure 5. Programmable Early Warning Zone (PEWZ)

When a WRITE or WRITE FILE MARK (WFM) command writes data or filemark upon first reaching the PEWZ, Atape driver sets ENOSPC for Write and WFM to indicate that the current position reaches the PEWZ. After PEWZ is reached and before Early Warning is reached, all further writes and WFMs are allowed. The **TRAILER** parameter and the current design for LEOM (Logical End of Medium/Partition, or Early Warning Zone) and PEOM (Physical End of Medium/Partition) have no effect on the driver behavior in PEWZ.

For the application developers:

1. Two methods are used to determine PEWZ when the *errno* is set to ENOSPC for **Write** or **Write FileMark** command, since ENOSPC is returned for either EW or PEW.
   - Method 1: Issue a **Request Sense** IOCTL, check the sense key and ASC-ASCQ, and if it is 0x0/0x0007 (PROGRAMMABLE EARLY WARNING DETECTED), the tape is in PEW. If the sense key ASC-ASCQ is 0x0/0x0000 or 0x0/0x0002, the tape is in EW.
   - Method 2: Call **Read Position** IOCTL in long or extended form and check bpew and eop bits. If bpew = 1 and eop = 0, the tape is in PEW. If bpew = 1 and eop = 1, the tape is in EW.

   Atape driver requests the tape drive to save the mode page indefinitely. The PEW size is modified in the drive until a new setup is requested from the driver or application. The application must be programmed to issue the **Set** IOCTL to zero when PEW support is no longer needed, as Atape drivers do not complete this function. PEW is a setting of the drive and not tape. Therefore, it is the same on each partition, should partitions exist.
2. Encountering the PEWZ does not cause the device server to run a synchronize operation or terminate the command. It means that the data or filemark is written in the cartridge when a check condition with PROGRAMMABLE EARLY WARNING DETECTED is returned. But, the Atape driver still returns the counter to less than zero (**-1**) for a **write** command or a failure for **Write FileMark** IOCTL call with ENOSPC error. In this way, it forces the application to use one of the methods to check PEW or EW. When the application determines ENOSPC comes from PEW, it reads the requested write data or filemark that are written into the cartridge and reach or pass the PEW point. The application can issue a **Read position** IOCTL to validate the tape position.

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 input/output control (IOCTL) command flushes the tape buffers to the tape immediately.

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);
    }
```

STIOCDM

This IOCTL command displays and manipulates one or two messages on the message display. The message that is sent with this call does not always remain on the display. It depends on the current state of the tape device.

The input data structure is

```
#define MAXMSGLEN     8
struct stdm_s
   {
  char dm_func;                      /* function code */
                                     /* function selection */
   #define DMSTATUSMSG 0x00          /* general status message */
   #define DMDVMSG     0x20          /* demount/verify message */
   #define DMMIMMED    0x40          /* mount with immediate action indicator*/
   #define DMDEMIMMED  0xE0          /* demount/mount with immediate action */
                                     /* message control */
   #define DMMSG0      0x00          /* display message 0 */
   #define DMMSG1      0x04          /* display message 1 */
   #define DMFLASHMSG0 0x08          /* flash message 0 */
   #define DMFLASHMSG1 0x0C          /* flash message 1 */
   #define DMALTERNATE 0x10          /* alternate message 0 and message 1 */
  char dm_msg0[MAXMSGLEN];           /* message 0 */
  char dm_msg1[MAXMSGLEN];           /* message 1 */
  };
```

An example of the **STIOCDM** command is

```
#include <sys/Atape.h>
struct stdm_s stdm;
stdm.dm_func=DMSTATUSMSG|DMMSG0;
bcopy("SSD",stdm.dm_msg0,8);
if (ioctl(tapefd,STIOCDM,&stdm)<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 **block_type** and **curpos** fields are used during a **set** operation. The tape position is defined as where the next **read** or **write** operation occurs. The **query** function can be used independently or with the **set** function. Also, the **set** function can be used independently or with the **query** function.

The **block_type** field is set to **QP_LOGICAL** when a SCSI logical **blockid** format is wanted. During a query operation, the **curpos** field is set to a simple **unsigned int**.

On IBM 3490 tape drives only, the **block_type** field can be set to **QP_PHYSICAL**. Setting this **block_type** on any other device is ignored and defaults to **QP_LOGICAL**. After a **set** operation, the position is at the logical block that is indicated by the **curpos** field. If the **block_type** field is set to **QP_PHYSICAL**, the **curpos** field that is returned is a vendor-specific **blockid** format from the tape device. When **QP_PHYSICAL** is used for a **query** operation, the **curpos** field is used only in a subsequent set operation with **QP_PHYSICAL**. This function completes a high speed **locate** operation. Whenever possible, use **QP_PHYSICAL** because it is faster. This advantage is obtained only when the **set** operation uses the **curpos** field from the **QP_PHYSICAL** query.

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 - 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, 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. The **bcu** and **bycu** settings indicate whether these fields contain valid data. The block ID of the next block of data that transferred to or from the physical tape is returned in the **tapepos** field.

The partition number field that is 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          */
   #define QP_LOGICAL  0             /* SCSI logical block ID format            */
   #define QP_PHYSICAL 1             /* 3490 only, vendor-specific block ID format */
                                     /* ignored for all other devices           */
   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        */
   boolean bcu;    /* number of blocks in buffer is unknown     */
   boolean bycu;   /* number of bytes in buffer is unknown      */
   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;
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 that is collected from the tape device, or it issues a new **Request Sense** command and returns the collected data. If **LASTERROR** is requested, 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, 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 volume logging is set to On, the log sense data is 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_RECOVER_BUFFER

This IOCTL command recovers the buffer data from the tape device. It is typically used after an error occurs during a **write** operation that prevents the data in the tape device buffers from being written to tape. The **STIOCQRYPOS** command can be used before this IOCTL command to determine the number of blocks and the bytes of data that is in the device buffers.

Each **STIOC_RECOVER_BUFFER** IOCTL call returns one block of data from the device. This **ioctl** command can be issued multiple times to completely recover all the buffered data from the device.

After the IOCTL command is completed, the **ret_len** field contains the number of bytes returned in the application buffer for the block. If no blocks are in the tape device buffer, then the **ret_len** value is set to zero.

The output data structure is

```
struct buffer_data
    {
     char *buffer;
     int bufsize;
     int ret_len;
    };
```

An example of the **STIOC_RECOVER_BUFFER** command is

```
struct buffer_data bufdata;

  bufdata.bufsize = 256 * 1024;
  bufdata.buffer = malloc(256 * 1024);

  if (ioctl(tapefd,STIOC_RECOVER_BUFFER,&bufdata)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    }
  else
    {
    printf("Returned bytes=%d",bufdata.ret_len);
    }
```

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 with 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 currently 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 unload the current tape. The volume name can be queried and set by using the **STIOCQRYP** and **STIOCSETP** IOCTLs.

The argument that is 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 in the **/var/adm/ras** system directory. The device driver stores up to three dumps in this directory. The first dump that is created is named **Atape.rmtx.dump1**, where *x* is the device number, for example, **rmt0**. The second and third dumps are **dump2** and **dump3**. After a third dump file is created, the next dump starts at **dump1** again and overlays the previous **dump1** file.

The argument that is used for this command is NULL to dump to the system directory. Or, it is a character pointer to a buffer that contains the path and file name 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 3590.dump */
   char *dump_name = "3590.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 be retrieved from the device by 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, it stores the dump to either a host specified file or in the **/var/adm/ras** system directory. The device driver stores up to three dumps in this directory. The first dump that is created is named **Atape.rmtx.dump1**, where *x* is the device number, for example **rmt0**. The second and third dumps are **dump2** and **dump3**. After a 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 by using the **STIOC_FORCE_DUMP** IOCTL.

The argument that is used for this command is NULL to dump to the system directory. Or, it is a character pointer to a buffer that contains the path and file name 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 3590.dump */
   char *dump_name = "3590.dump";
   if (ioctl(tapefd,STIOC_READ_DUMP,dump_name)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }
```

STIOC_LOAD_UCODE

This IOCTL command downloads microcode to the device. The argument that is used for this command is a character pointer to a buffer that contains the path and file name 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_FMR_TAPE

This IOCTL command creates an FMR tape. The tape is created with the current microcode loaded in the tape device.

There are no arguments for this IOCTL command.

An example of the **STIOC_FMR_TAPE** command is

```
 /* create fmr tape */
    if (ioctl(tapefd,STIOC_FMR_TAPE,NULL)<0)
    {
    printf("IOCTL failure. errno=%d",errno);
    exit(errno);
    }
```

MTDEVICE (Obtain device number)

This IOCTL command obtains the device number that is used for communicating with the IBM TotalStorage Enterprise library 3494.

The structure of the IOCTL request is

```
int device;
if (ioctl(tapefd,MTDEVICE,&device)<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 used normally 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 that are used with this IOCTL are

```
typedef 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 */
            res_1                 :5;  /* reserved */
     uchar  reserved[2];               /* reserved */
     uchar  bits_per_mm[3];            /*bits per mm */
     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;
 }
```

STIOC_GET_DENSITY and STIOC_SET DENSITY

The **STIOC_GET_DENSITY** IOCTL is used to query the current write density format settings on the tape drive. The current density code from the drive **Mode Sense** header, the **Read/Write Control Mode** page default density, and pending density are returned.

The **STIOC_SET_DENSITY** IOCTL is used to set a new write density format on the tape drive by using the default and pending density fields. The **density code** field is not used and ignored on this IOCTL. The application can specify a new write density for the current loaded tape only or as a default for all tapes. Refer to the examples.

The application must get the current density settings first before the current settings are modified. If the application specifies a new density for the current loaded tape only, then the application must issue another **Set Density** IOCTL after the current tape is unloaded and the next tape is loaded to either the default maximum density or a new density. This action ensures the tape drive uses the correct density. If the application specifies a new default density for all tapes, the setting remains in effect until changed by another set density IOCTL or the tape drive is closed by the application.

Following is the structure for the **STIOC_GET_DENSITY** and **STIOC_SET_DENSITY** IOCTLs.

```
struct density_data_t
{
    char  density_code;        /* mode sense header density code         */
    char  default_density;     /* default write density                  */
    char  pending_density;     /* pending write density                  */
    char  reserved[9];
};
```

Note:

1. The IOCTLs are only supported on tape drives that can write multiple density formats. Refer to the Hardware Reference for the specific tape drive to determine whether multiple write densities are supported. If the tape drive does not support the IOCTLs, *errno* EINVAL is returned.
2. The device driver always sets the default maximum write density for the tape drive on every open system call. Any previous **STIOC_SET_DENSITY** IOCTL values from the last open are not used.
3. If the tape drive detects an invalid density code or cannot complete the operation on the **STIOC_SET_DENSITY** IOCTL, the *errno* is returned. Then, the current drive density settings before the IOCTL are restored.
4. The `struct density_data_t` defined in the header file is used for both IOCTLs. The **density_code** field is not used and ignored on the **STIOC_SET_DENSITY** IOCTL.

Examples

```
struct density_data_t data;

/* open the tape drive          */
/* get current density settings */
rc = ioctl(fd, STIOC_GET_DENSITY, %data);

/* set 3592 J1A density format for current loaded tape only */
data.default_density = 0x7F;
data.pending_density = 0x51;
rc = ioctl(fd, STIOC_SET_DENSITY, %data);

/* unload tape    */
/* load next tape */
/* set 3592 E05 density format for current loaded tape only */
data.default_density = 0x7F;
data.pending_density = 0x52;
rc = ioctl(fd, STIOC_SET_DENSITY, %data);

/* unload tape    */
/* load next tape */
/* set default maximum density for current loaded tape */
data.default_density = 0;
data.pending_density = 0;
rc = ioctl(fd, STIOC_SET_DENSITY, %data);

/* close the tape drive             */
```

```
/* open the tape drive              */
/* set 3592 J1A density format for current loaded tape and all subsequent tapes */
data.default_density = 0x51;
data.pending_density = 0x51;

rc = ioctl(fd, STIOC_SET_DENSITY, %data);
```

STIOC_CANCEL_ERASE

The **STIOC_CANCEL_ERASE** IOCTL is used to cancel an erase operation currently in progress. This action happens when an application issued the **STIOCTOP** IOCTL with the **st_op** field that specifies **STERASE_IMM**. The application that issued the erase and is waiting for it to complete then returns immediately with *errno* ECANCELLED. This IOCTL always returns 0 whether an **erase immediate** operation is in progress or not.

This IOCTL can be issued only when the openx() extended parameter **SC_TMCP** is used to open the device. It happens when the application that issued the erase still has the device currently open. There is no argument for this IOCTL and the **arg** parameter is ignored.

GET_ENCRYPTION_STATE

This IOCTL command queries the drive's encryption method and state. The data structure that is used for this IOCTL is for all of the supported operating systems.

```
struct encryption_status {
      uchar encryption_capable;    /* (1)Set this field as a boolean based on the
    capability of the drive */
      uchar encryption_method;     /* (2)Set this field to one of the
     #defines METHOD_* below              */
#define METHOD_NONE 0             /* Only used in GET_ENCRYPTION_STATE */
#define METHOD_LIBRARY 1          /* Only used in GET_ENCRYPTION_STATE */
#define METHOD_SYSTEM 2           /* Only used in GET_ENCRYPTION_STATE */
#define METHOD_APPLICATION 3      /* Only used in GET_ENCRYPTION_STATE */
#define METHOD_CUSTOM 4           /* Only used in GET_ENCRYPTION_STATE */
#define METHOD_UNKNOWN 5          /* Only used in GET_ENCRYPTION_STATE */

       uchar encryption_state;     /* (3) Set this field to one of the
                               #defines STATE_* below */
#define STATE_OFF  0               /* Used in GET/SET_ENCRYPTION_STATE */
#define STATE_ON 1                /* Used in GET/SET_ENCRYPTION_STATE */
#define STATE_NA 2                /* Only used in GET_ENCRYPTION_STATE*/

       uchar[13] reserved;
    };
```

An example of the **GET_ENCRYPTION_STATE** command is

```
int qry_encrytion_state (void)
{
   int rc = 0;
   struct encryption_status encryption_status_t;

   printf("issuing query encryption status...\n");
   memset(,&encryption_status_t 0, sizeof(struct encryption_status));
   rc = ioctl(fd, GET_ENCRYPTION_STATE, &encryption_status_t);
   if(rc == 0)
   {
      if(encryption_status_t.encryption_capable)
   printf("encryption capable......Yes\n");
      else
   printf("encryption capable......No\n");
      switch(encryption_status_t.encryption_method)
      {
```

```
    case METHOD_NONE:
     printf("encryption method.......METHOD_NONE\n");
     break;
    case METHOD_LIBRARY:
     printf("encryption method.......METHOD_LIBRARY\n");
     break;
    case METHOD_SYSTEM:
     printf("encryption method.......METHOD_SYSTEM\n");
     break;
    case METHOD_APPLICATION:
     printf("encryption method.......METHOD_APPLICATION\n");
     break;
    case METHOD_CUSTOM:
     printf("encyrpiton method.......METHOD_CUSTOM\n");
     break;
    case METHOD_UNKNOWN:
     printf("encryption method.......METHOD_UNKNOWN\n");
     break;

    default:
     printf("encryption method.......Error\n");
    }

    switch(encryption_status_t.encryption_state)
    {
    case STATE_OFF:
     printf("encryption state........OFF\n");
     break;
    case STATE_ON:
     printf("encryption state........ON\n");
     break;
    case STATE_NA:
     printf("encryption state........NA\n");
     break;

    default:
     printf("encryption state......Error\n");
    }
 }

 return rc;
}
```

SET_ENCRYPTION_STATE

This IOCTL command allows set encryption state only for application-managed encryption. On unload, some of the drive settings can be reset to default. To set the encryption state, the application must issue this IOCTL after a tape is loaded and at BOP.

The data structure used for this IOCTL is the same as the one for **GET_ENCRYPTION_STATE**. An example of the **SET_ENCRYPTION_STATE** command is

```
int set_encryption_state(int option)
{
   int rc = 0;
   struct encryption_status encryption_status_t;

   printf("issuing query encryption status...\n");
   memset(,&encryption_status_t 0, sizeof(struct encryption_status));
   rc = ioctl(fd, GET_ENCRYPTION_STATE, );&encryption_status_t
   if(rc < 0) return rc;

   if(option == 0)
       encryption_status_t.encryption_state = STATE_OFF;
   else if(option == 1)
```

```
        encryption_status_t.encryption_state = STATE_ON;
    else
    {
        printf("Invalid parameter.\n");
        return -EINVAL;
    }

    printf("Issuing set encryption state......\n");
    rc = ioctl(fd, SET_ENCRYPTION_STATE, &encryption_status_t);

    return rc;
}
```

SET_DATA_KEY

This IOCTL command allows set the data key only for application-managed encryption. The data structure that is used for this IOCTL is for all of the supported operating systems.

```
struct data_key
{
    uchar[12 data_key_index;
    uchar data_key_index_length;
    uchar[15] reserved1;
    uchar[32] data_key;
    uchar[48] reserved2;
};
```

An example of the **SET_DATA_KEY** command is

```
int set_datakey(void)
{
   int rc = 0;
   struct data_key encryption_data_key_t;

   printf("Issuing set encryption data key......\n");
   memset(&encryption_data_key_t, 0, sizeof(struct data_key));
   /* fill in your data key here, then issue the following ioctl*/
   rc = ioctl(fd, SET_DATA_KEY, &encryption_data_key_t);
   return rc;
}
```

READ_TAPE_POSITION

The **READ_TAPE_POSITION** IOCTL is used to return **Read Position** command data in either the short, long, or extended form. The type of data to return is specified by setting the **data_format** field to either **RP_SHORT_FORM**, **RP_LONG_FORM**, or **RP_EXTENDED_FORM**.

The data structures that are used with this IOCTL are

```
#define RP_SHORT_FORM        0x00
#define RP_LONG_FORM         0x06
#define RP_EXTENDED_FORM     0x08

struct short_data_format {
  uint bop:1,                /* beginning of partition              */
       eop:1,                /* end of partition                    */
       locu:1,               /* 1 means num_buffer_logical_obj field is
                                unknown                             */
       bycu:1,               /* 1 means the num_buffer_bytes field is
                                unknown                             */
       rsvd :1,
       lolu:1,               /* 1 means the first and last logical
                                obj position fields are unknown     */
       perr: 1,              /* 1 means the position fields have
```

```
                                 overflowed and can not be reported      */
       bpew :1;                /*  beyond programmable early warning      */
  uchar active_partition;      /* current active partition               */
  char reserved[2];
  uint first_logical_obj_position; /* current logical object position     */
  uint last_logical_obj_position;  /* next logical object to be transferred
                                     to tape                             */
  uint num_buffer_logical_obj; /* number of logical objects in buffer     */
  uint num_buffer_bytes;       /* number of bytes in buffer               */
  char reserved1;
  };

struct long_data_format {
  uint bop:1,           /* beginning of partition       */
       eop:1,           /* end of partition             */
       rsvd1:2,
       mpu:1,           /* 1 means the logical file id field
                           in unknown                   */
       lonu:1,          /* 1 means either the partition number
                           or logical obj number field
                           are unknown                  */
       rsvd2:1,
       bpew :1;         /*  beyond programmable early
                            warning                     */
  char reserved[6];
  uchar active_partition; /* current active partition     */
  ullong logical_obj_number; /* current logical object
                               position                 */
  ullong logical_file_id; /* number of filemarks from bop
                            and current logical position */
  ullong obsolete;
  };

struct extended_data_format {
  uint bop:1,            /* beginning of partition       */
       eop:1,            /* end of partition             */
       locu:1,           /* 1 means num_buffer_logical_obj field
                            is unknown                   */
       bycu:1,           /* 1 means the num_buffer_bytes field
                            is unknown                   */
       rsvd :1,
       lolu:1,           /* 1 means the first and last logical obj
                            position fields are unknown  */
       perr: 1,          /* 1 means the position fields have
                            overflowed and can not be reported */
       bpew :1;          /*  beyond programmable early warning */
  uchar active_partition; /* current active partition     */
  ushort additional_length;
  uint num_buffer_logical_obj;       /* number of logical objects in buffer */
  ullong first_logical_obj_position; /* current logical object position     */
  ullong last_logical_obj_position;  /* next logical object to be transferred
                                        to tape                             */
  ullong num_buffer_bytes;           /* number of bytes in buffer           */
  char reserved;
  };

struct read_tape_position{
  uchar data_format; /* Specifies the return data format either short,
 long or extended as defined above */
  union
    {
    struct short_data_format rp_short;
    struct long_data_format rp_long;
    struct extended_data_format rp_extended;
    char reserved[64];
    } rp_data;
  };
```

Example of the **READ_TAPE_POSITION** IOCTL

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

struct read_tape_position rpos;

    printf("Reading tape position long form....\n");
    rpos.data_format = RP_LONG_FORM;
    if (ioctl (fd, READ_TAPE_POSITION, &rpos) <0)
       return errno;

       if (rpos.rp_data.rp_long.bop)
       printf("    Beginning of Partition ..... Yes\n");
     else
       printf("    Beginning of Partition ..... No\n");
      if (rpos.rp_data.rp_long.eop)
       printf("    End of Partition ........... Yes\n");
     else
       printf("    End of Partition ........... No\n");
      if (rpos.rp_data.rp_long.bpew)
       printf("    Beyond Early Warning ... ... Yes\n");
     else
       printf("    Beyond Early Warning ....... No\n");
      if (rpos.rp_data.rp_long.lonu)
       {
       printf("    Active Partition ........... UNKNOWN \n");
       printf("    Logical Object Number ...... UNKNOWN \n");
       }
     else
       {
       printf("    Active Partition ... ....... %u \n",
            rpos.rp_data.rp_long.active_partition);
       printf("    Logical Object Number ...... %llu \n",
            rpos.rp_data.rp_long.logical_obj_number);
        }

      if (rpos.rp_data.rp_long.mpu)
       printf("    Logical File ID ............ UNKNOWN \n");
     else
       printf("    Logical File ID ............ %llu \n",
              rpos.rp_data.rp_long.logical_file_id);
```

SET_TAPE_POSITION

The **SET_TAPE_POSITION** IOCTL is used to position the tape in the current active partition to either a logical block id or logical filemark. The **logical_id_type** field in the IOCTL structure specifies either a logical block or logical filemark.

The data structure that is used with this IOCTL is

```
#define LOGICAL_ID_BLOCK_TYPE   0x00
#define LOGICAL_ID_FILE_TYPE    0x01

struct set_tape_position{
  uchar logical_id_type;    /* Block or file as defined above */
  ullong logical_id;        /* logical object or logical file to position to */
  char reserved[32];
  };
```

Examples of the **SET_TAPE_POSITION** IOCTL

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

  struct set_tape_position setpos;

  /* position to logical block id 10 */
  setpos.logical_id_type =  LOGICAL_ID_BLOCK_TYPE
```

```
  setpos.logical_id = 10;
  ioctl(fd, SET_TAPE_POSITION, &setpos);

  /* position to logical filemark 4 */
  setpos.logical_id_type =  LOGICAL_ID_FILE_TYPE
  setpos.logical_id = 4;
  ioctl(fd, SET_TAPE_POSITION, &setpos);
```

SET_ACTIVE_PARTITION

The **SET_ACTIVE_PARTITION** IOCTL is used to position the tape to a specific partition. Then, it becomes the current active partition for subsequent commands and a specific logical bock id in the partition. To position to the beginning of the partition, the **logical_block_id** field is set to 0.

The data structure that is used with this IOCTL is

```
struct set_active_partition {
  uchar partition_number;         /* Partition number 0-n to change to     */
  ullong logical_block_id;        /* Blockid to locate to within partition */
  char reserved[32];
  };
```

Examples of the **SET_ACTIVE_PARTITION** IOCTL

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

  struct set_active_partition partition;

  /* position the tape to partition 1 and logical block id 12 */
  partition.partition_number = 1;
  partition.logical_block_id = 12;
  ioctl(fd, SET_ACTIVE_PARTITION, &partition);

  /* position the tape to the beginning of partition 0 */
  partition.partition_number = 0;
  partition.logical_block_id = 0;
  ioctl(fd, SET_ACTIVE_PARTITION, &partition);
```

QUERY_PARTITION

The **QUERY_PARTITION** IOCTL is used to return partition information for the tape drive and the current media in the tape drive. It includes the current active partition the tape drive is using for the media. The **number_of partitions** field is the current number of partitions on the media and the **max_partitions** is the maximum partitions that the tape drive supports. The **size_unit** field can be either one of the defined values or another value such as 8. It is used with the **size array** field value for each partition to specify the actual size partition sizes. The **partition_method** field is either Wrap-wise Partitioning or Longitudinal Partitioning. Refer to “CREATE_PARTITION” on page 70 for details.

The data structure that is used with this IOCTL is

```
The define for "partition_method":
#define UNKNOWN_TYPE                       0 /* vendor-specific or unknown         */
#define WRAP_WISE_PARTITION                1 /* Wrap-wise Partitioning without RABF */
#define LONGITUDINAL_PARTITION             2 /* Longitudinal Partitioning          */
#define WRAP_WISE_PARTITION_WITH_FASTSYNC  3 /* Wrap-wise Partitioning with RABF    */
```

```
The define for "size_unit":
#define SIZE_UNIT_BYTES     0      /* Bytes                              */
#define SIZE_UNIT_KBYTES    3      /* Kilobytes                          */
#define SIZE_UNIT_MBYTES    6      /* Megabytes                          */
#define SIZE_UNIT_GBYTES    9      /* Gigabytes                          */
#define SIZE_UNIT_TBYTES   12      /* Terabytes                          */
```

```
struct query_partition {
  uchar max_partitions;          /* Max number of supported partitions      */
  uchar active_partition;        /* current active partition on tape       */
  uchar number_of_partitions;    /* Number of partitions from 1 to max      */
  uchar size_unit;               /* Size unit of partition sizes below      */
  ushort size[MAX_PARTITIONS];   /* Array of partition sizes in size units  */
                                 /* for each partition, 0 to (number - 1)   */
  uchar partition_method;        /* partitioning type                       */
  char reserved [31];
  };
```

Examples of the **QUERY_PARTITION** IOCTL

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

  struct query_partition partition;
  int i;

  if (ioctl(fd, QUERY_PARTITION, &partition) < 0)
     return errno;

  printf("  Max supported partitions ... %d\n",partition.max_partitions);
  printf("  Number of partitions ....... %d\n",partition.number_of_partitions);
  printf("  Active partition ........... %d\n",partition.active_partition);
  printf(" Partition Method ......... %d\n",partition.partition_method);
  if (partition.size_unit == SIZE_UNIT_BYTES)
    printf("  Partition size unit ........ Bytes\n");
  else if  (partition.size_unit == SIZE_UNIT_KBYTES)
    printf("  Partition size unit ........ Kilobytes\n");
  else if  (partition.size_unit == SIZE_UNIT_MBYTES)
    printf("  Partition size unit ........ Megabytes\n");
  else if  (partition.size_unit == SIZE_UNIT_GBYTES
    printf("  Partition size unit ........ Gigabytes\n");
  else if  (partition.size_unit == SIZE_UNIT_TBYTES)
    printf("  Partition size unit ........ Terabytes\n");
  else
    printf("  Partition size unit ........ %d\n",partition.size_unit);

  for (i=0; i < partition.number_of_partitions; i++)
    printf("  Partition %d size ........... %d\n",i,partition.size[i]);
```

CREATE_PARTITION

The **CREATE_PARTITION** IOCTL is used to format the current media in the tape drive into 1 or more partitions. The number of partitions to create is specified in the **number_of_partitions** field. When more than one partition is created, the **type** field specifies the type of partitioning, either FDP, SDP, or IDP. The tape must be positioned at the beginning of tape (partition 0 logical block id 0) before this IOCTL is used.

If the **number_of_partitions** field to create in the IOCTL structure is one partition, all other fields are ignored and not used. The tape drive formats the media by using its default partitioning type and size for a single partition.

When the **type** field in the IOCTL structure is set to either FDP or SDP, the **size_unit** and **size** fields in the IOCTL structure are not used. When the type field in the IOCTL structure is set to IDP, the **size_unit** with the **size** fields are used to specify the size for each partition.

There are two partition types: Wrap-wise Partitioning (Figure 6 on page 71) optimized for streaming performance, and Longitudinal Partitioning (Figure 7 on page 71) optimized for random access performance. Media is always partitioned into 1 by default. Or, more than one partition where the data partition always exists as partition 0 and other extra index partition 1 to n can exist.

A WORM media cannot be partitioned and the **Format Medium** commands are rejected. Attempts to scale a partitioned media is accepted. However, only if you use the correct **FORMAT** field setting, as part of scaling the volume is set to a single data partition cartridge.

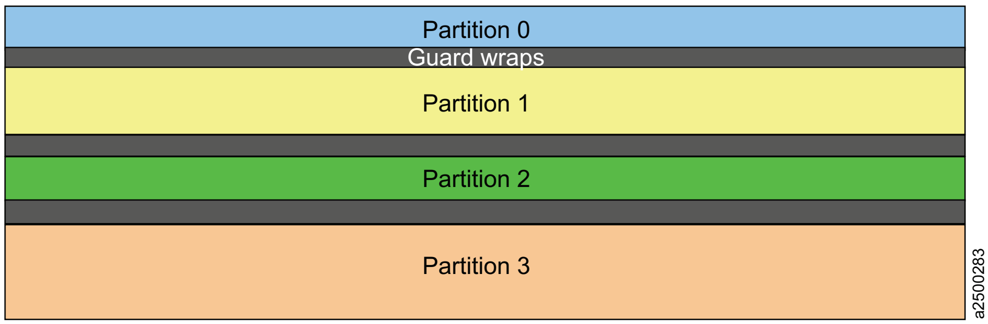

Figure 6. Wrap-wise partitioning

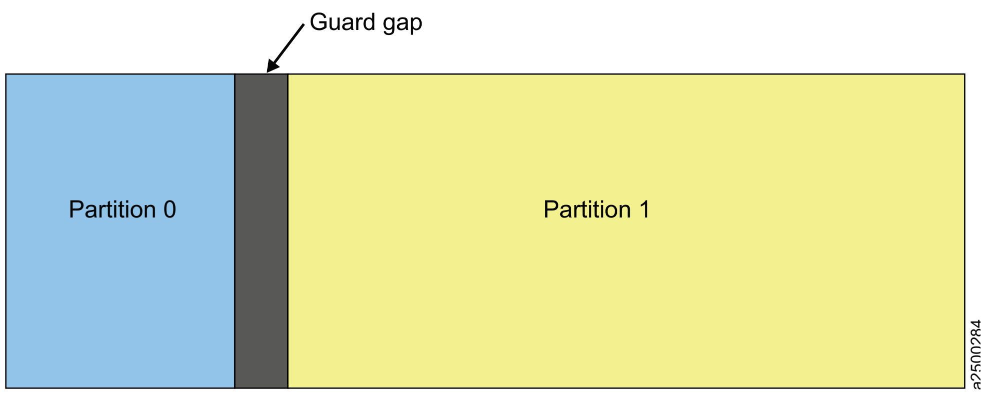

Figure 7. Longitudinal partitioning

The following chart lists the maximum number of partitions that the tape drive supports.

Table 3. Number of supported partitions

Drive type	Maximum number of supported partitions
LTO 5 (TS2250 and TS2350) and later	2 in Wrap-wise Partitioning
3592 E07 (TS 1140)	4 in Wrap-wise Partitioning 2 in Longitudinal Partitioning

The data structure that is used with this IOCTL is

```
The define for "partition_method":
#define UNKNOWN_TYPE                      0  /* vendor-specific or unknown        */
#define WRAP_WISE_PARTITION               1  /* Wrap-wise Partitioning without RABF */
#define LONGITUDINAL_PARTITION            2  /* Longitudinal Partitioning         */
#define WRAP_WISE_PARTITION_WITH_FASTSYNC 3  /* Wrap-wise Partitioning with RABF    */

The define for "type":
#define IDP_PARTITION       1      /* Initiator Defined Partition type   */
#define SDP_PARTITION       2      /* Select Data Partition type         */
```

```
#define FDP_PARTITION       3      /* Fixed Data Partition type          */

The define for "size_unit":
#define SIZE_UNIT_BYTES     0      /* Bytes                              */
#define SIZE_UNIT_KBYTES    3      /* Kilobytes                          */
#define SIZE_UNIT_MBYTES    6      /* Megabytes                          */
#define SIZE_UNIT_GBYTES    9      /* Gigabytes                          */
#define SIZE_UNIT_TBYTES   12      /* Terabytes                          */

struct tape_partition {
  uchar type;                      /* Type of tape partition to create      */
  uchar number_of_partitions;      /* Number of partitions to create        */
  uchar size_unit;                 /* IDP size unit of partition sizes below */
  ushort size[MAX_PARTITIONS];     /* Array of partition sizes in size units */
                                   /* for each partition,0 to (number - 1)   */
  uchar partition_method;          /* partitioning type                     */
  char reserved [31];
  };
```

Examples of the **CREATE_PARTITION** IOCTL

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

  struct tape_partition partition;

  /* create 2 SDP partitions on LTO-5 */
  partition.type = SDP_PARTITION;
  partition.number_of_partitions = 2;
  partition.partition_method = WRAP_WISE_PARTITION;
  ioctl(fd, CREATE_PARTITION, &partition);

   /* create 2 IDP partitions with partition 1 for 37 gigabytes and partition 0
   for the remaining capacity on LTO-5 */
   partition.type = IDP_PARTITION;
   partition.number_of_partitions = 2;
   partition.partition_method = WRAP_WISE_PARTITION;
   partition.size_unit = SIZE_UNIT_GBYTES;
   partition.size[0] = 0xFFFF;
   partition.size[1] = 37;
   ioctl(fd, CREATE_PARTITION, &partition);

   /* format the tape into 1 partition */
   partition.number_of_partitions = 1;
   ioctl(fd, CREATE_PARTITION, &partition);

   /* create 4 IDP partitions on 3592 JC volume in Wrap-wise partitioning
   with partition 0 and 2 for 94.11 gigabytes (minimum size)and partition 1 and 3
   to use the remaining capacity equally around 1.5 TB on 3592 E07 */
   partition.type = IDP_PARTITION;
   partition.number_of_partitions = 4;
   partition.partition_method = WRAP_WISE_PARTITION;
   partition.size_unit = 8;      /* 100 megabytes */
   partition.size[0] = 0x03AD;
   partition.size[1] = 0xFFFF;
   partition.size[2] = 0x03AD;
   partition.size[3] = 0x3AD2;
```

ALLOW_DATA_OVERWRITE

The **ALLOW_DATA_OVERWRITE** IOCTL is used to set the drive to allow a subsequent data write type command at the current position. Or, it allows a **CREATE_PARTITION** IOCTL when data safe (append-only) mode is enabled.

For a subsequent write type command, the **allow_format_overwrite** field must be set to 0. The **partition_number** and **logical_block_id** fields must be set to the current partition and position within the partition where the overwrite occurs.

For a subsequent **CREATE_PARTITION** IOCTL, the **allow_format_overwrite** field must be set to 1. The **partiton_number** and **logical_block_id** fields are not used. However, the tape must be at the beginning of tape (partition 0 logical block id 0) before the **CREATE_PARTITION** IOCTL is issued.

The data structure that is used with this IOCTL is

```
struct allow_data_overwrite{
  uchar partition_number;        /* Partition number 0-n to overwrite     */
  ullong logical_block_id;       /* Blockid to overwrite to within partition */
  uchar allow_format_overwrite;  /* allow format if in data safe mode       */
  char reserved[32];
  };
```

Examples of the **ALLOW_DATA_OVERWRITE** IOCTL

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

  struct read_tape_position rpos;
  struct allow_data_overwrite data_overwrite;
  struct set_active_partition partition;

  /* get current tape position for a subsequent write type command and */
  /* set the allow_data_overwrite fields with the current position for the next
write type command */
  rpos.data_format = RP_LONG_FORM;
  if (ioctl (fd, READ_TAPE_POSITION, &rpos) &lt;0)
     retun errno;

  data_overwrite.partition_number = rpos.rp_data.rp_long.active_partition;
  data_overwrite.logical_block_id = rpos.rp_data.rp_long.logical_obj_number;
  data_overwrite.allow_format_overwrite = 0;
  ioctl (fd, ALLOW_DATA_OVERWRITE, &data_overrite;);

  /* set the tape position to the beginning of tape and */
  /* prepare a format overwrite for the CREATE_PARTITION ioctl */
  partition.partition_number = 0;
  partition.logical_block_id = 0;
  if (ioctl(fd, SET_ACTIVE_PARTITION, &partition;) &10)
    return errno;

  data_overwrite.allow_format_overwrite = 1;
  ioctl (fd, ALLOW_DATA_OVERWRITE, &data_overwrite);
```

QUERY_LOGICAL_BLOCK_PROTECTION

The IOCTL queries whether the drive can support this feature, what Logical Block Protection (LBP) method is used, and where the protection information is included.

The **lbp_capable** field indicates whether the drive has logical block protection capability. The **lbp_method** field displays if LBP is enabled and what the protection method is. The LBP information length is shown in the **lbp_info_length** field. The fields of **lbp_w**, **lbp_r**, and **rbdp** present that the protection information is included in write, read, or recover buffer data.

The data structure that is used with this IOCTL is

```
struct logical_block_protection
{
   uchar lbp_capable;     /* [OUTPUT] the capability of lbp for QUERY ioctl only */
   uchar lbp_method;      /* lbp method used for QUERY [OUTPUT]and SET [INPUT] ioctls */
     #define  LBP_DISABLE           0x00
     #define  REED_SOLOMON_CRC      0x01
   uchar lbp_info_length; /* lbp info length for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_w;           /* protection info included in write data */
```

```
                           /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_r;            /* protection info included in read data */
                           /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar rbdp;             /* protection info included in recover buffer data */
                           /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar reserved[26];
};
```

Examples of the **QUERY_LOGICAL_BLOCK_PROTECTION** IOCTL

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

  struct logical_block_protection lbp_protect;

  printf("Querying Logical Block Protection....\n");

  if (ioctl(fd, QUERY_LOGICAL_BLOCK_PROTECTION, &lbp_protect) < 0)
      return errno;
  printf("  Logical Block Protection capable........ %d\n",lbp_protect.lbp_capable);
  printf("  Logical Block Protection method.......... %d\n",lbp_protect.lbp_method);
  printf("  Logical Block Protection Info Length... %d\n",lbp_protect.lbp_info_length);
  printf("  Logical Block Protection for Write........ %d\n",lbp_protect.lbp_w);
  printf("  Logical Block Protection for Read....... %d\n",lbp_protect.lbp_r);
  printf("  Logical Block Protection for RBDP...... %d\n",lbp_protect.rbdp);
```

SET_LOGICAL_BLOCK_PROTECTION

The IOCTL enables or disables Logical Block Protection, sets up what method is used, and where the protection information is included.

The **lbp_capable** field is ignored in this IOCTL by the Atape driver. If the **lbp_method** field is 0 (LBP_DISABLE), all other fields are ignored and not used. When the **lbp_method** field is set to a valid non-zero method, all other fields are used to specify the setup for LBP.

The data structure that is used with this IOCTL is

```
struct logical_block_protection
{
   uchar lbp_capable;    /* [OUTPUT] the capability of lbp for QUERY ioctl only */
   uchar lbp_method;     /* lbp method used for QUERY [OUTPUT] and SET [INPUT] ioctls */
     #define  LBP_DISABLE           0x00
     #define  REED_SOLOMON_CRC      0x01
   uchar lbp_info_length;  /* lbp info length for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_w;          /* protection info included in write data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_r;          /* protection info included in read data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar rbdp;           /* protection info included in recover buffer data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar reserved[26];
};
```

Examples of the **SET_LOGICAL_BLOCK_PROTECTION** IOCTL

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

 int rc;
  struct logical_block_protection lbp_protect;

  printf("Setting Logical Block Protection....\n\n");

  printf ("Enter Logical Block Protection method:      ");
  gets (buf);
  lbp_protect.lbp_method= atoi(buf);
  printf ("Enter Logical Block Protection Info Length: ");
```

```
gets (buf);
lbp_protect.lbp_info_length= atoi(buf);
printf ("Enter Logical Block Protection for Write:   ");
gets (buf);
lbp_protect.lbp_w= atoi(buf);
printf ("Enter Logical Block Protection for Read:    ");
gets (buf);
lbp_protect.lbp_r= atoi(buf);
printf ("Enter Logical Block Protection for RBDP:    ");
gets (buf);
lbp_protect.rbdp= atoi(buf);

rc = ioctl(fd, SET_LOGICAL_BLOCK_PROTECTION, &lbp_protect);

if (rc)
   printf ("Set Logical Block Protection Fails (rc %d)",rc);
else
   printf ("Set Logical Block Protection Succeeds");
```

Note:

1. The drive always expects a CRC attached with a data block when LBP is enabled for **lbp_r** and **lbp_w**. Without the CRC bytes attachment, the drive fails the **Read and Write** command. To prevent the CRC block transfer between the drive and application, the maximum block size limit must be determined by application. Call the **STIOCQRYP** IOCTL and get the system maximum block size limit. Call the **Read Block Limits** command to get the drive maximum block size limit. Then, use the minimum of the two limits.
2. When a unit attention with a power-on and device reset (Sense key/Asc-Ascq x6/x2900) occurs, the LBP enable bits (**lbp_w**, **lbp_r**, and **rbdp**) are reset to OFF by default. Atape tape driver returns EIO for an IOCTL call in this situation. Once the application determines it is a reset unit attention in the sense data, it responses to query LBP setup and reissues this IOCTL to set up LBP properly.
3. The LBP setting is controlled by the application and not the device driver. If an application enables LBP, it must also disable LBP when it closes the drive, as this action is not done by the device driver.

STIOC_READ_ATTRIBUTE

The IOCTL is issued to read attribute values that belongs to a specific partition from medium auxiliary memory.

The input or output data structure is

```
#define MAX_ATTR_LEN 1024
struct read_attribute
{
   uchar service_action;   /* [IN] service action   */
   uchar partition_number; /* [IN] the partition which the attributes belong to */
    ushort first_attr_id;  /* [IN] first attribute id to be returned   */
   uint attr_data_len;     /* [OUT] length of attribute data returned  */
   uchar reserved[8];
   char data[MAX_ATTR_LEN]; /* [OUT] read attributes data   */
} ;
```

An example of the **STIOC_READ_ATTRIBUTE** command is

```
#include <sys/Atape.h>
int rc,attr_len;
struct read_attribute rd_attr;

memset(&rd_attr,0,sizeof(struct read_attribute));
rd_attr.service_action=0x00;
rd_attr.partition_number=1;
```

```
rd_attr.first_attr_id=0x800;

printf("Read attribute command ....\n");
rc=ioctl(fd, STIOC_READ_ATTRIBUTE, &rd_attr);

if (rc)
  printf ("Read Attribute failed (rc %d)",rc);
else
 {
  printf ("Read Attribute Succeeds!");
  dump_bytes (rd_attr.data, min(MAX_ATTR_LEN, rd_attr.attr_data_len),
"Attribute Data");
}
```

STIOC_WRITE_ATTRIBUTE

The IOCTL sets the attributes in medium auxiliary memory at a specific partition.

Following is the structure for **STIOC_WRITE_ATTRIBURE** IOCTL

```
struct write_attribute
{
  uchar write_cache;       /* [IN] WTC - Write-through cache */
  uchar partition_number;  /* [IN] the partition which the attribute is belonged to */
  uint parm_list_len;      /* [IN] parameter list length */
  uchar reserved[10];
  char data[MAX_ATTR_LEN];   /* [IN] write attributes data */
} ;
```

An example of the **STIOC_WRITE_ATTRIBUTE** command is

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


int rc;
struct write_attribute wr_attr;

memset(&wr_attr,0,sizeof(struct write_attribute));

wr_attr.write_cache=0;
wr_attr.parm_list_len=0x11;
wr_attr.data[3]=0x0D;
wr_attr.data[4]=0x08;
wr_attr.data[6]=0x01;
wr_attr.data[8]=0x08;
wr_attr.data[9]='I';
wr_attr.data[10]='B';
wr_attr.data[11]='M';
wr_attr.data[12]=' ';
wr_attr.data[13]='T';
wr_attr.data[14]='E';
wr_attr.data[15]='S';
wr_attr.data[16]='T';

printf("Issuing a sample Write Attribute command ....\n\n");
rc=ioctl(fd, STIOC_WRITE_ATTRIBUTE, &wr_attr);

if (rc)
 printf ("Write Attribute failed (rc %d)",rc);
else
 printf ("Write Attribute Succeeds");
```

VERIFY_TAPE_DATA

The IOCTL issues a **VERIFY** command. This command causes data to be read from the tape and passed through the drive's error detection and correction hardware.

This action determines whether it can be recovered from the tape, whether the protection information is present, and validates correctly on logical block on the medium. The driver returns the IOCTL a failure or a success if the `VERIFY SCSI` command is completed in a **Good SCSI** status.

Note:

1. When an application sets the VBF method, it considers the driver's close operation in which the driver can write filemarks in its close, which the application did not explicitly request. For example, some drivers write two consecutive filemarks that mark the end of data on the tape in its close, if the last tape operation was a `WRITE` command.
2. Per the user's or application's request, Atape driver sets the block size in the field of **Block Length** in mode block descriptor for `Read and Write` commands. Then, it maintains this block size setting in a whole open. For instance, the tape driver sets a zero in the **Block Length** field for the variable block size. This act causes the missing of an overlength condition on a SILI Read. **Block Length** must be set to a non-zero value.

   Before Fixed bit is set to ON with VTE or VBF ON in `Verify` IOCTL, the application is requested to set the block size in mode block descriptor. Then, the drive uses it to verify the length of each logical block. For example, a 256 KB length is set in **Block Length** field to verify the data. The setup overrides the early setting from IBM tape driver.

   When the application completes the `Verify` IOCTL call, the original block size setting must be restored for `Read and Write` commands, the application either issues `set block size` IOCTL. Or, it closes the drive immediately and reopens the drive for the next tape operation. It is recommended to reopen the drive for the next tape operation. Otherwise, it causes next `Read and Write` command misbehavior.
3. To support DPF for `Verify` command with FIXED bit on, it is requested to issue an IBM tape driver to set **blksize** in `STIOCSETP` IOCTL. IBM tape driver sets the **block length** in mode block descriptor same as the block size and save the block size in kernel memory. The driver restores the **block length** before it tries the `Verify SCSI` command again. Otherwise, it causes the `Verify` command to fail.
4. The IOCTL can be returned longer than the timeout when DPF occurs.

The structure is defined for this IOCTL as

```
struct verify_data
{
  uint     : 2, /* reserved */
        vte: 1, /* [IN] verify to end-of-data */
      vlbpm: 1, /* [IN] verify logical block protection info */
        vbf: 1, /* [IN] verify by filemarks */
      immed: 1, /* [IN] return SCSI status immediately */
     bytcmp: 1, /* No use currently */
      fixed: 1; /* [IN] set Fixed bit to verify the length of each logical block */
uchar reseved[15];
uint verify_length; /* [IN] amount of data to be verified */
} ;
```

An example of the `VERIFY_TAPE_DATA` command is to verify all of logical block from the current position to end of data. It includes a verification that each logical block uses the logical block protection method that is specified in the **Control Data Protection** mode page, when vte is set to 1 with vlbpm on.

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

int rc;
struct verify_data vrf_data;

memset(&vrf_data,0,sizeof(struct verify_data));
vrf_data.vte=1;
vrf_data.vlbpm=1;
vrf_data.vbf=0;
vrf_data.immed=0;
vrf_data.fixed=0;
vrf_data.verify_length=0;

printf("Verify Tape Data command ....\n");
rc=ioctl(fd,VERIFY_TAPE_DATA, &vrf_data);

if (rc)
 printf ("Verify Tape Data failed (rc %d)",rc);
else printf
 ("Verify Tape Data Succeeded!");
```

QUERY_RAO_INFO

The IOCTL is used to query the maximum number and size of User Data Segments (UDS) that are supported from tape drive and driver for the wanted **uds_type**. The application calls this IOCTL before the **GENERATE_RAO** and **RECEIVE_RAO** IOCTLs are issued. The application uses the return data to limit the number of UDS requested in the **GENERATE_RAO** IOCTL.

The structure that is defined for this IOCTL as

```
struct query_rao_info {
    char   uds_type;         /* [IN]  0: UDS_WITHOUT_GEOMETRY                       */
                             /*       1: UDS_WITH_GEOMETRY                          */
    char   reserved[7];
    ushort max_uds_number;   /* [OUT] Max UDS number supported from drive           */
    ushort max_uds_size;     /* [OUT] Max single UDS size supported from drive in byte */
 ushort max_host_uds_number; /* [OUT] Max UDS number supported from driver          */
}
```

An example of the **QUERY_RAO_INFO** command is

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

  int rc;
  struct query_rao_info qrao;

 bzero(&qrao,sizeof(struct query_rao_info));

 qrao.uds_type=uds_type;

 rc=ioctl(fd,QUERY_RAO_INFO,&qrao);

 if (rc)
     printf("QUERY_RAO_INFO fails with rc %d\n",rc);
   else
     {
     max_host_uds_num=qrao.max_host_uds_number;
     max_uds_size=qrao.max_uds_size;
     }

    return rc;
```

GENERATE_RAO

The IOCTL is called to send a GRAO list to request the drive to generate a **Recommending Access Order** list.

The process method is either 1 or 2 to create a RAO list, and the type of UDS is either with or without the geometry. The **uds_number** must be not larger than `max_host_uds_number` in the **`QUERY_RAO_INFO`** IOCTL. The application allocates a memory with **grao_list_leng** (`uds_number * sizeof(struct grao_uds_desc) +8`) for the pointer of **grao_list**. 8 bytes is the size that is needed for the header portion on of the return data.

The structures for the **`GENERATE_RAO`** IOCTL are

```
struct generate_rao {
    char   process;    /* [IN] Requested process to generate RAO list        */
                       /*      0: no reorder UDS and no calculate locate time */
                       /*          (not currently supported by the drive)     */
                       /*      1: no reorder UDS but calculate locate time    */
                       /*      2: reorder UDS and calculate locate time       */
    char   uds_type;   /* [IN] 0: UDS_WITHOUT_GEOMETRY                        */
                       /*      1: UDS_WITH_GEOMETRY                           */
char    reserved1[2];
    uint    grao_list_leng; /* [IN] The data length is allocated for GRAO list. */
char   *grao_list;     /* [IN] the pointer is allocated to the size of grao_list_leng */
                       /*     (uds_number * sizeof(struct grao_uds_desc)
                       /*     +sizeof(struct grao_list_header))               */
                       /*     and contains the data of GRAO parameter list.   */
                       /*     The uds number isn't larger than max_host_uds_number */
                       /*     in QUERY_RAO ioctl.                             */
    char   reserved2[8];
}
```

The **grao** list is in the format and the parameter data can be generated by using the structures that are defined here.

```
 -- List Header
 -- UDS Segment Descriptor (first)
     ......
 -- UDS Segment Descriptor (last)

struct grao_list_header {
    uchar  reserved[4];
    uint   addl_data;              /* additional data              */
}

struct grao_uds_desc {
    ushort desc_leng;              /* descriptor length            */
    char    reserved[3];
    char    uds_name[10];          /* uds name given by application  */
    char    partition;             /* Partition number 0-n to overwrite */
    ullong beginning_loi;          /* Beginning logical object ID    */
    ullong ending_loi;             /* Ending logical object ID       */
}
```

An example of the **`GENERATE_RAO`** command is

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

 int rc;
 struct generate_rao grao;

 bzero(&grao,sizeof(struct generate_rao));

 grao.process=2;
```

```
grao.uds_type=uds_type;
grao.grao_list_leng=max_host_uds_num * sizeof(struct grao_uds_desc)
+ sizeof(struct grao_list_header);

if (!(grao.grao_list=malloc(grao.grao_list_leng)))
     {
     perror("Failure allocating memory");
     return (errno);
     }
memset(grao.grao_list, 0, grao.grao_list_leng);
grao.grao_list[3]=36;

rc=ioctl(fd,GENERATE_RAO,&grao);
if (rc)
   printf("GENERATE_RAO fails with rc %d\n",rc);
else
   printf("GENERATE_RAO succeeds\n");

free(grao.grao_list);

return rc;
```

RECEIVE_RAO

After a **GENERATE_RAO** IOCTL is completed, the application calls the **RECEIVE_RAO** IOCTL to receive a recommended access order of UDS from the drive. To avoid a system crash, it is important that the application allocates a large enough block of memory for the ***rrao_list** pointer and notifies the driver of the allocated size. It is done by indicating the size of the buffer in bytes to the *rrao_list_leng* variable as an input to the **receive_rao_list** structure.

The structure for the **RECEIVE_RAO** IOCTL is

```
struct receive_rao_list {
         uint    rrao_list_offset;  /* [IN] The offset of receive RAO list to */
                                    /*      begin returning data             */
         uint    rrao_list_leng;    /* [IN/OUT] number byte of data length    */
                                    /* [IN] The data length is allocated for RRAO */
                                    /*      list by application the length is     */
                                    /*      (max_uds_size * uds_number +          */
                                    /*      sizeof(struct rrao_list_header)       */
                                    /*      max_uds_size is reported in           */
                                    /*      sizeof(struct rrao_list_header)       */
                                    /*      uds_number is the total UDS number    */
                                    /*      requested from application in         */
                                    /*      GENERATE_RAO ioctl                    */
                                    /* [OUT] the data length is actual returned   */
                                    /*      in RRAO list from the driver          */
         char   *rrao_list;         /* [IN/OUT] the data pointer of RRAO list     */
         char   reserved[8];
};
```

The **rrao** list is in this format.

```
List Header

UDS descriptor (first)
   -- Basic UDS descriptor
   -- Additional UDS info descriptor (first)
      ......
   -- Additional UDS info descriptor (last)

......

UDS descriptor (last)
   -- Basic UDS descriptor
```

```
-- Additional UDS info descriptor (first)
   ......
-- Additional UDS info descriptor (last)
```

The sample code is

```
int rc;
  struct receive_rao_list rrao;

  bzero(&rrao,sizeof(struct receive_rao_list));

  rrao.rrao_list_offset=0;
  rrao.rrao_list_leng=max_host_uds_num * max_uds_size + 8;
  /* 8 is the header of rrao list */

  if (!(rrao.rrao_list=malloc(rrao.rrao_list_leng)))
     {
     perror("Failure allocating memory");
     return (errno);
     }
  memset(rrao.rrao_list, 0, rrao.rrao_list_leng);

  rc=ioctl(fd,RECEIVE_RAO,&rrao);
  if (rc)
    pintf("RECEIVE_RAO fails with rc %d\n",rc);
  else
    printf("rrao_list_leng %d\n",rrao.rrao_list_leng);

  free(rrao.rrao_list);

 return rc;
```

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, these IOCTL operations 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 is restricted to **/dev/rmt*.smc** and any attempt to access them through **/dev/rmt*** fails.

Overview

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.

SMCIOC_READ_CARTIDGE_LOCATION
: Returns the cartridge location information for storage elements in the library.

These IOCTL commands and their associated structures are defined by including the **/usr/include/sys/Atape.h** header file in the C program by using the functions.

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. 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 Range** command. It 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

```
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 information about the four element types. The software application processes the input data (the number of elements about which it requires information). Then, it 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 */
           impexp:1,          /* import/export placed by operator or robot */
           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 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;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, it loads 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, it moves a tape to the first empty slot in the magazine if the slot address is specified as zero. If the IOCTL is issued to the **/dev/rmt** special file, the tape is automatically rewound and unloaded 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 used normally 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 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

```
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;
 }
```

SMCIOC_READ_CARTIDGE_LOCATION

The **SMCIOC_READ_CARTIDGE_LOCATION** IOCTL is used to return the cartridge location information for storage elements in the library. The **element_address** field specifies the starting element address to return. The **number_elements** field specifies how many storage elements are returned. The **data** field is a pointer to the buffer for return data. The buffer must be large enough for the number of elements that are returned. If the storage element contains a cartridge, then the **ASCII identifier** field in return data specifies the location of the cartridge.

Note: This IOCTL is supported only on the TS3500 (3584) library.

The data structures that are used with this IOCTL are

```
struct cartridge_location_data
{
    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 */
    uchar  resvd2[3];          /* 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 */
    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[24];     /* slot identification */
};

struct read_cartridge_location
{
    ushort element_address;                  /* starting element address */
    ushort number_elements;                  /* number of elements       */
    struct cartridge_location_data *data;    /* storage element pages    */
    char reserved[8];                        /* reserved                 */
};
```

Example of the **SMCIOC_READ_CARTRIDGE_LOCATION** IOCTL

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

  int i;
  struct cartridge_location_data *data, *elemp;
  struct read_cartridge_location cart_location;
  struct element_info element_info;

  /* get the number of slots and starting element address */
  if (ioctl(fd, SMCIOC_ELEMENT_INFO, &element_info) < 0)
     return errno;

  if (element_info.slots == 0)
     return 0;

  data = malloc(element_info.slots * sizeof(struct cartridge_location_data));
  if (data == NULL)
    return ENOMEM;

  /* Read cartridge location for all slots */
  bzero(data,element_info.slots * sizeof(struct cartridge_location_data));
  cart_location.data = data;
  cart_location.element_address = element_info.slot_addr;
  cart_location.number_elements = element_info.slots;
```

```
  if (ioctl (fd, SMCIOC_READ_CARTRIDGE_LOCATION, &cart_location) < 0)
     {
     free(data);
     return errno;
     }

    elemp = data;
   for (i = 0; i < element_info.slots; i++, elemp++)
     {
     if (elemp->address == 0
)       continue;

      printf("Slot Address %d\n",elemp->address);
     if (elemp->except)
       printf("  Slot State ..................... Abnormal\n");
     else
       printf("  Slot State ..................... Normal\n");
     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");
     printf("  Volume Tag ..................... %0.36s\n", elemp->volume);
     printf("  Cartridge Location ............. %0.24s\n", elemp->identifier);


}    free(data);
     return 0;
```

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 input/output error (EIO), the application can issue the **STIOCQRYSENSE** IOCTL command with the LASTERROR option. Or, it can issue the **SIOC_REQSENSE** IOCTL command to analyze the sense data and determine why the error occurred.

Codes for all operations

The following codes and their descriptions apply to all operations.

[EACCES]
: Data encryption access denied.

[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 complete the wanted function.

[ETIMEDOUT]
A command that is timed out in the device.

[ENOCONNECT]
The device did not respond to selection.

[ECONNREFUSED]
The device driver detected that the device vital product data (VPD) 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 that is 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.

[EINPROGRESS]
This *errno* is returned when the extended open flag **SC_KILL_OPEN** is used to kill all processes that currently have the device opened.

Write error codes

The following codes and their descriptions apply to **write** operations.

[EINVAL]
The operation that is 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 or the programmable early warning zone (PEWZ) while it was in label-processing mode. This return code is returned only once when the early warning or the programmable early warning zone (PEWZ) 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 a general error occurred 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 opened with the **O_WRONLY** flag.

[EINVAL]
The operation that is 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 codes

The following codes and their descriptions apply to **close** operations.

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

[ENOTREADY]
A command that is 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 that is requested has invalid parameters or an invalid combination of parameters.

This error code also results if the IOCTL is not supported for the device.

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

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

[ECANCELLED]
The **STIOCTOP** IOCTL with the **st_op** field that specifies **STERASE_IMM** was canceled by another process that issued the **STIOC_CANCEL_ERASE** IOCTL.

Chapter 3. HP-UX tape and medium changer device driver

HP-UX programming interface

The HP-UX programming interface to the Advanced Tape Device Driver (ATDD) software conforms to the standard HP-UX tape device driver interface. The following user callable entry points are supported.

- "open"
- "close" on page 98
- "read" on page 99
- "write" on page 99
- "ioctl" on page 100

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. More 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 Tape 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 allows only 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 that is 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 that is mounted fails.

O_NDELAY
This option indicates to the driver not to wait until the tape drive is ready before device is opened and commands are sent. If the flag is not set, an open call requires a physical tape to be loaded and ready. The open without the flag fails and an EIO is returned if the tape drive is not ready.

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 that is 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 is flushed out to tape before completion of the close. If any error occurs in flushing the data, an error code is returned by the close routine.

An application must explicitly issue the close() call when the I/O resource is no longer necessary or in preparation for termination. The operating system implicitly issues the close() call for an application that terminates without closing the resource itself. If an application terminates unexpectedly but leaves behind child processes that inherited the file descriptor for the open resource, the operating system does not implicitly close the file descriptor because it believes that 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 run while it was opened. The commands are issued to the tape drive during the close operation according to the following logic and rules.

```
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 SPACE1 FILEMARK

if special file is REWIND ON CLOSE
   REWIND
```

Rules:

1. Return EIO and release the drive when a unit attention happens before the close().
2. Fail the command, return EIO, and release the drive if a unit attention occurs during the close().
3. If a SCSI command fails during close processing, only the **SCSI RELEASE** is attempted thereafter.
4. The return code from the **SCSI RELEASE** command is ignored.

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 that is 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 (and zero indicates a file mark).

variable block size

When in variable block size mode, the **bufsize** parameter can be any value valid to the drive. The amount of data that is returned equals 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 is lost because the next read starts from the start 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) is returned. If the **bufsize** parameter is valid, the **read** command always returns the amount of data that is requested unless a file mark is encountered. In that case, it returns all data that occurred before the filemark and `actual` equals 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. Physical limitations of the drive can cause **write** to fail (for example, attempting to write past the physical end of tape).

The following code fragment shows 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 is returned (EINVAL). If the write size exceeds the device maximum block size or the configured buffer size of the tape drive, an error is returned (EINVAL).

ioctl

The ATDD software supports all input/output control (IOCTL) commands that are supported by the HP-UX native drivers, **tape2**, and **stape**. See the following HP-UX **man** pages for information.

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

IOCTL operations

The following sections describe IOCTL operations that are supported by the ATDD. Usage, syntax, and examples are given.

The IOCTL operations that are supported by the driver are described in

- "General SCSI IOCTL operations"
- "SCSI medium changer IOCTL operations" on page 108
- "SCSI tape drive IOCTL operations" on page 118
- "Base operating system tape drive IOCTL operations" on page 149
- "Service aid IOCTL operations" on page 151

The following files must be included by user programs that issue the IOCTL commands described in this section to access the tape device 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 whether 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_REQUEST_SENSE
Return the device sense data.

IOC_LOG_SENSE_PAGE
Return a log sense page from the device.

IOC_LOG_SENSE10_PAGE
Return the log sense data by using a 10-byte CDB with optional subpage.

IOC_MODE_SENSE
Return the mode sense data from the device.

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. It is installed with the HP-UX Advanced Tape Device Driver (ATDD) package. Any application program that issues these commands must include one or both header files.

IOC_TEST_UNIT_READY

This command determines whether 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               : 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 when a nonzero page code is requested. For inquiry pages 0x80, data that is mapped by structures **inq_pg_80_t** is returned in the data array. Otherwise, an array of data is returned in the data array.

The following data structure 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, 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 wanted 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 parameters.

- Log Page
  - Log Page Header
    - Page Code
    - Page Length
  - Log Parameter(s) (One or more might 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 rc;
  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);
     printf ("\n");
     scsi_request_sense ();
   }
   return (rc);
   }
```

IOC_LOG_SENSE10_PAGE

This IOCTL command is enhanced to add a *Subpage* variable from **IOC_LOG_SENSE_PAGE**. It returns a log sense page and Subpage from the device. The wanted page is selected by specifying the **page_code** and **subpage_code** in the **log_sense10_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 must be set to zero. To obtain the entire log page that starts at a specific parameter code, set the **parm_pointer** field to the wanted code and the **len** field to zero. To obtain a specific number of parameter bytes, set the **parm_pointer** field to the wanted code. Then, 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 are always the log page header. See the appropriate device manual to determine the supported log pages and content. The data structure is

```
/* log sense page and subpage structure */

typedef struct {
   uchar page_code;            /* [IN] Log sense page */
   uchar subpage_code;         /* [IN] Log sense subpage */
   uchar reserved[2];           /* unused */
   unsigned short len;         /* [OUT] number of valid bytes in data
                                   (log_page_header_size + page_length) */
   unsigned short parm_pointer;  /* [IN] specific parameter number at which
                                   the data begins */
   char data[LOGSENSEPAGE];     /* [OUT] log data */
} log_sense10_page_t;
```

IOC_MODE_SENSE

This command returns a mode sense page from the device. The wanted 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 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 and the 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 that is 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
Complete 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.

SMCIOC_READ_CARTRIDGE_LOCATION
Returns the cartridge location information for all storage elements in the library.

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

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 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 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 run 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

```
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 **SMCIOC_READ_ELEMENT_DEVIDS** command is

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

static int smc_read_element_devids ( )
{
  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. 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);
```

SMCIOC_READ_CARTRIDGE_LOCATION

The **SMCIOC_READ_CARTIDGE_LOCATION** IOCTL is used to return the cartridge location information for storage elements in the library. The **element_address** field specifies the starting element address to return and the **number_elements** field specifies how many storage elements are returned. The data field is a pointer to the buffer for return data. The buffer must be large enough for the number of elements that are returned. If the storage element contains a cartridge, then the **ASCII identifier** field in return data specifies the location of the cartridge.

Note: This IOCTL is supported only on the TS3500 (3584) library.

The 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  resvd2[3];                /* reserved                */
    uchar  svalid:1,                 /* element address valid   */
           invert:1,                 /* medium inverted         */
           :6;                       /* reserved                */
    ushort source;                   /* source storage elem addr */
    uchar  volume[36];               /* primary volume tag      */
    uchar  :4,                       /* reserved                */
           code_set:4;               /* code set                */
    uchar  :4,                       /* reserved                */
```

```
        ident_type:4;              /* identifier type         */
    uchar  resvd3;                 /* reserved                */
    uchar  ident_len;              /* identifier length       */
    uchar  identifier[24];         /* slot identification     */
} cartridge_location_data_t;

typedef struct
{
    ushort element_address;           /* starting element address */
    ushort number_elements;           /* number of elements       */
    cartridge_location_data_t *data;  /* storage element pages    */
    char reserved[8];                 /* reserved                 */
} read_cartridge_location_t;
```

An example of the **SMCIOC_READ_CARTRIDGE_LOCATION** command is

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

   int rc;
   int available_slots=0;
   cartridge_location_data_t *slot_devid;
   read_cartridge_location_t slot_devids;

   slot_devids.element_address = (ushort)element_address;
   slot_devids.number_elements = (ushort)number_elements;

   if (rc = ioctl(dev_fd,SMCIOC_ELEMENT_INFO,&element_info))
       {
       PERROR("SMCIOC_ELEMENT_INFO failed");
       PRINTF("\n");
       scsi_request_sense();
       return (rc);
       }

     if (element_info.cell_count == 0)
        {
        printf("No slots found in element information...\n");
        errno = EIO;
        return errno;
        }

     if ((slot_devids.element_address==0) && (slot_devids.number_elements==0))
        {
        slot_devids.element_address=element_info.cell_address;
        slot_devids.number_elements=element_info.cell_count;
        printf("Reading all locations...\n");
        }
     if ((element_info.cell_address > slot_devids.element_address)
     (slot_devids.element_address >
     (element_info.cell_address+element_info.cell_count-1)))
        {
        printf("Invalid slot address %d\n",element_address);
        errno = EINVAL;
        return errno;
        }

      available_slots = (element_info.cell_address+element_info.cell_count)
-slot_devids.element_address;
     if (available_slots>slot_devids.number_elements)
      available_slots=slot_devids.number_elements;
      slot_devid = malloc(element_info.cell_count *
      sizeof(cartridge_location_data_t));
     if (slot_devid == NULL)
       {
       errno = ENOMEM;
       return errno;
       }
```

```
bzero((caddr_t)slot_devid,element_info.cell_count * sizeof(cartridge_location_data_t));
  slot_devids.data = slot_devid;

  rc = ioctl (dev_fd, SMCIOC_READ_CARTRIDGE_LOCATION, &slot_devids);

  free(slot_devid);
  return rc;
```

SCSI tape drive IOCTL operations

A set of enhanced IOCTL commands gives applications access to extra features of IBM tape drives.

The following commands are supported.

STIOC_TAPE_OP
: Runs standard tape drive operations.

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.

STIOC_REPORT_DENSITY_SUPPORT
: Return supported densities from the tape device.

STIOC_GET_DENSITY
: Query the current write density format settings on the tape drive. The current density code from the drive **Mode Sense** header, the **Read/Write Control Mode** page default density, and pending density are returned.

STIOC_SET_DENSITY
: Set a new write density format on the tape drive by using the default and pending density fields. The application can specify a new write density for the current loaded tape only or as a default for all tapes.

GET_ENCRYPTION_STATE
: This IOCTL can be used for application, system, and library-managed encryption. It allows a query only of the encryption status.

SET_ENCRYPTION_STATE
: This IOCTL can be used only for application-managed encryption. It sets the encryption state for application-managed encryption.

SET_DATA_KEY
: This IOCTL can be used only for application-managed encryption. It sets the data key for application-managed encryption.

CREATE_PARTITION
: Create one or more tape partitions and format the media.

QUERY_PARTITION
: Query tape partitioning information and current active partition.

SET_ACTIVE_PARTITION
: Set the current active tape partition.

ALLOW_DATA_OVERWRITE
: Set the drive to allow a subsequent data overwrite type command at the current position or allow a `CREATE_PARTITION` IOCTL when data safe (append-only) mode is enabled.

READ_TAPE_POSITION
: Read current tape position in either short, long, or extended form.

SET_TAPE_POSITION
: Set the current tape position to either a logical object or logical file position.

QUERY_LOGICAL_BLOCK_PROTECTION
: Query **Logical Block Protection** (LBP) support and its setup.

SET_LOGICAL_BLOCK_PROTECTION
: Enable/disable **Logical Block Protection** (LBP), set the protection method, and how the protection information is transferred.

VERIFY_TAPE_DATA
: Allows the drive to verify data from the tape to determine whether it can be recovered. Or, whether the protection information is present and validates correctly on logical block on the medium.

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

STIOC_TAPE_OP

This command runs standard tape drive operations. It is similar to the `MTIOCTOP` IOCTL command defined in the**/usr/include/sys/mtio.h** system header file. However, the `STIOC_TAPE_OP` command uses the **ST_OP** opcodes and the data structure that is 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 149.

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 run. 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 specific **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 <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 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 data structure **mtget** 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 150.

The following data structure is returned by the driver.

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

The **mt_flags** field, which returns the type of automatic cartridge stacker or loader that are 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 configuration information about the tape drive. The **STIOC_GET_DEVICE_INFO** command uses the following data structure that is defined in the **/usr/include/sys/st.h** 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 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, which returns the current type of media, 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 information about the tape position.

The tape position is defined as where the next read or write operation occurs. The **STIOC_GET_POSITION** and **STIOC_SET_POSITION** commands can be used independently or with one another.

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             */
  uchar partition;               /* partition number                */
  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. They are used for high-speed **locate** operations that are implemented by the tape drive. Only the IBM 3490E Magnetic Tape Subsystem and the IBM TotalStorage Enterprise Virtual Tape Servers (VTS) support the **PHYSICAL_BLK** type. All devices support the **LOGICAL_BLK** type.

The **block_type** is the only field that must be filled out by the caller. 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 quickly return 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 that remains in the buffer.

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

The **bot** and **eot** fields indicate whether 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 occurs. The **STIOC_GET_POSITION** and **STIOC_SET_POSITION** commands can be used independently or with one another.

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              */
  uchar partition;               /* partition number                */
  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, it is set to **PHYSICAL_BLK** for composite tape positions that are used for high-speed **locate** operations that are implemented by the tape drive. Only the IBM 3490E Magnetic Tape Subsystem **or a virtual drive in a VTS** support the **PHYSICAL_BLK** type. All 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 that is 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. The IBM 3490E Magnetic Tape Subsystem drives in VTSs do not support **set_position** to **eot**.

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 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 that are indicated for the specific **type**.

The **type** field, which is filled out by the caller, must be set to one of the following values.

BLOCKSIZE
: Block Size (0–2097152 [2 MB]).

  A value of zero indicates variable block size. Only the IBM 3590 Tape System supports 2 MB maximum block size. All other devices support 256 KB maximum block size.

COMPRESSION
: Compression Mode (0 or 1).

  If this mode is enabled, data is compressed by the tape device before it is stored 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 does 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 that uses the trailer label processing options must stop normal data writing when LEOM (Logic End of Medium) is reached. Then, it runs end of volume processing. Such processing typically consists of writing a final data record, a filemark, and a "trailing" type label. Finally, two more filemarks indicate the end of data (EOD).

WRITEPROTECT
Write Protect Mode.

This configuration parameter returns the current write protection status of the mounted cartridge. The **writeprotect** is not applied to the VTS with logical volumes only. 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. Setting the tape to this value resets the PERSISTENT and ASSOCIATED logical write protection modes. It does not reset the WORM logical or the PHYSICAL write protection modes.
- PHYS_PROTECT

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

  Note: Only IBM 3590 and MP 3570 Tape Subsystems recognize the following values:
- WORM_PROTECT

  The tape is logically write protected in WORM mode. When the tape is protected in this mode, it is permanently write protected. The only method of returning the tape to a writable state is to format the cartridge, erasing all data.
- PERS_PROTECT

  The tape is logically write protected in PERSISTENT mode. A tape that is protected in this mode is write protected for all uses (across mounts). This logical write protection mode can be reset by using the **NO_PROTECT** value.
- ASSC_PROTECT

  The tape is logically write protected in ASSOCIATED mode. A tape that is protected in this mode is write protected only while it is associated with a tape drive (mounted). When the tape is unloaded from the drive, the associated write protection is reset. This logical write protection mode can also be reset by using the **NO_PROTECT** value.

ACFMODE
Automatic Cartridge Facility Mode.

Note: NOTE: This mode is not supported for Ultrium devices.

This configuration parameter is read only. ACF modes can be established only through the tape drive operator panel. The device driver can query only the ACF mode; it cannot change it. The ACFMODE parameter applies only to the IBM 3590 Tape System and the IBM Magstar MP Tape Subsystem. The following values are recognized.

- NO_ACF

  There is no ACF attached to the tape drive.
- SYSTEM_MODE

  The ACF is in the **System** mode. This mode allows explicit load and unloads to be issued through the device driver. An **`unload`** or **`offline`** command causes the tape drive to unload the cartridge and the ACF to replace the cartridge in its original magazine slot. A subsequent load command causes the ACF to load the cartridge from the next sequential magazine slot into the drive.

- RANDOM_MODE

  The ACF is in the **Random** mode. This mode provides random access to all of the cartridges in the magazine. The ACF operates as a standard SCSI medium changer device.
- MANUAL_MODE

  The ACF is in the **Manual** mode. This mode does not allow ACF control through the device driver. Cartridge load and unload operations can be run only through the tape drive operator panel. Cartridges are imported and exported through the priority slot.
- ACCUM_MODE

  The ACF is in the **Accumulate** mode. This mode is similar to **Manual** mode. However, rather than cartridges that are exported through the priority slot, they are put away in the next available magazine slot.
- AUTO_MODE

  The ACF is in the **Automatic** mode. This mode causes cartridges to be accessed sequentially under ACF control. When a tape finished processing, it is put back in its magazine slot. Then, the next tape is loaded without an explicit `unload` and `load` command from the host.
- LIB_MODE

  The ACF is in the **Library** mode. This mode is available only if the tape drive is installed in an automated tape library that supports the ACF (3495).

SCALING
Capacity Scaling.

Note: This configuration is not supported for Ultrium devices.

This configuration parameter sets the capacity or logical length of the currently mounted tape. The `SCALING` parameter is not supported on the IBM 3490E Magnetic Tape Subsystem nor in VTS drives. The following values are recognized.

- SCALE_100

  The current tape capacity is 100%.
- SCALE_75

  The current tape capacity is 75%.
- SCALE_50

  The current tape capacity is 50%.
- SCALE_25

  The current tape capacity is 25%.
- Other values (0x00 - 0xFF)

  For 3592 tape drive only.

SILI Suppress Illegal Length Indication.

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

DATASAFE
Data safe mode.

This parameter queries the current drive setting for data safe (append-only) mode. On a set operation, it changes the current data safe mode setting on the drive. On a set operation a parameter value of zero sets the drive to normal (non-data safe) mode. A value of 1 sets the drive to data safe mode.

PEWSIZE
Programmable early Warning.

The PEW is a setting of the drive and not a specific tape. Therefore, it is the same on each partition if partitions exist. Once this setting is made in the drive, it remains on until the application sets the PEW size to zero. At which point, it does not have a PEW zone until it is again set up by the application. The size of the PEW is set in the **parm_data_t** structure with the "**value**" parameter. The parameter establishes the programmable early warning zone size. The value specifies how many MBs before the standard end-of-medium early warning zone to place the programmable early warning indicator. The user application is warned that the tape is running out of space when the tape head reaches the PEW location. ENOSPC is returned on the first write operation to detect PEW.

Supported on 11iv3, however 11iv2 allows for auto blocking that can return inaccurate results.

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 with the **STIOC_GET_PARM** command.

The ATDD 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 revert to the defaults when the tape drive is closed and reopened.

To change the default configuration settings, see the *IBM TotalStorage and System Storage Tape 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 that are indicated for the specific **type**.

The **type** field, which is filled out by the caller, can be set to one of the following values.

BLOCKSIZE
Block Size (0–2097152 [2 MB]).

A value of zero indicates variable block size. Only the IBM 3590 Tape System supports 2 MB maximum block size. All other devices support 256 KB maximum block size.

COMPRESSION
Compression Mode (0 or 1).

If this mode is enabled, data is compressed by the tape device before it is stored 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 does 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 establishes the current write protection status of the mounted cartridge. The IBM Virtual Tape Server does not support the **write_protect** mode to a logical cartridge. The parameter applies only to the IBM 3590 and MP 3570 Tape Subsystems. 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. Setting the tape to this value resets the PERSISTENT and ASSOCIATED logical write protection modes. It does not reset the WORM logical or the PHYSICAL write protection modes.
- WORM_PROTECT

The tape is logically write-protected in WORM mode. When the tape is protected in this mode, it is permanently write protected. The only method of returning the tape to a writable state is to format the cartridge, erasing all data.

- PERS_PROTECT

  The tape is logically write protected in PERSISTENT mode. A tape that is protected in this mode is write protected for all uses (across mounts). This logical write protection mode can be reset by using the **NO_PROTECT** value.

- ASSC_PROTECT

  The tape is logically write protected in ASSOCIATED mode. A tape that is protected in this mode is write protected only while it is associated with a tape drive (mounted). When the tape is unloaded from the drive, the associated write protection is reset. This logical write protection mode can also be reset by using the **NO_PROTECT** value.

- PHYS_PROTECT

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

ACFMODE
Automatic Cartridge Facility Mode.

Note: This mode is not supported for Ultrium devices.

This configuration parameter is read only. ACF modes can be established only through the tape drive operator panel. This type value is not supported by the `STIOC_SET_PARM` IOCTL.

SCALING
Capacity Scaling.

Note: This configuration is not supported for Ultrium devices.

This configuration parameter sets the capacity or logical length of the currently mounted tape. The tape must be at BOT to change this value. Changing the scaling value destroys all existing data on the tape. The SCALING parameter is not supported on the IBM 3490E Magnetic Tape Subsystem or VTS drives. The following values are recognized.

- SCALE_100

  Sets the tape capacity to 100%.

- SCALE_75

  Sets the tape capacity to 75%.

- SCALE_50

  Sets the tape capacity to 50%.

- SCALE_25

  Sets the tape capacity to 25%.

- Other values (0x00 - 0xFF)

  For 3592 tape drive only.

SILI Suppress Illegal Length Indication.

If this mode is enabled and a larger block of data is requested than is read from the tape block, the tape device suppresses raising a check condition.

This action eliminates error processing that is normally run by the device driver and results in improved read performance for some situations.

DATASAFE
Data safe mode.

This parameter queries the current drive setting for data safe (append-only) mode or on a set operation changes the current data safe mode setting on the drive. On a set operation, a parameter value of zero sets the drive to normal (non-data safe) mode and a value of *1* sets the drive to data safe mode.

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.

Note: This command is not supported for Ultrium devices.

The message that is sent with this call does not always remain on the display. It depends on the current drive activity.

Note: All messages must be padded to MSGLEN bytes (8). Otherwise, garbage characters (meaningless data) are displayed in the message.

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

```
typedef struct {
  uchar function;                    /* message function code */
  char msg_0[MSGLEN];                /* message 0 */
  char msg_1[MSGLEN];                /* message 1 */
} msg_data_t;
```

The **function** field, which is filled out by the caller, is set by combining (by using logical OR), a **Message Type** flag and a **Message Control** flag.

Message Type Flags:

GENSTATUS (General Status Message)
Message 0, Message 1, or both are displayed according to the **Message Control** flag until the drive next initiates tape motion or the message is updated with a new message.

DMNTVERIFY (Demount/Verify Message)
Message 0, Message 1, or both are displayed according to the **Message**

Control flag until the current volume is unloaded. If the volume is unloaded, the message display is not changed and the command runs no operation.

MNTIMMED (Mount with Immediate Action Indicator)
Message 0, Message 1, or both are displayed according to the **Message Control** flag until the volume is loaded. An attention indicator is activated. If the volume is loaded, the message display is not changed and the command runs no operation.

DMNTIMMED (Demount/Mount with Immediate Action Indicator)
When the **Message Control** flag is set to a value of *ALTERNATE*, Message 0 and Message 1 are displayed alternately until the currently mounted volume, if any, is unloaded. When the **Message Control** flag is set to any other value, Message 0 is displayed until the currently mounted volume, if any, is unloaded. Message 1 is displayed from the time the volume is unloaded (or immediately, if the volume is already unloaded) until another volume is loaded. An attention indicator is activated.

Message Control Flag:

DISPMSG0
Display message 0.

DISPMSG1
Display message 1.

FLASHMSG0
Flash message 0.

FLASHMSG1
Flash message 1.

ALTERNATE
Alternate flashing message 0 and message 1.

An example of the **STIOC_DISPLAY_MSG** command is

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

msg_data_t msg_data;
msg_data.function = GENSTATUS | ALTERNATE;
memcpy (msg_data.msg_0, "Hello   ", 8);
memcpy (msg_data.msg_1, "World!!!", 8);

if (!(ioctl (dev_fd, STIOC_DISPLAY_MSG, &msg_data))) {
  printf ("The STIOC_DISPLAY_MSG ioctl succeeded.\n");
}

else {
  perror ("The STIOC_DISPLAY_MSG 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 ();
}
```

STIOC_ REPORT_ DENSITY_ SUPPORT

This 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 that are used with this IOCTL are

```
typedef struct density_report
{
    uchar  primary_density_code;      /* primary density code */
    uchar  secondary_density_code;    /* secondary densuty 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 report_density_support
{
    uchar  media;              /* report all or current media as defined above */
    uchar  number_reports;     /* number of density reports returned in array  */
    struct density_report reports[MAX_DENSITY_REPORTS];
} rpt_dens_sup_t;
```

Examples of the **STIOC_REPORT_DENSITY_SUPPORT** command are

```
static int st_report_density_support ()
{
  int rc;
  int i;
  rpt_dens_sup_t density;

  int bits_per_mm = 0;
  int media_width = 0;
  int tracks = 0;
  int capacity = 0;

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

  density.media = ALL_MEDIA_DENSITY;
  density.number_reports = 0;

  if (!(rc = ioctl (dev_fd, STIOC_REPORT_DENSITY_SUPPORT, &density))) {
     PRINTF ("STIOC_REPORT_DENSITY_SUPPORT succeeded.\n");
```

```
    printf("Total number of densities reported: %d\n",density.number_reports);
}
else {
        PERROR ("STIOC_REPORT_DENSITY_SUPPORT failed");
        PRINTF ("\n");
        scsi_request_sense ();
 }

 for (i = 0; i < density.number_reports; i++)
 {
        bits_per_mm = (int)density.reports[i].bits_per_mm[0] << 16;
        bits_per_mm |= (int)density.reports[i].bits_per_mm[1] << 8;
        bits_per_mm |= (int)density.reports[i].bits_per_mm[2];

        media_width |= density.reports[i].media_width[0] << 8;
        media_width |= density.reports[i].media_width[1];

        tracks |= density.reports[i].tracks[0] << 8;
        tracks |= density.reports[i].tracks[1];

        capacity = density.reports[i].capacity[0] << 24;
        capacity |= density.reports[i].capacity[1] << 16;
        capacity |= density.reports[i].capacity[2] << 8;
        capacity |= density.reports[i].capacity[3];

        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",bits_per_mm);
        printf("  Media Width.................. %d\n",media_width);
        printf("  Tracks....................... %d\n",tracks);
        printf("  Capacity (megabytes)......... %d\n",capacity);


        if (interactive) {
               printf ("\nHit <enter> to continue...");
               getchar ();
        }

} /* end for all media density*/

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

```
density.media = CURRENT_MEDIA_DENSITY;
density.number_reports = 0;

if (!(rc = ioctl (dev_fd, STIOC_REPORT_DENSITY_SUPPORT, &density))) {
  printf ("STIOC_REPORT_DENSITY_SUPPORT succeeded.\n");
  printf("Total number of densities reported: %d\n",density.number_reports);
}
else {
        perror ("STIOC_REPORT_DENSITY_SUPPORT failed");
        printf ("\n");
        scsi_request_sense ();
}

for (i = 0; i < density.number_reports; i++)
{

        bits_per_mm = density.reports[i].bits_per_mm[0] << 16;
        bits_per_mm |= density.reports[i].bits_per_mm[1] << 8;
        bits_per_mm |= density.reports[i].bits_per_mm[2];

        media_width |= density.reports[i].media_width[0] << 8;
        media_width |= density.reports[i].media_width[1];

        tracks |= density.reports[i].tracks[0] << 8;
        tracks |= density.reports[i].tracks[1];

        capacity = density.reports[i].capacity[0] << 24;
        capacity |= density.reports[i].capacity[1] << 16;
        capacity |= density.reports[i].capacity[2] << 8;
        capacity |= density.reports[i].capacity[3];

        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",bits_per_mm);
        printf("  Media Width.................. %d\n",media_width);
        printf("  Tracks....................... %d\n",tracks);
        printf("  Capacity (megabytes)......... %d\n",capacity);

        if (interactive) {
                printf ("\nHit <enter> to continue...");
                getchar ();
              }
```

```
  }

  return (rc);
}
```

STIOC_GET_DENSITY and STIOC_SET_DENSITY

The **STIOC_GET_DENSITY** IOCTL is used to query the current write density format settings on the tape drive. The current density code from the drive **Mode Sense** header, the **Read/Write Control Mode** page default density, and pending density are returned.

The **STIOC_SET_DENSITY** IOCTL is used to set a new write density format on the tape drive by using the default and pending density fields. The density code field is not used and ignored on this IOCTL. The application can specify a new write density for the current loaded tape only or as a default for all tapes. Refer to the examples.

The application must get the current density settings first before the current settings are modified. If the application specifies a new density for the current loaded tape only, then the application must issue another set density IOCTL. This action happens after the current tape is unloaded and the next tape is loaded to either the default maximum density or a new density to ensure the tape drive uses the correct density. If the application specifies a new default density for all tapes, the setting remains in effect until changed by another set density IOCTL or the tape drive is closed by the application.

Following is the structure for the **STIOC_GET_DENSITY** and **STIOC_SET_DENSITY** IOCTLs.

```
struct density_data_t
{
    char  density_code;         /* mode sense header density code     */
    char  default_density;      /* default write density              */
    char  pending_density;      /* pending write density              */
    char  reserved[9];
};
```

Note:

1. These IOCTLs are supported only on tape drives that can write multiple density formats. Refer to the Hardware Reference for the specific tape drive to determine whether multiple write densities are supported. If the tape drive does not support the IOCTLs, *errno* EINVAL is returned.
2. The device driver always sets the default maximum write density for the tape drive on every open system call. Any previous **STIOC_SET_DENSITY** IOCTL values from the last open are not used.
3. If the tape drive detects an invalid density code or cannot run the operation on the **STIOC_SET_DENSITY** IOCTL, the *errno* is returned and the current drive density settings before the IOCTL is restored.
4. The **struct density_data_t** defined in the header file is used for both IOCTLs. The **density_code** field is not used and ignored on the **STIOC_SET_DENSITY** IOCTL.

Examples

```
struct density_data_t data;

/* open the tape drive              */
```

```
/* get current density settings */
rc = ioctl(fd, STIOC_GET_DENSITY, &data);

/* set 3592 J1A density format for current loaded tape only */
data.default_density = 0x7F;
data.pending_density = 0x51;
rc = ioctl(fd, STIOC_SET_DENSITY, &data);

/* unload tape      */
/* load next tape */
/* set 3592 E05 density format for current loaded tape only */
data.default_density = 0x7F;
data.pending_density = 0x52;
rc = ioctl(fd, STIOC_SET_DENSITY, &data);

/* unload tape      */
/* load next tape */
/* set default maximum density for current loaded tape */
data.default_density = 0;
data.pending_density = 0;
rc = ioctl(fd, STIOC_SET_DENSITY, &data);

/* close the tape drive               */
/* open the tape drive               */
/* set 3592 J1A density format for current loaded and all subsequent tapes*/
data.default_density = 0x51;
data.pending_density = 0x51;
rc = ioctl(fd, STIOC_SET_DENSITY, &data);
```

GET_ENCRYPTION_STATE

This IOCTL command queries the drive's encryption method and state.

The data structure that is used for this IOCTL is as follows on all of the supported operating systems.

```
typedef struct encryption_status  {
    uchar encryption_capable; /* Set this field as a boolean based on the
                                 capability of the drive */
                              /* encryption_method used for GET ioctl only  */
    uchar encryption_method;  /* Set this field to one of the defines below */
        #define METHOD_NONE        0  /* Only used in GET_ENCRYPTION_STATE  */
       #define METHOD_LIBRARY      1  /* Only used in GET_ENCRYPTION_STATE  */
       #define METHOD_SYSTEM       2  /* Only used in GET_ENCRYPTION_STATE  */
       #define METHOD_APPLICATION  3  /* Only used in GET_ENCRYPTION_STATE  */
       #define METHOD_CUSTOM       4  /* Only used in GET_ENCRYPTION_STATE  */
       #define METHOD_UNKNOWN      5  /* Only used in GET_ENCRYPTION_STATE  */
    uchar encryption_state;   /*  Set this field to one of the defines below */
       #define STATE_OFF           0  /* Used in GET/SET_ENCRYPTION_STATE   */
       #define STATE_ON            1  /* Used in GET/SET_ENCRYPTION_STATE   */
       #define STATE_NA            2  /* Used in GET_ENCRYPTION_STATE       */
    uchar reserved[13];
} encryption_status_t;
```

An example of the **GET_ENCRYPTION_STATE** command is

```
int qry_encryption_state (void) {
   int rc = 0;
   struct encryption_status encryption_status_t;

   printf("issuing query encryption status...\n");
   memset(&encryption_status_t, 0, sizeof(struct encryption_status));
   rc = ioctl (fd, GET_ENCRYPTION_STATE, &encryption_status_t);

   if(rc == 0) {
     if(encryption_status_t.encryption_capable)
        printf("encryption capable......Yes\n");
```

```
    else
        printf("encryption capable......No\n");
    switch(encryption_status_t.encryption_method) {
         case METHOD_NONE:
             printf("encryption method.......METHOD_NONE\n");
             break;
        case METHOD_LIBRARY:
             printf("encryption method.......METHOD_LIBRARY\n");
             break;
        case METHOD_SYSTEM:
             printf("encryption method.......METHOD_SYSTEM\n");
             break;
        case METHOD_APPLICATION:
             printf("encryption method.......METHOD_APPLICATION\n");
             break;
        case METHOD_CUSTOM:
             printf("encryption method.......METHOD_CUSTOM\n");
            break;
        case METHOD_UNKNOWN:
             printf("encryption method.......METHOD_UNKNOWN\n");
            break;
        default:
            printf("encryption method.......Error\n");
     }


     switch(encryption_status_t.encryption_state) {
         case STATE_OFF:
             printf("encryption state........OFF\n");
             break;
         case STATE_ON:
             printf("encryption state........ON\n");
             break;
         case STATE_NA:
             printf("encryption state........NA\n");
            break;
        default:
             printf("encryption state......Error\n");
       }
  }
return rc;
}
```

SET_ENCRYPTION_STATE

This IOCTL command allows setting the encryption state only for application-managed encryption. On unload, some of the drive settings can be reset to default. To set the encryption state, the application must issue this IOCTL after a tape is loaded and at BOP.

The data structure that is used for this IOCTL is the same as the one for **GET_ENCRYPTION_STATE**.

An example of the **SET_ENCRYPTION_STATE** command is

```
int set_encryption_status(int option) {
  int rc = 0;
  struct encryption_status encryption_status_t;

  printf("issuing query encryption status...\n");
  memset(&encryption_status_t, 0, sizeof(struct encryption_status));
  rc = ioctl(fd, GET_ENCRYPTION_STATE, &encryption_status_t);
  if(rc < 0) return rc;
  if(option == 0)
     encryption_status_t.encryption_state = STATE_OFF;
  else if(option == 1)
```

```
      encryption_status_t.encryption_state = STATE_ON;
   else {
      printf("Invalid parameter.\n");
      return (EINVAL);
   }

   printf("Issuing set encryption status......\n");
   rc = ioctl(fd, SET_ENCRYPTION_STATE, &encryption_status_t);

   return rc;
}
```

SET_DATA_KEY

This IOCTL command allows setting the data key only for application-managed encryption.

The data structure that is used for this IOCTL is as follows on all of the supported operating systems.

```
struct data_key {
  uchar data_key_index[12];       /* The DKi */
  uchar data_key_index_length;    /* The DKi length */
  uchar reserved1[15];
  uchar data_key[32];             /* The DK  */
  uchar reserved2[48];
};
```

An example of the **SET_DATA_KEY** command is

```
int set_datakey(void) {
  int rc = 0;
  struct data_key encryption_data_key_t;

  printf("Issuing set encryption data key......\n");
  memset(&encryption_status_t, 0, sizeof(struct data_key));

  /* fill in your data key here, then issue the following ioctl*/
  rc = ioctl(fd, SET_DATA_KEY, &encryption_status_t);
  return rc;
}
```

QUERY_PARTITION

The **QUERY_PARTITION** IOCTL is used to return partition information for the tape drive and the current media in the tape drive. The data includes the current active partition the tape drive is using for the media. The **number_of partitions** field is the current number of partitions on the media. The **max_partitions** is the maximum partitions that the tape drive supports. The **size_unit** field can be either one of the defined values. Or, it can be another value (such as 8) and is used with the **size array** field value for each partition to specify the actual size partition sizes. The **partition_method** field can be Wrap-wise Partitioning or Longitudinal Partitioning. Refer to "CREATE_PARTITION" on page 140 for details.

The data structure that is used with this IOCTL is

```
The define for "partition_method":
#define UNKNOWN_TYPE                        0 /* vendor-specific or unknown */
#define WRAP_WISE_PARTITION                 1 /* Wrap-wise Partitioning     */
#define LONGITUDINAL_PARTITION              2 /* Longitudinal Partitioning  */

The define for "size_unit":
#define SIZE_UNIT_BYTES       0      /* Bytes                              */
#define SIZE_UNIT_KBYTES      3      /* Kilobytes                          */
#define SIZE_UNIT_MBYTES      6      /* Megabytes                          */
```

```
#define SIZE_UNIT_GBYTES      9      /* Gigabytes                          */
#define SIZE_UNIT_TBYTES     12      /* Terabytes                          */

typedef struct query_partition
{
  uchar max_partitions;           /* Max number of supported partitions      */
  uchar active_partition;         /* current active partition on tape        */
  uchar number_of_partitions;     /* Number of partitions from 1 to max      */
  uchar size_unit;                /* Size unit of partition sizes below      */
  ushort size[MAX_PARTITIONS];    /* Array of partition sizes in size units  */
                                  /* for each partition, 0 to (number - 1)   */
  uchar  partition_method;        /* partition type                          */
char reserved [31];
} query_partition_t;
```

Example of the **QUERY_PARTITION** IOCTL

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

   int rc,i;
   struct query_partition q_partition;

   memset((char *)&q_partition, 0, sizeof(struct query_partition));

   rc = ioctl(dev_fd, QUERY_PARTITION, &q_partition);
   if(!rc)
   {

     printf("QUERY PARTITION ioctl succeed\n");
      printf(" Partition Method = %d\n",q_partition.partition_method);
      printf("Max partitions = %d\n",q_partition.max_partitions);
      printf("Number of partitions = %d\n",q_partition.number_of_partitions);
      {
         printf("Size of Partition # %d = %d ",i,q_partition.size[i]);
         switch(q_partition.size_unit)
         {
            case SIZE_UNIT_BYTES:
               printf(" Bytes\n");
            break;
            case SIZE_UNIT_KBYTES:
               printf(" KBytes\n");
            break;
            case SIZE_UNIT_MBYTES:
               printf(" MBytes\n");
            break;
            case SIZE_UNIT_GBYTES:
               printf(" GBytes\n");
            break;
            case SIZE_UNIT_TBYTES:
               printf(" TBytes\n");
            break;
            default:
               printf("Size unit 0x%d\n",q_partition.size_unit);
         }
      }
      printf("Current active partition = %d\n",q_partition.active_partition);
   } else {
      printf("QUERY PARTITION ioctl failed\n");
   }

   return rc;
```

CREATE_PARTITION

The **CREATE_PARTITION** IOCTL is used to format the current media in the tape drive into 1 or more partitions. The number of partitions to create is specified in the

number_of_partitions field. When more than one partition is created, the type field specifies the type of partitioning, either **FDP**, **SDP**, or **IDP**. The tape must be positioned at the beginning of tape (partition 0 logical block id 0) before this IOCTL is used.

If the **number_of_partitions** field to create in the IOCTL structure is one partition, all other fields are ignored and not used. The tape drive formats the media with its default partitioning type and size for a single partition

When the type field in the IOCTL structure is set to either **FDP** or **SDP**, the **size_unit** and **size** fields in the IOCTL structure are not used. When the type field in the IOCTL structure is set to **IDP**, the **size_unit** with the **size** fields are used to specify the size for each partition.

There are two partition types in 3592 E07: Wrap-wise Partitioning (Figure 8) same as LTO 5 optimized for streaming performance and Longitudinal Partitioning (Figure 9) optimized for random access performance. Media is always partitioned into 1 by default or more than one partition. The data partition always exists as partition 0 and other extra index partition 1 to n can exist.

WORM media cannot be partitioned and the `Format Medium` commands are rejected. Attempts to scale a partitioned media is accepted. However, only if you use the correct **FORMAT** field setting, as part of scaling the volume is set to a single data partition cartridge.

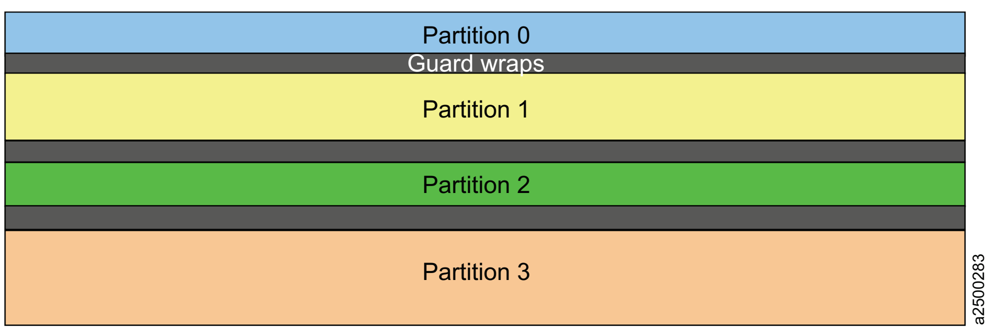

Figure 8. Wrap-wise partitioning

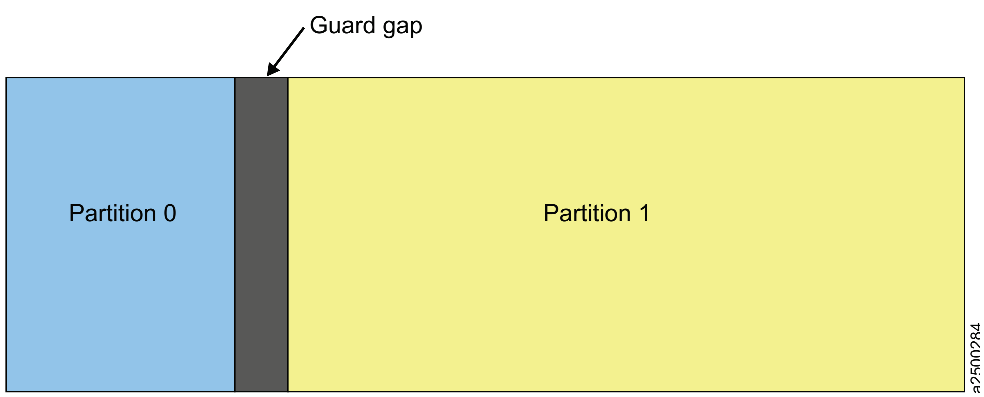

Figure 9. Longitudinal partitioning

The following chart lists the maximum number of partitions that the tape drive supports.

Table 4. Number of supported partitions

Drive type	Maximum number of supported partitions
LTO 5 (TS2250 and TS2350) and later	2 in Wrap-wise Partitioning
3592 E07 (TS 1140)	4 in Wrap-wise Partitioning 2 in Longitudinal Partitioning

The data structure that is used with this IOCTL is

```
The define for "partition_method":
#define UNKNOWN_TYPE                      0 /* vendor-specific or unknown */
#define WRAP_WISE_PARTITION               1 /* Wrap-wise Partitioning     */
#define LONGITUDINAL_PARTITION            2 /* Longitudinal Partitioning  */

The define for "type":
#define IDP_PARTITION      1      /* Initiator Defined Partition type   */
#define SDP_PARTITION      2      /* Select Data Partition type         */
#define FDP_PARTITION      3      /* Fixed Data Partition type          */

The define for "size_unit":
#define SIZE_UNIT_BYTES     0      /* Bytes                           */
#define SIZE_UNIT_KBYTES    3      /* Kilobytes                       */
#define SIZE_UNIT_MBYTES    6      /* Megabytes                       */
#define SIZE_UNIT_GBYTES    9      /* Gigabytes                       */
#define SIZE_UNIT_TBYTES   12      /* Terabytes                       */

typedef struct tape_partition
{
  uchar type;                     /* Type of tape partition to create      */
  uchar number_of_partitions;     /* Number of partitions to create        */
  uchar size_unit;                /* IDP size unit of partition sizes below */
  ushort size[MAX_PARTITIONS];    /* Array of partition sizes in size units */
                                  /* for each partition, 0 to (number - 1)  */
  uchar  partition_method;     /* partitioning type                      */
char reserved [31];
} tape_partition_t;
```

Examples of the **CREATE_PARTITION** IOCTL.

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

 struct tape_partition partition;

  /* create 2 SDP partitions for LTO-5*/
  partition.type = SDP_PARTITION;
  partition.number_of_partitions = 2;
  partition.partition_method = WRAP_WISE_PARTITION;
  ioctl(dev_fd, CREATE_PARTITION, &partition);

 /* create 2 IDP partitions with partition 1 for 37 gigabytes and partition 0
 for the remaining capacity on LTO-5*/
  partition.type = IDP_PARTITION;
  partition.number_of_partitions = 2;
  partition.partition_method = WRAP_WISE_PARTITION;
  partition.size_unit = SIZE_UNIT_GBYTES;
  partition.size[0] = 0xFFFF;
  partition.size[1] = 37;
  ioctl(dev_fd, CREATE_PARTITION, &partition);

 /* format the tape into 1 partition */
  partition.number_of_partitions = 1;
  ioctl(dev_fd, CREATE_PARTITION, &partition);

/* create 4 IDP partitions on 3592 JC volume in Wrap-wise partitioning
```

```
with partition 0 and 2 for 94.11 gigabytes (minimum size) and partition 1 and 3
to use the remaining capacity
 equally around 1.5 TB on 3592 E07 */
 partition.type = IDP_PARTITION;
 partition.number_of_partitions = 4;
 partition.partition_method = WRAP_WISE_PARTITION;
 partition.size_unit = 8;      /* 100 megabytes */
 partition.size[0] = 0x03AD;
 partition.size[1] = 0xFFFF;
 partition.size[2] = 0x03AD;
 partition.size[3] = 0x3AD2;
 ioctl(dev_fd, CREATE_PARTITION, &partition);
```

SET_ACTIVE_PARTITION

The **SET_ACTIVE_PARTITION** IOCTL is used to position the tape to a specific partition. Then, it becomes the current active partition for subsequent commands and a specific logical bock id in the partition. To position to the beginning of the partition, the **logical_block_id** field must be set to 0.

The data structure that is used with this IOCTL is

```
struct set_active_partition {
  uchar partition_number;         /* Partition number 0-n to change to     */
  ullong logical_block_id;        /* Blockid to locate to within partition */
  char reserved[32];
  };
```

Examples of the **SET_ACTIVE_PARTITION** IOCTL.

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

  struct set_active_partition partition;

  /* position the tape to partition 1 and logical block id 12 */
  partition.partition_number = 1;
  partition.logical_block_id = 12;
  ioctl(dev_fd, SET_ACTIVE_PARTITION, &partition);

  /* position the tape to the beginning of partition 0 */
  partition.partition_number = 0;
  partition.logical_block_id = 0;
  ioctl(dev_fd, SET_ACTIVE_PARTITION, &partition);
```

ALLOW_DATA_OVERWRITE

The **ALLOW_DATA_OVERWRITE** IOCTL is used to set the drive to allow a subsequent data write type command at the current position. Or, it allows a **CREATE_PARTITION** IOCTL when data safe (append-only) mode is enabled.

For a subsequent write type command, the **allow_format_overwrite** field must be set to 0. The **partition_number** and **logical_block_id** fields must be set to the current partition and position within the partition where the overwrite occurs.

For a subsequent **CREATE_PARTITION** IOCTL, the **allow_format_overwrite** field must be set to 1. The **partition number** and **logical_block_id** fields are not used. However, the tape must be at the beginning of tape (partition 0 logical block id 0) before the **CREATE_PARTITION** IOCTL is issued.

The data structure that is used with this IOCTL is

```
struct allow_data_overwrite{
  uchar partition_number;        /* Partition number 0-n to overwrite     */
  ullong logical_block_id;       /* Blockid to overwrite to within partition */
  uchar allow_format_overwrite;  /* allow format if in data safe mode      */
  char reserved[32];
  };
```

Examples of the **ALLOW_DATA_OVERWRITE** IOCTL.

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

  struct read_tape_position rpos;
  struct allow_data_overwrite data_overwrite;
  struct set_active_partition partition;

  /* set the allow_data_overwrite fields with the current position
 for the next write type command */
  data_overwrite.partition_number = rpos.rp_data.rp_long.active_partition;
  data_overwrite.logical_block_id = rpos.rp_data.rp_long.logical_obj_number;
  data_overwrite.allow_format_overwrite = 0;
  ioctl (dev_fd, ALLOW_DATA_OVERWRITE, &data_overwrite);

   /* set the tape position to the beginning of tape and */
   /* prepare a format overwrite for the CREATE_PARTITION ioctl */
   partition.partition_number = 0;
   partition.logical_block_id = 0;
   if (ioctl(dev_fd, SET_ACTIVE_PARTITION, &partition) <0)
     return errno;

   data_overwrite.allow_format_overwrite = 1;
  ioctl (dev_fd, ALLOW_DATA_OVERWRITE, &data_overwrite);
```

READ_TAPE_POSITION

The **READ_TAPE_POSITION** IOCTL is used to return **Read Position** command data in either the short, long, or extended form. The type of data to return is specified by setting the **data_format** field to either **RP_SHORT_FORM**, **RP_LONG_FORM**, or **RP_EXTENDED_FORM**.

The data structures that are used with this IOCTL are

```
#define RP_SHORT_FORM        0x00
#define RP_LONG_FORM         0x06
#define RP_EXTENDED_FORM     0x08

struct short_data_format {
  uchar bop:1,            /* beginning of partition */
        eop:1,            /* end of partition */
       locu:1,            /* 1 means num_buffer_logical_obj field is unknown */
       bycu:1,            /* 1 means the num_buffer_bytes field is unknown */
       svd :1,
       lolu:1,         /* 1 means the first and last logical obj position fields
 are unknown */
       err: 1,         /* 1 means the position fields have overflowed and can not
 be reported */
       bpew :1;        /*  beyond programmable early warning */
  uchar active_partition;         /* current active partition */
  char reserved[2];
  uint first_logical_obj_position; /* current logical object position */
  uint last_logical_obj_position;  /* next logical object to be transferred to tape */
  uint num_buffer_logical_obj;     /* number of logical objects in buffer */
  uint num_buffer_bytes;           /* number of bytes in buffer */
  char reserved1;
  };

struct long_data_format {
```

```
  uchar bop:1,                    /* beginning of partition */
            eop:1,              /* end of partition */
          rsvd1:2,
            mpu:1,     /* 1 means the logical file id field in unknown */
           lonu:1,     /* 1 means either the partition number or logical obj number
  field are unknown */
          rsvd2:1,
          bpew :1;     /*  beyond programmable early warning */
  char reserved[6];
  uchar active_partition;         /* current active partition */
  ullong logical_obj_number;      /* current logical object position */
  ullong logical_file_id;         /* number of filemarks from bop and current
 logical position */
  ullong obsolete;
  };

struct extended_data_format {
  uchar    bop      : 1,             /* beginning of partition              */
           eop      : 1,             /* end of partition                    */
           locu     : 1,             /* 1 means num_buffer_logical_obj field  */
                                     /* is unknown */
           bycu     : 1,             /* 1 means the num_buffer_bytes field is */
                                     /* unknown                             */
           rsvd     : 1,
           lolu     : 1,             /* 1 means the first and last logical    */
                                     /* obj position fields are unknown      */
           perr     : 1,             /* 1 means the position fields have     */
                                     /* overflowed and can not be reported   */
           bpew     : 1;             /* beyond programmable early warning    */

  uchar    active_partition;         /* current active partition             */
  ushort   additional_length;
  uint     num_buffer_logical_obj;   /* number of logical objects in buffer   */
  ullong   first_logical_obj_position;/* current logical object position      */
  ullong   last_logical_obj_position; /* next logical object to be transferred */
                                     /* to tape */
  ullong   num_buffer_bytes;         /* number of bytes in buffer            */
  char     reserved;
} extended_data_format_t;

typedef struct read_tape_position
{
 uchar data_format;       /* IN:  Specifies the return data format */
                           /* either short, long or extended       */
 union                    /* OUT: position data                    */
 {
  short_data_format_t    rp_short;
  long_data_format_t     rp_long;
  extended_data_format_t rp_extended;
  char reserved[64];
 } rp_data;
} read_tape_position_t ;
```

Example of the **READ_TAPE_POSITION** IOCTL.

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

 struct read_tape_position rpos;

    printf("Reading tape position long form....\n");
    rpos.data_format = RP_LONG_FORM;
    if (ioctl (dev_fd, READ_TAPE_POSITION, &rpos) <0)
       return errno;

       if (rpos.rp_data.rp_long.bop)
       printf("    Beginning of Partition ..... Yes\n");
     else
```

```
      printf("    Beginning of Partition ..... No\n");

     if (rpos.rp_data.rp_long.eop)
      printf("    End of Partition ........... Yes\n");
    else
      printf("    End of Partition ........... No\n");
     if (rpos.rp_data.rp_long.bpew)
      printf("    Beyond Early Warning ... ... Yes\n");
    else
      printf("    Beyond Early Warning ....... No\n");
     if (rpos.rp_data.rp_long.lonu)
      {
      printf("    Active Partition ........... UNKNOWN \n");
      printf("    Logical Object Number ...... UNKNOWN \n");
      }
    else
      {
      printf("    Active Partition ... ....... %u \n",
           rpos.rp_data.rp_long.active_partition);
      printf("    Logical Object Number ...... %llu \n",
           rpos.rp_data.rp_long.logical_obj_number);
       }

     if (rpos.rp_data.rp_long.mpu)
      printf("    Logical File ID ............ UNKNOWN \n");
    else
      printf("    Logical File ID ............ %llu \n",
             rpos.rp_data.rp_long.logical_file_id);
```

SET_TAPE_POSITION

The **SET_TAPE_POSITION** IOCTL is used to position the tape in the current active partition to either a logical block id or logical filemark. The **logical_id_type** field in the IOCTL structure specifies either a logical block or logical filemark.

The data structure that is used with this IOCTL is

```
#define LOGICAL_ID_BLOCK_TYPE   0x00
#define LOGICAL_ID_FILE_TYPE         0x01

struct set_tape_position{
  uchar logical_id_type;    /* Block or file as defined above */
  ullong logical_id;        /* logical object or logical file to position to */
  char reserved[32];
  };
```

Examples of the **SET_TAPE_POSITION** IOCTL.

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

  struct set_tape_position setpos;

  /* position to logical block id 10 */
  setpos.logical_id_type =  LOGICAL_ID_BLOCK_TYPE
  setpos.logical_id = 10;
  ioctl(dev_fd, SET_TAPE_POSITION, &setpos);

  /* position to logical filemark 4 */
   setpos.logical_id_type =  LOGICAL_ID_FILE_TYPE
   setpos.logical_id = 4;
   ioctl(dev_fd, SET_TAPE_POSITION, &setpos);
```

QUERY_LOGICAL_BLOCK_PROTECTION

The IOCTL queries whether the drive can support this feature, what logical block protection (LBP) method is used, and where the protection information is included.

The **lbp_capable** field indicates whether the drive has LBP capability. The **lbp_method** field displays if LBP is enabled and what the protection method is. The LBP information length is shown in the **lbp_info_length** field. The fields of **lbp_w**, **lbp_r**, and **rbdp** show that the protection information is included in write, read, or recover buffer data.

The data structure that is used with this IOCTL is

```
struct logical_block_protection
{
   uchar lbp_capable;     /* [OUTPUT] the capability of lbp for  QUERY ioctl only */
   uchar lbp_method;      /* lbp method used for QUERY [OUTPUT] and SET [INPUT] ioctls */
     #define  LBP_DISABLE          0x00
     #define  REED_SOLOMON_CRC     0x01
   uchar lbp_info_length; /* lbp info length for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_w;           /* protection info included in write data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_r;           /* protection info included in read data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar rbdp;            /* protection info included in recover buffer data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar reserved[26];
};
```

Examples of the **QUERY_LOGICAL_BLOCK_PROTECTION** IOCTL.

```
#include <sys/st.h>
  int rc;
  struct logical_block_protection lbp_protect;

  printf("Querying Logical Block Protection....\n");

  if (rc=ioctl(dev_fd, QUERY_LOGICAL_BLOCK_PROTECTION, &lbp_protect))
      return rc;

  printf("  Logical Block Protection capable...... %d\n",lbp_protect.lbp_capable);
  printf("  Logical Block Protection method........ %d\n",lbp_protect.lbp_method);
  printf("  Logical Block Protection Info Length.. %d\n",lbp_protect.lbp_info_length);
  printf("  Logical Block Protection for Write...... %d\n",lbp_protect.lbp_w);
  printf("  Logical Block Protection for Read...... %d\n",lbp_protect.lbp_r);
  printf("  Logical Block Protection for RBDP..... %d\n",lbp_protect.rbdp);
```

SET_LOGICAL_BLOCK_PROTECTION

The IOCTL enables or disables Logical Block Protection, sets up what method is used, and where the protection information is included.

The **lbp_capable** field is ignored in this IOCTL by the IBMtape driver. If the **lbp_method** field is 0 (**LBP_DISABLE**), all other fields are ignored and not used. When the **lbp_method** field is set to a valid non-zero method, all other fields are used to specify the setup for LBP.

The data structure that is used with this IOCTL is

```
struct logical_block_protection
{
   uchar lbp_capable;     /* [OUTPUT] the capability of lbp for  QUERY ioctl only */
   uchar lbp_method;      /* lbp method used for QUERY [OUTPUT] and SET [INPUT] ioctls */
     #define  LBP_DISABLE          0x00
     #define  REED_SOLOMON_CRC     0x01
   uchar lbp_info_length; /* lbp info length for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_w;           /* protection info included in write data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_r;           /* protection info included in read data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
```

```
   uchar rbdp;             /* protection info included in recover buffer data */
                           /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar reserved[26];
};
```

Examples of the **SET_LOGICAL_BLOCK_PROTECTION** IOCTL.

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

  int rc;
  struct logical_block_protection lbp_protect;

  printf("Setting Logical Block Protection....\n\n");

  printf ("Enter Logical Block Protection method:      ");
  gets (buf);
  lbp_protect.lbp_method= atoi(buf);
  printf ("Enter Logical Block Protection Info Length: ");
  gets (buf);
  lbp_protect.lbp_info_length= atoi(buf);
  printf ("Enter Logical Block Protection for Write:   ");
  gets (buf);
  lbp_protect.lbp_w= atoi(buf);
  printf ("Enter Logical Block Protection for Read:    ");
  gets (buf);
  lbp_protect.lbp_r= atoi(buf);
  printf ("Enter Logical Block Protection for RBDP:    ");
  gets (buf);
  lbp_protect.rbdp= atoi(buf);

  rc = ioctl(dev_fd, SET_LOGICAL_BLOCK_PROTECTION, &lbp_protect);

  if (rc)
    printf ("Set Logical Block Protection Fails (rc %d)",rc);
  else
    printf ("Set Logical Block Protection Succeeds");
```

Note:

1. The drive always expects a CRC attached with a data block when LBP is enabled for **lbp_r** and **lbp_w**. Without the CRC bytes attachment, the drive fails the **Read and Write** command. To prevent the CRC block transfer between the drive and application, the maximum block size limit must be determined by application.
2. The LBP setting is controlled by the application and not the device driver. If an application enables LBP, it must also disable LBP when it closes the drive, as this action is not done by the device driver.

VERIFY_TAPE_DATA

All parameters are INPUT parameters (specified by the programmer).

```
vte: verify to end of data
vlbpm: verify logical block protection information
vbf: verify by filemark
immed: return immediately, do not wait for command to complete
bytcmp: unused
fixed: verify the length of each logical block
```

Upon receiving this IOCTL, the tape drive runs the type of verification that is specified by the parameters. It returns SUCCESS if data is correct or appropriate sense data if the data is not correct.

```
typedef struct verify_data
{
  uchar       :  2,  /* reserved                              */
```

```
       vte    :  1,  /* [IN] verify to end-of-data          */
       vlbpm  :  1,  /* [IN] verify logical block
                        protection information             */
       vbf    :  1,  /* [IN] verify by filemarks            */
       immed  :  1,  /* [IN] return SCSI status immediately  */
       bytcmp :  1,  /*  No use currently                   */
       fixed  :  1;  /* [IN] set Fixed bit to verify the
                        length of each logical block       */
  uchar reseved[15];
  uint  verify_length;   /* [IN] amount of data to be verified   */

} verify_data_t;

#include <sys/st.h>
int rc;
verify_data_t  vrf_data;
memset(&vrf_data,0,sizeof(verify_data_t));

vrf_data.vte=1;
vrf_data.vlbpm=1;
vrf_data.vbf=0;
vrf_data.immed=0;
vrf_data.fixed=0;
vrf_data.verify_length=0;

printf("Verify Tape Data command ....\n");
rc=ioctl(fd,VERIFY_TAPE_DATA, &vrf_data);
if (rc)
   printf ("Verify Tape Data failed (rc %d)",rc);
else
   printf ("Verify Tape Data Succeeded!");
```

Base operating system tape drive IOCTL operations

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

The following commands are supported.

MTIOCTOP
: Run 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 runs 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 that is 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. It is 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 run. 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 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 data structure that is defined in the **/usr/include/sys/mtio.h** system header file. The two IOCTL commands are interchangeable. See "STIOC_GET_DEVICE_STATUS" on page 121.

An example of the **MTIOCGET** command is

```
#include <sys/mtio.h>
mtget mtget;

if  (!(ioctl (dev_fd, MTIOCGET, &mtget))) {
   printf ("The MTIOCGET ioctl succeeded.\n");
```

```
   printf ("\nThe device status data is:\n");
   dump_bytes ((char *)&mtget, sizeof (mtget));
} else {
   perror ("The MTIOCGET ioctl failed");
   scsi_request_sense ();
}
```

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 complete 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.

STIOC_QUERY_PATH
: Return the primary path and information for the first alternate path.

STIOC_DEVICE_PATH
: Return the primary path and all the alternate paths information.

STIOC_ENABLE_PATH
: Enable a path from the disabled state.

STIOC_DISABLE_ PATH
: Disable a path from the enabled state.

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

STIOC_DEVICE_SN

This command queries the serial number of the device that is used by the IBM 3494 tape library and the IBM TotalStorage Enterprise Virtual Tape Server.

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 run a diagnostic dump. The IBM 3490E Magnetic Tape Subsystem and the IBM TotalStorage Enterprise VTS do not support this command.

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. The IBM 3490E Magnetic Tape Subsystem and the IBM TotalStorage Enterprise VTS do not support this command.

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 must be set to one of the following values.

VEND_MODE
Vendor-specific mode.

DSCR_MODE
Descriptor mode.

DNLD_MODE
Download mode.

The **id** field must 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 must be set to one of the following values.

VEND_MODE
Vendor-specific mode.

DSCR_MODE
Descriptor mode.

DNLD_MODE
Download mode.

The **id** field must 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 ();
}
```

STIOC_QUERY_PATH

This IOCTL returns the primary path and information for the first alternate path.

The data structure is

```
typedef struct scsi_path_type
{
  char primary_name[15];              /* primary logical device name              */
  char primary_parent[15];            /* primary SCSI parent name, "Host" name    */
  uchar primary_id;                   /* primary target address of device, "Id" value*/
  uchar primary_lun;                  /* primary logical unit of device, "lun" value */
  uchar primary_bus;                  /* primary SCSI bus for device, "Channel" value*/
  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           */
  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         */
  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                     */
  char persistent_dsf[30];            /* persistent logical device name on 11i v3 */
  char reserved[30];
} scsi_path_t;
```

An example of the **STIOC_QUERY_PATH** command is

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

scsi_path_t  path;
memset(&path, 0, sizeof(scsi_path_t));
printf("Querying SCSI paths...\n");
rc = ioctl(dev_fd, STIOC_QUERY_PATH, &path);
if(rc == 0)
  show_path(&path);
```

STIOC_DEVICE_PATH

This IOCTL returns the primary path and all of the alternate paths information for a physical device. This IOCTL is supported only for a medium changer device.

The data structure is

```
struct device_path_type
{
 char name[30];                    /* logical device name             */
 char parent[30];                  /* logical parent name             */
 uchar id_valid;                   /* SCSI id/lun/bus fields valid    */
 uchar id;                         /* SCSI target address of device   */
 uchar lun;                        /* SCSI logical unit of device     */
 uchar bus;                        /* SCSI bus for device             */
 uchar fcp_id_valid;               /* FCP scsi/lun/id fields valid    */
 unsigned long long fcp_scsi_id;   /* FCP SCSi id of device           */
 unsigned long long fcp_lun_id;    /* FCP logical unit of device      */
 unsigned long long fcp_ww_name;   /* FCP world wide name             */
 uchar enabled;                    /* path enabled                    */
 uchar drive_port_valid;           /* drive port field valid          */
 uchar drive_port;                 /* drive port number               */
 uchar fenced;                     /* path fenced by disable path ioctl */
 uchar host;                       /* host bus adapter id             */
 char reserved[62];
};

#define MAX_SCSI_FAILOVER_PATH_DISPLAY 16


typedef struct device_paths
{
int  number_paths;                  /* number of paths configured       */
  int  cur_path;                    /* current active path             */
   device_path_t device_path[MAX_SCSI_FAILOVER_PATH_DISPLAY];
};
```

An example of the **STIOC_DEVICE_PATH** command is

```
#include "svc.h"
int rc = 0;
  struct device_paths paths;
  int i;

  PRINTF("Querying device paths...\n");

  if(!(rc = ioctl(dev_fd, STIOC_DEVICE_PATH, &paths)))
    {
      PRINTF("\n");
      for (i=0; i < paths.number_paths; i++)
 {
   if (i == 0)
     {
       PRINTF("Primary Path Number 1\n");
     }
   else
     {
       PRINTF("Alternate Path Number %d\n", i+1);
       PRINTF("  Logical Device....... %s\n",paths.device_path[i].name);
       PRINTF("  Host Bus Adapter..... %s\n",paths.device_path[i].parent);
     }
   if (paths.device_path[i].id_valid)
     {
       PRINTF("  SCSI Channel......... %d\n",paths.device_path[i].bus);
       PRINTF("  Target ID............ %d\n",paths.device_path[i].id);
       PRINTF("  Logical Unit......... %d\n",paths.device_path[i].lun);
     }
```

```
    if (paths.device_path[i].enabled)
      {
        PRINTF("  Path Enabled.................. Yes\n");
      }
    else
      {
        PRINTF("  Path Enabled.................. No \n");
      }
    if (paths.device_path[i].fenced)
      {
        PRINTF("  Path Manually Disabled......... Yes\n");
      }
    else
      {
        PRINTF("  Path Manually Disabled......... No \n");
      }

    PRINTF("\n");
  }

      PRINTF("Total paths configured.. %d\n",paths.number_paths);
     }
   return rc;
```

STIOC_ENABLE_PATH

This IOCTL enables the path that is specified by the path special file. This IOCTL is supported only for a medium changer device.

An example of the **STIOC_ENABLE_PATH** command is

```
#include "svc.h"
if (stat(path_name, &statbuf)!=0)
 {
   printf("Unable to stat path.\n");
   return -1;
 }

      if ((statbuf.st_rdev)&0xF00)
 {
   dev_t tempdev=(statbuf.st_rdev)&0xE00;
   tempdev>>=1; // this is the same as shift left 1 and 0xF00
   (statbuf.st_rdev)&=0xFFFFF0FF;
   (statbuf.st_rdev)|=tempdev;
 }

      devt=statbuf.st_rdev;

 if(!(rc = ioctl(dev_fd, STIOC_ENABLE_PATH, &devt)))
 {
   PRINTF("SCSI path enabled. \n");
 }
      else
 {
   PRINTF("Unabled to enable SCSI path, make sure this path is to the
same library as the opened path. \n Run Display Paths to see what paths
are connected to the opened path.\n");
 }
```

STIOC_DISABLE_PATH

This IOCTL disables the path that is specified by the path special file. This IOCTL is supported only for a medium changer device.

An example of the **STIOC_DISABLE_PATH** command is

```
#include "svc.h"

   if (stat(path_name, &statbuf)!=0)
 {
   printf("Unable to stat path.\n");
   return -1;
 }
      if ((statbuf.st_rdev)&0xF00)
 {
   dev_t tempdev=(statbuf.st_rdev)&0xE00;
   tempdev>>=1; // this is the same as shift left 1 and 0xF00
   (statbuf.st_rdev)&=0xFFFFF0FF;
   (statbuf.st_rdev)|=tempdev;
 }
      devt=statbuf.st_rdev;

if(!(rc = ioctl(dev_fd, STIOC_DISABLE_PATH,  &devt)))
 {
   PRINTF("SCSI path disabled. \n");
 }
      else
 {
   PRINTF("Unabled to enable SCSI path, make sure this path is to the
same library as the opened path. \n Run Display Paths to see what paths
are connected to the opened path.\n");
 }
```

Chapter 4. Linux tape and medium changer device driver

IBM supplies a tape drive and medium changer device driver for the Linux platform called **IBMtape**. IBM also supplies an open source device driver for Linux called **lin_tape**. Both **IBMtape** and **lin_tape** have the same programming reference as documented in this manual.

Software interface

Entry points

IBMtape supports the following Linux-defined entry points.
- "open"
- "close"
- "read" on page 160
- "write" on page 160
- "ioctl" on page 161

open

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

The programmer can access IBMtape devices with one of 3 access modes: **write only**, **read only**, or **read and write**.

IBMtape also support the **append open** flag. When the **open** function is called with the **append** flag set to TRUE, IBMtape attempts to **rewind and** seek two consecutive filemarks and place the initial tape position between them. **Open append** fails [*errno*: EIO] if no tape is loaded or two consecutive filemarks are not 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 issues a SCSI **reserve** command to the target device. If the **reserve** command fails, **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 special files, such as **/dev/IBMtape0n**, if the last command before the **close** function was a successful **write**, IBMtape writes two consecutive filemarks that marks 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 extra filemark is written that marks the end of data. Then, the tape position is set between the two consecutive filemarks.

For non-rewinding special files, if the last tape command before the close function is **write**, but the write fails with sense key 6 (Unit Attention) and ASC/ASCQ 29/00 (Power On, Reset, or Bus Device Reset Occurred) or sense key 6 and

ASC/ASCQ 28/00 (Not Ready to Ready Transition, Medium May Have Changed), IBMtape does not write two consecutive tape file marks that mark the end of data during **close** processing. If the last tape command before the close function is **write one file mark** and that command fails with one of the above two errors, IBMtape does not write one extra file mark that marks the end of data during close processing.

For rewind devices, such as **/dev/IBMtape0**, if the last command before the **close** function was a successful **write**, IBMtape writes two consecutive filemarks that mark the end of data and issues a **rewind** command. If the last command before the **close** function successfully wrote one filemark, one extra filemark is written marking the end of data, and the **rewind** command is issued. If the **write filemark** command fails, no **rewind** command is issued.

The application writers must be aware that a Unit Attention sense data that is presented means that the tape medium might be in an indeterminate condition, and no assumptions can be made about current tape positioning or whether the medium that was previously in the drive is still in the drive. IBM suggests that after a Unit Attention is presented, the tape special file be closed and reopened, label processing/verification be run (to determine that the correct medium is mounted), and explicit commands be run to locate to the wanted location. Extra processing might also be needed for particular applications.

If an **SIOC_RESERVE ioctl** was 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 that are 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** must 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 completed when a tape is loaded in the device.

IBMtape supports two modes of **read** operation. If the **read_past_filemark** flag is set to TRUE (with the **STIOCSETP** input/output control [IOCTL]), then when a **read** operation encounters a filemark, it returns the number of bytes read before it encounters the filemark and sets the tape position after the filemark. If the **read_past_filemark** flag is set to FALSE (by default or with **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, then **read** returns 0 bytes read and positions the tape after the filemark.

If the **read** function reaches end of the data on the tape, input/output error (EIO) is returned and ASC, ASCQ keys (obtained by request sense **IOCTLs**) indicate the 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 completed when a tape is loaded in the device.

IBMtape supports early warning processing. When the **trailer_labels** flag is set to TRUE (by default or with **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 suppresses all warning messages from the device that is 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 is returned, and the ASC and ASCQ keys (obtained by the request sense **IOCTL**) confirm the end of physical medium condition. When the **trailer_labels** flag is set to FALSE (with **STIOCSETP** IOCTL call), IBMtape returns the ENOSPACE *errno* when any **`write`** command is attempted in the early warning zone.

ioctl

This entry point provides a set of drive SCSI-specific functions. It allows Linux applications to access and control the features and attributes of the drive 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 attempts a SCSI **`reserve`** command to the target device. If the **`reserve`** command fails, **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 was 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 that are 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 robotic device programmatically.

General IOCTL operations

This chapter describes the IOCTL commands that provide access and control to the tape and medium changer devices.

These commands are available for all tape and medium changer devices. They can be issued to any one of the **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_LOG_SENSE10_PAGE
Return the log sense data by using a 10-byte CDB with optional subpage.

SIOC_ENH_LOG_SENSE
Return the page data with a requested length from the application if no kernel memory restriction exists.

SIOC_MODE_SENSE_PAGE
Return the mode sense data.

SIOC_MODE_SENSE
Return the mode sense data with optional subpage.

SIOC_INQUIRY_PAGE
Return the inquiry data for a specific page.

SIOC_PASS_THROUGH
Pass through custom built SCSI commands.

SIOC_QUERY_PATH
Return the primary path and information for the first alternate path.

SIOC_DEVICE_PATHS
Return the primary path and information for all the alternate paths.

SIOC_ENABLE_PATH
Enable a path from the disabled state.

SIOC_DISABLE_PATH
Disable a path.

These IOCTL commands and their associated structures are defined in the **IBM_tape.h** header file, which can be found in **/usr/include/sys** after IBMtape is installed. The **IBM_tape.h** header file must be included in the corresponding C programs that call functions that are provided by IBMtape.

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(&inqdata, 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                     */
   unchar 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   */
   unchar asc;                      /* additional sense code          */
   unchar ascq;                     /* additional sense code qualifier */
   unchar 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     */
   unchar field[2];                 /* field pointer                  */
   unchar 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** 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/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();
 }
```

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();
}
```

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, SIOC_LOG_SENSE10_PAGE, and SIOC_ENH_LOG_SENSE

These IOCTL commands return log sense data from the device. The differences between the three is

- **SIOC_LOG_SENSE_PAGE** allows the user to retrieve a particular log page up to length LOGSENSEPAGE.
- **SIOC_LOG_SENSE10_PAGE** allows for a subpage to be returned up to length LOGSENSEPAGE.
- **SIOC_ENH_LOG_SENSE** returns the page data with a requested length from application if no kernel memory restriction exists.

For both **SIOC_LOG_SENSE_PAGE** and **SIOC_LOG_SENSE10_PAGE**, to obtain the entire log page, the **len** and **parm_pointer** fields must be set to zero. To obtain the entire log page that starts at a specific parameter code, set the **parm_pointer** field to the wanted code and the **len** field to zero. To obtain a specific number of parameter bytes, set the **parm_pointer** field to the wanted code. Then, 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 are always the log page header. In the Enhanced log sense page (**SIOC_ENH_LOG_SENSE**), the length cannot be set to zero as it indicates the allocated memory size that **char *logdatap** is pointing to. The minimum number for this value is 4, as it returns the first 4 bytes that is the log page header. See the appropriate device manual to determine the supported log pages and content.

The data structures are

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

struct log_sense10_page {
       unchar page_code;
       unchar subpage_code;
       unchar reserved[2];
       unsigned short len;
       unsigned short parm_pointer;
       char data[LOGSENSEPAGE];
};
struct enh_log_sense {
  uchar page_code;       /* [IN] Log sense page     */
  uchar subpage_code;    /* [IN] Log sense sub-page */
  uchar page_control;    /* [IN] Page control       */
  uchar reserved[5];
  unsigned short len;    /* [IN] specific allocation length for logdatap */
                         /* by application                               */
                         /* [OUT] the length of return data at           */
                         /* logdatap from driver                         */
  unsigned short parm_pointer; /* [IN] specific parameter number at      */
                               /* which the data begins                  */
  char *logdatap;        /* [IN] the pointer for log sense data allocated*/
                         /* by application                               */
                         /* [OUT] log sense data returned from driver    */
  };
```

The first two IOCTLs are identical, except if a specific subpage is wanted, `log_sense10_page` must be used and `subpage_code` must be assigned by the user application.

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();
}
```

An example of the **SIOC_ENH_LOG_SENSE** command is

```
include <sys/IBM_tape.h>

#define LOG_PAGE_HEADER 4

struct enh_log_sense enh_log_page;
unsigned short length;

memset((char*)&enh_log_page, 0, sizeof(struct enh_log_sense));
enh_log_page.page_code = 0x17;
enh_log_page.subpage_code = 0x02;
enh_log_page.len = LOG_PAGE_HEADER;
enh_log_page.logdatap = malloc(LOG_PAGE_HEADER);

if(enh_log_page.logdatap == NULL){
    printf ("Unable to malloc LOG_PAGE_HEADER. Closing\n");
    exit(-1);
}

if (!ioctl (fd, SIOC_ENH_LOG_SENSE, &enh_log_page)) {
    printf ("The SIOC_ENH_LOG_SENSE ioctl succeeded\n");
    sprintf(text,"Log enhanced page header 0x%02X subpage 0x%02X",
      enh_log_page.page_code, enh_log_page.subpage_code);
    dump_bytes(enh_log_page.logdatap, enh_log_page.len, text);
}
else {
    perror ("The SIOC_ENH_LOG_SENSE ioctl failed");

}

length = (enh_log_page.logdatap[2] << 8) +enh_log_page.logdatap[3];
free(enh_log_page.logdatap);
enh_log_page.logdatap = NULL;

enh_log_page.len = length;
enh_log_page.logdatap = malloc(length);

if(enh_log_page.logdatap==NULL) {
    printf("Unable to malloc enh_log_page big size %d\n", length);
    if(length > 1024) {
     enh_log_page.logdatap = malloc(1024);
     enh_log_page.len = 1024;
     if(enh_log_page.logdatap == NULL) {
      printf("Unable to malloc enh_log_page 1024 size\n");
      exit(-1);
   }
  }
 }

 if (!ioctl (fd, SIOC_ENH_LOG_SENSE, &enh_log_page)) {
      printf ("The SIOC_ENH_LOG_SENSE ioctl succeeded\n");
      sprintf(text,"Enhanced Log Sense: page 0x%02X subpage 0x%02X length %d",
       enh_log_page.page_code, enh_log_page.subpage_code);

     dump_bytes(enh_log_page.logdatap, enh_log_page.len, text);
 }
 else {
     perror ("The SIOC_ENH_LOG_SENSE ioctl failed");
 }
 free(enh_log_page.logdatap);
```

SIOC_MODE_SENSE_PAGE and SIOC_MODE_SENSE

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 structures are

```
struct mode_sense_page {
        unchar page_code;
        char data[MAX_MDSNS_LEN];
};

struct mode_sense {
        unchar page_code;
        unchar subpage_code;
        unchar reserved[6];
        unchar cmd_code;
        char data[MAX_MDSNS_LEN];
};
```

The IOCTLs are identical, except that if a specific subpage is desired, **mode_sense** must be used and **subpage_code** must be assigned by the user application. Under the current implementation, **cmd_code** is not assigned by the user and must be left with a value 0.

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_page.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 with I/O buffer pointers to the lower SCSI layer. Status is returned from the lower SCSI layer to the caller with the **ASC and ASCQ values** and **SenseKey** fields. The ASC and ASCQ and sense key fields are valid only when the **SenseDataValid** field is true.

The data structure is

```
#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 mode. `SCSI_DATA_IN` indicates receiving data into the initiator (host bus adapter), also known as read mode. `SCSI_DATA_NONE` indicates that no data is transferred.

An example of the **`SCSI_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, SCSI_PASS_THROUGH, &PassThrough)) {
   printf ("The SCSI_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 SCSI_PASS_THROUGH ioctl failed");
   sioc_request_sense();
}
```

SIOC_QUERY_PATH

This IOCTL command returns the primary path and the first alternate path information for a physical device.

The data structure is

```
struct scsi_path
 {
   char primary_name[30];    /* primary logical device name               */
   char primary_parent[30];  /* primary SCSI parent name, "Host" name     */
   unchar primary_id;        /* primary target address of device, "Id" value*/
   unchar primary_lun;       /* primary logical unit of device, "lun" value */
   unchar primary_bus;       /* primary SCSI bus for device, "Channel" value*/
   unsigned long long primary_fcp_scsi_id;   /* not supported              */
   unsigned long long primary_fcp_lun_id;    /* not supported              */
   unsigned long long primary_fcp_ww_name;   /* not supported              */
   unchar primary_enabled;          /* primary path enabled                */
   unchar primary_id_valid;         /* primary id/lun/bus fields valid     */
   unchar primary_fcp_id_valid;     /* not supported                       */
   unchar alternate_configured;     /* alternate path configured           */
   char alternate_name[30];         /* alternate logical device name       */
   char alternate_parent[30];       /* alternate SCSI parent name          */
   unchar alternate_id;             /* alternate target address of device  */
   unchar alternate_lun;            /* alternate logical unit of device    */
   unchar alternate_bus;            /* alternate SCSI bus for device       */
   unsigned long long alternate_fcp_scsi_id;  /* not supported             */
   unsigned long long alternate_fcp_lun_id;   /* not supported             */
   unsigned long long alternate_fcp_ww_name;  /* not supported             */
   unchar alternate_enabled;                  /* alternate path enabled    */
   unchar alternate_id_valid;          /* alternate id/lun/bus fields valid */
   unchar alternate_fcp_id_valid;      /* not supported                    */
   unchar primary_drive_port_valid;    /* not supported                    */
   unchar primary_drive_port;          /* not supported                    */
   unchar alternate_drive_port_valid;  /* not supported                    */
   unchar alternate_drive_port;        /* not supported                    */
   unchar primary_fenced;         /* primary fenced by disable path ioctl   */
   unchar alternate_fenced;       /* alternate fenced by disable path ioctl */
   unchar primary_host;           /* primary host bus adapter id            */
   unchar alternate_host;       /* alternate host bus adapter id            */
   char reserved[56];
 };
```

An example of the **SIOC_QUERY_PATH** command is

```
#include <sys/IBM_tape.h>
struct scsi_path path;
memset(&path, 0, sizeof(struct scsi_path));
printf("Querying SCSI paths...\n");
rc = ioctl(fd, SIOC_QUERY_PATH, &path);
if(rc == 0)
    show_path(&path);
```

SIOC_DEVICE_PATHS

This IOCTL command returns the primary path and all of the alternate paths information for a physical device. This IOCTL supports only the 3592 tape drives. The data structure for this IOCTL command is

```
struct device_path_t
 {
   char name[30];                     /* logical device name              */
   char parent[30];                   /* logical parent name              */
   unchar id_valid;                   /* SCSI id/lun/bus fields valid     */
   unchar id;                         /* SCSI target address of device    */
   unchar lun;                        /* SCSI logical unit of device      */
   unchar bus;                        /* SCSI bus for device              */
   unchar fcp_id_valid;               /* not supported                    */
   unsigned long long fcp_scsi_id;    /* not supported                    */
   unsigned long long fcp_lun_id;     /* not supported                    */
   unsigned long long fcp_ww_name;    /* not supported                    */
   unchar enabled;                    /* path enabled                     */
   unchar drive_port_valid;           /* not supported                    */
   unchar drive_port;                 /* not supported                    */
   unchar fenced;                     /* path fenced by diable path ioctl */
   unchar host;                       /* host bus adapter id              */
   char reserved[62];
 };

 struct device_paths
 {
   int number_paths;                  /* number of paths configured       */
   struct device_path_t path[MAX_SCSI_PATH];
 };
```

An example of this IOCTL command is

```
#include <sys/IBM_tape.h>
struct device_paths device_path;
memset(%device_path, 0, sizeof(struct device_paths));
printf("Querying device paths...\n");
rc = ioctl(fd, SIOC_DEVICE_PATHS, &device_path);
 if(rc == 0)
    {
      printf("\n");
      for (i=0; i < device_path.number_paths; i++)
        {
          if (i == 0)
            printf("Primary Path Number 1\n");
          else
            printf("Alternate Path Number %d\n", i+1);
          printf("  Logical Device............ %s\n",device_path.path[i].name);
          printf("  Host Bus Adapter.......... %s\n",device_path.path[i].parent);

          if (device_path.path[i].id_valid)
            {
              printf("  SCSI Host ID.............. %d\n",device_path.path[i].host);
              printf("  SCSI Channel.............. %d\n",device_path.path[i].bus);
              printf("  Target ID................. %d\n",device_path.path[i].id);
              printf("  Logical Unit.............. %d\n",device_path.path[i].lun);
            }

          if (device_path.path[i].enabled)
            printf("  Path Enabled................... Yes\n");
          else
            printf("  Path Enabled................... No \n");
          if (device_path.path[i].fenced)
            printf("  Path Manually Disabled......... Yes\n");
          else
            printf("  Path Manually Disabled......... No \n");
```

```
        printf("\n");
      }

    printf("Total paths configured...... %d\n",device_path.number_paths);
  }
```

SIOC_ENABLE_PATH

This IOCTL enables the path that is specified by the path number. This command supports only the 3592 tape drives.

An example of this IOCTL command is

```
#include <sys/IBM_tape.h>
if (path == PRIMARY_SCSI_PATH)
    printf("Enabling primary SCSI path 1...\n");
  else
    printf("Enabling alternate SCSI path %d...\n",path);

  rc = ioctl(fd, SIOC_ENABLE_PATH, path);
```

SIOC_DISABLE_PATH

This IOCTL disables the path that is specified by the path number. This command supports only the 3592 tape drives.

An example of this IOCTL command is

```
#include <sys/IBM_tape.h>
if (path == PRIMARY_SCSI_PATH)
    printf("Disabling primary SCSI path 1...\n");
else
    printf("Disabling alternate SCSI path %d...\n",path);
rc = ioctl(fd, SIOC_DISABLE_PATH, path);
```

Tape drive IOCTL operations

The device driver supports the set of tape IOCTL commands that is available with the base Linux operating system. In addition, a set of expanded tape IOCTL commands gives applications access to extra features and functions of the tape drives.

Overview

The following IOCTL commands are supported.

STIOCTOP
: Run the 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.

STIOCDM
: Displays and manipulates one or two messages.

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.

MTDEVICE
Returns the device number that is used for communicating with an Enterprise Tape Library 3494.

STIOC_GET_DENSITY
Query the current write density format settings on the tape drive. The current density code from the drive **Mode Sense** header, the **Read/Write Control Mode** page default density, and the pending density are returned.

STIOC_SET_DENSITY
Set a new write density format on the tape drive by using the default and pending density fields. The application can specify a new write density for the currently loaded tape only. Or, it can specify a new write density as a default for all tapes.

GET_ENCRYPTION_STATE
This IOCTL can be used for application, system, and library-managed encryption. It allows only a query of the encryption status.

SET_ENCRYPTION_STATE
This IOCTL can be used only for application-managed encryption. It sets the encryption state for application-managed encryption.

SET_DATA_KEY
This IOCTL can be used only for application-managed encryption. It sets the data key for application-managed encryption.

STIOC_QUERY_PARTITION
This IOCTL queries for partition information on applicable tapes. It displays maximum number of possible partitions, number of partitions currently on tape, the active partition, the size unit (bytes, kilobytes, and so on), and the sizes of each partition.

STIOC_CREATE_PARTITION
This IOCTL creates partitions on applicable tapes. The user is allowed to specify the number and type of partitions and the size of each partition.

STIOC_SET_ACTIVE_PARTITION
This IOCTL allows the user to set the partition on which to complete tape operations.

STIOC_ALLOW_DATA_OVERWRITE
This IOCTL allows tape data to be overwritten when in data safe mode.

STIOC_READ_POSITION_EX
This IOCTL reads the tape position and includes support for the long and extended formats.

STIOC_LOCATE_16
This IOCTL sets the tape position by using a long tape format.

STIOC_QUERY_BLK_PROTECTION
This IOCTL queries the current capability and status of Logical Block Protection in the drive.

STIOC_SET_BLK_PROTECTION
This IOCTL sets the status of Logical Block Protection in the drive.

STIOC_VERIFY_TAPE_DATA
This IOCTL instructs the tape drive to scan the data on its current tape to check for errors.

STIOC_QUERY_RAO
The IOCTL is used to query the maximum number and size of User Data Segments (UDS) that are supported from tape drive and driver for the wanted **uds_type**.

STIOC_GENERATE_RAO
The IOCTL is called to send a **GRAO** list to request that the drive generate a **Recommended Access Order** list.

STIOC_RECEIVE_RAO
After a `STIOC_GENERATE_RAO` IOCTL is completed, the application calls the `STIOC_RECEIVE_RAO` IOCTL to receive a recommended access order of UDS from the drive.

| **STIOC_SET_SPDEV**
| This IOCTL is for usage through IBMSpecial open handle only. It sets the
| drive that processes the command requests, and to do so it needs the serial
| number of the drive as input.

These IOCTL commands and their associated structures are defined in the **IBM_tape.h** header file that can be found in the **lin_tape** source rpm package. This header must be included in the corresponding C program by using the IOCTL commands.

STIOCTOP

This IOCTL command runs 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
: Run the rewind operation. The tape devices run the retension operation automatically when needed.

STWEOF
: Write **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. The operation of the **SCSI Load** command varies depending on the type of device. See the appropriate hardware reference manual.

Note: If zero is used for operations that require the **st_count** parameter, then the command is not issued to the device. The device driver returns 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)) {
   printf("ioctl failure. errno=%d",errno);
   exit(errno);
}
```

STIOCQRYP or STIOCSETP

The **STIOCQRYP** command allows the program to query the tape device, device driver, and the media parameters. The **STIOCSETP** command allows the program to change the tape device, the device driver, and the media parameters.

Before the **STIOCSETP** command is issued, use the **STIOCQRYP** command to query and fill the fields of the data structure 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, each record that is written to the tape is immediately transferred to the tape. This operation guarantees that each record is on the tape, but it impacts performance.

Unchangeable parameters

The following parameters that are returned by the **STIOCQRYP** command cannot be changed by the **STIOCSETP** command.

hkwrd
: This parameter is accepted but ignored.

logical_write_protect
: This parameter sets the type of logical write protection for the tape that is loaded in the drive.

write_protect
: If the currently mounted tape is write protected, 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 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. Capacity Scaling is not currently supported in IBMtape.

density_code
This parameter is the density setting for the currently loaded tape. Some tape devices support multiple densities and 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.

read_sili_bit
This parameter is accepted but ignored. SILI bit is not supported due to Linux system environment limitations.

Changeable parameters

The following parameters can be changed by using the **STIOCSETP** IOCTL command.

trace This parameter turns the trace for the tape device On or Off.

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 that is allowed per SCSI command. In the IBMtape driver 3.0.3 or lower level, this value is 256 KB (262144 bytes) by default and changeable through the **STIOCSETP** IOCTL. In the IBMtape driver 3.0.5 or above and the open source driver lin_tape, this parameter is not changeable any more. It is determined by the maximum transfer size of the Host Bus Adapter that the tape drive is attached to.

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 that detects 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 (which are suppressed). 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 runs faster. However, the next tape command can take longer to finish because the actual physical rewind operation must complete before the next tape command can start.

logging
This parameter turns the volume logging for the tape device On or Off.

disable_sim_logging
: If this parameter is Off, the SIM/MIM data is automatically retrieved by the IBMtape device driver whenever it is available in the tape device.

disable_auto_drive_dump
: If this parameter is Off, the drive dump is automatically retrieved by the IBMtape device driver whenever a drive dump is in the tape device. It can also be set for all devices at modprobe configuration by adding disable_auto_drive_dump=1.

logical_write_protect
: This parameter sets the type of logical write protection for the tape that is loaded in the drive. See the hardware reference guide for the specific device for different types of logical write protect.

capacity_scaling
: This field can be changed only when the tape is positioned at the beginning of the tape. When a change is accepted, IBMtape rescales the tape capacity by formatting the loaded tape. See the *IBM Enterprise Tape System 3592 SCSI Reference* for the specific device for different types of capacity scaling.

IBM 3592 tape cartridges have two formats available, the 300 GB format and the 60 GB Fast Access format. The format of a cartridge can be queried under program control by issuing the **STIOCQRYP** IOCTL and checking the returned value of capacity_scaling_value (in hex).

If the capacity_scaling_value is 0x00, your 3592 tape cartridge is in 300 GB format. If the capacity_scaling_value is 0x35, your tape cartridge is in 60 GB Fast Access format. If the capacity_scaling_value is some other value, your tape cartridge format is undefined. (IBM can later define other supported cartridge formats. If so, they are documented in later versions of the *IBM TotalStorage Enterprise Tape System 3592 SCSI Reference*).

If you want to change your cartridge format, you can use the **STIOCSETP** IOCTL to change the capacity scaling value of your cartridge.

Note: All data on the cartridge is lost when the format is changed.
If you want to set it to the 300 GB format, set capacity_scaling_value to 0x00 and capacity_scaling to SCALE_VALUE. If you want to set it to the 60 GB Fast Access format, set capacity_scaling_value to 0x35 and capacity_scaling to SCALE_VALUE. Setting capacity_scaling to SCALE_VALUE is required.

Note: All data on the tape is deleted and is not recoverable.

read_past_file_mark
: This parameter changes the behavior of the **read** function when a filemark is encountered. If the **read_past_filemark** flag is TRUE when a **read** operation encounters a file mark, IBMtape returns the number of bytes read before the filemark is encountered and sets the tape position at the EOT side of the file mark.

If the **read_past_filemark** flag is FALSE (by default) when a read operation encounters a filemark, if data was read, the **read** function returns the number of bytes read, and positions the tape at the BOT side of the filemark. If no data was read, the **read** returns 0 bytes and positions the tape at the EOT side of the filemark.

limit_read_recov
: If this flag is TRUE, automatic recovery from read errors is limited to 5 seconds. If it is FALSE, the default is restored and the tape drive takes an arbitrary amount of time for read error recovery.

limit_write_recov
: If this flag is TRUE, automatic recovery from write errors is limited to 5 seconds. If it is FALSE, the default is restored and the tape drive takes an arbitrary amount of time for write error recovery.

data_safe_mode
: If this flag is TRUE, **data_safe_mode** is set in the drive. This action prevents data on the tape from being overwritten to avoid accidental data loss. If the value is FALSE, **data_safe_mode** is turned off.

pews This parameter establishes the programmable early warning zone size. It is a 2-byte numerical value that specifies how many MB before the standard end-of-medium early warning zone to place the programmable early warning indicator. If this value is set to a positive integer, a user application is warned that the tape is running out of space when the tape head reaches the PEW location. If **pews** is set to 0, then there no early warning zone occurs and the user is notified only at the standard early warning location.

The input or output data structure is

```
struct stchgp_s {
        int blksize;                /* new block size                   */
        boolean trace;              /* TRUE = message trace on          */
        uint hkwrd;                 /* trace hook word                  */
        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;            /* enable or disable 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 tranfer len         */
        char volid[16];             /* volume id                        */
        unchar 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
        unchar record_space_mode;   /* fsr/bsr space mode               */
#define SCSI_SPACE_MODE     1
#define AIX_SPACE_MODE      2
        unchar logical_write_protect; /* logical write protect          */
#define NO_PROTECT          0
#define ASSOCIATED_PROTECT  1
#define PERSISTENT_PROTECT  2
#define WORM_PROTECT        3
        unchar capacity_scaling;    /* capacity scaling                 */
#define SCALE_100           0
#define SCALE_75            1
#define SCALE_50            2
#define SCALE_25            3
#define SCALE_VALUE         4
        unchar retain_reservation;  /* retain reservation               */
```

```
        unchar alt_pathing;           /* alternate pathing active        */
        boolean emulate_autoloader;   /* emulate autoloader in random mode*/
        unchar medium_type;           /* tape medium type                */
        unchar density_code;          /* tape density code               */
        boolean disable_sim_logging;  /* disable sim/mim error logging   */
        boolean read_sili_bit;        /* SILI bit setting for read commands*/
        unchar read_past_filemark;    /* fixed block read pass the filemark*/
        boolean disable_auto_drive_dump; /* disable auto drive dump logging*/
        unchar capacity_scaling_value;   /* hex value of capacity scaling */
        boolean wfm_immediate;        /* buffer write file mark          */
        boolean limit_read_recov;     /* limit read recovery to 5 seconds */
        boolean limit_write_recov;    /* limit write recovery to 5 seconds*/
        boolean data_safe_mode;       /* turn data safe mode on/off      */
        unchar pews[2];               /* programmable early warn zone size*/
        unchar reserve_type;          /* if set persistent reserve will be used */
        unchar reserved[12];
};
```

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

```
#include <sys/IBM_tape.h>
if (ioctl(tapefd,STIOCSYNC,NULL)) {
   printf("ioctl failure. errno=%d",errno);
   exit(errno);
}
```

STIOCDM

This IOCTL command shows and manipulates one or two messages on the message display. The message that is sent with this call does not always remain on the display. It depends on the current state of the tape device. Refer to the IBM 3590 manuals for a description of the message display functions.

The input data structure is

```
#define MAXMSGLEN 8
struct stdm_s
{
   char dm_function;                /* function code */
   /* function selection */
   #define DMSTATUSMSG  0x00       /* general status message */
   #define DMDVMSG 0x20            /* demount verify message */
   #deinfe DMMIMMED 0x40            /* mount with immediate action indicator */
   #define DMDEMIMMED 0xE0         /* demount/mount with immediate action */
```

```
   /* message control */
   #define DMMSG0  0x00           /* display message 0 */
   #define DMMSG1  0x04           /* display message 1 */
   #define DMFLASHMSG0 0x08        /* flash message 0 */
   #define DMFLASHMSG1 0x0C        /* flash message 1 */
   #define DMALTERNATE 0x10        /* alternate message 0 and message 1 */
   char dm_msg0[MAXMSGLEN];        /* message 0 */
   char dm_msg1[MAXMSGLEN];        /* message 1 */
};
```

An example of the **STIOCDM** command is

```
#include <sys/IBM_tape.h>
struct stdm_s stdm;
memset(&stdm, 0, sizeof(struct stdm_s));
stdm.dm_func = DMSTATUSMSG|DMMSG0;
bcopy("SSG", stdm.dm_msg0, 8);
if(ioctl(tapefd, STIOCDM, &stdm)<0)
{
  printf("IOCTL failure, errno = %d", errno);
  exit(errno);
}
```

STIOCQRYPOS

This command queries the tape position. Tape position is defined as the location where the next read or write operation occurs. The query function can be used independently of, or with, the **STIOCSETPOS** IOCTL command.

A write filemark of count 0 is always issued to the drive, which flushes all data from the buffers to the tape media. After the write filemark finishes, 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, **lbot** contains the value **LBOT_UNKNOWN**.

Note: **lbot** indicates the last block of data that is transferred to the tape.

The number of blocks and number of bytes currently in the tape device buffers is returned in the **num_blocks** and **num_bytes** fields.

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 returned **partition_number** field is the current partition of the loaded tape.

The block ID of the next block of data to 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*/
   unchar     partition_number;        /* Current partition number on tape*/
   unchar     reserved1[2];
   blockid_t tapepos;                  /* Next block to be transferred.   */
   unchar      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)) {
   printf("ioctl failure.  errno=%d",errno);
   exit(errno);
}
oldposition=stpos.curpos;
```

STIOCSETPOS

This IOCTL command issues a high speed **locate** operation to the position specified on the tape. It uses the same position data structure that is described for **STIOCQRYPOS**, however, only the **block_type** and **curpos** fields are used during a **set** operation. **STIOCSETPOS** can be used independently of or with **STIOCQRYPOS**.

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_LOGICAL;
if (ioctl(tapefd,STIOCQRYPOS,&stpos)) {
   printf("ioctl failure.  errno=%d",errno);
   exit(errno);
}
oldposition=stpos.curpos;

stpos.curpos=oldposition;
stpos.block_type=QP_LOGICAL;
if (ioctl(tapefd,STIOCSETPOS,&stpos)) {
   printf("ioctl failure.  errno=%d",errno);
   exit(errno);
}
```

STIOCQRYSENSE

This IOCTL command returns the last sense data that is collected from the tape device. Or, it issues a new **Request Sense** command and returns the collected data. If **sense_type** equals **LASTERROR**, then the sense data is valid only if the last tape operation had an error that caused a sense command to be issued to the device. If the sense data is valid, then the IOCTL command finishes 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 */
   unchar sense[MAXSENSE];    /* actual sense data */
   int   len;                 /* length of valid sense data returned */
   int   residual_count;      /* residual count from last operation */
   unchar 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)) {
   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
   BYTE b3;
   /*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 */
   BYTE additional_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)) {
   printf("ioctl failure.  errno=%d\n",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 by 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)) {
   printf("ioctl failure. errno=%d\n",errno);
   exit(1);
}
```

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

An example of the **STIOC_RESET_DRIVE** command is

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

STIOC_PREVENT_MEDIUM_REMOVAL

This IOCTL command prevents an operator from removing media 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 media from the device. This command is normally used after the **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. It returns either ALL supported densities or only 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 that are used with this IOCTL is

```
struct density_report {
   unchar primary_density_code;   /* primary density code */
   unchar 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 {
   unchar 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;
}
```

MTDEVICE (Obtain Device Number)

This IOCTL command obtains the device number that is used for communicating with a 3494 library.

An example of the **MTDEVICE** command is

```
int device;
if(ioctl(tapefd, MTDEVICE, &device)<0)
{
  printf("IOCTL failure, errno = %d\n", errno);
  exit(errno);
}
printf("Device number is %X\n", device);
```

STIOC_GET DENSITY and STIOC_SET_DENSITY

The **STIOC_GET_DENSITY** IOCTL is used to query the current write density format settings on the tape drive. The current density code from the drive **Mode Sense** header, the **Read/Write Control Mode** page default density and pending density are returned.

The **STIOC_SET_DENSITY** IOCTL is used to set a new write density format on the tape drive by using the default and pending density fields. The density code field is not used and ignored on this IOCTL. The application can specify a new write density for the current loaded tape only or as a default for all tapes. Refer to the examples below.

The application must get the current density settings first before the current settings are modified. If the application specifies a new density for the current loaded tape only, then the application must issue another set density IOCTL after the current tape is unloaded and the next tape is loaded to either the default maximum density or a new density to ensure the tape drive uses the correct density. If the application specifies a new default density for all tapes, the setting remains in effect until changed by another set density IOCTL or the tape drive is closed by the application.

The structure for the **STIOC_GET_DENSITY** and **STIOC_SET_DENSITY** IOCTLs is

```
struct density_data_t
{
    char  density_code;         /* mode sense header density code   */
    char  default_density;      /* default write density            */
    char  pending_density;    /* pending write density              */
    char  reserved[9];
};
```

Note:

1. These IOCTLs are supported only on tape drives that can write multiple density formats. Refer to the Hardware Reference for the specific tape drive to determine whether multiple write densities are supported. If the tape drive does not support these IOCTLs, *errno* EINVAL is returned.
2. The device driver always sets the default maximum write density for the tape drive on every open system call. Any previous **STIOC_SET_DENSITY** IOCTL values from the last open are not used.

3. If the tape drive detects an invalid density code or cannot run the operation on the **STIOC_SET_DENSITY** IOCTL, the *errno* is returned and the current drive density settings before the IOCTL are restored.
4. The **struct density_data_t** defined in the header file is used for both IOCTLs. The **density_code** field is not used and ignored on the **STIOC_SET_DENSITY** IOCTL.

Examples

```
struct density_data_t data;

/* open the tape drive         */
/* get current density settings */
rc = ioctl(fd, STIOC_GET_DENSITY, %data);
/* set 3592 J1A density format for current loaded tape only */
data.default_density = 0x7F;
data.pending_density = 0x51;
rc = ioctl(fd, STIOC_SET_DENSITY, %data);
/* unload tape    */
/* load next tape */
/* set 3592 E05 density format for current loaded tape only */
data.default_density = 0x7F;
data.pending_density = 0x52;
rc = ioctl(fd, STIOC_SET_DENSITY, %data);
/* unload tape    */
/* load next tape */
/* set default maximum density for current loaded tape */
data.default_density = 0;
data.pending_density = 0;
rc = ioctl(fd, STIOC_SET_DENSITY, %data);

/* close the tape drive             */
/* open the tape drive              */
/* set 3592 J1A density format for current loaded tape and all subsequent tapes */
data.default_density = 0x51;
data.pending_density = 0x51;

rc = ioctl(fd, STIOC_SET_DENSITY, %data);
```

GET_ENCRYPTION_STATE

This IOCTL command queries the drive's encryption method and state. The data structure that is used for this IOCTL is as follows on all of the supported operating systems

```
      struct encryption_status
      {
     uchar encryption_capable;    /* (1)Set this field as a boolean based on the
                                capability of the drive  */
     uchar encryption_method;     /* (2)Set this field to one of the following */
  #define METHOD_NONE       0 /* Only used in GET_ENCRYPTION_STATE */
  #define METHOD_LIBRARY    1 /* Only used in GET_ENCRYPTION_STATE */
  #define METHOD_SYSTEM     2 /* Only used in GET_ENCRYPTION_STATE */
  #define METHOD_APPLICATION  3 /* Only used in GET_ENCRYPTION_STATE */
  #define METHOD_CUSTOM    4 /* Only used in GET_ENCRYPTION_STATE */
  #define METHOD_UNKNOWN    5 /* Only used in GET_ENCRYPTION_STATE */

      uchar encryption_state;      /* (3) Set this field to one of the following */
  #define STATE_OFF         0 /* Used in GET/SET_ENCRYPTION_STATE */
  #define STATE_ON         1  /* Used in GET/SET_ENCRYPTION_STATE */
  #define STATE_NA         2  /* Only used in GET_ENCRYPTION_STATE*/
    uchar[13] reserved;
      };
```

An example of the **GET_ENCRYPTION_STATE** command is

```
int qry_encrytion_state (void)
{
   int rc = 0;
   struct encryption_status encryption_status_t;

   printf("issuing query encryption status...\n");
   memset(,&encryption_status_t 0, sizeof(struct encryption_status));
   rc = ioctl(fd, GET_ENCRYPTION_STATE, );&encryption_status_t
   if(rc == 0)
   {
      if(encryption_status_t.encryption_capable)
   printf("encryption capable......Yes\n");
      else
   printf("encryption capable......No\n");
      switch(encryption_status_t.encryption_method)
      {
      case METHOD_NONE:
       printf("encryption method.......METHOD_NONE\n");
       break;
      case METHOD_LIBRARY:
       printf("encryption method.......METHOD_LIBRARY\n");
       break;
      case METHOD_SYSTEM:
       printf("encryption method.......METHOD_SYSTEM\n");
       break;
      case METHOD_APPLICATION:
       printf("encryption method.......METHOD_APPLICATION\n");
       break;
      case METHOD_CUSTOM:
       printf("encyrpiton method.......METHOD_CUSTOM\n");
       break;
      case METHOD_UNKNOWN:
       printf("encryption method.......METHOD_UNKNOWN\n");
       break;

      default:
       printf("encrption method.......Error\n");
      }

      switch(encryption_status_t.encryption_state)
      {
      case STATE_OFF:
       printf("encryption state........OFF\n");
       break;
      case STATE_ON:
       printf("encryption state........ON\n");
       break;
      case STATE_NA:
       printf("encryption state........NA\n");
       break;

      default:
       printf("encryption state......Error\n");
      }
   }

   return rc;
}
```

SET_ENCRYPTION_STATE

This IOCTL command allows setting the encryption state only for application-managed encryption. On unload, some drive settings might be reset to default. To set the encryption state, the application must issue this IOCTL after a tape is loaded and at BOP.

The data structure that is used for this IOCTL is the same as the one for **GET_ENCRYPTION_STATE**. An example of the **SET_ENCRYPTION_STATE** command is

```
int set_encryption_state(int option)
{
   int rc = 0;
   struct encryption_status encryption_status_t;

   printf("issuing query encryption status...\n");
   memset(,&encryption_status_t 0, sizeof(struct encryption_status));
   rc = ioctl(fd, GET_ENCRYPTION_STATE, );&encryption_status_t
   if(rc < 0) return rc;

   if(option == 0)
       encryption_status_t.encryption_state = STATE_OFF;
   else if(option == 1)
       encryption_status_t.encryption_state = STATE_ON;
   else
   {
       printf("Invalid parameter.\n");
       return -EINVAL;
   }

   printf("Issuing set encryption state......\n");
   rc = ioctl(fd, SET_ENCRYPTION_STATE, &encryption_status_t);

   return rc;
}
```

SET_DATA_KEY

This IOCTL command allows the data key to be set only for application-managed encryption. The data structure that is used for this IOCTL is as follows on all of the supported operating systems.

```
struct data_key
{
    uchar[12] data_key_index;
    uchar data_key_index_length;
    uchar[15] reserved1;
    uchar[32] data_key;
    uchar[48] reserved2;
};
```

An example of the **SET_DATA_KEY** command is

```
int set_datakey(void)
{
   int rc = 0;
   struct data_key encryption_data_key_t;

   printf("Issuing set encryption data key......\n");
   memset(,&encryption_data_key_t 0, sizeof(struct data_key));
   /* fill in your data key here, then issue the following ioctl*/
   rc = ioctl(fd, SET_DATA_KEY, &encryption_data_key_t);
   return rc;
}
```

STIOC_QUERY_PARTITION

This IOCTL queries and displays information for tapes that support partitioning. The data structure that is used for this IOCTL is

```
#define MAX_PARTITIONS 255
struct query_partition {
 unchar max_partitions;
 unchar active_partition;
```

```
 unchar number_of_partitions;
 unchar size_unit;
 ushort size[MAX_PARTITIONS];
 char reserved[32];
};
```

- **max_partitions** is the maximum number of partitions that the tape allows.
- **active_partition** is the current partition to which tape operations apply.
- **number_of_partitions** is the number of partitions currently on the tape.
- **size_unit** describes the units for the size of the tape, which is given as a logarithm to the base 10.

For example, 0 refers to 10^0 = 1, the most basic unit, which is bytes. All sizes that are reported are in bytes. 3 refers to 10^3, or kilobytes. Size is an array of the size of the partitions on tape, one array element per partition, in size_units.

An example of the **STIOC_QUERY_PARTITION** IOCTL is

```
int stioc_query_partition()
{
struct query_partition qry;
int rc = 0, i = 0;

memset(&qry, '\0', sizeof(struct query_partition));
  printf("Issuing IOCTL...\n");
rc = ioctl(fd, STIOC_QUERY_PARTITION, &qry);

  if(rc) {
      printf("Query partition failed: %d\n", rc);
    goto EXIT_LABEL;
 } /* if */

 printf("\nmax possible partitions: %d\n", qry.max_partitions);
 printf("number currently on tape: %d\n", qry.number_of_partitions);
 printf("active: %d\n", qry.active_partition);
 printf("unit: %d\n", qry.size_unit);

 for(i = 0; i < qry.number_of_partitions; i++)
    printf("size[%d]: %d\n", i, qry.size[i]);

 EXIT_LABEL:

 return rc;
 } /* stioc_query_partition() */
```

STIOC_CREATE_PARTITION

This IOCTL creates partitions on tapes that support partitioning. The data structure that is used for this IOCTL is

```
#define IDP_PARTITION    (1)
#define SDP_PARTITION    (2)
#define FDP_PARTITION    (3)
struct tape_partition {
 unchar type;
 unchar number_of_partitions;
 unchar size_unit;
 ushort size[MAX_PARTITIONS];
 char reserved[32];
};
```

Type is the type of partition, whether **IDP_PARTITION** (initiator defined partition), **SDP_PARTITION** (select data partition), or **FDP_PARTITION** (fixed data partition). The behavior of these options is described in the SCSI reference for your tape drive.

- **number_of_partitions** is the number of partitions the user wants to create.
- **size_unit** is as defined in the **STIOC_QUERY_PARTITION** section.
- **size** is an array of requested sizes, in size_units, one array element per partition.

An example of the **STIOC_CREATE_PARTITION** IOCTL is

```
int stioc_create_partition()
{
 int rc = 0, i = 0, char_cap = 0, short_cap = 0;
 struct tape_partition crt;
 char* input = NULL;

 char_cap = pow(2, sizeof(char) * BITS_PER_BYTE) - 1;
 short_cap = pow(2, sizeof(short) * BITS_PER_BYTE) - 1;

 input = malloc(DEF_BUF_SIZE / 16);
 if(!input) {
  rc = ENOMEM;
  goto EXIT_LABEL;
 } /* if */
 memset(input, '\0', DEF_BUF_SIZE / 16);

 memset(&crt, '\0', sizeof(struct tape_partition));

   while(atoi(input) < IDP_PARTITION || atoi(input) > FDP_PARTITION + 1) {
   printf("%d) IDP_PARTITION\n", IDP_PARTITION);
   printf("%d) SDP_PARTITION\n", SDP_PARTITION);
   printf("%d) FDP_PARTITION\n", FDP_PARTITION);
   printf("%d) Cancel\n", FDP_PARTITION + 1);
   printf("\nPlease select: ");

    fgets(input, DEF_BUF_SIZE / 16, stdin);
   if(atoi(input) == FDP_PARTITION + 1) {
    rc = 0;
    goto EXIT_LABEL;
   } /* if */
  } /* while */

   crt.type = atoi(input);

   memset(input, '\0', DEF_BUF_SIZE / 16);
   while(input[0] < '1' || input[0] > '9') {
   printf("Enter desired number of partitions (0 to cancel): ");
   fgets(input, DEF_BUF_SIZE / 16, stdin);
   if(input[0] == '0') {
    rc = 0;
    goto EXIT_LABEL;
   } /* if */

    if(atoi(input) > MAX_PARTITIONS) {
    printf("Please select number <= %d\n", MAX_PARTITIONS);
    input[0] = '\0';
   } /* if */
  } /* while */

   crt.number_of_partitions = atoi(input);

   if(crt.type == IDP_PARTITION && crt.number_of_partitions > 1) {
   memset(input, '\0', DEF_BUF_SIZE / 16);
   while(input[0] < '0' || input[0] > '9') {
    printf("Enter size unit (0 to cancel): ");
    fgets(input, DEF_BUF_SIZE / 16, stdin);
    if(input[0] == '0') {
     rc = 0;
     goto EXIT_LABEL;
    } /* if */
```

```
    if(atoi(input) > char_cap) {
    printf("Please select number <= %d\n", char_cap);
    input[0] = '\0';
   } /* if */
  } /* while */
  crt.size_unit = atoi(input);

   for(i = 0; i < crt.number_of_partitions; i++) {
   memset(input, '\0', DEF_BUF_SIZE / 16);
   while(input[0] != '-' &&
    (input[0] < '0' || input[0] > '9')) {
    printf("Enter size[%d] (0 to cancel, < 0 for "\
    "remaining space on cartridge): ", i);
    fgets(input, DEF_BUF_SIZE / 16, stdin);
    if(input[0] == '0') {
     rc = 0;
     goto EXIT_LABEL;
    } /* if */

     if(atoi(input) > short_cap) {
     printf("Please select number <= %d\n",
     short_cap);
     input[0] = '\0';
    } /* if */
   } /* while */
   if(input[0] == '-' && atoi(&input[1]) > 0)
   crt.size[i] = 0xFFFF;
   else crt.size[i] = atoi(input);
  } /* for */
 } /* if */

  printf("Issuing IOCTL...\n");
 rc = ioctl(fd, STIOC_CREATE_PARTITION, &crt);

  if(rc) {
  printf("Create partition failed: %d\n", rc);
  goto EXIT_LABEL;
 } /* if */

 EXIT_LABEL:

  if(input) free(input);
 return rc;
} /* stioc_create_partition() */
```

STIOC_SET_ACTIVE_PARTITION

This IOCTL allows the user to specify the partition on which to run subsequent tape operations. The data structure that is used for this IOCTL is

```
struct set_active_partition {
 unchar partition_number;
 unsigned long long logical_block_id;
 char reserved[32];
};
```

| • **partition_number** is the number of the requested active partition.
| • **logical_block_id** is the requested block position within the new active partition.

An example of the **STIOC_SET_ACTIVE_PARTITION** IOCTL is

```
int stioc_set_partition()
{
 int rc = 0;
 struct set_active_partition set;
 char* input = NULL;
```

```
 input = malloc(DEF_BUF_SIZE / 16);
 if(!input) {
  rc = ENOMEM;
  goto EXIT_LABEL;
 } /* if */
 memset(input, '\0', DEF_BUF_SIZE / 16);

 memset(&set, '\0', sizeof(struct set_active_partition));
  while(input[0] < '0' || input[0] > '9') {
   printf("Select partition (< 0 to cancel): ");
   fgets(input, DEF_BUF_SIZE / 16, stdin);

    if(input[0] == '-' && atoi(&input[1]) > 0) {
    rc = 0;
    goto EXIT_LABEL;
   } /* if */

    if(atoi(input) > MAX_PARTITIONS) {
    printf("Please select number &lt; %d\n", MAX_PARTITIONS);
    input[0] = '\0';
   } /* if */
  }  /* while */
  set.partition_number = atoi(input);

   printf("Issuing IOCTL...\n");
  rc = ioctl(fd, STIOC_SET_ACTIVE_PARTITION, &set);
   if(rc) {
   printf("Set partition failed: %d\n", rc);
   goto EXIT_LABEL;
  } /* if */

  EXIT_LABEL:

   if(input) free(input);
  return rc;
 } /* stioc_set_partition() */
```

STIOC_ALLOW_DATA_OVERWRITE

This IOCTL allows data on the tape to be overwritten when in data safe mode. The data structure that is used for this IOCTL is

```
struct allow_data_overwrite {
        unchar partition_number;
        unsigned long long logical_block_id;
        unchar allow_format_overwrite;
        char reserved[32];
};
```

- **partition_number** is the number of the drive partition on which to allow the overwrite.
- **logical_block_id** is the block that you want to overwrite.
- **allow_format_overwrite**, if set to TRUE, instructs the tape drive to allow a format of the tape and accept the **CREATE_PARTITION** ioctl.

If **allow_format_overwrite** is TRUE, **partition_number** and **logical_block_id** are ignored.

An example of the use of the **STIOC_ALLOW_DATA_OVERWRITE** IOCTL is

```
int stioc_allow_overwrite()
{
        int rc = 0, i = 0, brk = FALSE;
        struct allow_data_overwrite ado;
        char* input = NULL;
```

```
memset(&ado, '\0', sizeof(struct allow_data_overwrite));
input = malloc(DEF_BUF_SIZE / 4);
if(!input) {
      rc = ENOMEM;
      goto EXIT_LABEL;
 } /* if */
 memset(input, '\0', DEF_BUF_SIZE / 4);

 while(input[0] < '0' || input[0] > '1') {
       printf("0. Write Data 1. Create Partition (< 0 to cancel): ");
       fgets(input, DEF_BUF_SIZE / 4, stdin);

       if(input[0] == '-' && atoi(&input[1]) > 0) {
               rc = 0;
               goto EXIT_LABEL;
        } /* if */
 } /* while */

 ado.allow_format_overwrite = atoi(&input[0]);
 switch(ado.allow_format_overwrite) {
 case 0:
        memset(input, '\0', DEF_BUF_SIZE / 4);
        while((input[0] < '0' || input[0] > '9')
                && (input[0] < 'a' || input[0] > 'f')) {
                brk = FALSE;
                printf("Enter partition in hex (< 0 to cancel): 0x");
                fgets(input, DEF_BUF_SIZE / 4, stdin);

                if(input[0] == '-' && atoi(&input[1]) > 0) {
                    rc = 0;
                    goto EXIT_LABEL;
                } /* if */

                while(strlen(input) &&
                    isspace(input[strlen(input) - 1]))
                    input[strlen(input) - 1] = '\0';
                if(!strlen(input)) continue;

                for(i = 0; i < strlen(input); i++) {
                     if(input[i] >= 'A' && input[i] <= 'F')
                          input[i] = input[i] - 'A' + 'a';
                     else if(((input[i] < '0' || input[i] > '9') &&
                          (input[i] < 'a' || input[i] > 'f')) ||
                          i >= sizeof(unchar) * 2) {
                          printf("Input must be from 0 to 0xFF\n");
                          memset(input, '\0', DEF_BUF_SIZE / 4);
                          brk = TRUE;
                          break;
                      }  /* else if */
                 } /* for */
                 if(brk) continue;

  } /* while */

  ado.partition_number = char_to_hex(input);

  memset(input, '\0', DEF_BUF_SIZE / 4);
  while((input[0] < '0' || input[0] > '9')
                 && (input[0] < 'a' || input[0] > 'f')) {
                 brk = FALSE;
                 printf("Enter block ID in hex (< 0 to cancel): 0x");
                 fgets(input, DEF_BUF_SIZE / 4, stdin);

                 if(input[0] == '-' && atoi(&input[1]) > 0) {
                         rc = 0;
                         goto EXIT_LABEL;
                 } /* if */
```

```
                    while(strlen(input) &&
                            isspace(input[strlen(input) - 1]))
                            input[strlen(input) - 1] = '\0';
                    if(!strlen(input)) continue;

                    for(i = 0; i < strlen(input); i++) {
                        if(input[i] >= 'A' && input[i] <= 'F')
                            input[i] = input[i] - 'A' + 'a';
                        else if(((input[i] < '0' || input[i] > '9') &&
                            (input[i] &lt;'a' || input[i] > 'f')) ||
                            i >= sizeof(unsigned long long) * 2) {
                            printf("Input out of range\n");
                            memset(input, '\0', DEF_BUF_SIZE / 4);
                            brk = TRUE;
                            break;
                         }  /* else if */
                   } /* for */
                   if(brk) continue;

          } /* while */

          ado.logical_block_id = char_to_hex(input);
          break;
     case 1:
          break;
     default:
          assert(!"Unreachable.");
     } /* switch */

     printf("Issuing IOCTL...\n");
     rc = ioctl(fd, STIOC_ALLOW_DATA_OVERWRITE, &ado);

     if(rc) {
           printf("Allow data overwrite failed: %d\n", rc);
           goto EXIT_LABEL;
     } /* if */

 EXIT_LABEL:

     if(input) free(input);
     return rc;
} /* stioc_allow_overwrite() */
```

STIOC_READ_POSITION_EX

This IOCTL returns tape position with support for the short, long, and extended formats. The definitions and data structures that are used for this IOCTL follow. See the **READ_POSITION** section of your tape drive's SCSI documentation for details on the **short_data_format**, **long_data_format**, and **extended_data_format** structures.

```
#define RP_SHORT_FORM (0x00)
#define RP_LONG_FORM (0x06)
#define RP_EXTENDED_FORM (0x08)

struct short_data_format {
#if defined __LITTLE_ENDIAN
        unchar bpew : 1;
        unchar perr : 1;
        unchar lolu : 1;
        unchar rsvd : 1;
        unchar bycu : 1;
        unchar locu : 1;
        unchar eop : 1;
        unchar bop : 1;
```

```
#elif defined __BIG_ENDIAN
        unchar bop : 1;
        unchar eop : 1;
        unchar locu : 1;
        unchar bycu : 1;unchar rsvd : 1;
        unchar lolu : 1;
        unchar perr : 1;
        unchar bpew : 1;
#else
                error
#endif
        unchar active_partition;
        char reserved[2];
        unchar first_logical_obj_position[4];
        unchar last_logical_obj_position[4];
        unchar num_buffer_logical_obj[4];
        unchar num_buffer_bytes[4];
        char reserved1;
};

struct long_data_format {
#if defined __LITTLE_ENDIAN
        unchar bpew : 1;
        unchar rsvd2 : 1;
        unchar lonu : 1;
        unchar mpu : 1;
        unchar rsvd1 : 2;
        unchar eop : 1;
        unchar bop : 1;
#elif defined __BIG_ENDIAN
        unchar bop : 1;
        unchar eop : 1;
        unchar rsvd1 : 2;
        unchar mpu : 1;
        unchar lonu : 1;
        unchar rsvd2 : 1;
        unchar bpew : 1;
#else
                error
#endif
        char reserved[6];
        unchar active_partition;
        unchar logical_obj_number[8];
        unchar logical_file_id[8];
        unchar obsolete[8];
};

struct extended_data_format {
#if defined __LITTLE_ENDIAN
        unchar bpew : 1;
        unchar perr : 1;
        unchar lolu : 1;
        unchar rsvd : 1;
        unchar bycu : 1;
        unchar locu : 1;
        unchar eop : 1;
        unchar bop : 1;
#elif defined __BIG_ENDIAN
        unchar bop : 1;
        unchar eop : 1;
        unchar locu : 1;
        unchar bycu : 1;
        unchar rsvd : 1;
        unchar lolu : 1;
        unchar perr : 1;
        unchar bpew : 1;
#else
```

```
                error
#endif
        unchar active_partition;
        unchar additional_length[2];
        unchar num_buffer_logical_obj[4];
        unchar first_logical_obj_position[8];
        unchar last_logical_obj_position[8];
        unchar num_buffer_bytes[8];
        unchar reserved;
};

struct read_tape_position {
        unchar data_format;
        union {
                struct short_data_format rp_short;
                struct long_data_format rp_long;
                struct extended_data_format rp_extended;
        } rp_data;
};
```

`data_format` is the format in which you want to receive your data, as defined here. It can take the value `RP_SHORT_FORM`, `RP_LONG_FORM`, or `RP_EXTENDED_FORM`. When the IOCTL finishes, data is returned to the corresponding structure within the **rp_data** union.

An example of the use of the **`STIOC_READ_POSITION_EX`** IOCTL is

```
int stioc_read_position_ex(void)
{
        int rc = 0;
        char* input = NULL;
        struct read_tape_position rp = {0};

        printf("Note: only supported on LTO 5 and higher drives\n");
        input = malloc(DEF_BUF_SIZE / 16);
        if(!input) {
                rc = ENOMEM;
                goto EXIT_LABEL;
        } /* if */
        memset(input, '\0', DEF_BUF_SIZE / 16);

        while(input[0] == '\0' || atoi(input) &lt 0 || atoi(input) > 3) {
                printf("0) Cancel\n");
                printf("1) Short Form\n");
                printf("2) Long Form\n");
                printf("3) Extended Form\n");

                printf("\nPlease select: ");

                fgets(input, DEF_BUF_SIZE / 16, stdin);
                if(!atoi(input)) {
                        rc = 0;
                        goto EXIT_LABEL;
                } /* if */
        } /* while */

        memset(&rp, '\0', sizeof(struct read_tape_position));

        switch(atoi(input)) {
        case 1:
                 rp.data_format = RP_SHORT_FORM;
                 break;
        case 2:
                 rp.data_format = RP_LONG_FORM;
                 break;
        case 3:
```

```
                rp.data_format = RP_EXTENDED_FORM;
                break;
        default:
                rc = EINVAL;
                goto EXIT_LABEL;
        } /* switch */

        rc = ioctl(fd, STIOC_READ_POSITION_EX, &rp);

        if(rc) {
                printf("Cannot get position: %d\n", rc = errno);
                goto EXIT_LABEL;
        } /* if */

        print_read_position_ex(&rp);

  EXIT_LABEL:
        if(input) free(input);
        return rc;
 } /* stioc_read_position_ex() */
```

STIOC_LOCATE_16

This IOCTL sets the tape position by using the long tape format. The definitions and structure that are used for this IOCTL are

```
#define LOGICAL_ID_BLOCK_TYPE (0x00)
#define LOGICAL_ID_FILE_TYPE (0x01)

struct set_tape_position {
        unchar logical_id_type;
        unsigned long long logical_id;
        char reserved[32];
};
```

`logical_id_type` can take the values `LOGICAL_ID_BLOCK_TYPE` or `LOGICAL_ID_FILE_TYPE`. The values specify whether the tape head is located to the block with the specified logical_id or to the file with the specified logical_id. An example on how to use the **`STIOC_LOCATE_16`** IOCTL follows. The snippet assumes the declaration of global variables *filetype* and *blockid*.

```
int stioc_locate_16(void)
{
        int rc = 0;
        struct set_tape_position pos;

        memset(&pos, '\0', sizeof(struct set_tape_position));
        printf("\nLocating to %s ID %u (0x%08X)...\n",
               filetype ? "File" : "Block", blockid, blockid);

        pos.logical_id_type = filetype;
        pos.logical_id = (long long) blockid;

        rc = ioctl(fd, STIOC_LOCATE_16, &pos);
        return rc;
 } /* stioc_locate_16() */
```

STIOC_QUERY_BLK_PROTECTION

This IOCTL queries capability and status of the drive's Logical Block Protection. The structures and defines are

```
#define LBP_DISABLE            (0x00)
#define REED_SOLOMON_CRC       (0x01)

struct logical_block_protection {
```

```
        unchar lbp_capable;
        unchar lbp_method;
        unchar lbp_info_length;
        unchar lbp_w;
        unchar lbp_r;
        unchar rbdp;
        unchar reserved[26];
};
```

The `lbp_capable` is set to True if the drive supports logical block protection, or False otherwise.

A lbp_method method of LBP_DISABLE indicates that the logical block protection feature is turned off. A value of `REED_SOLOMON_CRC` indicates that logical block protection is used, with a Reed-Solomon cyclical redundancy check algorithm to run the block protection.

The `lbp_w` indicates that logical block protection is run for write commands. The `lbp_r` indicates that logical block protection is run for read commands. The `rbdp` indicates that logical block protection is run for recover buffer data. To use this IOCTL, issue the following call.

```
rc = ioctl(fd, STIOC_QUERY_BLK_PROTECTION, &lbp);
```

STIOC_SET_BLK_PROTECTION

This IOCTL sets status of the drive's Logical Block Protection. All fields are configurable except **lbp_capable** and **reserved**. The structures and defines are the same as for **`STIOC_QUERY_BLK_PROTECTION`**. To use this IOCTL, issue the following call.

```
rc = ioctl(fd, STIOC_SET_BLK_PROTECTION, &lbp);
```

STIOC_VERIFY_TAPE_DATA

This IOCTL instructs the tape drive to scan the data on its current tape to check for errors. The structure is defined as follows.

```
struct verify_data {

#if defined __LITTLE_ENDIAN
        unchar fixed     : 1;
        unchar bytcmp    : 1;
        unchar immed     : 1;
        unchar vbf       : 1;
        unchar vlbpm     : 1;
        unchar vte       : 1;
        unchar reserved1 : 2;
#elif defined __BIG_ENDIAN
        unchar reserved1 : 2;
        unchar vte       : 1;
        unchar vlbpm     : 1;
        unchar vbf       : 1;
        unchar immed     : 1;
        unchar bytcmp    : 1;
        unchar fixed     : 1;
#else
        error
#endif
        unchar verify_length[3];
        unchar reserved2[15];
};
```

`vte` instructs the drive to verify from the current tape head position to end of data.

vlbpm instructs the drive to verify that the logical block protection method that is specified in the **Control Data Protection** mode page is used for each block.

If vbf is set, then the **verify_length** field contains the number of filemarks to be traversed, rather than the number of blocks or bytes.

immed specifies that status is to be returned immediately after the command descriptor block is validated. Otherwise, the command does not return status until the entire operation finishes.

bytcmp is set to 0.

fixed indicates a fixed-block length, and that verify_length is interpreted as blocks rather than bytes.

verify_length specifies the length to verify in files, blocks or bytes, depending on the values of the **vbf** and **fixed** fields. If vte is set to 1, verify_length is ignored.

An example of the use of **STIOC_VERIFY_TAPE_DATA** is as follows.

```
int stioc_verify()
{
        int rc = 0, i = 0, cont = TRUE, len = 0;
        char* input = NULL;
        struct verify_data* vfy = NULL;

        struct {
                char* desc;
                int idx;
        } table[] = {
                {"Verify to EOD", VFY_VTE},
                {"Verify Logical Block Protection", VFY_VLBPM},
                {"Verify by Filemarks", VFY_VBF},
                {"Return immediately", VFY_IMMED},
                {"Fixed", VFY_FIXED},
                {NULL, 0}
        };

        input = malloc(DEF_BUF_SIZE / 16);
        if(!input) {
                rc = ENOMEM;
                goto EXIT_LABEL;
        } /* if */
        memset(input, '\0', DEF_BUF_SIZE / 16);

        vfy = malloc(sizeof(struct verify_data));
        if(!vfy) {
                rc = ENOMEM;
                goto EXIT_LABEL;
        } /* if */
        memset(vfy, '\0', sizeof(struct verify_data));

        printf("\n");
        for(i = 0; table[i].desc; i++) {
                while(tolower(input[0]) != 'y' && tolower(input[0]) != 'n') {
                        printf("%s (y/n/c to cancel)? ", table[i].desc);
                        fgets(input, DEF_BUF_SIZE / 16, stdin);
                        if(tolower(input[0]) == 'c') {
                                rc = 0;
                                goto EXIT_LABEL;
                        } /* if */
                } /* while */

                if(tolower(input[0]) == 'y') {
```

```
                switch(table[i].idx) {
                        case VFY_VTE:   vfy->vte   = 1; break;
                        case VFY_VLBPM: vfy->vlbpm = 1; break;
                        case VFY_VBF:   vfy->vbf   = 1; break;
                        case VFY_IMMED: vfy->immed = 1; break;
                       default: break;
                } /* switch */
        } /* if */
        memset(input,  '\0', DEF_BUF_SIZE / 16);
} /* for */

if(!vfy->vte) {
        while(cont) {
                cont = FALSE;

                printf("Verify length in decimal (c to cancel): ");
                fgets(input, DEF_BUF_SIZE / 16, stdin);

                while(strlen(input) && isspace(input[strlen(input)-1]))
                        input[strlen(input) - 1] = '\0';

                if(!strlen(input)) {
                        cont = TRUE;
                        continue;
                } /* if */

                if(tolower(input[0]) == 'c') {
                        rc = 0;
                        goto EXIT_LABEL;
                } /* if */

                for(i = 0; i < strlen(input); i++) {
                        if(!isdigit(input[i])) {
                                memset(input, '\0', DEF_BUF_SIZE / 16);
                                cont = TRUE;
                        } /* if */
                } /* for */

        } /* while */

        len = atoi(input);
        vfy->verify_length[0] = (len >> 16) & 0xFF;
        vfy->verify_length[1] = (len >> 8) & 0xFF;
        vfy->verify_length[2] = len & 0xFF;
} /* if */

rc = ioctl(fd, STIOC_VERIFY_TAPE_DATA, &vfy);
printf("VERIFY_TAPE_DATA returned %d\n", rc);
if(rc) printf("errno: %d\n", errno);

EXIT_LABEL:

        if(input) free(input);
        if(vfy) free(vfy);
        return rc;
} /* stioc_verify() */
```

STIOC_QUERY_RAO

The IOCTL is used to query the maximum number and size of User Data Segments (UDS) that are supported from tape drive and driver for the wanted **uds_type**.

The application calls this IOCTL before the **STIOC_GENERATE_RAO** and **STIOC_RECEIVE_RAO** IOCTLs are issued. The application uses the return data to limit the number of UDS requested in the **GENERATE_RAO** IOCTL.

The structure that is defined for this IOCTL is

```
struct query_rao_info{
          char   uds_type;         /* [IN]   0: UDS_WITHOUT_GEOMETRY    */
                                   /*        1: UDS_WITH_GEOMETRY       */
          char   reserved[7];
        ushort max_uds_number;     /* [OUT] Max UDS number supported from drive  */
        ushort max_uds_size;       /* [OUT] Max single UDS size supported from   */
                                   /*       drive in byte                        */
        ushort max_host_uds_number;/* [OUT] Max UDS number supported from driver */
};
```

An example of the **QUERY_RAO_INFO** command is

```
#include  <sys/IBMtape.h>
   int rc;
   struct query_rao_info stQueryRao;

bzero( (void *) &stQueryRao, sizeof(struct query_rao_info));

stQueryRao.uds_type = uds_type;

rc = ioctl(fd, STIOC_QUERY_RAO, &stQueryRao);

if(rc)
  printf("STIOC_QUERY_RAO fails with rc:   %d\n", rc);
else{
      max_host_uds_num = stQueryRao.max_host_uds_number;
      max_uds_size = stQueryRao.max_uds_size;
  }
return rc;
```

STIOC_GENERATE_RAO

The IOCTL is called to send a **GRAO** list to request that the drive generate a **Recommended Access Order** list. The process method is either 1 or 2 to create a **RAO** list, and the type of UDS is either with or without the geometry. The **uds_number** must be not larger than **max_host_uds_number** in the **STIOC_QUERY_RAO** IOCTL. The application allocates a block of memory with **grao_list_leng** (uds_number*sizeof(struct grao_uds_desc)+8) for the pointer of **grao_list**.

The structure for the **STIOC_GENERATE_RAO** IOCTL is

```
struct generate_rao {
          char   process;      /* [IN] Requested process to generate RAO list */
                               /*     0: no reorder UDS and no calculate      */
                               /*        locate time(not currently supported  */
                               /*        by the drive)                        */
                               /*     1: no reorder UDS but calculate locate  */
                               /*        time                                 */
                               /*     2: reorder UDS and calculate locate time*/
          char   uds_type;     /* [IN]0: UDS_WITHOUT_GEOMETRY                 */
                               /*     1: UDS_WITH_GEOMETRY                    */
          char   reserved1[2];
          uint    grao_list_leng;   /* [IN] The data length is allocated for GRAO */
          char   *grao_list;        /* [IN] the pointer is allocated to the size  */
                                    /*      of grao_list_leng   (uds_number       */
                                    /*      * sizeof(struct grao_uds_desc)        */
                                    /*      + sizeof(struct grao_list_header))    */
                                    /*      and contains the data of GRAO         */
                                    /*      parameter list. The uds number is     */
                                    /*      less than max_host_uds_number in      */
                                    /*      QUERY_RAO ioctl.                      */
          char   reserved2[8];
};
```

The **grao** list header and UDS segments make up the parameter data and are to be put in the following order.

-- List Header
-- UDS Segment Descriptor (first)
......
-- UDS Segment Descriptor (last)

The device driver does not supply the header or UDS segment Descriptor structures. That structure is to be supplied by the application.

Examples of the data structures are

```
struct grao_list_header{
  unchar reserved[4];
  char  addl_data[4];   /* additional data                   */
} ;

struct grao_uds_desc{
  unchar desc_leng[2];  /* descriptor length                 */
  char   reserved[3];
  char   uds_name[10];  /* uds name given by application     */
  unchar partition;     /* Partition number 0-n to overwrite  */
  unchar beginning_loi[8] ;    /* Beginning logical object ID */
  unchar ending_loi[8];        /* Ending logical object ID    */
};
```

A sample of **STIOC_GENERATE_RAO** is

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

int rc;
struct generate_rao grao;
bzero(&grao,sizeof(struct generate_rao));
grao.process=2;
grao.uds_type=uds_type;
grao.grao_list_leng=max_host_uds_num * sizeof(struct grao_uds_desc)
     + sizeof(struct grao_list_header);
if(!(grao.grao_list=malloc(grao.grao_list_leng)))
{
  perror("Failure allocating memory");
  return (errno);
}

memset(grao.grao_list, 0, grao.grao_list_leng);

rc=ioctl(fd,GENERATE_RAO,&grao);

if (rc)
   printf("GENERATE_RAO fails with rc%d\n",rc);
else
   printf("GENERATE_RAO succeeds\n");

free(grao.grao_list);

return rc;
```

STIOC_RECEIVE_RAO

After a **STIOC_GENERATE_RAO** IOCTL is completed, the application calls the **STIOC_RECEIVE_RAO** IOCTL to receive a recommended access order of UDS from the drive. To avoid a system crash, it is important that the application allocates a large enough block of memory for the ***rrao_list** pointer and notifies the driver of the allocated size. It is done by indicating the size of the buffer in bytes to the *rrao_list_leng* variable as an input to the **receive_rao_list** structure.

The structure for the **STIOC_RECEIVE_RAO** IOCTL is

```
struct receive_rao_list {
        uint    rrao_list_offset;  /* [IN] The offset of receive RAO list to */
                                   /*      begin returning data              */
        uint    rrao_list_leng;    /* [IN/OUT] number byte of data length    */
                                   /* [IN] The data length is allocated for RRAO */
                                   /*      list by application the length is     */
                                   /*      (max_uds_size * uds_number +          */
                                   /*      sizeof(struct rrao_list_header)       */
                                   /*      max_uds_size is reported in           */
                                   /*      sizeof(struct rrao_list_header)       */
                                   /*      uds_number is the total UDS number    */
                                   /*      requested from application in         */
                                   /*      GENERATE_RAO ioctl                    */
                                   /* [OUT] the data length is actual returned   */
                                   /*      in RRAO list from the driver          */
        char   *rrao_list;         /* [IN/OUT] the data pointer of RRAO list     */
        char   reserved[8];
};
```

The sample code is

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

int rc;
struct receive_rao_list rrao;

bzero(&rrao,sizeof(struct receive_rao_list));
rrao.rrao_list_offset=0;
rrao.rrao_list_leng= max_host_uds_num * max_uds_size + 8;
/* 8 is the header of rrao list */

if (!(rrao.rrao_list=malloc(rrao.rrao_list_leng)))
{
  perror("Failure allocating memory");
  return (errno);
}

memset(rrao.rrao_list, 0, rrao.rrao_list_leng);

rc=ioctl(fd,STIOC_RECEIVE_RAO,&rrao);

if (rc)
  printf("STIOC_RECEIVE_RAO fails with rc %d\n",rc);
else
  printf("rrao_list_leng %d\n",rrao.rrao_list_leng);

free(rrao.rrao_list);

return rc;
```

|
STIOC_SET_SPDEV

| With the latest lin_tape versions, the IBMSpecial device is created. It allows the use
| of ioctls for preemption purposes. Applications must use it cautiously and manage
| persistent reservation properly.

| This ioctl is for usage through IBMSpecial open handle only. It sets the drive that
| processes the command requests, and to do so it needs the serial number of the
| drive as input. If /dev/IBMSpecial is not created, it is not supported.

The data structure is

```
#define DD_MAX_DEVICE_SERIAL     36
struct sp_dev{
    char device_serial[DD_MAX_DEVICE_SERIAL];
};
```

An example of the **STIOC_SET_SPDEV** command is

```
#include <sys/IBM_tape.h>
struct sp_dev spd;

setDriveSN(spd.device_serial);
if (!ioctl (fd, STIOC_SET_SPDEV, &spd)) {
printf ("The STIOC_SET_SPDEV ioctl succeeded\n");
}
else {
perror ("The STIOC_SET_SPDEV ioctl failed the drive to work with was not set");
}
```

When the **STIOC_SET_SPDEV** ioctl succeeds, it is possible to send any of these ioctls to the drive previously set and identified by serial number: **STIOC_READ_RESERVEKEYS**, **STIOC_READ_RESERVATIONS**, **STIOC_REGISTER_KEY**, **STIOC_REMOVE_REGISTRATION**, **STIOC_CLEAR_ALL_REGSITRATION**.

Tape drive compatibility IOCTL operations

The following IOCTL commands help provide compatibility for previously compiled programs. Where practical, such programs must be recompiled to use the preferred IOCTL commands in the IBMtape device driver.

MTIOCTOP

This IOCTL command is similar in function to the **st** **MTIOCTOP** command. It is provided as a convenience for precompiled programs that 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 that 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 that call that IOCTL command. Refer to **/usr/include/sys/mtio.h** or **/usr/include/linux/mtio.h** for information on the **MTIOCPOS** command.

Medium changer IOCTL operations

This chapter describes the IOCTL commands that provide access and control of the SCSI medium changer functions. These IOCTL operations can be issued to the medium changer special file, such as **IBMchanger0**.

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.

SCSI IOCTL commands

These IOCTL commands and their associated structures are defined in the **IBM_tape.h** header file, which can be found in `/usr/include/sys` after IBMtape is installed. The **IBM_tape.h** header file is included in the corresponding C program by using the functions.

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 ((unchar *) &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. The second moves the cartridge that was previously in the **destination1** element to the **destination 2** element. This function is available only 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>
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 this IOCTL command is issued.

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). Then, it 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            */
         impexp  :1,  /* import/export placed by operator or robot */
         full    :1;  /* element contains medium           */
   unchar resvd1;     /* reserved                          */
   unchar asc;        /* additional sense code             */
   unchar 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               */
   unchar scsi;       /* SCSI bus address                  */
   unchar resvd2;     /* reserved                          */
   uint  svalid  :1,  /* element address valid             */
         invert  :1,  /* medium inverted                   */
                 :6;  /* reserved                          */
   ushort source;     /* source storage element address    */
   unchar volume[36]; /* primary volume tag                */
```

```
   unchar 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_info element_info;
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(&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 ((unchar *)(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 ((unchar *)(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 ((unchar *)(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 ((unchar *)(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, it loads 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/IBM_tape.h>
/* load cartridge from slot 3 */
if (ioctl (tapefd, SMCIOC_LOAD_MEDIUM,3)) {
   printf ("IOCTL failure. errno=%d\n",errno);
   exit(1);
}
/* load first cartridge from magazine */
if (ioctl (tapefd, SMCIOC_LOAD_MEDIUM,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, it moves a tape 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 fails with *errno* EIO.

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)) {
   printf ("IOCTL failure. errno=%d\n",errno);
   exit(1);
}
/* unload cartridge to first empty slot in magazine */
if (ioctl (tapefd, SMCIOC_UNLOAD_MEDIUM,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 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 of elements that are 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            */
   unchar resvd1;          /* reserved                           */
   unchar asc;             /* additional sense code              */
   unchar 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                */
   unchar scsi;            /* scsi bus address                   */
   unchar 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                    */
   unchar resvd3;          /* reserved                           */
   unchar ident_len;       /* identifier length                  */
   unchar 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;
}
```

Return codes

This chapter describes error codes that are generated by IBMtape when an error occurs during an operation. On error, the operation returns negative one (**-1**), and the external variable *errno* is set to one of the listed error codes. *Errno* values are defined in **/usr/include/errno.h** (and other files that it includes). Application programs must include **errno.h** 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 by using the appropriate hardware or SCSI reference for that device.

General error codes

The following codes apply to all operations.

[EBUSY]
An excessively busy state was encountered in the device.

[EFAULT]
A memory failure occurred due to an invalid pointer or address.

[EIO]
An error due to one of the following conditions:
- An unrecoverable media error was detected by the device.
- The device was not ready for operation or a tape was not in the drive.
- The device did not respond to SCSI 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 run the desired function.

[ETIMEDOUT]
A command timed out in the device.

Open error codes

The following codes apply to **open** operations.

[EACCES]
The **open** requires write access when the cartridge loaded in the drive is physically write-protected.

[EAGAIN]
The device was already open when an **open** was attempted.

[EBUSY]
The device was reserved by another initiator or an excessively busy state was encountered.

[EINVAL]
The operation that is 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. Run failure analysis.

[ENOMEM] Insufficient memory was available for an internal memory operation.

[EPERM] One of the following situations occurred:

- 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.

Close error codes

The following codes apply to **close** operations.

[EBUSY] The SCSI subsystem was busy.

[EFAULT] Memory reallocation failed.

[EIO] A command that is issued during **close**, such as a **`rewind`** command, failed because the device was not ready. An I/O error occurred during the operation. Run failure analysis.

Read error codes

The following codes apply to **read** operations.

[EFAULT] Failure copying from user to kernel space or vice versa.

[EINVAL] One of the following situations occurred:

- The operation that is 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 apply to **write** operations.

[EFAULT] Failure copying from user to kernel space or vice versa.

[EINVAL]
: One of the following conditions occurred:
- The operation that is 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 one time 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 apply to IOCTL operations.

[EBUSY]
: SCSI subsystem was busy.

[EFAULT]
: Failure copying from user to kernel space or vice versa.

[EINVAL]
: The operation that is 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, if you are 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. Run 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 might 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.

Chapter 5. Solaris tape and medium changer device driver

IOCTL operations

The following sections describe the IOCTL operations supported by the IBMtape device driver for Solaris. Usage, syntax, and examples are given.

The IOCTL operations supported by the Solaris tape and medium changer device driver support are described in

- “General SCSI IOCTL operations”
- “SCSI medium changer IOCTL operations” on page 232
- “SCSI tape drive IOCTL operations” on page 242
- “Base operating system tape drive IOCTL operations” on page 278
- “Downward compatibility tape drive IOCTL operations” on page 281
- “Service aid IOCTL operations” on page 287

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 whether the device is ready for operation.

IOC_INQUIRY
: Collect the inquiry data from the device.

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

IOC_REQUEST_SENSE
: Return the device sense data.

IOC_LOG_SENSE_PAGE
: Collect the log sense page data from the device.

IOC_LOG_SENSE10_PAGE
: Enhanced to add a Subpage variable from `IOC_LOG_SENSE_PAGE`. It returns a log sense page or Subpage from the device.

IOC_ENH_LOG_SENSE
: Enhanced to define the *len* variable as input by using program allocated memory with a pointer that is limited only by available kernel memory.

IOC_MODE_SENSE
: Return the mode sense data for a specific page.

IOC_MODE_SENSE_SUBPAGE
: Return the mode sense data for a specific page and Subpage.

SIOC_MODE_SENSE
: Return whole mode sense data and support for Mode Sense Subpage.

IOC_DRIVER_INFO
: Return the driver information.

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 that is installed with the IBMtape package. Any application program that issues these commands must include this header file.

IOC_TEST_UNIT_READY

This command determines whether 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 */
        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 x80 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, 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 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 parameters.

```
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.

```
#define IOC_LOG_SENSE_PAGE (_IOWR('S',6, log_sns_pg_t)
#define LOGSENSEPAGE 1024 /* The maximum data length which this */
                /* ioctl can return, including the    */
                /* log page header. This value is not */
                /* application modifiable.       */
typedef struct log_sns_pg_s {
    uchar page_code;    /* Log page to be returned.          */
    uchar subpage_code;   /* Log subpage to be returned.        */
    uchar reserved1[1];   /* Reserved for IBM future use.      */
    uchar reserved2[2];   /* Reserved for IBM future use.      */
    uchar data[LOGSENSEPAGE];  /* Log page data will be placed here. */
} log_sns_pg_t;
```

An example of the **IOC_LOG_SENSE_PAGE** command is

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

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

if(!(ioctl(dev_fd, IOC_LOG_SENSE_PAGE,&log_sns_pg))){
     log_data_len = (uint)(((log_page_hdr_p->len[2]<<8) |
     log_page_hdr_p->len[3])+4);
     returned_len = MIN(log_data_len,sizeof log_sns_pg.data);
     printf ("\n Log Sense Page ioctl succeeded.\n");
     printf(" Log Page 0x%X data, length %d(%d returned):
     \n",page,log_data_len,returned_len);
     dump_bytes((char*)log_page_p,returned_len);
}
else {
     perror("The IOC_INQUIRY ioctl failed");
     scsi_request_sense(); }
```

IOC_LOG_SENSE10_PAGE

This IOCTL command is enhanced to add a Subpage variable from **IOC_LOG_SENSE_PAGE**. It returns a log sense page or Subpage from the device.

The data structure that is used with this IOCTL is

```
#define LOGSENSEPAGE 1024      /* The maximum data length which this     */
                               /* ioctl can return, including the        */
                               /* log page header.  This value is not    */
                               /* application modifiable.                */
}

typedef struct  {
    uchar page_code;     /*  Log sense page */
    uchar subpage_code;  /*  Log sense subpage */
    uchar reserved[2];   /*  Reserved for IBM future use. */
    ushort len;          /*  number of valid bytes in data
                             (log_page_header_size+page_length) */
    ushort parm_pointer;       /*  specific parameter number at which
                                   the data begins */
    char data[LOGSENSEPAGE];   /*  log data */
}log_sense10_page_t;
```

Examples of the **IOC_LOG_SENSE10_PAGE** IOCTL.

```
#include<sys/st.h>
log_sense10_page_t log_sns_pg;
memset((char*)&log_sns_pg,0,sizeof(log_sense10_page_t));
log_sns_pg.page_code = page;
log_sns_pg.page_code =subpage;
log_sns_pg.parm_pointer =parm;

if(!(ioctl(dev_fd, IOC_LOG_SENSE10_PAGE,&log_sns_pg))){
```

```
log_data_len = (uint)(((log_page_hdr_p->len[2]<<8) | log_page_hdr_p-
>len[3])+4);
returned_len = MIN(log_data_len,sizeof log_sns_pg.data);
printf ("\n Log Sense Page ioctl succeeded.\n");
printf(" Log Page 0x%X data, length %d(%d returned):
\n",page,log_data_len,returned_len);
dump_bytes((char*)log_page_p,returned_len);
}
else { perror("The IOC_LOG_SENSE10_PAGE ioctl failed");
scsi_request_sense(); }
}
```

IOC_ENH_LOG_SENSE

This IOCTL command is enhanced to define the *len* variable as input and limited for available kernel memory from the **IOC_LOG_SENSE10_PAGE**.

The data structure that is used with this IOCTL is

```
typedef struct
{
  uchar page_code;         /* [IN] Log sense page     */
  uchar subpage_code;      /* [IN] Log sense sub-page */
  uchar page_control;      /* [IN] Page control       */
  uchar reserved[5];
  unsigned short len ;     /* [IN] specific allocation length for logdatap by application */
                           /* [OUT] the length of return data at logdatap from driver     */
  unsigned short parm_pointer; /* [IN] specific parameter number at which the data begins */
  char *logdatap;          /* [IN] the pointer for log sense data allocated by application*/
                           /* [OUT] log sense data returned from driver */
} enh_log_sense;
```

Examples of the **IOC_ENH_LOG_SENSE** IOCTL.

```
#include<sys/st.h>
enh_log_sense log_sns_pg;
memset((char*)&log_sns_pg,0,sizeof(enh_log_sense));
log_sns_pg.page_code = page; log_sns_pg.page_code =subpage;
log_sns_pg.parm_pointer =parm;

if(!(ioctl(dev_fd, IOC_ENH_LOG_SENSE,&log_sns_pg))){
   log_data_len = (uint)(((log_page_hdr_p->len[2]<<8) | log_page_hdr_p-
>len[3])+4);
   returned_len = MIN(log_data_len,sizeof log_sns_pg.logdatap);
   printf ("Log Sense Page ioctl succeeded.");
   printf(" Log Page 0x%X data, length %d(%d
 returned):",page,log_data_len,returned_len);
   dump_bytes((char*)log_page_p,returned_len);
 }
 else {
    perror("The ENH_LOG_SENSE  ioctl failed");
    scsi_request_sense();
 }
```

IOC_MODE_SENSE

This command returns a mode sense page from the device. The 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 *)&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_MODE_SENSE_SUBPAGE

This command returns the mode sense data for a specific page and Subpage from the device. The page and Subpage are selected by specifying the **page_code** and **subpage_code** in the **mode_sns_subpage_t** structure.

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

```
#define MAX_MS_SUBDATA  10240         /* The maximum subpage data length which */
                                      /* this ioctl can return, including   */
                                      /* headers and block descriptors.     */

typedef struct {
   uchar      page_code;              /* Page Code:  Set this field with    */
                                      /*    the desired mode page number    */
                                      /*    before issuing the ioctl        */
   uchar      subpage_code;           /* Subpage Code: Set this field with  */
                                      /*    the desired mode page subpage   */
                                      /*    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      reserved[13];
   uchar      data[MAX_MS_SUBDATA];    /* Mode Subpage Data:  Upon return,   */
                                       /*    this field contains the mode    */
                                       /*    Subpage data up to the max      */
                                       /*    length supported by the ioctl   */
} mode_sns_subpage_t;
```

An example of the **IOC_MODE_SENSE_SUBPAGE** command is

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

   int              rc;
   int              header_len;
   int              blk_dsc_len = 0;
   int              mode_data_len = 0;
   int              mode_data_returned_len = 0;
   int              max_mdsnspg_data_len = 0;
   uchar            cmd_code;
   uchar            medium_type;
   uchar            density_code;
   uchar            wrt_prot;
   char             *header_p;
   char             *blkdsc_p;
   void             *mode_data_p;
   mode_sns_subpage_t   mode_subpage;

   memset ((char *)&mode_subpage, 0, sizeof(mode_sns_subpage_t));
   mode_subpage.page_code = page;
   mode_subpage.subpage_code = subpage;

   if (!(rc = ioctl (dev_fd, IOC_MODE_SENSE_SUBPAGE, &mode_subpage))) {
      printf ("IOC_MODE_SENSE_SUBPAGE succeeded.\n");
      header_p = (char *)&mode_subpage.data;
      cmd_code = mode_subpage.cmd_code;
      if ( cmd_code == MODESNS_6_CMD ) {
         header_len             = sizeof(mode_hdr6_t);
         mode_data_len          = (uint) ((mode_hdr6_t *)header_p)->data_len;
         blk_dsc_len            = (uint) ((mode_hdr6_t *)header_p)->blk_dsc_len;
         max_mdsnspg_data_len   = MAX_MS_SUBDATA - header_len - blk_dsc_len;
         mode_data_returned_len = MIN( mode_data_len + 1, max_mdsnspg_data_len);
         medium_type            = (uchar)((mode_hdr6_t *)(header_p))->medium_type;
         wrt_prot               = (uchar)((mode_hdr6_t *)(header_p))->wrt_prot;
      }
      else if ( cmd_code == MODESNS_10_CMD ) {
         header_len    = sizeof(mode_hdr10_t);
         mode_data_len = (uint) ((((mode_hdr10_t *)header_p)->data_len[0] << 8)
               | ((mode_hdr10_t *)header_p)->data_len[1]);
         blk_dsc_len   = (uint) ((((mode_hdr10_t *)header_p)->blk_dsc_len[0] << 8)
               | ((mode_hdr10_t *)header_p)->blk_dsc_len[1] );
         max_mdsnspg_data_len   = MAX_MS_SUBDATA - header_len - blk_dsc_len;
         mode_data_returned_len = MIN(mode_data_len+2, max_mdsnspg_data_len);
         medium_type            = (uchar)((mode_hdr10_t *)(header_p))->medium_type;
         wrt_prot               = (uchar)((mode_hdr10_t *)(header_p))->wrt_prot;
      }
      else {
         fprintf (stderr, "mode sense: Unknown mode sense command code '0x%X'.\n",
  cmd_code);
         return (1);
      }        blkdsc_p     = header_p + header_len;
      mode_data_p  = blkdsc_p + blk_dsc_len;
      density_code = (blk_dsc_len ? ( unsigned char )((blkdsc_t
 *)(blkdsc_p))->density_code : 0);
      printf ("Page Code      x'%2.2X'\n", page);
      printf ("SubPage Code   x'%2.2X'\n", subpage);
      printf ("Command Code   x'%2.2X'\n", mode_subpage.cmd_code);
      printf ("Mode Data Len  %4d\n", mode_data_len);
```

```
        printf ("Blk Desc Len   %4d\n", blk_dsc_len);
        printf ("Returned Len   %4d\n", mode_data_returned_len);
        printf ("Write Protect  x'%2.2X'\t\n", wrt_prot);
        printf ("Medium Type    x'%2.2X'\t\n", medium_type);
        if (blk_dsc_len != 0)
           printf ("Density Code   x'%2.2X'\t\n", density_code);

        printf ("\nHeader:\n");
        DUMP_BYTES ((char *)(header_p), header_len);
        if (blk_dsc_len != 0) {
           printf ("\nBlock Descriptor:\n");
           DUMP_BYTES ((char *)(blkdsc_p), blk_dsc_len);
        }
        printf ("\nMode Page:\n");
        DUMP_BYTES ((char *)(mode_data_p), (mode_data_returned_len - header_len -
  blk_dsc_len));
     }
     else {
        perror ("mode sense subpage");
     }
     return (rc);
```

SIOC_MODE_SENSE

This command returns the mode sense data for a specific page and Subpage from the device. The page and Subpage are selected by specifying the **page_code** and **subpage_code** in the **mode_sense_t** structure.

```
#define MAX_MS_SUBDATA  10240          /* The maximum subpage 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 */
  #define MODESENSEPAGE 255   /* max data xfer for mode sense/select page ioctl  */

  typedef struct   {
    uchar   page_code;      /* mode sense page code       */
    uchar   subpage_code;   /* mode sense subpage code    */
    uchar   reserved[6];    /*Reserved for IBM future use.*/
    uchar   cmd_code;       /* SCSI Command Code: this field is set with     */
                            /* SCSI command code which the device responded. */
                               /*  x'5A' = Mode Sense (10) */
                               /*  x'1A' = Mode Sense (6)  */
    char   data[MODESENSEPAGE];  /* whole mode sense data include header,
                                    block descriptor and page */
    } mode_sense_t;
```

An example of the **SIOC_MODE_SENSE** command is

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

     int               header_len;
     int               blk_dsc_len = 0;
     int               mode_data_len = 0;
     int               mode_data_returned_len = 0;
     int               max_mdsnspg_data_len = 0;
     uchar             cmd_code;
     uchar             medium_type;
     uchar             density_code;
     uchar             wrt_prot;
     char              *header_p;
     char              *blkdsc_p;
     void              *mode_data_p;
     mode_sense_t mode_sns;
```

```
memset ((char *)&mode_sns, 0, sizeof(mode_sense_t));

mode_sns.page_code = page;
mode_sns.subpage_code = subpage;

if (!(rc = ioctl (dev_fd, SIOC_MODE_SENSE, &mode_sns))) {
   header_p = (char *)&mode_sns.data;
   cmd_code = mode_sns.cmd_code;
   if ( cmd_code == MODESNS_6_CMD ) {
      header_len             = sizeof(mode_hdr6_t);
      mode_data_len          = (uint) ((mode_hdr6_t *)header_p)->data_len;
      blk_dsc_len            = (uint) ((mode_hdr6_t *)header_p)->blk_dsc_len;
      max_mdsnspg_data_len   = MAX_MS_SUBDATA - header_len - blk_dsc_len;
      mode_data_returned_len = MIN( mode_data_len + 1, max_mdsnspg_data_len);
      medium_type            = (uchar)((mode_hdr6_t *)(header_p))->medium_type;
      wrt_prot               = (uchar)((mode_hdr6_t *)(header_p))->wrt_prot;
}
else if ( cmd_code == MODESNS_10_CMD ) {
      header_len    = sizeof(mode_hdr10_t);
      mode_data_len = (uint) ((((mode_hdr10_t *)header_p)->data_len[0] << 8)
            | ((mode_hdr10_t *)header_p)->data_len[1]);

      blk_dsc_len   = (uint) ((((mode_hdr10_t *)header_p)->blk_dsc_len[0] << 8)
            | ((mode_hdr10_t *)header_p)->blk_dsc_len[1] );
      max_mdsnspg_data_len   = MAX_MS_SUBDATA - header_len - blk_dsc_len;
      mode_data_returned_len = MIN(mode_data_len+2, max_mdsnspg_data_len);
      medium_type            = (uchar)((mode_hdr10_t *)(header_p))->medium_type;
      wrt_prot               = (uchar)((mode_hdr10_t *)(header_p))->wrt_prot;
}
else {
      fprintf (stderr, "mode sense: Unknown mode sense command code
      '0x%X'.\n", cmd_code);
      return (1);
}
  blkdsc_p     = header_p + header_len;
  mode_data_p  = blkdsc_p + blk_dsc_len;
  density_code = (blk_dsc_len
                 ? ( unsigned char )((blkdsc_t *)(blkdsc_p))->density_code  :
                 0 );
  PRINTF ("\nHeader:\n");
  DUMP_BYTES ((char *)(header_p), header_len);
  if (blk_dsc_len != 0) {
      PRINTF ("\nBlock Descriptor:\n");
      DUMP_BYTES ((char *)(blkdsc_p), blk_dsc_len);
  }
  PRINTF ("\nMode Page:\n");
  DUMP_BYTES ((char *)(mode_data_p),
 (mode_data_returned_len - header_len - blk_dsc_len));
}
else {
  PERROR ("mode sense page");
  PRINTF ("\n");
  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, &IBMtape_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 that is 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.

SMCIOC_MOVE_MEDIUM
Transport 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_ELEMENT_INFO
Return the information about the device elements.

SMCIOC_INVENTORY
Return the information about the medium changer elements.

SMCIOC_AUDIT
Run an audit of the element status.

SMCIOC_AUDIT_RANGE
Run an audit for a particular range of elements.

SMCIOC_LOCK_DOOR
Lock and unlock the library access door.

SMCIOC_READ_ELEMENT_DEVIDS
Return the device ID element descriptors for drive elements.

SMCIOC_READ_CARTRIDGE_LOCATION
Returns the cartridge location information for all storage elements in the library.

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

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_EXCHANGE_MEDIUM

This command exchanges a cartridge from one element to another element. This command is equivalent to two **SCSI Move Medium** commands. The first moves the cartridge from the source element to the **destination1** element. 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 following data structure is filled out and supplied by the caller.

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

An example of the **SMCIOC_EXCHANGE_MEDIUM** command is

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

    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)dst1;
    exchange_medium.destination2 = (short)dst2;

    if (!(rc = ioctl (dev_fd, SMCIOC_EXCHANGE_MEDIUM, &exchange_medium))) {
       PRINTF ("SMCIOC_MOVE_MEDIUM succeeded.\n");
    }
    else {
       PERROR ("SMCIOC_EXCHANGE_MEDIUM failed");
       PRINTF ("\n");
       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 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");
   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 run 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_AUDIT_RANGE

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

```
#include <sys/smc.h>
  element_range_t elements;
  /*audit slots 32 to 36 */
 elements.element_address =32;
 elements.number_elements =5;
 if (!ioctl (dev_fd, SMCIOC_AUDIT_RANGE, &elements))
       printf ("The SMCIOC_AUDIT_RANGE ioctl succeeded \n");
 else
 {
       perror ("The SMCIOC_AUDIT_RANGE 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>
/*-------------------------------------------------------------------------*/
/* Name: smc_read_element_devids                                           */
/*------------------------------------------------------------------------ */
static int smc_read_element_devids(void)
   {
   int rc;
   int i,j;
   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;
  }
  j=0;
  elemp = elem_devid;
  for (i = 0; i < element_info.drive_count; i++, elemp++)
  {
  /* don't overflow screen if menu mode */
  if (interactive && j == 2)
  {
     j=0;
     printf ("\nHit to continue...");
     getchar();
  }
  j++;
  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");
    if (interactive)
       {
       printf ("\nHit to continue...");
       getchar();
       }
    }
    free(elem_devid);
    return errno;
}
```

SMCIOC_READ_CARTRIDGE_LOCATION

The **SMCIOC_READ_CARTRIDGE_LOCATION** IOCTL is used to return the cartridge location information for storage elements in the library. The **element_address** field specifies the starting element address to return and the **number_elements** field specifies how many storage elements are returned. The data field is a pointer to the buffer for return data. The buffer must be large enough for the number of elements that are returned. If the storage element contains a cartridge, then the **ASCII identifier** field in return data specifies the location of the cartridge.

Note: This IOCTL is supported only on the TS3500 (3584) library.

The 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  resvd2[3];                        /* reserved                */
    uchar  svalid:1,                         /* element address valid    */
               invert:1,                     /* medium inverted  */
                         :6;                 /* reserved        */
    ushort source;                           /* source storage elem addr */
    uchar  volume[36];                       /* primary volume tag      */
    uchar               :4,                  /* reserved        */
           code_set:4;                       /* code set        */
    uchar               :4,                  /* reserved        */
```

```
        ident_type:4;                      /* identifier type  */
    uchar  resvd3;                         /* reserved         */
    uchar  ident_len;                      /* identifier length     */
    uchar  identifier[24];                 /* slot identification   */
} cartridge_location_data_t;

typedef struct
{
    ushort element_address;                 /* starting element address */
    ushort number_elements;                 /* number of elements       */
    cartridge_location_data_t *data;        /* storage element pages    */
    char reserved[8];                       /* reserved /
} read_cartridge_location_t;
```

An example of the **SMCIOC_READ_CARTRIDGE_LOCATION** command is

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

   int rc;
   int available_slots=0;
   cartridge_location_data_t *slot_devid;
   read_cartridge_location_t slot_devids;

   slot_devids.element_address = (ushort)element_address;
   slot_devids.number_elements = (ushort)number_elements;

   if (rc = ioctl(dev_fd,SMCIOC_ELEMENT_INFO,&element_info))
       {
       PERROR("SMCIOC_ELEMENT_INFO failed");
       PRINTF("\n");
       scsi_request_sense();
       return (rc);

       }
     if (element_info.cell_count == 0)
       {
       printf("No slots found in element information...\n");
       errno = EIO;
       return errno;
       }
     if ((slot_devids.element_address==0) && (slot_devids.number_elements==0))
       {
       slot_devids.element_address=element_info.cell_address;
       slot_devids.number_elements=element_info.cell_count;
       printf("Reading all locations...\n");
       }
    if ((element_info.cell_address > slot_devids.element_address)
     || (slot_devids.element_address >
 (element_info.cell_address+element_info.cell_count-1)))
       {
       printf("Invalid slot address %d\n",element_address);
       errno = EINVAL;
       return errno;
       }

     available_slots = (element_info.cell_address+element_info.cell_count)
-slot_devids.element_address;
    if (available_slots>slot_devids.number_elements)
       available_slots=slot_devids.number_elements;

     slot_devid = malloc(element_info.cell_count
 * sizeof(cartridge_location_data_t));
    if (slot_devid == NULL
)        {
       errno = ENOMEM;
       return errno;
       }
```

```
    bzero((caddr_t)slot_devid,element_info.cell_count * sizeof
(cartridge_location_data_t));
    slot_devids.data = slot_devid;
    rc = ioctl (dev_fd, SMCIOC_READ_CARTRIDGE_LOCATION, &slot_devids);

    free(slot_devid);
    return rc;
```

SCSI tape drive IOCTL operations

A set of enhanced IOCTL commands gives applications access to extra features of IBM tape drives.

The following commands are supported.

STIOC_TAPE_OP
Perform a 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 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 messages on the tape drive console.

STIOC_SYNC_BUFFER
Flush the drive buffers to the tape.

STIOC_REPORT_DENSITY_SUPPORT
Return supported densities from the tape device.

GET_ENCRYPTION_STATE
This IOCTL can be used for application, system, and library-managed encryption. It allows a query only of the encryption status.

SET_ENCRYPTION_STATE
This IOCTL can be used only for application-managed encryption. It sets encryption state for application-managed encryption.

SET_DATA_KEY
This IOCTL can be used only for application-managed encryption. It sets the data key for application-managed encryption.

CREATE_PARTITION
Create one or more tape partitions and format the media.

QUERY_PARTITION
Query tape partitioning information and current active partition.

SET_ACTIVE_PARTITION
Set the current active tape partition.

ALLOW_DATA_OVERWRITE
Set the drive to allow a subsequent data overwrite type command at the current position or allow a **CREATE_PARTITION** IOCTL when data safe (append-only) mode is enabled.

READ_TAPE_POSITION
Read current tape position in either short, long, or extended form.

SET_TAPE_POSITION
Set the current tape position to either a logical object or logical file position.

QUERY_LOGICAL_BLOCK_PROTECTION
Query Logical Block Protection (LBP) support and its setup.

SET_LOGICAL_BLOCK_PROTECTION
Enable/disable Logical Block Protection (LBP), set the protection method, and how the protection information is transferred.

VERIFY_TAPE_DATA
Issues **VERIFY** command to cause data to be read from the tape and passed through the drive's error detection and correction hardware. This action determines whether it can be recovered from the tape. Or, whether the protection information is present and validates correctly on logical block on the medium.

QUERY_RAO_INFO
The IOCTL is used to query the maximum number and size of User Data Segments (UDS) that are supported from tape drive and driver for the wanted **uds_type**.

GENERATE_RAO
The IOCTL is called to send a **GRAO** list to request that the drive create a **Recommended Access Order** list.

RECEIVE_RAO
After a **GENERATE_RAO** IOCTL is completed, the application calls the **RECEIVE_RAO** IOCTL to receive a recommended access order of UDS from the drive.

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

STIOC_TAPE_OP

This command runs the standard tape drive operations. It is identical to the **MTIOCTOP** IOCTL command that is defined in the **/usr/include/sys/mtio.h** system header file. The **STIOC_TAPE_OP** and **MTIOCTOP** commands both use the same data structure that is 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 278.

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

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 run. A `Test Unit Ready` command is issued to the drive to retrieve status information.

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
: Space backward **mt_count** filemarks, then space backward before all data records in that tape file. For a specific **MTNBFS** operation with **mt_count = n**, the equivalent position can be achieved with **MT_BSF** and **MT_FSF**, as follows.

```
MTBSF with mt_count = n + 1
MTFSF with mt_count = 1
```

MTGRSZ
: Return the current record (block) size. The `mt_count` parameter contains the value.

MTSRSZ
: Set the working record (block) size to **mutant**.

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 that is 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 that is 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 279.

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 that are 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. It is identical to the **MTIOCGETDRIVETYPE** IOCTL command that is 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 that is 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 279.

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 occurs. The **STIOC_GET_POSITION** and **STIOC_SET_POSITION** commands can be used independently or 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 */
  uchar partition;              /* partition number */
  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. They are used for high-speed **locate** operations that are implemented by the tape drive. Only the IBM

3490E Magnetic Tape Subsystem or a virtual drive in a VTS supports the **PHYSICAL_BLK** type. All devices support the **LOGICAL_BLK** type.

The **block_type** is the only field that must be filled out by the caller. 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 quickly return 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 that remains in the buffer.

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

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

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 occurs. The **STIOC_GET_POSITION** and **STIOC_SET_POSITION** commands can be used independently or 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 */
  uchar partition;             /* partition number */
  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. They are used for high-speed **locate** operations that are implemented by the tape drive. Only the IBM 3490E Magnetic Tape Subsystem and the IBM Virtual Tape Servers support the **PHYSICAL_BLK** type. All 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 that is 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. The IBM 3490E Magnetic Tape Subsystem drives in VTSs do not support position to end of 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 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 that are indicated for the specific **type**.

The **type** field, which is filled out by the caller, is set to one of the following values.

BLOCKSIZE
: Block Size (0- [2 MB]/Maximum dma size).

  A value of zero indicates variable block size. Only the IBM 359x Tape System supports 2 MB maximum block size or maximum dma transfer size that is supported by the host adapter if it is larger than 2 MB. All other devices support 256 KB maximum block size.

COMPRESSION
: Compression Mode (0 or 1).

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

- NO_IMMEDIATE (0)

  If IMMEDIATE is set to zero, SCSI commands that support the immediate bit in the CDB run to completion before status is returned.
- GEN_IMMEDIATE (1)

  If IMMEDIATE is set to GEN_IMMEDIATE, the SCSI commands **Write FM**, **Locate**, **Load-Unload**, **Erase**, and **Rewind** return with status before the command actually completes on the tape drive.
- REW_IMMEDIATE (2)

  If IMMEDIATE is set to REW_IMMEDIATE, the **SCSI rewind** command returns with status before the command actually completes on the tape drive.

TRAILER
Trailer Label Mode (0 or 1).

This mode affects write behavior after logical end of medium (LEOM) is reached. See “Writing to a special file” on page 296 for information about write operations that approach LEOM. With trailer label processing disabled (TRAILER=0), writing past logical end of medium (LEOM) is not allowed. After LEOM is reached, all further writes fail, returning **-1**, with the *errno* system variable set to ENOSPC (no space that is left on device).

With trailer label processing enabled (TRAILER=1), writing past logical end of medium (LEOM) is allowed. After LEOM is reached, all subsequent writes succeed until physical end of medium (PEOM) is reached. Write requests for multiple fixed blocks can encounter short writes. See “Writing to a special file” on page 296 for information about short writes. After PEOM is reached, all further writes fail, returning **-1**, with the *errno* system variable set to ENOSPC (nospace that is left on device).

An application that uses the trailer label processing option must stop normal data writing when LEOM is reached, and run end of volume processing. Such processing typically consists of writing a final data record, a filemark, and a "trailing" tape label. Finally, two more filemarks are written to indicate 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. Setting the tape to this value resets the **PERSISTENT** and **ASSOCIATED** logical write protection modes. It does not reset the WORM logical or the PHYSICAL write protection modes.
- 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, and it is not alterable through device driver functions.

Note: Only IBM 359x and Magstar MP 3570 Tape Subsystem recognize the following values.

- WORM_PROTECT

  The tape is logically write protected in **WORM** mode. When the tape is protected in this mode, it is **permanently** write-protected. The only method to return the tape to a writable state is to format the cartridge, erasing all data.

- PERS_PROTECT

  The tape is logically write protected in **PERSISTENT** mode. A tape that is protected in this mode is write-protected for all uses (across mounts). This logical write protection mode can be reset by using the **NO_PROTECT** value.

- ASSC_PROTECT

  The tape is logically write protected in **ASSOCIATED** mode. A tape that is protected in this mode in write protected only while it is associated with a tape drive (mounted). When the tape is unloaded from the drive, the associated write protection is reset. This logical write protection mode can also be reset by using the **NO_PROTECT** value.

ACFMODE

Automatic Cartridge Facility mode.

This configuration parameter is read-only. ACF modes can be established only through the tape drive operator panel. The device driver can query only the ACF mode; it cannot change it. The ACFMODE parameter applies only to the IBM 3590 Tape System and the IBM Magstar MP 3570 Tape Subsystem. The following values are recognized:

- NO_ACF

  There is no ACF attached to the tape drive.

- SYSTEM_MODE

  The ACF is in the **system** mode. This mode allows explicit load and unloads to be issued through the device driver. An unload or offline command causes the tape drive to unload the cartridge and the ACF to replace the cartridge in its original magazine slot. A subsequent load command causes the ACF to load the cartridge from the next sequential magazine slot into the drive.

- RANDOM_MODE

  The ACF is in the **random** mode. This mode provides random access to all of the cartridges in the magazine. The ACF operates as a standard SCSI medium changer device.

- MANUAL_MODE

  The ACF is in the **manual** mode. This mode does not allow ACF control through the device driver. Cartridge load and unload operations can be run only through the tape drive operator panel. Cartridges are imported and exported through the priority slot.

- ACCUM_MODE

  The ACF is in the **accumulate** mode. This mode is similar to the manual mode. However, rather than cartridges that are exported through the priority slot, they are put away in the next available magazine slot.

- AUTO_MODE

  The ACF is in the **automatic** mode. This mode causes cartridges to be accessed sequentially under ACF control. When a tape finishes processing, it is put back in its magazine slot. Then, the next tape is loaded without an explicit unload and load command from the host.
- LIB_MODE

  The ACF is in the **library** mode. This mode is available only if the tape drive is installed in an automated tape library that supports the ACF (3495).

SCALING
Capacity Scaling.

This configuration parameter returns the capacity or logical length or the currently mounted tape. The `SCALING` parameter is not supported on the IBM 3490E Magnetic Tape Subsystem, nor in VTS drives. The following values are recognized.

- SCALE_100

  The current tape capacity is 100%.
- SCALE_75

  The current tape capacity is 75%.
- SCALE_50

  The current tape capacity is 50%.
- SCALE_25

  The current tape capacity is 25%.
- Other values (0x00 - 0xFF)

  For 3592 tape drive only.

SILI Suppress Illegal Length Indication.

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

DATASAFE
Data safe mode.

This parameter queries the current drive setting for data safe (append-only) mode. Or, on a set operation changes the current data safe mode setting on the drive. On a set operation a parameter value of zero sets the drive to normal (non-data safe) mode. A value of 1 sets the drive to data safe mode.

PEW_SIZE
Programmable early warning zone.

Using the tape parameter, the application is allowed to request the tape drive to create a zone that is called the programmable early warning zone (PEWZ) in the front of Early Warning (EW).

When a `WRITE` or `WRITE FILE MARK` (WFM) command writes a data or filemark upon reaching the PEWZ, a check condition status arises associated with a sense data with EOM and **PROGRAMMABLE EARLY WARNING DETECTED**. The WRITE or WFM commands in PEWZ is completed with a good status.

For the application developers, two methods are used to determine PEWZ when the *errno* is set to ENOSPC for **WRITE** or **WRITE FILE MARK** command, since ENOSPC is returned for either EW or PEW.

- Method 1: Issue the **Request Sense** IOCTL, check the sense key and ASC-ASCQ, and if it is `0x0/0x0007` (PROGRAMMABLE EARLY WARNING DETECTED), the tape is in PEW. If the sense key ASC-ASCQ is `0x0/0x0000` or `0x0/0x0002`, the tape is in EW.
- Method 2: **Call Read Position** IOCTL in long or extended form and check BPEW and EOP bits. If bpew = 1 and eop = 0, the tape is in PEW. If bpew = 1 and eop = 1, the tape is in EW.

The IBMtape driver requests the tape drive to save the mode page indefinitely. The PEW size is modified in the drive until a new setup is requested from the driver or application. The application must be programmed to issue the **Set** IOCTL to zero when PEW support is no longer needed, as the IBMtape drivers does not run this function. PEW is a setting of the drive and not tape. Therefore, it is the same on each partition, should partitions exist.

Encountering the PEWZ does not cause the device server to run a synchronize operation or terminate the command. It means that the data or filemark is written in the cartridge when a check condition with **PROGRAMMABLE EARLY WARNING DETECTED** is returned. But, IBMtape driver still returns the counter to less than zero (**-1**) for a write command or a failure for **Write FileMark** IOCTL call with ENOSPC error. In this way, it forces the application to use one of the methods to check PEW or EW. Once the application determines ENOSPC comes from PEW, it reads the requested write data or filemark that is written into the cartridge and reach or pass the PEW point. The application can issue a **Read position** IOCTL to validate the tape position.

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 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 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 revert to the defaults when the tape drive is closed and reopened.

Note: The **COMPRESSION**, **WRITEPROTECT**, **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 that is 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. The default write protect and scaling modes are established through the presently mounted cartridge.

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 that are indicated for the specific **type**.

The **type** field, which is filled out by the caller, can be set to one of the following values.

BLOCKSIZE
: Block Size (0-2097152 [2 MB]/Maximum dma size).

  A value of zero indicates variable block size. Only the IBM 359x Tape System supports 2 MB maximum block size or maximum dma transfer size that is supported by the host adapter if it is larger than 2 MB. All other devices support 256 KB maximum block size.

COMPRESSION
: Compression Mode (0 or 1).

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

  - NO_IMMEDIATE (0)

    If IMMEDIATE is set to zero, SCSI commands that support the immediate bit in the CDB run to completion before status is returned.
  - GEN_IMMEDIATE (1)

    If IMMEDIATE is set to **GEN_IMMEDIATE**, the SCSI commands **Write FM**, **Locate**, **Load-Unload**, **Erase**, and **Rewind** return with status before the command completes on the tape drive.
  - REW_IMMEDIATE (2)

    If IMMEDIATE is set to **REW_IMMEDIATE**, the **SCSI rewind** command returns with status before the command actually completes on the tape drive.

TRAILER
: Trailer Label Mode (0 or 1).

  This mode affects write behavior after logical end of medium (LEOM) is reached. See “Writing to a special file” on page 296 for information about write operations that approach LEOM. With trailer label processing disabled (TRAILER = 0), writing past logical end of medium (LEOM) is not allowed. After LEOM is reached, all further writes fail, returning **-1**, with the *errno* system variable set to ENOSPC (no space that is left on device). With trailer label processing enabled (TRAILER = 1), writing past logical end of medium (LEOM) is allowed. After LEOM is reached, all subsequent writes succeed until physical end of medium (PEOM) is reached. Write requests for multiple fixed blocks can encounter short writes. See “Writing to a special file” on page 296 for information about short writes. After PEOM is reached, all further writes fail, returning **-1**, with the *errno* system variable set to ENOSPC (no space that is left on device).

An application that uses the trailer label processing option stops normal data writing when LEOM is reached, and runs end of volume processing. Such processing typically consists of writing a final data record, a filemark, and a trailing tape label. Finally, two more filemarks are run to indicate end of data (EOD).

WRITEPROTECT
Write-Protect Mode.

This configuration parameter establishes the current write protection status of the mounted cartridge. The **WRITEPROTECT** parameter applies only to the IBM 359x Tape System and the IBM Magstar MP 3570 Tape Subsystem. 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. Setting the tape to this value resets the **PERSISTENT** and **ASSOCIATED** logical write protection modes. It does not reset the **WORM** logical or the **PHYSICAL** write protection modes.
- WORM_PROTECT

  The tape is logically write-protected in **WORM** mode. When the tape is protected in this mode, it is permanently write-protected. The only method to return the tape to a writable state is to format the cartridge, erasing all data.
- PERS_PROTECT

  The tape is logically write-protected in **PERSISTENT** mode. A tape that is protected in this mode is write protected for all uses (across mounts). This logical write protection mode can be reset by using the **NO_PROTECT** value.
- ASSC_PROTECT

  The tape is logically write-protected in **ASSOCIATED** mode. A tape that is protected in this mode in write-protected only while it is associated with a tape drive (mounted). When the tape is unloaded from the drive, the associated write protection is reset. This logical write protection mode can also be reset by using the **NO_PROTECT** value.
- PHYS_PROTECT

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

ACFMODE
Automatic Cartridge Facility Mode.

This configuration parameter is read-only. ACF modes can be established only through the tape drive operator panel. This type value is not supported by the **STIOC_SET_PARM** IOCTL.

SCALING
Capacity Scaling.

This configuration parameter sets the capacity or logical length or the currently mounted tape. The tape must be at BOT to change this value. Changing the scaling value destroys all existing data on the tape. The SCALING parameter is not supported on the IBM 3490E Magnetic Tape Subsystem or VTS drives. The following values are recognized.

- SCALE_100

  Sets the tape capacity to 100%.
- SCALE_75

  Sets the tape capacity to 75%.
- SCALE_50

  Sets the tape capacity to 50%.
- SCALE_25

  Sets the tape capacity to 25%.
- Other values (0x00 - 0xFF)

  For 3592 tape drive only.

SILI Suppress Illegal Length Indication.

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

DATASAFE
Data safe mode.

This parameter queries the current drive setting for data safe (append-only) mode. Or,on a set operation it changes the current data safe mode setting on the drive. On a set operation, a parameter value of zero sets the drive to normal (non-data safe) mode. A value of *1* sets the drive to data safe mode.

PEW_SIZE
Programmable early warning zone.

Using the tape parameter, the application is allowed to request the tape drive to create a zone that is called the programmable early warning zone (PEWZ) in the front of Early Warning (EW).

When a **WRITE** or **WRITE FILE MARK** (WFM) command writes a data or filemark upon reaching the PEWZ, a check condition status arises associated with a sense data with EOM and PROGRAMMABLE EARLY WARNING DETECTED. The **WRITE** or **WRITE FILE MARK** commands in PEWZ are completed with a good status.

For the application developers.

1. Two methods are used to determine PEWZ when the *errno* is set to ENOSPC for **WRITE** or **WRITE FILE MARK** command, since ENOSPC is returned for either EW or PEW.
   - Method 1: Issue the **Request Sense** IOCTL, check the sense key and ASC-ASCQ, and if it is `0x0/0x0007` (PROGRAMMABLE EARLY

WARNING DETECTED), the tape is in PEW. If the sense key ASC-ASCQ is 0x0/0x0000 or 0x0/0x0002, the tape is in EW.

- Method 2: Call **Read Position** IOCTL in long or extended form and check bpew and eop bits. If bpew = 1 and eop = 0, the tape is in PEW. If bpew = 1 and eop = 1, the tape is in EW.

The IBMtape driver requests the tape drive to save the mode page indefinitely. The PEW size is modified in the drive until a new setup is requested from the driver or application. The application must be programmed to issue the **Set** IOCTL to zero when PEW support is no longer needed. The IBMtape drivers do not run this function. PEW is a setting of the drive and not tape. Therefore, it is the same on each partition, should partitions exist.

2. Encountering the PEWZ does not cause the device server to run a synchronize operation or terminate the command. It means that the data or filemark is written in the cartridge when a check condition with **PROGRAMMABLE EARLY WARNING DETECTED** is returned. But, the IBMtape driver still returns the counter to less than zero (**-1**) for a write command or a failure for **Write FileMark** IOCTL call with ENOSPC error. In this way, it forces the application to use one of the previous methods to check PEW or EW. Once the application determines ENOSPC comes from PEW, it reads the requested write data or filemark that is written into the cartridge and reaches or passes the PEW point. The application can issue a **Read position** IOCTL to validate the tape position.

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.

The message that is sent by using this call does not always remain on the display. It depends on the current drive activity.

Note: All messages must be padded to MSGLEN bytes (8). Otherwise, garbage characters (meaningless data) can be displayed in the message.

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

```
typedef struct {
  uchar function;                   /* message function code */
  char msg_0[MSGLEN];               /* message 0 */
  char msg_1[MSGLEN];               /* message 1 */
} msg_data_t;
```

The function field, which is filled out by the caller, is set by combining (by using logical OR) a **Message Type** flag and a **Message Control Flag**.

Message Type Flags

GENSTATUS (General Status Message)
: Message 0, Message 1, or both are displayed according to the **Message Control** flag, until the drive next initiates tape motion or the message is updated with a new message.

DMNTVERIFY (Demount/Verify Message)
: Message 0, Message 1, or both are displayed according to the **Message Control** flag, until the current volume is unloaded. If the volume is unloaded, the message display is not changed and the command completes no operation.

MNTIMMED (Mount with Immediate Action Indicator)
: Message 0, Message 1, or both are displayed according to the **Message Control** flag, until the volume is loaded. An attention indicator is activated. If the volume is loaded, the message display is not changed and the command completes no operation.

DMNTIMMED (Demount/Mount with Immediate Action Indicator)
: When the **Message Control** flag is set to a value of ALTERNATE, Message 0 and Message 1 are displayed alternately until the currently mounted volume, if any, is unloaded. When the **Message Control** flag is set to any other value, Message 0 is displayed until the currently mounted volume, if any, is unloaded. Message 1 is displayed from the time the volume is unloaded (or immediately, if the volume is already unloaded) until another volume is loaded. An attention indicator is activated.

Message Control Flags

DISPMSG0
: Display message 0.

DISPMSG1
: Display message 1.

FLASHMSG0
: Flash message 0.

FLASHMSG1
: Flash message 1.

ALTERNATE
: Alternate flashing message 0 and message 1.

An example of the **STIOC_DISPLAY_MSG** command is

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

msg_data_t msg_data;
msg_data.function = GENSTATUS | ALTERNATE;
memcpy (msg_data.msg_0, "Hello   ", 8);
memcpy (msg_data.msg_1, "World!!!", 8);

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

```
else {
  perror ("The STIOC_DISPLAY_MSG 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 ();
}
```

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 that are used with this IOCTL are

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

Examples of the **STIOC_REPORT_DENSITY_SUPPORT** command are

```
/*-------------------------------------------------------------------------*/
/*        Name: st_report_density_support                                  */
/*    Synopsis: Report the supported densities for the device.             */
```

```
/*     Returns: Error code from /usr/include/sys/errno.h.                   */
/*------------------------------------------------------------------------*/
static int st_report_density_support ()
{
        int rc;
        int i;
        rpt_dens_sup_t density;

        int bits_per_mm = 0;
        int media_width = 0;
        int tracks = 0;
        int capacity = 0;

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

        density.media = ALL_MEDIA_DENSITY;
        density.number_reports = 0;

        if (!(rc = ioctl (dev_fd, STIOC_REPORT_DENSITY_SUPPORT, &density))) {
             printf ("STIOC_REPORT_DENSITY_SUPPORT succeeded.\n");
             printf("Total densities reported: %d\n",density.number_reports);
        }
        else {
                perror ("STIOC_REPORT_DENSITY_SUPPORT failed");
                printf ("\n");
                scsi_request_sense ();
        }

        for (i = 0; i < density.number_reports; i++)
        {
                bits_per_mm = (int)density.reports[i].bits_per_mm[0] << 16;
                bits_per_mm |= (int)density.reports[i].bits_per_mm[1] << 8;
                bits_per_mm |= (int)density.reports[i].bits_per_mm[2];

                media_width |= density.reports[i].media_width[0] << 8;
                media_width |= density.reports[i].media_width[1];

                tracks |= density.reports[i].tracks[0] << 8;
                tracks |= density.reports[i].tracks[1];

                capacity = density.reports[i].capacity[0] << 24;
                capacity |= density.reports[i].capacity[1] << 16;
                capacity |= density.reports[i].capacity[2] << 8;
                capacity |= density.reports[i].capacity[3];

                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",bits_per_mm);
        printf("  Media Width.................. %d\n",media_width);
        printf("  Tracks....................... %d\n",tracks);
        printf("  Capacity (megabytes)......... %d\n",capacity);


        if (interactive) {
                printf ("\nHit <ENTER> to continue...");
                getchar ();
        }

} /* end for all media density*/

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

density.media = CURRENT_MEDIA_DENSITY;
density.number_reports = 0;

if (!(rc = ioctl (dev_fd, STIOC_REPORT_DENSITY_SUPPORT, &density))) {
        printf ("STIOC_REPORT_DENSITY_SUPPORT succeeded.\n");
        printf("Total number of densities reported: %d\n",
                 density.number_reports);
}
else {
        perror ("STIOC_REPORT_DENSITY_SUPPORT failed");
        printf ("\n");
        scsi_request_sense ();
}

for (i = 0; i < density.number_reports; i++)
{

        bits_per_mm = density.reports[i].bits_per_mm[0] << 16;
        bits_per_mm |= density.reports[i].bits_per_mm[1] << 8;
        bits_per_mm |= density.reports[i].bits_per_mm[2];

        media_width |= density.reports[i].media_width[0] << 8;
        media_width |= density.reports[i].media_width[1];

        tracks |= density.reports[i].tracks[0] << 8;
        tracks |= density.reports[i].tracks[1];

        capacity = density.reports[i].capacity[0] << 24;
        capacity |= density.reports[i].capacity[1] << 16;
        capacity |= density.reports[i].capacity[2] << 8;
        capacity |= density.reports[i].capacity[3];

        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",bits_per_mm);
            printf("  Media Width................. %d\n",media_width);
            printf("  Tracks...................... %d\n",tracks);
            printf("  Capacity (megabytes)......... %d\n",capacity);

            if (interactive) {
                printf ("\nHit <ENTER> to continue...");
                getchar ();
            }
        }


        return (rc);
}
```

STIOC_GET_DENSITY

STIOC_GET_DENSITY is used to query the current write density format settings on the tape drive for 3592 E05 or later model drive only.

The **STIOC_GET_POSITION** and **STIOC_SET_POSITION** commands can be used independently or with each other.

Following is the structure for the **STIOC_GET_DENSITY** and **STIOC_SET_DENSITY** IOCTLs.

```
struct density_data_t
{
    char  density_code;       /* mode sense header density code           */
    char  default_density;    /* default write density                    */
    char  pending_density;    /* pending write density                    */
    char  reserved[9];
};
```

The **density_code** field returns the current density of the tape that is loaded in the tape drive from the block descriptor of Mode sense. The **default_density** field returns the default write density in Mode sense (**Read/Write Control**). The **pending_density** field returns the pending write density in Mode sense (**Read/Write Control**). An example of the **STIOC_SET_DENSITY** command is

```
#include <sys/st.h>
density_data_t density_data;

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

STIOC_SET_DENSITY

STIOC_SET_DENSITY is used to set a new write density format on the tape drive by using the default and pending density fields in 3592 E05 or later model drive only. For example, this command is used if the user wants to write the data to the tape in 3592 J1A format (0x51) in 3592 E05 drive, not in the default 3592 E05 format (0x52). The application can specify a new write density for the current loaded tape only or as a default for all tapes. Refer to the examples.

The **STIOC_GET_POSITION** and **STIOC_SET_POSITION** commands can be used independently or with each other. The application gets the current density settings first before the current settings are modified. If the application specifies a new density for the current loaded tape only, then the application must issue another set density IOCTL after the current tape is unloaded and the next tape is loaded to either the default maximum density or a new density to ensure the tape drive uses the correct density. If the application specifies a new default density for all tapes, the setting remains in effect until changed by another set density IOCTL or the tape drive is closed by the application.

Following is the structure for the **STIOC_GET_DENSITY** and **STIOC_SET_DENSITY** IOCTLs.

```
struct density_data_t
{
    char  density_code;       /* mode sense header density code     */
    char  default_density;    /* default write density              */
    char  pending_density;    /* pending write density              */
    char  reserved[9];
};
```

Note:

1. These IOCTLs are supported only on tape drives that can write multiple density formats. Refer to the hardware reference for the specific tape drive to determine whether multiple write densities are supported. If the tape drive does not support these IOCTLs, *errno* EINVAL is returned.
2. The device driver always sets the default maximum write density for the tape drive on every open system call. Any previous **STIOC_SET_DENSITY** IOCTL values from the last open are not used.
3. If the tape drive detects an invalid density code or cannot complete the operation on the **STIOC_SET_DENSITY** IOCTL, the *errno* is returned and the current drive density settings before the IOCTL is restored.
4. The struct **density_data_t** defined in the header file of **st.h** is used for both IOCTLs. The **density_code** field is not used and ignored on the **STIOC_SET_DENSITY** IOCTL.
5. A new write density is allowed only when positioned at BOP (logical block 0), and is ignored at any other location in the tape drive. The new density is applied on the next write-type operation (Write, Write Filemarks (>0), Erase, Format Medium, and so on). It is not reported in the **STIOC_GET_DENSITY** IOCTL **density_code** field before the format is completed.

Here are study cases on how to set the default write density and pending write density for a new write density before the IOCTL is issued.

```
struct density_data_t density_data;
```

Case 1: Set 3592 J1A density format for current loaded tape only.

```
density_data.default_density = 0x7F;
density_data.pending_density = 0x51;
```

Case 2: Set 3592 E05 density format for current loaded tape only.

```
density_data.default_density = 0x7F;
density_data.pending_density = 0x52;
```

Case 3: Set default maximum density for current loaded tape.

```
density_data.default_density = 0;
density_data.pending_density = 0;
```

Case 4: Set 3592 J1A density format for current loaded tape and all subsequent tapes.

```
density_data.default_density = 0x51;
density_data.pending_density = 0x51;
```

An example of the **STIOC_SET_DENSITY** command is

```
#include <sys/st.h>
density_data_t density_data;

/* set 3592 J1A density format  (0x51) for current loaded tape only */
density_data.default_density = 0x7F;
density_data.pending_density = 0x51;

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

GET_ENCRYPTION_STATE

This IOCTL command queries the drive's encryption method and state.

The data structure that is used for this IOCTL is as follows on all of the supported operating systems.

```
struct encryption_status {
    uchar encryption_capable;        /* Set this field as a boolean based on the
                                 capability of the drive */
    uchar encryption_method;         /* Set this field to one of the
                                 defines below */
       #define METHOD_NONE              0    /* Only used in
                                         GET_ENCRYPTION_STATE */
       #define METHOD_LIBRARY           1    /* Only used in
                                         GET_ENCRYPTION_STATE */
       #define METHOD_SYSTEM            2    /* Only used in
                                         GET_ENCRYPTION_STATE */
       #define METHOD_APPLICATION       3    /* Only used in
                                         GET_ENCRYPTION_STATE */
       #define METHOD_CUSTOM            4    /* Only used in
                                         GET_ENCRYPTION_STATE */
       #define METHOD_UNKNOWN           5    /* Only used in
                                         GET_ENCRYPTION_STATE */
    uchar encryption_state;              /* Set this field to one of the
                                       defines below */
       #define STATE_OFF                0    /* Used in GET/SET_ENCRYPTION_STATE */
```

```
      #define STATE_ON                 1    /* Used in GET/SET_ENCRYPTION_STATE */
      #define STATE_NA                 2    /* Used in GET_ENCRYPTION_STATE */
   uchar reserved[13];
 };
```

An example of the **GET_ENCRYPTION_STATE** command is

```
int qry_encryption_state (void) {
   int rc = 0;
   struct encryption_status encryption_status_t;

   printf("issuing query encryption status...\n");
   memset(&encryption_status_t, 0, sizeof(struct encryption_status));
   rc = ioctl (fd, GET_ENCRYPTION_STATE, &encryption_status_t);

   if(rc == 0) {
     if(encryption_status_t.encryption_capable)
        printf("encryption capable......Yes\n");
   else
       printf("encryption capable......No\n");
   switch(encryption_status_t.encryption_method) {
        case METHOD_NONE:
            printf("encryption method.......METHOD_NONE\n");
            break;
       case METHOD_LIBRARY:
            printf("encryption method.......METHOD_LIBRARY\n");
            break;
       case METHOD_SYSTEM:
            printf("encryption method.......METHOD_SYSTEM\n");
            break;
       case METHOD_APPLICATION:
            printf("encryption method.......METHOD_APPLICATION\n");
            break;
       case METHOD_CUSTOM:
            printf("encryption method.......METHOD_CUSTOM\n");
           break;
       case METHOD_UNKNOWN:
            printf("encryption method.......METHOD_UNKNOWN\n");
           break;
       default:
           printf("encryption method.......Error\n");
    }


    switch(encryption_status_t.encryption_state) {
        case STATE_OFF:
            printf("encryption state........OFF\n");
            break;
        case STATE_ON:
            printf("encryption state........ON\n");
            break;
        case STATE_NA:
            printf("encryption state........NA\n");
           break;
       default:
            printf("encryption state......Error\n");
      }
  }
return rc;
}
```

SET_ENCRYPTION_STATE

This IOCTL command allows setting the encryption state only for application-managed encryption. On unload, some of the drive settings might be reset to default. To set the encryption state, the application issues this IOCTL after a tape is loaded and at BOP.

The data structure that is used for this IOCTL is the same as the one for **GET_ENCRYPTION_STATE**.

An example of the **SET_ENCRYPTION_STATE** command is

```
int set_encryption_status(int option) {
  int rc = 0;
  struct encryption_status encryption_status_t;

  printf("issuing query encryption status...\n");
  memset(&encryption_status_t, 0, sizeof(struct encryption_status));
  rc = ioctl(fd, GET_ENCRYPTION_STATE, &encryption_status_t);
  if(rc < 0) return rc;
  if(option == 0)
     encryption_status_t.encryption_state = STATE_OFF;
  else if(option == 1)
     encryption_status_t.encryption_state = STATE_ON;
  else {
     printf("Invalid parameter.\n");
     return (EINVAL);
  }

  printf("Issuing set encryption status......\n");
  rc = ioctl(fd, SET_ENCRYPTION_STATE, &encryption_status_t);

  return rc;
}
```

SET_DATA_KEY

This IOCTL command allows setting the data key only for application-managed encryption.

The data structure that is used for this IOCTL is as follows on all of the supported operating systems.

```
struct data_key {
  uchar data_key_index[12];       /* The DKi */
  uchar data_key_index_length;    /* The DKi length */
  uchar reserved1[15];
  uchar data_key[32];             /* The DK  */
  uchar reserved2[48];
};
```

An example of the **SET_DATA_KEY** command is

```
int set_datakey(void) {
  int rc = 0;
  struct data_key encryption_data_key_t;

  printf("Issuing set encryption data key......\n");
  memset(&encryption_status_t, 0, sizeof(struct data_key));

  /* fill in your data key here, then issue the following ioctl*/
  rc = ioctl(fd, SET_DATA_KEY, &encryption_status_t);
  return rc;
}
```

QUERY_PARTITION

The **QUERY_PARTITION** IOCTL is used to return partition information for the tape drive and the current media in the tape drive, including the current active partition the tape drive is using for the media. The **number_of partitions** field is the current number of partitions on the media and the **max_partitions** is the maximum partitions that the tape drive supports. The **size_unit** field can be either one of the defined values or another value such as 8. It is used with the **size array** field value for each partition to specify the actual size partition sizes. The **partition_method** field is either Wrap-wise Partitioning or Longitudinal Partitioning. Refer to "CREATE_PARTITION" on page 268 for details.

The data structure that is used with this IOCTL is

```
The define for "partition_method":
#define UNKNOWN_TYPE                       0 /* vendor-specific or unknown         */
#define WRAP_WISE_PARTITION                1 /* Wrap-wise Partitioning without RABF */
#define LONGITUDINAL_PARTITION             2 /* Longitudinal Partitioning          */
#define WRAP_WISE_PARTITION_WITH_FASTSYNC 3 /* Wrap-wise Partitioning with RABF    */

The define for "size_unit":
define SIZE_UNIT_BYTES       0    /* Bytes            */
#define SIZE_UNIT_KBYTES     3    /* Kilobytes        */
#define SIZE_UNIT_MBYTES     6    /* Megabytes        */
#define SIZE_UNIT_GBYTES     9    /* Gigabytes        */
#define SIZE_UNIT_TBYTES    12    /* Terabytes        */

struct query_partition {
  uchar max_partitions;          /* Max number of supported partitions      */
  uchar active_partition;        /* current active partition on tape        */
  uchar number_of_partitions;    /* Number of partitions from 1 to max      */
  uchar size_unit;               /* Size unit of partition sizes below      */
  ushort size[MAX_PARTITIONS];   /* Array of partition sizes in size units  */
                                 /* for each partition, 0 to (number - 1)   */
  uchar partition_method;        /* partitioning type                       */
char reserved [31];
};
```

Example of the **QUERY_PARTITION** IOCTL

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

int rc,i;
struct query_partition q_partition;

memset((char *)&q_partition, 0, sizeof(struct query_partition));
rc = ioctl(dev_fd, QUERY_PARTITION, &q_partition);
if(!rc)
{
    printf("QUERY PARTITION ioctl succeed\n");
    printf(" Partition Method = %d\n",q_partition.partition_method);
    printf("Max partitions = %d\n",q_partition.max_partitions);
    printf("Number of partitions = %d\n",q_partition.number_of_partitions);
    for(i=0;i<q partition.number of partitions;i++)
   {
      printf("Size of Partition # %d = %d ",i,q_partition.size[i]);
      switch(q_partition.size_unit)
      {
         case SIZE_UNIT_BYTES:
            printf(" Bytes\n");
         break;
         case SIZE_UNIT_KBYTES:
            printf(" KBytes\n");
         break;
         case SIZE_UNIT_MBYTES:
```

```
            printf(" MBytes\n");
         break;
         case SIZE_UNIT_GBYTES:
            printf(" GBytes\n");
         break;
         case SIZE_UNIT_TBYTES:
            printf(" TBytes\n");
         break;
         default:
            printf("Size unit 0x%d\n",q_partition.size_unit);
      }
   }
   printf("Current active partition = %d\n",q_partition.active_partition);
} else {
   printf("QUERY PARTITION ioctl failed\n");
}

return rc;
```

CREATE_PARTITION

The **CREATE_PARTITION** IOCTL is used to format the current media in the tape drive into 1 or more partitions. The number of partitions to create is specified in the **number_of_partitions** field. When more than one partition is created, the **type** field specifies the type of partitioning, either **FDP**, **SDP**, or **IDP**. The tape must be positioned at the beginning of tape (partition 0 logical block id 0) before this IOCTL is used.

If the **number_of_partitions** field to create in the IOCTL structure is one partition, all other fields are ignored and not used. The tape drive formats the media by using its default partitioning type and size for a single partition.

When the **type** field in the IOCTL structure is set to either **FDP** or **SDP**, the **size_unit** and **size** fields in the IOCTL structure are not used. When the type field in the IOCTL structure is set to IDP, the **size_unit** with the size fields are used to specify the size for each partition.

There are two partition types: Wrap-wise Partitioning (Figure 10 on page 269) optimized for streaming performance, and Longitudinal Partitioning (Figure 11 on page 269) optimized for random access performance. Media is always partitioned into 1 by default or more than one partition where the data partition always exists as partition 0 and other extra index partition 1 to n might exist.

A WORM media cannot be partitioned and the **Format Medium** commands are rejected. Attempts to scale a partitioned media is accepted. However, only if you use the correct **FORMAT** field setting, as part of scaling the volume is set to a single data partition cartridge.

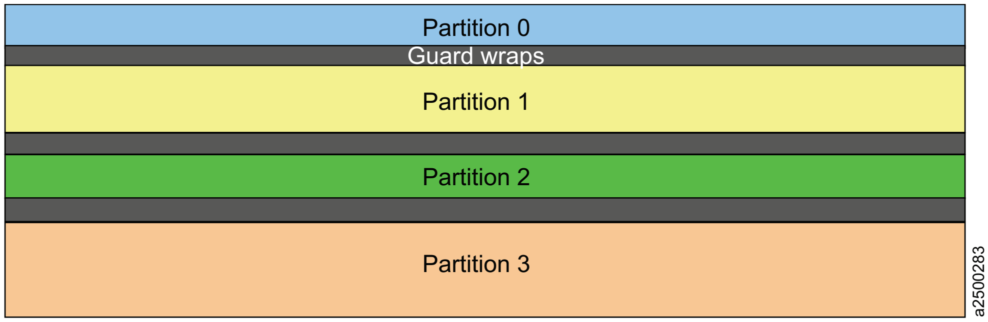

Figure 10. Wrap-wise partitioning

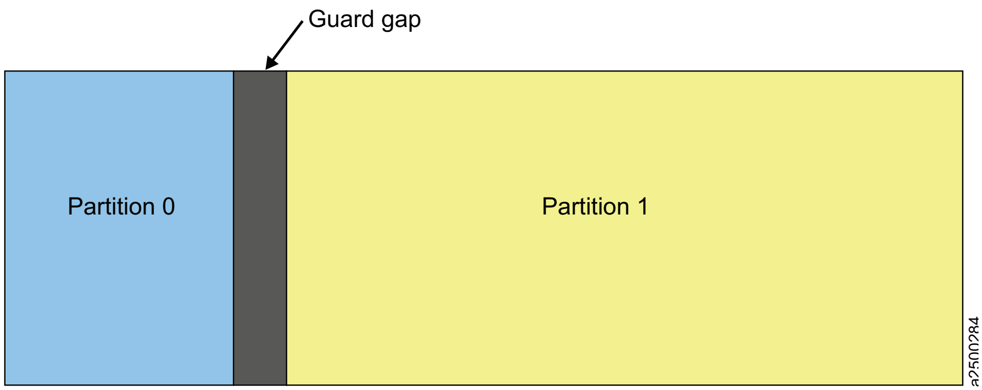

Figure 11. Longitudinal partitioning

The following chart lists the maximum number of partitions that the tape drive supports.

Table 5. Number of supported partitions

Drive type	Maximum number of supported partitions
LTO 5 (TS2250 and TS2350) and later	2 in Wrap-wise Partitioning
3592 E07 (TS 1140)	4 in Wrap-wise Partitioning 2 in Longitudinal Partitioning

The data structure that is used with this IOCTL is

```
The define for "partition_method":
#define UNKNOWN_TYPE                      0 /* vendor-specific or unknown         */
#define WRAP_WISE_PARTITION               1 /* Wrap-wise Partitioning without RABF */
#define LONGITUDINAL_PARTITION            2 /* Longitudinal Partitioning          */
#define WRAP_WISE_PARTITION_WITH_FASTSYNC 3 /* Wrap-wise Partitioning with RABF    */

The define for "type":
#define IDP_PARTITION        1      /* Initiator Defined Partition type   */
#define SDP_PARTITION        2      /* Select Data Partition type         */
#define FDP_PARTITION        3      /* Fixed Data Partition type          */

The define for "size_unit":
#define SIZE_UNIT_BYTES      0      /* Bytes                              */
#define SIZE_UNIT_KBYTES     3      /* Kilobytes                          */
#define SIZE_UNIT_MBYTES     6      /* Megabytes                          */
#define SIZE_UNIT_GBYTES     9      /* Gigabytes                          */
#define SIZE_UNIT_TBYTES    12      /* Terabytes                          */
```

```
struct tape_partition {
  uchar type;                    /* Type of tape partition to create      */
  uchar number_of_partitions;    /* Number of partitions to create        */
  uchar size_unit;               /* IDP size unit of partition sizes below */
  ushort size[MAX_PARTITIONS];   /* Array of partition sizes in size units */
                                 /* for each partition, 0 to (number - 1)  */
  uchar partition_method;        /* partitioning type                     */
  char reserved [31];
  };
```

Examples of the **CREATE_PARTITION** IOCTL.

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

  struct tape_partition partition;

   /* create 2 SDP partitions for LTO-5 */
  partition.type = SDP_PARTITION;
  partition.number_of_partitions = 2;
  partition.partition_method = WRAP_WISE_PARTITION;
  ioctl(dev_fd, CREATE_PARTITION, &partition);

  /* create 2 IDP partitions with partition 1 for 37 gigabytes and
  partition 0 for the remaining capacity on LTO-5*/
  partition.type = IDP_PARTITION;
  partition.number_of_partitions = 2;
  partition.partition_method = WRAP_WISE_PARTITION;
  partition.size_unit = SIZE_UNIT_GBYTES;
  partition.size[0] = 0xFFFF;
  partition.size[1] = 37;
  ioctl(dev_fd, CREATE_PARTITION, &partition);

  /* format the tape into 1 partition */
   partition.number_of_partitions = 1;
  ioctl(dev_fd, CREATE_PARTITION, &partition);

  /* create 4 IDP partitions on 3592 JC volume in Wrap-wise partitioning with
  partition 0 and 2 for 94.11 gigabytes (minimum size) and partition 1 and 3 to use
  the remaining capacity equally around 1.5 TB on 3592 E07 */
  partition.type = IDP_PARTITION;
  partition.number_of_partitions = 4;
  partition.partition_method = WRAP_WISE_PARTITION;
  partition.size_unit = 8;      /* 100 megabytes */
  partition.size[0] = 0x03AD;
  partition.size[1] = 0xFFFF;
  partition.size[2] = 0x03AD;
  partition.size[3] = 0x3AD2;
  ioctl(dev_fd, CREATE_PARTITION, &partition);
```

SET_ACTIVE_PARTITION

The **SET_ACTIVE_PARTITION** IOCTL is used to position the tape to a specific partition. This partition becomes the current active partition for subsequent commands and a specific logical block id in the partition. To position to the beginning of the partition, the **logical_block_id** field must be set to 0.

The data structure that is used with this IOCTL is

```
struct set_active_partition {
  uchar partition_number;        /* Partition number 0-n to change to     */
  ullong logical_block_id;       /* Blockid to locate to within partition */
  char reserved[32];
  };
```

Examples of the **SET_ACTIVE_PARTITION** IOCTL.

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

struct set_active_partition partition;

/* position the tape to partition 1 and logical block id 12 */
partition.partition_number = 1;
partition.logical_block_id = 12;
ioctl(dev_fd, SET_ACTIVE_PARTITION, &partition);

/* position the tape to the beginning of partition 0 */
 partition.partition_number = 0;
 partition.logical_block_id = 0;
 ioctl(dev_fd, SET_ACTIVE_PARTITION, &partition);
```

ALLOW_DATA_OVERWRITE

The **ALLOW_DATA_OVERWRITE** IOCTL is used to set the drive to allow a subsequent data write type command at the current position. Or, allow a **CREATE_PARTITION** IOCTL when data safe (append-only) mode is enabled.

For a subsequent write type command, the **allow_format_overwrite** field must be set to 0. The **partition_number** and **logical_block_id** fields must be set to the current partition and position within the partition where the overwrite occurs.

For a subsequent **CREATE_PARTITION** IOCTL, the **allow_format_overwrite** field must be set to 1. The **partition_number** and **logical_block_id** fields are not used. However, the tape must be at the beginning of tape (partition 0 logical block id 0) before the **CREATE_PARTITION** IOCTL is issued.

The data structure that is used with this IOCTL is

```
struct allow_data_overwrite{
  uchar partition_number;         /* Partition number 0-n to overwrite       */
  ullong logical_block_id;        /* Blockid to overwrite to within partition */
  uchar allow_format_overwrite;   /* allow format if in data safe mode       */
  char reserved[32];
  };
```

Examples of the **ALLOW_DATA_OVERWRITE** IOCTL.

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

struct read_tape_position rpos;
struct allow_data_overwrite data_overwrite;
struct set_active_partition partition;

/* get current tape position for a subsequent write type command and */
rpos.data_format = RP_LONG_FORM;
if (ioctl (dev_fd, READ_TAPE_POSITION, &rpos) <0)
   retun errno;

  /* set the allow_data_overwrite fields with the current position
 for the next write type command */
 data_overwrite.partition_number = rpos.rp_data.rp_long.active_partition;
 data_overwrite.logical_block_id = rpos.rp_data.rp_long.logical_obj_number;
 data_overwrite.allow_format_overwrite = 0;
  ioctl (dev_fd, ALLOW_DATA_OVERWRITE, &data_overwrite);

 /* set the tape position to the beginning of tape and */
 /* prepare a format overwrite for the CREATE_PARTITION ioctl */
 partition.partition_number = 0;
 partition.logical_block_id = 0;
 if (ioctl(dev_fd, SET_ACTIVE_PARTITION, &partition;) <0)
  return errno;
```

```
data_overwrite.allow_format_overwrite = 1;
ioctl (dev_fd, ALLOW_DATA_OVERWRITE, &data_overwrite);
```

READ_TAPE_POSITION

The **READ_TAPE_POSITION** IOCTL is used to return **Read Position** command data in either the short, long, or extended form. The type of data to return is specified by setting the **data_format** field to either **RP_SHORT_FORM**, **RP_LONG_FORM**, or **RP_EXTENDED_FORM**.

The data structures that are used with this IOCTL are

```
#define RP_SHORT_FORM              0x00
#define RP_LONG_FORM               0x06
#define RP_EXTENDED_FORM           0x08

struct short_data_format {
  uchar bop:1,                 /* beginning of partition           */
       eop:1,                  /* end of partition                 */
       locu:1,                 /* 1 means num_buffer_logical_obj field
                                  is unknown                       */
       bycu:1,                 /* 1 means the num_buffer_bytes field
                                  is unknown                       */
       rsvd :1,
       lolu:1,                 /* 1 means the first and last logical obj
                                  position fields are unknown      */
       perr: 1,                /* 1 means the position fields have overflowed
                                  and can not be reported          */
       bpew :1;                /* beyond programmable early warning */
  uchar active_partition;      /* current active partition         */
  char reserved[2];
  uint first_logical_obj_position;/* current logical object position */
  uint last_logical_obj_position; /* next logical object to be transferred
                                     to tape                       */
  uint num_buffer_logical_obj;    /* number of logical objects
                                     in buffer                     */
  uint num_buffer_bytes;          /* number of bytes in buffer      */
  char reserved1;
  };

struct long_data_format {
  uchar bop:1,                    /* beginning of partition         */
        eop:1,                    /* end of partition               */
        rsvd1:2,
        mpu:1,                    /* 1 means the logical file id field
                                     is unknown                    */
        lonu:1,                   /* 1 means either the partition number
                                     or logical obj number field are
                                     unknown                       */
        rsvd2:1,
        bpew :1;                  /* beyond programmable early
                                     warning                       */
  char reserved[6];
  uchar active_partition;         /* current active partition       */
  ullong logical_obj_number;      /* current logical object position */
  ullong logical_file_id;         /* number of filemarks from bop and
                                     current logical position      */
  ullong obsolete;
  };

struct extended_data_format {
  uchar bop:1,                   /* beginning of partition          */
        eop:1,                   /* end of partition                */
        locu:1,                  /* 1 means num_buffer_logical_obj field
                                    is unknown                      */
```

```
      bycu:1,                  /* 1 means the num_buffer_bytes field
                                  is unknown                      */
      rsvd :1,
      lolu:1,                  /* 1 means the first and last logical obj
                                  position fields are unknown      */
      perr: 1,                 /* 1 means the position fields have overflowed
                                  and can not be reported          */
      bpew :1;                 /* beyond programmable early warning*/
  uchar active_partition;      /* current active partition         */
  ushort additional_length;
  uint num_buffer_logical_obj;    /* number of logical objects in buffer */
  ullong first_logical_obj_position;/* current logical object position   */
  ullong last_logical_obj_position; /* next logical object to be transferred
                                     to tape                           */
  ullong num_buffer_bytes;        /* number of bytes in buffer          */
  char reserved;
  };

struct read_tape_position{
  uchar data_format; /* Specifies the return data format either short,
 long or extended as defined above */
  union
    {
    struct short_data_format rp_short;
    struct long_data_format rp_long;
    struct extended_data_format rp_extended;
    char reserved[64];
    } rp_data;
  };
```

Example of the **READ_TAPE_POSITION** IOCTL.

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

    struct read_tape_position rpos;

    printf("Reading tape position long form....\n");
    rpos.data_format = RP_LONG_FORM;
    if (ioctl (dev_fd, READ_TAPE_POSITION, &rpos) <0)
       return errno;

       if (rpos.rp_data.rp_long.bop)
       printf("    Beginning of Partition ..... Yes\n");
     else
       printf("    Beginning of Partition ..... No\n");
      if (rpos.rp_data.rp_long.eop)
       printf("    End of Partition ........... Yes\n");
     else
       printf("    End of Partition ........... No\n");

      if (rpos.rp_data.rp_long.bpew)
       printf("    Beyond Early Warning ... ... Yes\n");
     else
       printf("    Beyond Early Warning ....... No\n");

      if (rpos.rp_data.rp_long.lonu
)        {
       printf("    Active Partition ........... UNKNOWN \n");
       printf("    Logical Object Number ...... UNKNOWN \n");
       }
     else
       {
       printf("    Active Partition ... ....... %u \n",
            rpos.rp_data.rp_long.active_partition);
       printf("    Logical Object Number ...... %llu \n",
            rpos.rp_data.rp_long.logical_obj_number);
        }
```

```
      if (rpos.rp_data.rp_long.mpu
)       printf("    Logical File ID ............ UNKNOWN \n");
     else
       printf("    Logical File ID ............ %llu \n",
              rpos.rp_data.rp_long.logical_file_id);
```

SET_TAPE_POSITION

The **SET_TAPE_POSITION** IOCTL is used to position the tape in the current active partition to either a logical block id or logical filemark. The **logical_id_type** field in the IOCTL structure specifies either a logical block or logical filemark.

The data structure that is used with this IOCTL is

```
#define LOGICAL_ID_BLOCK_TYPE   0x00
#define LOGICAL_ID_FILE_TYPE    0x01

struct set_tape_position{
  uchar logical_id_type;    /* Block or file as defined above */
  ullong logical_id;        /* logical object or logical file to position to */
  char reserved[32];
  };
```

Examples of the **SET_TAPE_POSITION** IOCTL.

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

  struct set_tape_position setpos;

  /* position to logical block id 10 */
  setpos.logical_id_type =  LOGICAL_ID_BLOCK_TYPE
  setpos.logical_id = 10;
  ioctl(dev_fd, SET_TAPE_POSITION, &setpos);

  /* position to logical filemark 4 */
  setpos.logical_id_type =  LOGICAL_ID_FILE_TYPE
  setpos.logical_id = 4;
  ioctl(dev_fd, SET_TAPE_POSITION, &setpos);
```

QUERY_LOGICAL_BLOCK_PROTECTION

The IOCTL queries whether the drive can support this feature, what LBP method is used, and where the protection information is included.

The **lbp_capable** field indicates whether the drive has logical block protection (LBP) capability. The **lbp_method** field displays if LBP is enabled and what the protection method is. The LBP information length is shown in the **lbp_info_length** field. The fields of **lbp_w**, **lbp_r**, and **rbdp** present that the protection information is included in write, read, or recover buffer data.

The data structure that is used with this IOCTL is

```
struct logical_block_protection
{
   uchar lbp_capable;     /* [OUTPUT] the capability of lbp for  QUERY ioctl only */
   uchar lbp_method;      /* lbp method used for QUERY [OUTPUT] and SET [INPUT] ioctls */
     #define  LBP_DISABLE           0x00
     #define  REED_SOLOMON_CRC      0x01
   uchar lbp_info_length; /* lbp info length for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_w;           /* protection info included in write data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_r;           /* protection info included in read data */
                          /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
```

```
   uchar rbdp;           /* protection info included in recover buffer data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar reserved[26];
};
```

Examples of the **QUERY_LOGICAL_BLOCK_PROTECTION** IOCTL.

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

  int rc;
  struct logical_block_protection lbp_protect;

  printf("Querying Logical Block Protection....\n");

  if (rc=ioctl(dev_fd, QUERY_LOGICAL_BLOCK_PROTECTION, &lbp_protect))
      return rc;

  printf("  Logical Block Protection capable........ %d\n",lbp_protect.lbp_capable);
  printf("  Logical Block Protection method.......... %d\n",lbp_protect.lbp_method);
  printf("  Logical Block Protection Info Length... %d\n",lbp_protect.lbp_info_length);
  printf("  Logical Block Protection for Write........ %d\n",lbp_protect.lbp_w);
  printf("  Logical Block Protection for Read....... %d\n",lbp_protect.lbp_r);
  printf("  Logical Block Protection for RBDP...... %d\n",lbp_protect.rbdp);
```

SET_LOGICAL_BLOCK_PROTECTION

The IOCTL enables or disables Logical Block Protection, sets up what method is used, and where the protection information is included.

The **lbp_capable** field is ignored in this IOCTL by the IBMtape driver. If the **lbp_method** field is 0 (LBP_DISABLE), all other fields are ignored and not used. When the **lbp_method** field is set to a valid non-zero method, all other fields are used to specify the setup for LBP.

The data structure that is used with this IOCTL is

```
struct logical_block_protection
{
   uchar lbp_capable;    /* [OUTPUT] the capability of lbp for  QUERY ioctl only */
   uchar lbp_method;     /* lbp method used for QUERY [OUTPUT] and SET [INPUT] ioctls */
     #define  LBP_DISABLE           0x00
     #define  REED_SOLOMON_CRC      0x01
   uchar lbp_info_length;/* lbp info length for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_w;          /* protection info included in write data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar lbp_r;          /* protection info included in read data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar rbdp;           /* protection info included in recover buffer data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   uchar reserved[26];
};
```

Examples of the **SET_LOGICAL_BLOCK_PROTECTION** IOCTL.

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

 int rc;
  struct logical_block_protection lbp_protect;

  printf("Setting Logical Block Protection....\n\n");

  printf ("Enter Logical Block Protection method:      ");
  gets (buf);
  lbp_protect.lbp_method= atoi(buf);
  printf ("Enter Logical Block Protection Info Length: ");
  gets (buf);
```

```
lbp_protect.lbp_info_length= atoi(buf);
printf ("Enter Logical Block Protection for Write:   ");
gets (buf);
lbp_protect.lbp_w= atoi(buf);
printf ("Enter Logical Block Protection for Read:    ");
gets (buf);
lbp_protect.lbp_r= atoi(buf);
printf ("Enter Logical Block Protection for RBDP:    ");
gets (buf);
lbp_protect.rbdp= atoi(buf);

rc = ioctl(dev_fd, SET_LOGICAL_BLOCK_PROTECTION, &lbp_protect);

  if (rc)
   printf ("Set Logical Block Protection Fails (rc %d)",rc);
 else
   printf ("Set Logical Block Protection Succeeds");
```

Note:

1. The drive always expects a CRC attached with a data block when LBP is enabled for **lbp_r** and **lbp_w**. Without the CRC bytes attachment, the drive fails the **Read** and **Write** command. To prevent the CRC block transfer between the drive and application, the maximum block size limit must be determined by application. Call the **STIOC_GET_PARM** IOCTL to get the parameter of **MAX_SCSI_XFER** (the system maximum block size limit). Call the **STIOC_READ_BLKLIM** IOCTL to get the value of **max_blk_lim** (the drive maximum block size limit). Then, use the minimum of the two limits.
2. When a unit attention with a power-on and device reset (Sense key/Asc-Ascq x6/x2900) occurs, the LBP enable bits (**lbp_w**, **lbp_r**, and **rbdp**) are reset to OFF by default. The IBMtape tape driver returns EIO for an IOCTL call in this situation. Once the application determines it is a reset unit attention in the sense data, it responds to query LBP setup again and reissues this IOCTL to set up LBP properly.
3. The LBP setting is controlled by the application and not the device driver. If an application enables LBP, it must also disable LBP when it closes the drive, as this action is not done by the device driver.

VERIFY_TAPE_DATA

The IOCTL issues a **VERIFY** command to cause data to be read from the tape and passed through the drive's error detection and correction hardware. This action determines whether it can be recovered from the tape. Or, whether the protection information is present and validates correctly on logical block on the medium. The driver returns the IOCTL a failure or a success if the **VERIFY SCSI** command is completed in a Good SCSI status.

Note:

1. When an application sets the VBF method, it considers the driver's close operation in which the driver can write filemarks in its close that the application did not explicitly request. For example, some drivers write two consecutive filemarks that mark the end of data on the tape in its close, if the last tape operation was a **WRITE** command.
2. Per the user's or application's request, the IBMtape driver sets the block size in the field of **Block Length** in mode block descriptor for **Read** and **Write** commands. Then, it maintains this block size setting in a whole open. For instance, the tape driver set a zero in the **Block Length** field for the variable

block size. This act causes the missing of an overlength condition on a SILI Read (and cause problems for LTFS). Block Length must be set to a non-zero value.

Before the set Fixed bit ON with VTE or VBF ON in **VERIFY** IOCTL, the application is also requested to set the block size in mode block descriptor. The drive uses it to verify the length of each logical block. For example, a 256 KB length is set in **Block Length** field to verify the data. The setup overrides the early setting from the IBM tape driver.

Once the application completes **VERIFY** IOCTL call, the original block size setting needs to be restored for **Read** and **Write** commands. The application either issues **Set Block Size** IOCTL, or closes the drive immediately and reopens the drive for the next tape operation. It is recommended to reopen the drive for the next tape operation. Otherwise, it causes **Read** and **Write** command misbehavior.

3. To support DPF for **VERIFY** command with FIXED bit on, it is requested to issue IBM tape driver to set **blksize** standard IOCTL to set the block size. The IBM tape driver sets the **Block Length** in mode block descriptor same as the block size and save the block size in kernel memory. The driver restores the **Block Length** before it retries the **Verify SCSI** command. Otherwise, the retry **VERIFY** command fails.
4. The IOCTL can be returned longer than the timeout when DPF occurs.

The data structure that is used with this IOCTL is

```
typedef struct
{
   uchar             :  2,   /* reserved                                   */
                  vte:  1,   /* verify to end-of-data                      */
                vlbpm: 1,    /* verify logical block protection information */
                  vbf: 1,    /* verify by filemarks                        */
                immed:  1,   /* return SCSI status immediately             */
               bytcmp:  1,   /* Reserved for IBM future use.               */
                fixed:  1;   /* set Fixed bit to verify the length of
                                each logical block                         */
   uchar       reserved[15]; /* Reserved for IBM future use.               */
   uint      verify_length;  /* amount of data to be verified              */
}verify_data_t ;
```

Examples of the **VERIFY_TAPE_DATA** IOCTL.

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

   char buf[60];
   verify_data_t vd;
   unsigned int vlength=0;
   int i;

   bzero( (void *) &vd, sizeof( verify_data_T)  );

   printf("Enable field \'Verify to End Of Data\'[y/n]: ");
   gets( buf);
   vd.vte = ( tolower( buf[0] ) == 'y' );

   printf("Enable field \'verify logical block protection information\'[y/n]: ");
   gets( buf);
   vd.vlbpm = ( tolower( buf[0] ) == 'y' );

   printf("Enable field \'verify by filemarks\'[y/n]: ");
   gets( buf);
   vd.vbf = ( tolower( buf[0] ) == 'y' );

   printf("Enable field \'return SCSI status immediately\'[y/n]: ");
```

```
gets( buf);
vd.immed = ( tolower( buf[0] ) == 'y' );

printf("Enable field \'set Fixed bit to verify the length of each
logical block\'[y/n]: ");
gets( buf);
vd.fixed = ( tolower( buf[0] ) == 'y' );

printf("Get the amount of data to be verified: ");
gets( buf);
vlength = atoi( buf);

vd.verify_length = vlength;

printf("Data dump:\n");

for( i = 0; i &lt sizeof( struct verify_data); i++)
    printf("byte %d: 0x%02x\n", i, *((( char *) vd;) + i)   );

if (!ioctl ( dev_fd, VERIFY_TAPE_DATA, (void *) &vd)){
    printf ("The VERIFY_DATA ioctl succeeded\n");
}
else{
    perror ("The VERIFY_DATA ioctla failed");
}
```

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.

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/imple** directory. Any application program that issues these commands must include this header file.

MTIOCTOP

This command runs 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 that is defined in the **/usr/include/sys/mtio.h** system header file. The two IOCTL commands are interchangeable. See “STIOC_TAPE_OP” on page 243.

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 that is defined in the **/usr/include/sys/mtio.h** system header file. The two IOCTL commands are interchangeable. See "STIOC_GET_DEVICE_STATUS" on page 245.

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 that is defined in the **/usr/include/sys/mtio.h** system header file. The two IOCTL commands are interchangeable. See "STIOC_GET_DEVICE_INFO" on page 246.

USCSICMD

This command provides the user a SCSI command interface.

Attention: The **uscsi** command is powerful, but dangerous. So its use is restricted to processes that run 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. Avoid commands such as **Mode Select**, as they can damage data that is stored on the drive or system panics. Also, as the commands are not checked in any way by the device driver, any block can be overwritten. 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 *)&uscsi_cmd, (char)0, sizeof(uscsi_cmd));
memset ((char *)&rq_buf, (char)0, sizeof(rq_buf));
memset ((char *)&uscsi_buf, (char)0, sizeof(uscsi_buf));

printf("Enter the SCSI cdb in hex (f.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: Enter the command byte more or less
 than two digitals.\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 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);
```

Downward compatibility tape drive IOCTL operations

This set of IOCTL commands is provided for compatibility only with previous versions of the IBM SCSI Tape Device Driver (**IBMDDAst**) that supported the IBM 3490E Magnetic Tape Subsystem on the SunOS 4.1.3 operating system. The applications that are written for **IBMDDAst** are compatible with the device driver (**IBMtape**) on a source level only. Binary compatibility is not guaranteed.

Recompile the application by using the **/usr/include/sys/oldtape.h** header file (in place of the previously used **/usr/include/sys/Atape.h**).

Note: This interface is obsolete. It was superseded by the interface that is defined in the **/usr/include/sys/st.h** header file. New development efforts must use the **st.h** interface to ensure its compatibility with future releases of the Solaris Tape and Medium Changer Device Driver.

The following commands are supported.

STIOCQRYP
: Query the working parameters of the tape drive.

STIOCSETP
: Set the working parameters of the tape drive.

STIOCSYNC
: Flush the drive buffers to the tape.

STIOCDM
: Display messages on the tape drive console.

STIOCQRYPOS
: Query the physical position on the tape.

STIOCSETPOS
: Set the physical position on the tape.

STIOCQRYSENSE
: Return the sense data that is collected from the tape drive.

STIOCQRYINQUIRY
: Return the inquiry data that is collected from the tape drive.

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

Note: The **oldtape.h** header file replaces the **Atape.h** header file.

STIOCQRYP or STIOCSETP

These commands allow a program to query and set the working parameters of the tape drive.

Issue the `query` command to fill the fields of the data structure with the current data that you do not want to change. Make the changes to the required fields and issue the `set` command to process the required changes.

Changing certain fields (such as **buffered_mode** or **compression**) can affect the drive performance. If **buffered_mode** is disabled, each block that is written to the tape drive is immediately transferred to the tape. This process guarantees that each record is on the tape, but it degrades performance. If compression mode is enabled, the write performance can increase based on the compressibility of the data written.

The changes that are made through this IOCTL are effective only during the current open session. The tape drive reverts to the default working parameters established by the configuration file at the time of the next open operation.

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

```
struct stchgp_s  {
  int blksize;                    /* block size */
  struct sttrc_s {
    boolean trace;                /* not used */
    ulong hkwrd;                  /* not used */
  } sttrc;
  int sync_count;                 /* OBSOLETE AND UNSUPPORTED */
  boolean autoload;               /* OBSOLETE AND UNSUPPORTED */
  boolean buffered_mode;          /* on/off buffered mode */
  boolean compression;            /* on/off compression mode */
  boolean trailer_labels;         /* on/off write past EOM mode */
  boolean rewind_immediate;       /* on/off immediate rewind mode */
  boolean reserved[64];           /* reserved */
};
```

The data structure has the following fields.

- blksize

  This field defines the effective block size for the tape drive (0=variable).
- sync_count

  This field is obsolete. It is set to 0 by the **Query** command and ignored by the **Change** command.
- autoload

  This field is obsolete. It is set to 0 by the **Query** command and ignored by the **Change** command.
- buffered_mode

  This field enables or disables the buffered write mode.

  (0=disable, 1=enable).
- compression

  This field enables or disables the hardware compression mode.

  (0=disable, 1=enable).
- trailer_labels

  This field enables or disables the trailer-label processing mode.

  (0=disable, 1=enable).

  If this mode is enabled, writing records past the early warning mark on the tape is allowed. The first write operation to detect EOM returns ENOSPC. This write operation does 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.
- rewind_immediate

  This field enables or disables the immediate rewind mode.

  (0=disable, 1=enable).

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

An example of the **STIOCQRYP** and **STIOCSETP** commands is

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

struct stchgp_s stchgp;

/* QUERY OLD PARMS */
if (ioctl (tapefd, STIOCQRYP, &stchgp) < 0) {
  printf ("IOCTL failure, errno = %d", errno);
  exit (errno);
}
```

```
/* SET NEW PARMS */
stchgp.rewind_immediate = rewind_immediate;
stchgp.trailer_labels = trailer_labels;

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

STIOCSYNC

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 **STIOCSYNC** command is

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

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

STIOCDM

This command displays and manipulates one or two messages on the tape drive console.

The message that is sent by using this call does not always remain on the display. It depends on the current drive activity.

Note: All messages must be padded to 8 bytes. Otherwise, garbage characters (meaningless data) can be displayed in the message.

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

```
struct stdm_s {
  char dm_func;                         /* message function codes */
                                        /* Function Selection */
  #define DMSTATUSMSG            0x00   /* general status message */
  #define DMDVMSG                0x20   /* demount/verify message */
  #define DMMIMMED               0x40   /* mount with immediate action */
  #define DMDEMIMMED             0xE0   /* demount with immediate action */
                                        /* Message Control */
  #define DMMSG0                 0x00   /* display message 0 */
  #define DMMSG1                 0x04   /* display message 1 */
  #define DMFLASHMSG0            0x08   /* flash message 0 */
  #define DMFLASHMSG1            0x0C   /* flash message 1 */
  #define DMALTERNATE            0x10   /* alternate messages 0 and 1 */
  #define MAXMSGLEN                 8
  char dm_msg0[MAXMSGLEN];              /* message 0 */
  char dm_msg1[MAXMSGLEN];              /* message 1 */
};
```

An example of the **STIOCDM** command is

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

struct stdm_s stdm;

stdm.dm_func = DMSTATUSMSG | DMMSG0;
bcopy ("SSD", stdm.dm_msg0, 8);
```

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

STIOCQRYPOS or STIOCSETPOS

These commands allow a program to query and set the physical position on the tape.

Tape position is defined as where the next read or write operation occurs. The **STIOCQRYPOS** command and the **STIOCSETPOS** command can be used independently or with each other.

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

```
struct stpos_s
  {
  char block_type;                 /* format of block ID information */
    #define QP_LOGICAL         0
    #define QP_PHYSICAL        1
  boolean eot;                     /* early warning EOT */
  #define blockid_t                   unsigned int
  blockid_t curpos;                /* current or new tape position */
  blockid_t lbot;                  /* last block written to tape */
    #define LBOT_NONE          0xFFFFFFFF
    #define LBOT_UNKNOWN       0xFFFFFFFE
  char reserved[64];               /* reserved */
};
```

The **block_type** field is set to **QP_LOGICAL** for standard SCSI logical tape positions. It is set to **QP_PHYSICAL** for composite tape positions that are used for high-speed locate operations that are implemented by the tape drive.

For **STIOCSETPOS** commands, the **block_type** and **curpos** fields must be filled out by the caller. The other fields are ignored. The type of position that is specified in the **curpos** field must correspond with the type specified in the **block_type** field. Use the **QP_PHYSICAL** type for better performance. High-speed locate positions can be obtained with the **STIOCQRYPOS** command, saved, and used later with the **STIOCSETPOS** command to quickly return to the same location on the tape.

Following a **STIOCQRYPOS** command, the **lbot** field indicates the last block of data that was transferred physically to the tape. For example, if the application wrote to 12 blocks and **lbot** equals 8, four blocks are in the tape buffer. This field is valid only if the last command was a write operation. Otherwise, **LBOT_UNKNOWN** is returned. It does not reflect the number of application write operations because a single write operation can translate to multiple blocks.

An example of the **STIOCQRYPOS** and **STIOCSETPOS** commands is

```
#include <sys/oldtape.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;
```

```
/* do other stuff... */

stpos.curpos = oldposition;
stpos.block_type = QP_PHYSICAL;

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

STIOCQRYSENSE

This command returns the sense data that is collected from the tape drive.

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

```
struct stsense_s {
  /* INPUT */
  char sense_type;                  /* new sense or last error sense */
    #define FRESH               1
    #define LASTERROR           2
  /* OUTPUT */
  #define MAXSENSE              128
  char sense[MAXSENSE];             /* actual sense data */
  int len;                          /* length of sense data returned */
  char reserved[64];                /* reserved */
};
```

If **sense_type** is set to **LASTERROR**, the last sense data that is collected from the device is returned. If it is set to **FRESH**, a new **Request Sense** command is issued and the sense data is returned.

An example of the **STIOCQRYSENSE** command is

```
#include <sys/oldtape.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 (SENSE_KEY (&stsense.sense) == MEDIUM_ERROR) {
  printf ("We're in trouble now!");
  exit (SENSE_KEY (&stsense.sense));
}
```

STIOCQRYINQUIRY

This command returns the inquiry data that is collected from the tape drive.

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

```
struct inq_data_s {
  BYTE b0;                          /* peripheral device byte */
    #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;                            /* removable media/device type byte */
    #define RMB(x)                    ((x->b1 & 0x80)>>7)
    #define FIXED                     0
    #define REMOVABLE                 1
    #define device_type_qualifier(x)  (x->b1 & 0x7F)
  BYTE b2;                            /* standards version byte */
    #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;                            /* asynchronous event notification */
    #define AENC(x)                   ((x->b3 & 0x80)>>7)
    #define TrmIOP(x)                 ((x->b3 & 0x40)>>6)
    #define Response_Data_Format(x)   (x->b3 & 0x0F)
    #define SCSI1INQ                  0
    #define CCSINQ                    1
    #define SCSI2INQ                  2
  BYTE additional_length;
  BYTE res56[2];                      /* reserved bytes */
  BYTE b7;                            /* protocol byte */
    #define RelAdr(x)                 ((x->b7 & 0x80)>>7)
    #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];      /* vendor identification */
  char product_identification[16];    /* product identification */
  char product_revision_level[4];     /* product revision level */
};
struct st_inquiry {
    struct inq_data_s standard;
    BYTE vendor_specific[255-sizeof(struct inq_data_s)];
};
```

An example of the **STIOCQRYINQUIRY** command is

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

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");
}
```

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.

IOC_FORCE_DUMP
Force the device to complete 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.

IOC_DEVICE_PATH
Query the path information for a particular path or all of the paths for a particular parent device.

IOC_CHECK_PATH
Display the enable or disable information for each path in the path table.

IOC_ENABLE_PATH
Enable a path in the path table.

IOC_DISABLE_PATH
Disable a path in the path table.

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

STIOC_DEVICE_SN

This command returns the device number as used by the IBM Enterprise Tape Library and the Enterprise Model B18 Virtual Tape Server.

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 ();
}
```

IOC_FORCE_DUMP

This command forces the device to complete 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. The IBM 3490E Magnetic Tape Subsystem and the IBM Enterprise Model B18 Virtual Tape Server do not support this command.

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 ();
}
```

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 must be set to one of the following values.

VEND_MODE
: Vendor-specific mode.

DSCR_MODE
: Descriptor mode.

DNLD_MODE
: Download mode.

The **id** field must 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 ();
}
```

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 must be set to one of the following values.

VEND_MODE
Vendor-specific mode.

DSCR_MODE
Descriptor mode.

DNLD_MODE
Download mode.

The **id** field must 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; /* buffer_io should be initialized
    per the hardware ref*/
if (!(ioctl (dev_fd, STIOC_WRITE_BUFFER, &buffer_io))) {
  printf ("Buffer written successfully.\n");
}

else {
  perror ("Failure writing buffer");
  scsi_request_sense ();
}
```

IOC_DEVICE_PATH

This command returns the information about the path information for a particular path or all of the paths for a particular parent device.

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

```
typedef struct {
 int          instance;                        /* Instance Number of this path */
 int          tgt;                             /* SCSI target for this path    */
 int          lun;                             /* SCSI LUN for this path       */
 uint64_t     wwnn;                            /* WWNN for this fc path        */
 uint64_t     wwpn;                            /* WWPN for this fc path        */
 int          path_type;                       /* primary 0 or
    alt 1, 2, 3, ..., 15 */
                                        /* none 0xFF                  */
 int          enable;                          /* path enable 1, disable 0     */
 char         devpath[125];                    /* devices path of this path    */
 char         dev_ser[33];                     /* Device serial number         */
 char         ucode_level[32];                 /* Device microcode level       */
}  device_path_t;

typedef struct {
 int          number_paths;                    /* number of paths configured   */
```

An example of the **IOC_DEVICE_PATH** command is

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

device_paths_t device_paths;

if (rc = ioctl(dev_fd,IOC_DEVICE_PATHS, %device_paths)){
 perror ("IOC_DEVICE_PATHS failed");
 printf ("\n");
 return (rc);
}

printf ("\nEnter path number or <enter> for all of the paths:");
gets (buf);
if (buf[0] == '\0') {
  for (i=0; i<device_paths.number_paths)i++) {
    show_path (&device_paths.device_path[i]);
    printf ("\n---more---")
         if (interactive) getchar ();
  }
}
else {
  i = atoi(buf);
  if ((i>=device_paths.number_paths||(i<0) {
    printf ("\nInvalid Path Number selection.\n");
    return (FALSE);
  }
  show path (&device_paths_.device_path[i]);
}
```

IOC_CHECK_PATH

This command is used to display the enable or disable information for each path in the path table.

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

```
typedef struct {
  int  number_paths;                    /* number of paths configured       */
  path_enable_t       path_enable[MAX_SCSI_PATH];
} check_path_t;
```

See the example of the **IOC_CHECK_PATH** command in "IOC_ENABLE_PATH and IOC_DISABLE_PATH."

IOC_ENABLE_PATH and IOC_DISABLE_PATH

This command is used to enable or disable a path in the path table.

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

```
typedef struct {
  int path;                            /* Failover path:  primary path: 0  */
                                       /*  alternate path: 1, 2, 3, ..., 15 */
                                       /* No failover path : 0xFF          */
  int enable;                         /* path enable 1, disable 0         */
} path_enable_t;
```

An example of the commands is

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

check_path_t check_path;
path_enable_t path_enable;

if (!(rc = ioctl (dev_fd, IOC_CHECK_PATHS, &check_path))) {
      printf ("IOC_CHECK_PATHS succeeded.\n");
}

printf ("Enter selection (0=disable, 1=enable): ");
gets (buf);
if (*buf != '\0') {
    if (path_enable.enable) {
        if (rc = ioctl (dev_fd, IOC_ENABLE_PATH, &path_enable)) {
            perror ("IOC_ENABLE_PATH failed");
            printf ("\n");
            return (rc);
       }
   }
   else {
       if (rc = ioctl (dev_fd, IOC_DISABLE_PATH, &path_enable)) {
            perror ("IOC_DISABLE_PATH failed");
            printf ("\n");
            return (rc);
       }
    }
}
```

Return codes

The calls to the IBMtape device driver return error codes that describe the outcome of the call. The returned error codes 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 is either **0** for success, or **-1** for failure. Then, the system global variable *errno* contains the error value. For the **read** and **write** calls, the return code of the function call contains 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 or end of tape. If the read or write operation completely failed, the return code is set to **-1** and the error value is stored in the system global variable *errno*.

The error codes that are returned from IBMtape are described in the following section.

Note: The EIO return code indicates that a device-related input/output (I/O) error occurred. Further information about the error can be obtained by using the **IOC_REQUEST_SENSE** IOCTL command to retrieve sense data. This sense data can then be interpreted with the device hardware or SCSI reference.

General error codes

The following codes and their descriptions apply in general to all operations.

[EACCES]
An operation to modify the media was attempted illegally.

[EBADF]
A bad file descriptor was specified for the device.

[EBUSY]
An excessively 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 exceeds 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 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.

[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, 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.

[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.

[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.

[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 because of 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 is 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** operation.

[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 because of 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 beginning an I/O session with a tape drive or medium changer. For example:

```
fd = open ("/dev/rmt/0st", O_FLAGS);
```

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 **O_FLAGS** parameters are defined in the **/usr/include/sys/fcntl.h** system header file. Use bitwise inclusive OR to combine individual values together. 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 that is mounted fails.
- **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 that is mounted fails.
- **O_NDELAY** or **O_NONBLOCK**

  These two flags complete the same function. This option indicates to the driver not to wait until the tape drive is ready before the device is opened and commands are sent. Until the drive is ready, subsequent commands that require a physical tape to be loaded and ready fail. Other commands that do not require a tape to be loaded, such as inquiry or move medium commands, succeed. All special files support these flags.
- **O_APPEND**

  This flag is used 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 that is mounted fails.

  During an open for append operation, the tape is rewound and 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 fails. An example of writing to a tape drive is

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

where:
`count` is the return code from the write command.
`fd` is the file descriptor of a previously opened special file.
`buffer` is a pointer to the source data buffer.
`numbytes` is the number of bytes requested to be written.

If the device is configured to use a fixed block size, **numbytes** must be a whole number multiple of the block size. If the block size is variable, the value that is specified in **numbytes** is the size of the block written.

After each call to write is issued, the return code tells how many bytes were written. Normally, the return code is the same as the number of bytes requested in the write command. There are some exceptions, however. If the device is configured to use fixed block size, and a write is for multiple blocks, possibly only some of the requested blocks can be written. This action is called a **short write**. The return code from a **short write** is less than the number of bytes requested, but always a whole number multiple of the block size. Applications that write multiple fixed blocks must be prepared to handle short writes, and calculate from the return code which blocks were not transferred to tape. Short writes are not an error condition, and IBMtape does not set a value for the *errno* system variable.

- A return code of zero indicates that the logical end of medium (LEOM) is reached. None of the requested bytes were written. A return code of zero is not an error condition, and IBMtape does not set a value for the *errno* system variable.
- If the return code is less than zero, the **write** operation failed. None of the requested bytes were written. IBMtape sets an error code in the *errno* system variable.

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 fails. An example of reading from a tape drive is

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

where:
`count` is the return code from the read command.
`fd` is the file descriptor of a previously opened special file.
`buffer` is a pointer to the destination data buffer.
`numbytes` is the maximum number of bytes requested to be read.

If the device is configured for variable block size, a single block of up to **numbytes** bytes is read. However, if the block size on tape is greater than **numbytes**, the read fails, with *errno* set to ENOMEM. This action is called an **overlength read** condition.

If the device is configured to use a fixed block size, **numbytes** must be a whole number multiple of that block size. If **numbytes** is not such a multiple, IBMtape fails the read and sets *errno* to EINVAL. If the block size on tape does match the configured block size, whether larger or smaller, the read fails, with *errno* set to EIO. This action is called an **incorrect length** condition.

After the **read** is issued, if **count** is less than zero, the read failed, no data is returned. The system variable *errno* is set to indicate the type of error. See “Read error codes” on page 294 for a complete list of *errno* values and their meanings.

If **count** equals zero, then the end of medium (EOM) or a filemark was encountered before any data was read. This issue is not an error condition, and IBMtape does not set *errno*. If a second read returns zero, the application infers that EOM is reached. Otherwise, the application infers that a filemark was encountered. When a filemark is encountered while reading, the tape is left positioned on the end of medium (EOM) side of the filemark.

If greater than zero, **count** reports how many bytes were read from tape. Even though greater than zero, it can still be less than **numbytes**. If the device is configured for variable blocks, **count** can be any value between 1 and **numbytes**. If configured to use a fixed block size, **count** can always be a whole number multiple of that block size. In either case, such a condition is called an **underlength read** or **short read**.

Underlength reads are not error conditions, and IBMtape does not set *errno*. However, for variable block mode, some overhead processing that is incurred by underlength reads can be eliminated by setting the SILI parameter to **1**. This action can improve read performance. See "STIOC_GET_PARM" on page 249 for information on the **SILI** parameter.

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

If the **close** 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 close operation attempts to run 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 started 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 might be inconsistent.

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

```
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
```

Rules:
1. Return EIO and release the drive when a unit attention happens before the close().
2. Fail the command, return EIO and release the drive if a unit attention occurs during the close().
3. If a SCSI command fails during close processing, only the **SCSI RELEASE** is attempted thereafter.
4. If the tape is already unloaded from the driver, no **SYNC BUFFER (WFM(0))** or rewinding (only for rewind-on-close special files) of the tape is done.
5. The return code from the **SCSI RELEASE** command is ignored.

Issuing IOCTL operations to a special file

The IOCTL system call provides the mechanism for running 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 that is returned from the **open** system call. The `command` is the value of the IOCTL operation that is defined in the appropriate header file. Also, `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, and any other value indicates a failure as defined in the **/usr/include/sys/errno.h** header file.

The IOCTL operations that are supported by the Solaris Tape and Medium Changer Device Driver are defined in the following header files. They are included with the IBMtape package and installed in the **/usr/include/sys** subdirectory. These header files must be included by any application source file that requires access the IOCTL functions that are supported by the IBMtape device driver. (Existing applications that use the standard Solaris tape drive IOCTL operations that are 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)
- oldtape.h (downward compatible tape drive operations, **obsolete**)

Chapter 6. Windows tape device drivers

Windows programming interface

The programming interface conforms to the standard Microsoft Windows Server 2003, Windows Server 2008, and Windows Server 2012 tape device drivers interface. It is detailed in the Microsoft Developer Network (MSDN) Software Development Kit (SDK) and Driver Development Kit (DDK). Common documentation for these similar devices are indicated by 200x.

Windows IBMTape is conformed by two sets of device drivers,

- **ibmtpxxx.sys**, which supports the IBM TotalStorage or Magstar tape drives, where
  - **ibmtp2k3.sys, ibmtpbs2k3.sys, ibmtpft2k3.sys** are used for Windows Server 2003
  - **ibmtp2k8.sys, ibmtpbs2k8.sys, ibmtpft2k8.sys** are used for Windows Server 2008
  - **ibmtp2k12.sys, ibmtpbs2k12.sys, ibmtpft2k12.sys** are used for Windows Server 2012
- **ibmcgxxx.sys**, which supports the IBM TotalStorage or Magstar medium changer, where
  - **ibmcg2k3.sys, ibmcgbs2k3.sys, ibmcgft2k3.sys** are used for Windows Server 2003
  - **ibmcg2k8.sys, ibmcgbs2k8.sys, ibmcgft2k8.sys** are used for Windows Server 2008
  - **ibmcg2k12.sys, ibmcgbs2k12.sys, ibmcgft2k12.sys** are used for Windows Server 2012

The programming interface conforms to the standard Microsoft Windows 200x tape device driver interface. It is detailed in the Microsoft Developer Network (MSDN) Software Development Kit (SDK), and Driver Development Kit (DDK).

User-callable entry points

The following user-callable tape driver entry points are supported under **ibmtpxxx.sys**.

- CreateFile
- CloseHandle
- DeviceIoControl
- EraseTape
- GetTapeParameters
- GetTapePosition
- GetTapeStatus
- PrepareTape
- ReadFile
- SetTapeParameters
- SetTapePosition

- WriteFile
- WriteTapemark

Tape Media Changer driver entry points

If the Removable Storage Manager is stopped, then the following user-callable tape media changer driver entry points are supported under **ibmcgxxx.sys**:

- CreateFile
- CloseHandle
- DeviceIoControl

Users who want to write application programs to issue commands to IBM TotalStorage device drivers must obtain a license to the MSDN and the Microsoft Visual C++ Compiler. Users also need access to IBM hardware reference manuals for IBM TotalStorage devices.

Programs that access the IBM TotalStorage device driver must complete the following steps:

1. Include the following files in the application.

```
#include <ntddscsi.h>
#include <ntddchgr.h>
#include <ntddtape.h> /* Modified as indicated below */
```

2. Add the following lines to **ntddtape.h**.

```
#define LB_ACCESS FILE_READ_ACCESS | FILE_WRITE_ACCESS
#define M_MTI(x) CTL_CODE(IOCTL_BASE+2,x,METHOD_BUFFERED, LB_ACCESS)
#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_LOG_SENSE10 CTL_CODE(IOCTL_TAPE_BASE, 0x0833,
METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_ENH_TAPE_LOG_SENSE10  CTL_CODE(IOCTL_TAPE_BASE, 0x0835, METHOD_BUFFERED,
FILE_READ_ACCESS )
#define IOCTL_TAPE_REPORT_MEDIA_DENSITY CTL_CODE(IOCTL_TAPE_BASE, 0x081e,
METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_TAPE_OBTAIN_MTDEVICE (M_MTI(16))
#define IOCTL_CREATE_PARTITION CTL_CODE(IOCTL_TAPE_BASE, 0x0826, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_QUERY_PARTITION CTL_CODE(IOCTL_TAPE_BASE, 0x0825, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_SET_ACTIVE_PARTITION CTL_CODE(IOCTL_TAPE_BASE, 0x0827, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_QUERY_DATA_SAFE_MODE CTL_CODE(IOCTL_TAPE_BASE, 0x0823, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_SET_DATA_SAFE_MODE CTL_CODE(IOCTL_TAPE_BASE, 0x0824, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_ALLOW_DATA_OVERWRITE CTL_CODE(IOCTL_TAPE_BASE, 0x0828, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_SET_PEW_SIZE
CTL_CODE(IOCTL_TAPE_BASE, 0x082C, METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_QUERY_PEW_SIZE
CTL_CODE(IOCTL_TAPE_BASE, 0x082B, METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_VERIFY_TAPE_DATA
CTL_CODE(IOCTL_TAPE_BASE, 0x082A, METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_QUERY_RAO_INFO   CTL_CODE(IOCTL_TAPE_BASE, 0x082E, METHOD_BUFFERED,
FILE_READ_ACCESS )
```

```
#define IOCTL_GENERATE_RAO     CTL_CODE(IOCTL_TAPE_BASE, 0x082F, METHOD_BUFFERED,
FILE_READ_ACCESS )
#define IOCTL_RECEIVE_RAO      CTL_CODE(IOCTL_TAPE_BASE, 0x0834, METHOD_BUFFERED,
FILE_READ_ACCESS )
```

CreateFile

The **CreateFile** entry point is called to make the driver and device ready for input/output (I/O). Installing the driver in non-exclusive mode allows several handles to the same TotalStorage device. If the driver was installed in exclusive mode, by default only one active handle is allowed to a given TotalStorage device. However, if more than one handle is needed for the same device, the **dwCreationDisposition** parameter can be set to OPEN_ALWAYS in CreateFile() function to create an extra handle. By using this flag the resulting handle has limited functions. The driver allows the following IOCTLs only:

IOCTL_STORAGE_PERSISTENT_RESERVE_IN
IOCTL_STORAGE_PERSISTENT_RESERVE_OUT
IOCTL_SCSI_PASS_THROUGH_DIRECT with Cdb[0] set to INQUIRY (0x12)

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

```
HANDLE ddHandle0, ddHandle1; // file handle for LUN0 and LUN1

/*
** Open for reading/writing on LUN0,
** where the device special file name is in the form of tapex and
** x is the logical device 0 to n - can be determined from Registry
**
** Open for media mover operations on LUN1,
** where the device special file name is in the form of
** changerx and x is the logical device 0 to n - can be determined from Registry

ddHandle0 = CreateFile(
                                       "\\\\.\\tape0",
                                       DWORD dwDesiredAccess,
                                       DWORD dwShareMode,
                                       LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                       DWORD dwCreationDisposition,
                                       DWORD dwFlagsAndAttributes,
                                       HANDLE hTemplateFile
                                       );
ddHandle1 = CreateFile(
                                       "\\\\.\\changer0",
                                       DWORD dwDesiredAccess,
                                       DWORD dwShareMode,
                                       LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                       DWORD dwCreationDisposition,
                                       DWORD dwFlagsAndAttributes,
                                       HANDLE hTemplateFile
                                            );
/* Print msg if open failed for handle 0 or 1 */
if(ddHandlen == INVALID_HANDLE_VALUE)
{
  printf("open failed for LUNn\n");
  printf("System Error = %d\n",GetLastError());
  exit (-1);
}
```

The **CloseHandle** entry point is called to stop 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.

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 that is returned never exceeds the length parameter.

See "Variable and fixed block read/write processing" on page 331 for a full discussion of the read/write processing feature.

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. One example is attempting to write past the physical end of the tape.

See "Variable and fixed block read/write processing" on page 331 for a full discussion of the read/write processing feature.

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 also verifies that all the requested data was written by examining the **`lpNumberOfBytesWritten`** parameter. See "Write Tapemark" for details on committing data on the media.

Write Tapemark

Application writers who are using the **WriteFile** entry point to write data to tape must 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 might return a successful return code, but the data might not be on the media yet. Calling the **WriteTapemark** entry point and receiving a good return code, however, ensures that data is committed to tape media properly if all previous **WriteFile** calls were successful. However, applications that write large amounts of data to tape might not want to wait until writing a tapemark to know whether previous data was written to the media properly. For example:

```
WriteTapemark(
 HANDLE hDevice,
 DWORD dwTapemarkType,
 DWORD dwTapemarkCount,
 BOOL bImmediate
);
```

`dwTapemarkType` is the type of operation requested.

The only type that is supported is

```
TAPE_FILEMARKS
```

The WriteTapemark entry point might also be called with the **`dwTapemarkCount`** parameter set to 0 and the **`bImmediate`** parameter that is set to FALSE. This action commits any uncommitted data that is written by previous **WriteFile** calls (since the last call to **WriteTapemark**) to the media. If no error is returned by the **WriteFile** calls and the **WriteTapemark** call, the application can assume that all data is committed to the media successfully.

SetTapePosition

The **SetTapePosition** entry point is called to seek to a particular block of media data. For example:

```
SetTapePosition(
 HANDLE hDevice,
 DWORD dwPositionMethod,
 DWORD dwPartition,
 DWORD dwOffsetLow,
 DWORD dwOffsetHigh,
 BOOL bImmediate
);
```

`dwPositionMethod` is the type of positioning.

For Magstar devices, the following types of tapemarks and immediate values are supported.

TAPE_ABSOLUTE_BLOCK	bImmediate TRUE or FALSE
TAPE_LOGICAL_BLOCK	bImmediate TRUE or FALSE

For 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. For example:

```
GetTapePosition(
 HANDLE hDevice,
 DWORD dwPositionType,
 LPDWORD lpdwPartition,
 LPDWORD lpdwOffsetLow,
 LPDWORD lpdwOffsetHigh
);
```

`dwPositionType` is the type of positioning.

TAPE_ABSOLUTE_POSITION or **TAPE_LOGICAL_POSITION** might 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 drive.

The data structures are

```
struct{ // structure used by GET_TAPE_MEDIA_INFORMATION
        LARGE_INTEGER   Capacity;  /* invalid for Magstar */
        LARGE_INTEGER   Remaining; /* invalid for Magstar */
        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 prepare the tape for access or removal. For example,

```
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 completed from the current location. For example:

```
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

GetTapeStatus

The **GetTapeStatus** entry point is called to determine whether the tape device is ready to process tape commands. For example,

```
GetTapeStatus(
HANDLE hDevice
);
```

`hDevice` is the handle to the device for which to get the device status.

DeviceIoControl

The **DeviceIoControl** function is described in the *Microsoft Developer Network (MSDN) Software Developer Kit*(SDK) and *Device Driver Developer Kit* (DDK).

The **DeviceIoControl** function sends a control code directly to a specified device driver, causing the corresponding device to complete 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.

IOCTL_SCSI_PASS_THROUGH
: Tape and medium changer.

IOCTL_SCSI_PASS_THROUGH_DIRECT
: Tape and medium changer.

IOCTL_STORAGE_RESERVE
: Tape and medium changer.

IOCTL_STORAGE_RELEASE
: Tape and medium changer.

| **IOCTL_STORAGE_PERSISTENT_RESERVE_IN**
| : Tape and medium changer.

| **IOCTL_STORAGE_PERSISTENT_RESERVE_OUT**
| : Tape and medium changer.

IOCTL_CHANGER_EXCHANGE_MEDIUM
: Medium changer not all changers.

IOCTL_CHANGER_GET_ELEMENT_STATUS
: Medium changer if bar code Reader then VolTags supported.

IOCTL_CHANGER_GET_PARAMETERS
: Medium changer.

IOCTL_CHANGER_GET_PRODUCT_DATA
: Medium changer.

IOCTL_CHANGER_GET_STATUS
: Medium changer.

IOCTL_CHANGER_INITIALIZE_ELEMENT_STATUS
: Medium changer with range not supported by all changers.

IOCTL_CHANGER_MOVE_MEDIUM
: Medium changer.

IOCTL_CHANGER_SET_ACCESS
: Medium changer for IE Port only and not for all changers.

IOCTL_CHANGER_SET_POSITION
: Medium changer only some devices support the transport object.

An example of the use of **SCSI Pass Through** is contained in the sample code **SPTI.C** 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

The Removable Storage Manager (RSM) must be stopped to use these **ioctl** commands. RSM can be stopped from **Computer Management (Local)** > **Services and Applications** > **Services** > **Removable Storage**.

IOCTL commands

Not all source or destination addresses, exchanges, moves, or operations are allowed for a particular IBM Medium Changer. The user must issue an **IOCTL_CHANGER_GET_PARAMETER** to determine the type of operations that are allowed by a specific changer device. Further information on allowable commands for a particular changer can be found in the IBM hardware reference for that device. It is recommended that the user have a copy of the hardware reference before any applications for the changer device are constructed.

IOCTL_CHANGER_EXCHANGE_MEDIUM

The media from the source element is moved to the first destination element. The medium that previously occupied the first destination element is moved to the second destination element (the second destination element might be the same as the source) by sending an **ExchangeMedium (0xA6)** 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 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 an **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 recalibrates 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 **PositionToElemen (0x2B)** SCSI command to the device. The input data is a structure of **CHANGER_SET_POSITION**.

Preempt reservation

A reservation can be preempted by issuing the appropriate IOCTL. The current reservation key is needed to successfully preempt the reservation. The current reservation key can be queried by issuing an **IOCTL_STORAGE_PERSISTENT_RESERVE_IN** IOCTL:

```
PERSISTENT_RESERVE_COMMAND prcmd = { 0 };
 PPRI_RESERVATION_LIST prsl = NULL;
 ULONG AdditionalLength = 0;
 DWORD BytesReturned = 0;
 INT iStatus = 0, i, j;

 UCHAR  *bufDataRead = (UCHAR *) malloc(SENSE_BUFFER_SIZE * 2);

 ZeroMemory(bufDataRead, SENSE_BUFFER_SIZE * 2);

 prcmd.Size = sizeof(PERSISTENT_RESERVE_COMMAND);

 prcmd.PR_IN.ServiceAction    = RESERVATION_ACTION_READ_KEYS;
 prcmd.PR_IN.AllocationLength = sizeof(PRI_REGISTRATION_LIST);

 for (i = 0; i < 2; i++) {
         if (0 == i)
                 AdditionalLength = sizeof(PRI_RESERVATION_LIST);

         iStatus = DeviceIoControl(tape,
                                   IOCTL_STORAGE_PERSISTENT_RESERVE_IN,
                                   &prcmd,
                                   prcmd.Size,
                                   bufDataRead,
                                   AdditionalLength,
                                   &BytesReturned,
                                   NULL);

         if (0 == iStatus) {
                 free(bufDataRead);
```

```
                return FALSE;
            }
            prsl = (PPRI_RESERVATION_LIST)bufDataRead;

            if (0 == i) {
               AdditionalLength  = (ULONG)((prsl->AdditionalLength[0] & 0xff) << 12);
               AdditionalLength |= (ULONG)((prsl->AdditionalLength[1] & 0xff) << 8);
               AdditionalLength |= (ULONG)((prsl->AdditionalLength[2] & 0xff) << 4);
               AdditionalLength |= (ULONG) (prsl->AdditionalLength[3] & 0xff);

               AdditionalLength += sizeof(PRI_RESERVATION_LIST);

               prcmd.PR_IN.AllocationLength = AdditionalLength;
            } else if (1 == i) {
               for (j = 0; (j * sizeof(PRI_RESERVATION_DESCRIPTOR)
               + sizeof(PRI_RESERVATION_LIST))<= AdditionalLength; j++) {
                    printf("\nReservation 0x%08x%08x being examined at
                    descriptor index %d.\n",
                   ((LARGE_INTEGER*)(prsl->Reservations[j].ReservationKey))
                    ->HighPart,
                   ((LARGE_INTEGER*)(prsl->Reservations[j].ReservationKey))
                    ->LowPart,
                       j);
                    }
            }
    }
```

When the reservation key is known, it can be preempted, meaning a new host can be the reservation holder, thus being able to interact with the target as needed.

```
 PRI_RESERVATION_DESCRIPTOR reservation = { 0 };
        PERSISTENT_RESERVE_COMMAND prcmd = { 0 };
        PRO_PARAMETER_LIST prolist = { 0 };
        DWORD BytesReturned = 0;
        INT iStatus = 0;

        UCHAR  bufDataRead[sizeof(PERSISTENT_RESERVE_COMMAND)
        + sizeof(PRO_PARAMETER_LIST)] = { 0 };

        prcmd.Size = sizeof(PERSISTENT_RESERVE_COMMAND) + sizeof(PRO_PARAMETER_LIST);

        prcmd.PR_OUT.ParameterList[1] = 0x18;
        prcmd.PR_OUT.ServiceAction = 0x3;
        prcmd.PR_OUT.Type = 0x3;

        query_reserve(tape, &reservation);

        RtlCopyMemory(prolist.ReservationKey, reservation.ReservationKey, 8);
        RtlCopyMemory(prolist.ServiceActionReservationKey,
        reservation.ReservationKey, 8);

        RtlCopyMemory(bufDataRead, &prcmd, sizeof(PERSISTENT_RESERVE_COMMAND));
        RtlCopyMemory(bufDataRead + sizeof(PERSISTENT_RESERVE_COMMAND),
        &prolist, sizeof(PRO_PARAMETER_LIST));

        iStatus = DeviceIoControl(tape,
                            IOCTL_STORAGE_PERSISTENT_RESERVE_OUT,
                            bufDataRead,
                            sizeof(bufDataRead),
                            bufDataRead,
                            sizeof(bufDataRead),
                            &BytesReturned,
                            NULL);
```

The query_reserve function can be implemented as explained earlier. Finally, if the reservation needs to be preempted on a different system than the original reservation holder, the OPEN_ALWAYS flag comes in handy. It allows the user to query the target's serial number, then queries the reservation key, and preempts the reservation.

Caution is advised when preempting reservations due to inherent risk of data loss if done incorrectly. Applications must make sure that they are clearing or preempting the appropriate reservation.

Vendor-specific (IBM) device IOCTLs for DeviceIoControl

The following descriptions are of the IBM vendor-specific ioctl requests for tape and changer.

```
/*
    This macro is defined in ntddk.h and devioctl.h
    #define CTL_CODE(DeviceType, Function, Method, Access) \
        (((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
    */
```

The following **ioctl** commands are supported by the **ibmtp.sys** driver through DeviceIoControl.

```
#define LB_ACCESS FILE_READ_ACCESS | FILE_WRITE_ACCESS
#define M_MTI(x) CTL_CODE(IOCTL_BASE+2,x,METHOD_BUFFERED, LB_ACCESS)
#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_LOG_SENSE10 CTL_CODE(IOCTL_TAPE_BASE, 0x0833,
METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_ENH_TAPE_LOG_SENSE10  CTL_CODE(IOCTL_TAPE_BASE, 0x0835, METHOD_BUFFERED,
FILE_READ_ACCESS )
#define IOCTL_TAPE_REPORT_MEDIA_DENSITY CTL_CODE(IOCTL_TAPE_BASE, 0x081e,
METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_TAPE_OBTAIN_MTDEVICE (M_MTI(16))
#define IOCTL_CREATE_PARTITION CTL_CODE(IOCTL_TAPE_BASE, 0x0826, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_QUERY_PARTITION CTL_CODE(IOCTL_TAPE_BASE, 0x0825, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_SET_ACTIVE_PARTITION CTL_CODE(IOCTL_TAPE_BASE, 0x0827, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_QUERY_DATA_SAFE_MODE CTL_CODE(IOCTL_TAPE_BASE, 0x0823, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_SET_DATA_SAFE_MODE CTL_CODE(IOCTL_TAPE_BASE, 0x0824, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_ALLOW_DATA_OVERWRITE CTL_CODE(IOCTL_TAPE_BASE, 0x0828, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define IOCTL_SET_PEW_SIZE
CTL_CODE(IOCTL_TAPE_BASE, 0x082C, METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_QUERY_PEW_SIZE
CTL_CODE(IOCTL_TAPE_BASE, 0x082B, METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_VERIFY_TAPE_DATA
CTL_CODE(IOCTL_TAPE_BASE, 0x082A, METHOD_BUFFERED, FILE_READ_ACCESS )
#define IOCTL_QUERY_RAO_INFO   CTL_CODE(IOCTL_TAPE_BASE, 0x082E, METHOD_BUFFERED,
FILE_READ_ACCESS )
#define IOCTL_GENERATE_RAO     CTL_CODE(IOCTL_TAPE_BASE, 0x082F, METHOD_BUFFERED,
FILE_READ_ACCESS )
#define IOCTL_RECEIVE_RAO      CTL_CODE(IOCTL_TAPE_BASE, 0x0834, METHOD_BUFFERED,
FILE_READ_ACCESS )
```

IOCTL_TAPE_OBTAIN_SENSE

Issue this command after an error occurs to obtain sense information that is associated with the most recent error. To guarantee that the application can obtain sense information that is associated with an error, the application must issue this command before other commands to the device are issued. Subsequent operations (other than **IOCTL_TAPE_OBTAIN_SENSE**) reset the sense data field before the operation is run.

This IOCTL is only available for the tape path.

The following output structure is completed by the **IOCTL_TAPE_OBTAIN_SENSE** command that is passed by the caller.

```
#define MAG_SENSE_BUFFER_SIZE 96 /* Default request sense buffer size for \
                                                   Windows 200x */

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 completed 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 this command to reset all log pages follows.

```
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 that is returned is formatted according to the IBM Magstar hardware reference.

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_LOG_SENSE10

Issue this command to obtain the log data of the requested log page/subpage from IBM Magstar tape device. The data returned is formatted according to the IBM Magstar hardware reference. This IOCTL is only for the tape path.

The following input/output structure is used by the **IOCTL_TAPE_LOG_SENSE10** command.

```
#define MAX_LOG_SENSE 1024 // Maximum number of bytes the command will return
typedef struct _TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE{
  UCHAR         PageCode;        /* [IN] Log sense page       */
  UCHAR         SubPageCode;     /* [IN] Log sense subpage    */
  UCHAR         PC;              /* [IN] PC bit to be consistent with
                                    previous Log Sense IOCTL   */
  UCHAR         reserved[2];     /* unused                    */
  ULONG         PageLength;      /* [OUT] number of valid bytes in data
                                    (log_page_header_size+page_length)*/
  ULONG         parm_pointer;    /* [IN]  specific parameter number at which
                                    the data begins                 */
  CHAR          LogData[MAX_LOG_SENSE_DATA];   /* [OUT] log sense data */
} TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE, *PTAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE;
```

An example of the **IOCTL_TAPE_LOG_SENSE10** command is

```
DWORD cb;
TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE logsense;
logsense.PageCode=0x10;
logsense.PageCode=0x01;
logsense.PC = 1;
DeviceIoControl(hDevice,
IOCTL_TAPE_LOG_SENSE10,
&logsense, (long)sizeof(TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE,
&logsense, (long)sizeof(TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE,
&cb, (LPOVERLAPPED) NULL);
```

IOCTL_ENH_TAPE_LOG_SENSE10

Issue this command to obtain the log data of the requested log page/subpage from IBM TotalStorage tape device. The data that is returned is formatted according to the IBM TotalStorage hardware reference. This IOCTL is only for the tape path and is enhanced so the application can set the page length and provide the buffer enough to get the data back. The following input/output structure is used by the **IOCTL_ENH_TAPE_LOG_SENSE10** command.

```
typedef struct _ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE{
  UCHAR         PageCode;               /* [IN] Log sense page */
  UCHAR         SubPageCode;            /* [IN] Log sense subpage */
  UCHAR         PC;                     /* [IN] PC bit */
  UCHAR         reserved[5];            /* unused */
  ULONG         Length;                 /* [IN][OUT] number of valid bytes in data  */
                                        /* (log_page_header_size+page_length)      */
  ULONG         parm_pointer;           /* [IN]  specific parameter number at which */
                                        /* the data begins                  */
  CHAR          LogData[1];             /* [IN] log sense buffer allocated by */
                                        /* application                      */
                                        /* [OUT] log sense data             */
} ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE, *PENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE;
```

An example of the **IOCTL_ENH_TAPE_LOG_SENSE10** command is

```
DWORD cb;
char *logsense;
int pageLength = 256;
long lsize = sizeof(ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE) - sizeof
(CHAR) /*LogData[1]*/ + pageLength

logsense = malloc (lsize);
(ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE)logsense->PageCode=0x10;
(ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE)logsense->SubPageCode=0x01;
(ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE)logsense->PC = 1;
(ENH_TAPE_LOG_SENSE_PARAMETERS_WITH_SUBPAGE)logsense->Length = pageLength;
DeviceIoControl(hDevice,
```

```
                IOCTL_ENH_TAPE_LOG_SENSE10,
                &logsense, (long)lsize,
                &logsense, (long)lsize,
                &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 fails. This IOCTL is only for the tape path.

The following output structure is completed 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);
```

IOCTL_TAPE_OBTAIN_MTDEVICE

Issue this command to obtain the device number of a 3590 TotalStorage device in an IBM 3494 Enterprise Tape Library. An error is returned if it is issued against a 3570 drive.

The following output structure is filled in by the **IOCTL_TAPE_OBTAIN_MTDEVICE** command.

```
typedef ULONG TAPE_OBTAIN_MTDEVICE, *PTAPE_OBTAIN_MTDEVICE;
```

An example of the **IOCTL_TAPE_OBTAIN_MTDEVICE** command is

```
int   *rc_ptr
DWORD cb;
TAPE_OBTAIN_MTDEVICE mt_device;

 *rc_ptr = DeviceIoControl(gp->ddHandle0,
                           IOCTL_TAPE_OBTAIN_MTDEVICE,
                           NULL,
                           0,
                           &mt_device,
                           (long)sizeof(TAPE_OBTAIN_MTDEVICE),
                           &cb,
                           (LPOVERLAPPED) NULL);
```

```
 if(*rc_ptr)
    printf(fp, "\nntutil MTDevice Info : %x\n\n", mt_device);
 else
  /* Error handling code */
```

IOCTL_TAPE_GET_DENSITY

The IOCTL code for **IOCTL_TAPE_GET_DENSITY** is defined as follows.

```
#define IOCTL_TAPE_GET_DENSITY \
CTL_CODE(IOCTL_TAPE_BASE, 0x000c, METHOD_BUFFERED, \
 FILE_READ_ACCESS | FILE_WRITE_ACCESS).
```

The IOCTL reports density for supported devices by using the following structure.

```
typedef struct _TAPE_DENSITY
{
    UCHAR   ucDensityCode;
    UCHAR   ucDefaultDensity;
    UCHAR   ucPendingDensity;
} TAPE_DENSITY, *PTAPE_DENSITY;
```

An example of the **IOCTL_TAPE_GET_DENSITY** command is

```
TAPE_DENSITY tape_density = {0};

rc = DeviceIoControl(hDevice,
IOCTL_TAPE_GET_DENSITY,
NULL,
0,
&tape_density,
sizeof(TAPE_DENSITY),
&cb,
(LPOVERLAPPED) NULL);
```

IOCTL_TAPE_SET_DENSITY

The IOCTL code for **IOCTL_TAPE_SET_DENSITY** is defined as follows.

```
#define IOCTL_TAPE_SET_DENSITY \
CTL_CODE(IOCTL_TAPE_BASE, 0x000d, METHOD_BUFFERED, \
 FILE_READ_ACCESS | FILE_WRITE_ACCESS)
```

The IOCTL sets density for supported devices by using the following structure.

```
typedef struct _TAPE_DENSITY
{
    UCHAR   ucDensityCode;
    UCHAR   ucDefaultDensity;
    UCHAR   ucPendingDensity;
} TAPE_DENSITY, *PTAPE_DENSITY;
```

`ucDensityCode` is ignored. `ucDefaultDensity` and `ucPendingDensity` are set by using the tape drive's mode page 0x25. Caution must be taken when this IOCTL is issued. An incorrect tape density might lead to data corruption.

An example of the **IOCTL_TAPE_SET_DENSITY** command is

```
TAPE_DENSITY tape_density;

// Modify fields of tape_density.  For details, see the SCSI specification
// for your hardware.

rc = DeviceIoControl(hDevice,
IOCTL_TAPE_SET_DENSITY,
&tape_density,
sizeof(TAPE_DENSITY),
```

```
NULL,
0,
&cb,
(LPOVERLAPPED) NULL);
```

IOCTL_TAPE_GET_ENCRYPTION_STATE

This IOCTL command queries the drive's encryption method and state.

The IOCTL code for **IOCTL_TAPE_GET_ENCRYPTION_STATE** is defined as follows.

```
#define IOCTL_TAPE_GET_ENCRYPTION_STATE CTL_CODE(IOCTL_TAPE_BASE, 0x0820,
    METHOD_BUFFERED, FILE_READ_ACCESS )
```

The IOCTL gets encryption states for supported devices by using the following structure.

```
typedef struct _ENCRYPTION_STATUS
{
    UCHAR ucEncryptionCapable;   /* (1)Set this field as a boolean based on
                              the capability of the drive  */
    UCHAR ucEncryptionMethod;    /* (2)Set this field to one of the
                              defines METHOD_* below             */
    UCHAR ucEncryptionState;     /* (3)Set this field to one of the
                              #defines STATE_* below */
    UCHAR aucReserved[13];
} ENCRYPTION_STATUS, *PENCRYPTION_STATUS;
```

`#defines` for METHOD.

```
#define ENCRYPTION_METHOD_NONE      0 /* Only used in
                                    GET_ENCRYPTION_STATE */
#define ENCRYPTION_METHOD_LIBRARY    1 /* Only used in
                                    GET_ENCRYPTION_STATE */
#define ENCRYPTION_ METHOD_SYSTEM    2 /* Only used in
                                    GET_ENCRYPTION_STATE */
#define ENCRYPTION_ METHOD_APPLICATION  3 /* Only used in
                                       GET_ENCRYPTION_STATE */
#define ENCRYPTION_ METHOD_CUSTOM  4 /* Only used in
                                    GET_ENCRYPTION_STATE */
#define ENCRYPTION_ METHOD_UNKNOWN  5 /* Only used in
                                    GET_ENCRYPTION_STATE */
```

`#defines` for STATE.

```
#define  ENCRYPTION_STATE_OFF 0  /* Used in GET/SET_ENCRYPTION_STATE */
#define  ENCRYPTION_STATE_ON 1  /* Used in GET/SET_ENCRYPTION_STATE */
#define  ENCRYPTION_STATE_NA 2  /* Only used in GET_ENCRYPTION_STATE*/
```

An example of the **IOCTL_TAPE_GET_ENCRYPTION_STATE** command is

```
ENCRYPTION_STATUS scEncryptStat;
DeviceIoControl(hDevice,
                IOCTL_TAPE_GET_ENCRYPTION_STATE,
                &scEncryptStat,
                sizeof(ENCRYPTION_STATUS),
                &scEncryptStat,
                sizeof(ENCRYPTION_STATUS),
                ,&cb
                (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_SET_ENCRYPTION_STATE

This IOCTL command allows only set encryption state for application-managed encryption.

Note: On unload, some drive settings might be reset to default. To set the encryption state, the application must issue this IOCTL after a tape is loaded and at BOP.

The data structure that is used for this IOCTL is the same as for **IOCTL_GET_ENCRYPTION_STATE**.

```
#define IOCTL_TAPE_SET_ENCRYPTION_STATE CTL_CODE(IOCTL_TAPE_BASE, 0x0821,
    METHOD_BUFFERED,
    FILE_READ_ACCESS | FILE_WRITE_ACCESS )
```

An example of the **IOCTL_TAPE_SET_ENCRYPTION_STATE** command is

```
ENCRYPTION_STATUS scEncryptStat;
DeviceIoControl(hDevice,
                IOCTL_TAPE_SET_ENCRYPTION_STATE,
                &scEncryptStat,
                sizeof(ENCRYPTION_STATUS),
                ,&scEncryptStat
                sizeof(ENCRYPTION_STATUS),
                &cb,
                (LPOVERLAPPED) NULL);
```

IOCTL_TAPE_SET_DATA_KEY

This IOCTL command is used to set the data key only for application-managed encryption.

The IOCTL sets data keys for supported devices by using the following structure.

```
#define IOCTL_TAPE_SET_DATA_KEY CTL_CODE(IOCTL_TAPE_BASE, 0x0822,
    METHOD_BUFFERED,
    FILE_READ_ACCESS | FILE_WRITE_ACCESS )
```

```
#define DATA_KEY_INDEX_LENGTH      12
#define DATA_KEY_RESERVED1_LENGTH  15
#define DATA_KEY_LENGTH            32
#define DATA_KEY_RESERVED2_LENGTH  48
typedef struct _DATA_KEY
{
    UCHAR aucDataKeyIndex[DATA_KEY_INDEX_LENGTH];
    UCHAR ucDataKeyIndexLength;
    UCHAR aucReserved1[DATA_KEY_RESERVED1_LENGTH];
    UCHAR aucDataKey[DATA_KEY_LENGTH];
    UCHAR aucReserved2[DATA_KEY_RESERVED2_LENGTH];
} DATA_KEY, *PDATA_KEY;
```

An example of the **IOCTL_TAPE_SET_DATA_KEY** command is

```
DATA_KEY scDataKey;
/* fill in your data key and data key length, then issue DeviceIoControl */
DeviceIoControl(hDevice,
                IOCTL_TAPE_SET_DATA_KEY,
                &scDataKey,
                sizeof(DATA_KEY),
                &scDataKey,
                sizeof(DATA_KEY),
                &cb,
                (LPOVERLAPPED) NULL);
```

IOCTL_CREATE_PARTITION

This command is used to create one or more partitions on the tape. The tape must be at BOT (partition 0 logical block id 0) before the command is issued or it fails. The application must either issue this **IOCTL_CREATE_PARTITION** after a tape is initially loaded or issue the **IOCTL_SET_ACTIVE_PARTITION** with the **partition_number** and **logical_clock_id** fields that are set to 0 first.

The structure that is used to create partitions is

```
#define IOCTL_CREATE_PARTITION         CTL_CODE(IOCTL_TAPE_BASE, 0x0826,
METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
typedef struct _TAPE_PARTITION{
  UCHAR  type;                  /* Type of tape partition to create      */
  UCHAR  number_of_partitions;  /* Number of partitions to create        */
  UCHAR  size_unit;             /* IDP size unit of partition sizes below */
  USHORT size[MAX_PARTITIONS];  /* Array of partition sizes in size units */
                                /* for each partition, 0 to (number - 1)  */
                                /* Size can not be 0 and one partition    */
                                /* size must be 0xFFFF to use the         */
                                /* remaining capacity on the tape.        */
 UCHAR partition_method;        /* partitioning type                      */
 char reserved [31];
} TAPE_PARTITION, *PTAPE_PARTITION;
```

An example of the **IOCTL_CREATE_PARTITION** command is

```
 DWORD cb;
 TAPE_PARTITION tape_partition
        ...
DeviceIoControl(gp->ddHandle0,
               IOCTL_CREATE_PARTITION,
               &tape_partition,
               (long)sizeof(TAPE_PARTITION),
                NULL,
                0,
                &cb,
               (LPOVERLAPPED) NULL);
```

IOCTL_QUERY_PARTITION

This command returns partition information for the current loaded tape.

The following output structure is completed by the **IOCTL_QUERY_PARTITION** command.

```
#define IOCTL_QUERY_PARTITION          CTL_CODE(IOCTL_TAPE_BASE, 0x0825,
 METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define MAX_PARTITIONS 255
typedef struct _QUERY_PARTITION{
  UCHAR  max_partitions;         /* Max number of supported partitions     */
  UCHAR  active_partition;       /* current active partition on tape       */
  UCHAR  number_of_partitions;   /* Number of partitions from 1 to max     */
  UCHAR  size_unit;              /* Size unit of partition sizes below     */
  USHORT size[MAX_PARTITIONS];   /* Array of partition sizes in size units */
                                 /* for each partition, 0 to (number - 1)  */
  UCHAR  partition_method;       /* partitioning type                      */
char reserved [31];
} QUERY_PARTITION, *PQUERY_PARTITION;
```

An example of the **IOCTL_QUERY_PARTITION** command is

```
DWORD cb;
QUERY_PARTITION tape_query_partition;
DeviceIoControl(gp->ddHandle0,
               IOCTL_QUERY_PARTITION,
               NULL,
               0,
```

```
              &tape_query_partition,
              (long)sizeof(QUERY_PARTITION),
              &cb,
              (LPOVERLAPPED) NULL);
```

IOCTL_SET_ACTIVE_PARTITION

This command is used to set the current active partition that is used on tape and locate to a specific logical block id within the partition. If the logical block id is 0, the tape is positioned at BOP. If the partition number specified is 0 along with a logical block id 0, the tape is positioned at both BOP and BOT.

The structure for **IOCTL_SET_ACTIVE_PARTITION** command is

```
#define IOCTL_SET_ACTIVE_PARTITION      CTL_CODE(IOCTL_TAPE_BASE, 0x0827,
METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
typedef struct _SET_ACTIVE_PARTITION{
  UCHAR partition_number;                 /* Partition number 0-n to change to */
  ULONGLONG logical_block_id;  /* Blockid to locate to within partition */
  char reserved[32];
} SET_ACTIVE_PARTITION, *PSET_ACTIVE_PARTITION;
```

An example of the **IOCTL_SET_ACTIVE_PARTITION** command is

```
DWORD cb;
SET_ACTIVE_PARTITION set_partition;
...
DeviceIoControl(gp->ddHandle0,
              IOCTL_SET_ACTIVE_PARTITION,
              &set_partition,
              (long)sizeof(SET_ACTIVE_PARTITION),
              NULL,
              0,
              &cb,
              (LPOVERLAPPED) NULL);
```

IOCTL_QUERY_DATA_SAFE_MODE

This command reports if the Data Safe Mode is enabled or disabled.

The following output structure is completed by the **IOCTL_QUERY_DATA_SAFE_MODE** command.

```
#define IOCTL_QUERY_DATA_SAFE_MODE      CTL_CODE(IOCTL_TAPE_BASE, 0x0823,
 METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
typedef struct _DATA_SAFE_MODE{
  ULONG value;
} DATA_SAFE_MODE, *PDATA_SAFE_MODE;
```

An example of the **IOCTL_QUERY_DATA_SAFE_MODE** command is

```
DWORD cb;
DATA_SAFE_MODE tapeDataSafeMode;
DeviceIoControl(gp->ddHandle0,
              IOCTL_QUERY_DATA_SAFE_MODE,
              NULL,
              0,
              &tapeDataSafeMode,
              (long)sizeof(DATA_SAFE_MODE),
              &cb,
             (LPOVERLAPPED) NULL);
```

IOCTL_SET_DATA_SAFE_MODE

This command enables or disables Data Safe Mode.

The structure that is used to enable or disable Data Safe Mode is the same as **IOCTL_QUERY_DATA_SAFE_MODE**.

An example of the **IOCTL_SET_DATA_SAFE_MODE** command is

```
#define IOCTL_SET_DATA_SAFE_MODE        CTL_CODE(IOCTL_TAPE_BASE, 0x0824,
METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
DATA_SAFE_MODE tapeDataSafeMode;
...
DeviceIoControl(gp->ddHandle0,
               IOCTL_SET_DATA_SAFE_MODE,
               &tapeDataSafeMode,
               (long)sizeof(DATA_SAFE_MODE),
                NULL,
                0,
                &cb,
                (LPOVERLAPPED) NULL);
```

IOCTL_ALLOW_DATA_OVERWRITE

This command allows previously written data on the tape to be overwritten. This action happens when append only mode is enabled on the drive with either a write type command or a format command is allowed on the **IOCTL_CREATE_PARTITION**. Before this IOCTL is issued, the application must locate to the partition number and logical block id within the partition where the data overwrite or format occurs.

The data structure that is used for **IOCTL_ALLOW_DATA_OVERWRITE** to enable or disable is

```
#define IOCTL_ALLOW_DATA_OVERWRITE      CTL_CODE(IOCTL_TAPE_BASE, 0x0828,
METHOD_BUFFERED,
 FILE_READ_ACCESS | FILE_WRITE_ACCESS )
typedef struct ALLOW_DATA_OVERWRITE{
  UCHAR partition_number;            /* Partition number 0-n to overwrite  */
 ULONGULONG logical_block_id;  /* Blockid to overwrite to within partition */
  UCHAR allow_format_overwrite;      /* allow format if in data safe mode  */
  UCHAR reserved[32];
} ALLOW_DATA_OVERWRITE, *PALLOW_DATA_OVERWRITE;
```

An example of the **IOCTL_ALLOW_DATA_OVERWRITE** command is

```
ALLOW_DATA_OVERWRITE tapeAllowDataOverwrite;
...
DeviceIoControl(gp->ddHandle0,
               IOCTL_ALLOW_DATA_OVERWRITE,
               &tapeAllowDataOverwrite,
               (long)sizeof(ALLOW_DATA_OVERWRITE),
               NULL,
               0,
               &cb,
               (LPOVERLAPPED) NULL);
```

IOCTL_READ_TAPE_POSITION

This command returns Position data in either the short, long, or extended form. The type of data to return is specified by setting the **data_format** field to either **RP_SHORT_FORM**, **RP_LONG_FORM**, or **RP_EXTENDED_FORM**.

The data structures that are used with this IOCTL are

```
#define IOCTL_READ_TAPE_POSITION         CTL_CODE(IOCTL_TAPE_BASE, 0x0829,
METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define RP_SHORT_FORM          0x00
#define RP_LONG_FORM           0x06
#define RP_EXTENDED_FORM       0x08

typedef struct _SHORT_DATA_FORMAT {
  UCHAR bop:1,     /* beginning of partition */
        eop:1,     /* end of partition */
        locu:1,    /* 1 means num_buffer_logical_obj field is unknown */
        bycu:1,    /* 1 means the num_buffer_bytes field is unknown */
        rsvd :1,
        lolu:1,    /* 1 means the first and last logical obj position fields
                      are unknown */
        perr: 1,   /* 1 means the position fields have overflowed and
                      cannot be reported */
        bpew :1;   /*  beyond programmable early warning */
  UCHAR active_partition;   /* current active partition */
  UCHAR reserved[2];
  UCHAR first_logical_obj_position[4]; /* current logical object position */
  UCHAR last_logical_obj_position[4];  /* next logical object to be
                                          transferred to tape */
  UCHAR num_buffer_logical_obj[4];     /* number of logical objects in buffer */
  UCHAR num_buffer_bytes[4];           /* number of bytes in buffer */
  UCHAR reserved1;                     /* instead of the commented reserved1 */
} SHORT_DATA_FORMAT, *PSHORT_DATA_FORMAT;

typedef struct _LONG_DATA_FORMAT {
  UCHAR bop:1, /* beginning of partition */
        eop:1, /* end of partition */
        rsvd1:2,
        mpu:1, /* 1 means the logical file id field in unknown */
        lonu:1,/* 1 means either the partition number or logical obj number field
                  are unknown */
        rsvd2:1,
        bpew :1;/*  beyond programmable early warning */
  CHAR  reserved[6];
  UCHAR active_partition;     /* current active partition */
  UCHAR logical_obj_number[8];/* current logical object position */
  UCHAR logical_file_id[8];   /* number of filemarks from bop and
                                 current logical position */
  UCHAR obsolete[8];
}LONG_DATA_FORMAT, *PLONG_DATA_FORMAT;

typedef struct _EXTENDED_DATA_FORMAT {
  UCHAR bop:1,  /* beginning of partition */
        eop:1,  /* end of partition */
        locu:1, /* 1 means num_buffer_logical_obj field is unknown */
        bycu:1, /* 1 means the num_buffer_bytes field is unknown */
        rsvd :1,
        lolu:1, /* 1 means the first and last logical obj position fields
                   are unknown */
        perr: 1,/* 1 means the position fields have overflowed and
                   can not be reported */
        bpew :1;/*  beyond programmable early warning */
  UCHAR  active_partition;            /* current active partition */
  UCHAR  additional_length[2];
  UCHAR  num_buffer_logical_obj[4];    /* number of logical objects in buffer */
  UCHAR  first_logical_obj_position[8];/* current logical object position */
  UCHAR  last_logical_obj_position[8]; /* next logical object to be
                                          transferred to tape */
  UCHAR  num_buffer_bytes[8];          /* number of bytes in buffer */
  UCHAR reserved;
} EXTENDED_DATA_FORMAT, *PEXTENDED_DATA_FORMAT;

typedef struct READ_TAPE_POSITION{
```

```
  UCHAR data_format; /* Specifies the return data format either
                        short, long or extended*/
  union
  {
     SHORT_DATA_FORMAT rp_short;
     LONG_DATA_FORMAT rp_long;
     EXTENDED_DATA_FORMAT rp_extended;
     UCHAR reserved[64];
  } rp_data;
} READ_TAPE_POSITION, *PREAD_TAPE_POSITION;
```

An example of the **READ_TAPE_POSITION** command is

```
DWORD cb;
READ_TAPE_POSITION tapePosition;
*rc_ptr = DeviceIoControl(gp->ddHandle0,
                         IOCTL_READ_TAPE_POSITION,
                         &tapePosition,
                         (long)sizeof(READ_TAPE_POSITION),
                         &tapePosition,
                         (long)sizeof(READ_TAPE_POSITION),
                         &cb,
                         (LPOVERLAPPED) NULL);
```

IOCTL_SET_TAPE_POSITION

This command is used to position the tape in the current active partition to either a logical block id or logical filemark. The **logical_id_type** field in the IOCTL structure specifies either a logical block or logical filemark.

The data structure that is used with this IOCTL is

```
#define IOCTL_SET_TAPE_POSITION_LOCATE16 CTL_CODE(IOCTL_TAPE_BASE, 0x0830,
METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
#define LOGICAL_ID_BLOCK_TYPE   0x00
#define LOGICAL_ID_FILE_TYPE    0x01

typedef struct _SET_TAPE_POSITION{
  UCHAR     logical_id_type;    /* Block or file as defined above */
  ULONGLONG logical_id;         /* logical object or logical file to position to */
  UCHAR     reserved[32];
} SET_TAPE_POSITION, *PSET_TAPE_POSITION;
```

An example of the **SET_TAPE_POSITION** command is

```
DWORD cb;
SET_TAPE_POSITION tapePosition;

      *rc_ptr = DeviceIoControl(gp->ddHandle0,
                  IOCTL_SET_TAPE_POSITION_LOCATE16,
                  &tapePosition,
                  (long)sizeof(SET_TAPE_POSITION)
                   NULL,
                   0,
                   &cb,
                  (LPOVERLAPPED) NULL);
```

IOCTL_QUERY_LBP

This command returns logical block protection information. The following output structure is completed by the **IOCTL_QUERY_LBP** command.

```
#define IOCTL_QUERY_LBP    CTL_CODE(IOCTL_TAPE_BASE, 0x0831,
METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
typedef struct _LOGICAL_BLOCK_PROTECTION {
```

```
   UCHAR lbp_capable;  /* [OUTPUT] the capability of lbp for  QUERY ioctl only */
   UCHAR lbp_method;   /* lbp method used for QUERY [OUTPUT] and SET [INPUT] ioctls */
#define LBP_DISABLE 0x00
#define REED_SOLOMON_CRC 0x01
   UCHAR lbp_info_length; /* lbp info length for QUERY [OUTPUT] and SET [INPUT]
                             ioctls */
   UCHAR lbp_w;        /* protection info included in write data */
                       /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   UCHAR lbp_r;        /* protection info included in read data */
                       /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   UCHAR rbdp;         /* protection info included in recover buffer data */
                       /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   UCHAR reserved[26];
}LOGICAL_BLOCK_PROTECTION, *PLOGICAL_BLOCK_PROTECTION;
```

An example of the **IOCTL_QUERY_LBP** command is

```
*rc_ptr = DeviceIoControl(gp->ddHandle0,
                         IOCTL_QUERY_LBP,
                         NULL,
                         0,
                         &tape_query_LBP,
                         (long)sizeof(LOGICAL_BLOCK_PROTECTION),
                         &cb,
                         (LPOVERLAPPED) NULL);
```

IOCTL_SET_LBP

This command sets logical block protection information. The following input structure is sent to the **IOCTL_SET_LBP** command.

```
#define IOCTL_SET_LBP      CTL_CODE(IOCTL_TAPE_BASE, 0x0832,
METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS )
typedef struct _LOGICAL_BLOCK_PROTECTION {
   UCHAR lbp_capable;    /* [OUTPUT] the capability of lbp for  QUERY ioctl only */
   UCHAR lbp_method;     /* lbp method used for QUERY [OUTPUT] and SET [INPUT]
                            ioctls */
#define LBP_DISABLE 0x00
#define REED_SOLOMON_CRC 0x01
   UCHAR lbp_info_length; /* lbp info length for QUERY [OUTPUT] and SET [INPUT]
                             ioctls */
   UCHAR lbp_w;          /* protection info included in write data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   UCHAR lbp_r;          /* protection info included in read data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   UCHAR rbdp;           /* protection info included in recover buffer data */
                         /* a boolean for QUERY [OUTPUT] and SET [INPUT] ioctls */
   UCHAR reserved[26];
}LOGICAL_BLOCK_PROTECTION, *PLOGICAL_BLOCK_PROTECTION;
```

An example of the **IOCTL_SET_LBP** command is

```
*rc_ptr = DeviceIoControl(gp->ddHandle0,
IOCTL_SET_LBP,
&tape_set_LBP,
(long)sizeof(LOGICAL_BLOCK_PROTECTION),
NULL,
0,
&cb,
LPOVERLAPPED) NULL);
```

IOCTL_SET_PEW_SIZE

This command is used to set Programmable Early Warning size.

```
#define IOCTL_SET_PEW_SIZE
   CTL_CODE(IOCTL_TAPE_BASE, 0x082C, METHOD_BUFFERED, FILE_READ_ACCESS )
```

The structure that is used to set PEW size is

```
typedef struct _PEW_SIZE{
  USHORT value;
} PEW_SIZE, *PPEW_SIZE;
```

An example of the **IOCTL_SET_PEW_SIZE** command is

```
DWORD cb;
PEW_SIZE pew_size;
...
DeviceIoControl(gp->ddHandle0,
IOCTL_SET_PEW_SIZE,
&pew_size, (long)sizeof(PEW_SIZE),
NULL,
0,
&cb,
(LPOVERLAPPED) NULL);
```

IOCTL_QUERY_PEW_SIZE

This command is used to query Programmable Early Warning size.

```
#define IOCTL_QUERY_PEW_SIZE
   CTL_CODE(IOCTL_TAPE_BASE, 0x082B, METHOD_BUFFERED, FILE_READ_ACCESS )
```

The structure that is used to query PEW size is

```
typedef struct _PEW_SIZE{
  USHORT value;
} PEW_SIZE, *PPEW_SIZE;
```

An example of the **IOCTL_QUERY_PEW_SIZE** command is

```
DWORD cb;
PEW_SIZE pew_size;
...
DeviceIoControl(gp->ddHandle0,
IOCTL_QUERY_PEW_SIZE,
NULL,
0,
&pew_size,
(long)sizeof(PEW_SIZE),
&cb,
(LPOVERLAPPED) NULL);
```

IOCTL_VERIFY_TAPE_DATA

This command is used to verify tape data. It uses the drive's error detection and correction hardware to determine whether it can be recovered from the tape. It also checks whether the protection information is present and validates correctly on logical block on the medium. It returns a failure or a success.

```
#define IOCTL_VERIFY_TAPE_DATA
  CTL_CODE(IOCTL_TAPE_BASE, 0x082A, METHOD_BUFFERED, FILE_READ_ACCESS )
```

The structure that is used to verify tape data is

```
typedef struct _VERIFY_DATA {
    UCHAR reserved : 2;  /* Reserved                                         */
    UCHAR vte: 1;        /* [IN] verify to end-of-data                       */
    UCHAR vlbpm: 1;      /* [IN] verify logical block protection information  */
    UCHAR vbf: 1;        /* [IN] verify by filemarks                         */
    UCHAR immed: 1;      /* [IN] return SCSI status immediately              */
    UCHAR bytcmp: 1;     /*  No use currently                                */
    UCHAR fixed: 1;      /* [IN] set Fixed bit to verify the length of
```

```
                            each logical block                              */
    UCHAR reseved[15];
    ULONG verify_length; /* [IN] amount of data to be verified              */
}VERIFY_DATA, *PVERIFY_DATA;
```

An example of the **IOCTL_VERIFY_DATA** command is

```
DWORD cb;
VERIFY_DATA verify_data;
...
DeviceIoControl(gp->ddHandle0,
IOCTL_VERIFY_TAPE_DATA,
&verify_data,
sizeof(VERIFY_DATA),
NULL,
0,
&cb,
(LPOVERLAPPED) NULL);
```

IOCTL_QUERY_RAO_INFO

This command is used to query the maximum UDS number and UDS size before the **Recommended Access Order** list is generated and received (TS1140 or later). The structure for the **IOCTL_QUERY_RAO_INFO** command is

```
#define UDS_WITHOUT_GEOMETRY 0
#define UDS_WITH_GEOMETRY    1

typedef struct _QUERY_RAO_INF{
    CHAR uds_type;                          /*[IN] 0: UDS_WITHOUT_GEOMETRY
                                                   1: UDS_WITH_GEOMETRY          */
    CHAR reserved[7];USHORT max_uds_number;  /* [OUT] Max UDS number supported
                                              from drive                         */
    USHORT max_uds_size;                    /* [OUT] Max single UDS size supported
                                              from drive in bytes                */
    USHORT max_host_uds_number;             /* [OUT] Max UDS number supported from
                                              driver                             */
} QUERY_RAO_INFO, *PQUERY_RAO_INFO;
```

An example of the **IOCTL_QUERY_RAO_INFO** command is

```
QUERY_RAO_INFO qRAO;
...
qRAO.uds_type = udstype; //UDS_WITHOUT_GEOMETRY or  UDS_WITH_GEOMETRY
*rc_ptr = DeviceIoControl(hDevice,
                                 IOCTL_QUERY_RAO_INFO,&
                                 qRAO,
                                 sizeof(QUERY_RAO_INFO),
                                 &qRAO,
                                 sizeof(QUERY_RAO_INFO),
                                 &cb,
                                 (LPOVERLAPPED) NULL);
```

IOCTL_GENERATE_RAO

This command is used to generate a **Recommended Access Order** list (TS1140 or later). The UDS list is required as input. Use **USD_DESCRIPTOR** to build this list. The structure for the **IOCTL_GENERATE_RAO** command is

```
#define UDS_WITHOUT_GEOMETRY 0
#define UDS_WITH_GEOMETRY    1

typedef struct _GENERATE_RAO{
    CHAR process;            /* [IN] Requested process to generate RAO list       */
                             /*      0: no reorder UDS and no calculate locate time */
                             /*         (not currently supported by the drive)     */
                             /*      1: no reorder UDS but calculate locate time   */
```

```
                         /*      2: reorder UDS and calculate locate time      */
    CHAR uds_type;       /* [IN] 0: UDS_WITHOUT_GEOMETRY */
                         /*      1: UDS_WITH_GEOMETRY */
    CHAR reserved1[2];
    ULONG grao_list_length; /* [IN] The data length is allocated for GRAO list.  */
    CHAR reserved2[8];
    CHAR grao_list[1];   /* [IN] the pointer is allocated to the size          */
                         /*      of grao_list_leng                              */
                         /*      (uds_number * sizeof(struct grao_uds_desc)
                         /*      +sizeof(struct grao_list_header))              */
                         /*      and contains the data of GRAO parameter list. */
                         /*      The uds number isn't larger                    */
                         /*      than max_host_uds_number in QUERY_RAO ioctl.  */

} GENERATE_RAO, *PGENERATE_RAO;

typedef struct _UDS_DESCRIPTOR{
     CHAR descLength[2];
     CHAR reserved[3];
     CHAR UDSName[10];
     CHAR PartNum;
     CHAR beginningLOI[8];
     CHAR endingLOI[8];
}UDS_DESCRIPTOR, *PUDS_DESCRIPTOR;
```

An example of the **IOCTL_GENERATE_RAO** command is

```
# UDS_SIZE 32
# HEADER_SIZE 8

char *pGRAO;
...
long lGRAOsize = sizeof(GENERATE_RAO)-sizeof(CHAR)/*grao_list[1]
*/ + udsamount*UDS_SIZE + HEADER_SIZE;
pGRAO = malloc (lGRAOsize);
...
((PGRAO)pGRAO)->process = process;
((PGRAO)pGRAO)->uds_type = udstype;
((PGRAO)pGRAO)->grao_list_length = udsamount*UDS_SIZE + HEADER_SIZE;
...
((PGRAO_LIST_HEADER)((PGRAO)pGRAO)->grao_list)->addl_data = ((PGRAO)pGRAO)->
grao_list_length - HEADER_SIZE;
PopulateUDS ( (PGRAO_LIST_HEADER)(((PGRAO)pGRAO)->grao_list)+HEADER_SIZE, udsamount);

      *rc_ptr = DeviceIoControl(hDevice,
                                IOCTL_GENERATE_RAO,
                                pGRAO,
                                lGRAOsize,
                                NULL,
                                0,
                                &cb,
                                LPOVERLAPPED) NULL);
```

IOCTL_RECEIVE_RAO

This command is used to receive the generated **Recommended Access Order** list (TS1140 and later).

The structure for **IOCTL_RECEIVE_RAO** command is

```
typedef struct _RECEIVE_RAO_LIST {
    ULONG  rrao_list_offset; /* [IN] The offset of receive RAO list to
                             /*  begin returning data                    */
    ULONG  rrao_list_length; /* [IN/OUT] number byte of data length      */
                           /* [IN] The data length is allocated for RRAO list
                           /*  by application                            */
                           /* the length is (max_uds_size * uds_number + sizeof */
```

```
                        /* (struct rrao_list_header)                  */
                        /* max_uds_size is reported in QUERY_RAO_INFO ioctl */
                        /* uds_number is the total UDS number requested  */
                           from application                           */
                        /* in GENERATE_RAO ioctl                      */
                        /* [OUT] the data length is actual returned in RRAO list */
                        /* from the driver                            */
    CHAR  reserved[8];
    CHAR  rrao_list[1];    /* [IN/OUT] the data pointer of RRAO list */
}RECEIVE_RAO_LIST, *PRECEIVE_RAO_LIST;
```

An example of the **IOCTL_RECEIVE_RAO** command is

```
# HEADER_SIZE 8
...
char pRRAO;
...
long lRRAOsize=sizeof(RECEIVE_RAO_LIST)-sizeof(CHAR)/*rrao_list[1]
*/ + udsamount*udssize+UDS_HEADER;
...
pRRAO = malloc (lRRAOsize);
...
((PRECEIVE_RAO_LIST) pRRAO)->rrao_list_offset = offset;
((PRECEIVE_RAO_LIST) pRRAO)->rrao_list_length = udsamount*udssize+UDS_HEADER;
...
*rc_ptr = DeviceIoControl(hDevice,
                                IOCTL_RECEIVE_RAO,
                                pRRAO,
                                lRRAOsize,
                                pRRAO,
                                lRRAOsize,
                                &cb,
                                (LPOVERLAPPED) NULL);
```

IOCTL_CHANGER_OBTAIN_SENSE

Issue this command after an error occurs to obtain sense information that is associated with the most recent error. To guarantee that the application can obtain sense information that is associated with an error, the application must issue this command before other commands are issued to the device. Subsequent operations (other than **IOCTL_CHANGER_OBTAIN_SENSE**) reset the sense data field before the operation is run.

This IOCTL is only available for the changer path.

```
#define IOCTL_CHANGER_BASE          FILE_DEVICE_CHANGER
#define IOCTL_CHANGER_OBTAIN_SENSE
  CTL_CODE(IOCTL_CHANGER_BASE, 0x0819, METHOD_BUFFERED, FILE_READ_ACCESS)
```

The following output structure is completed by the **IOCTL_CHANGER_OBTAIN_SENSE** command that is passed by the caller.

```
#define MAG_SENSE_BUFFER_SIZE 96 /* Default request sense buffer size for \
Windows 200x */
typedef struct _CHANGER_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];
} CHANGER_OBTAIN_SENSE, *PCHANGER_OBTAIN_SENSE;
```

An example of the **IOCTL_CHANGER_OBTAIN_SENSE** command is

```
DWORD cb;
CHANGER_OBTAIN_SENSE sense_data;
DeviceIoControl(hDevice,
```

```
IOCTL_CHANGER_OBTAIN_SENSE,
NULL,
0,
&sense_data,
(long)sizeof(CHANGER_OBTAIN_SENSE),
&cb,
(LPOVERLAPPED) NULL);
```

IOCTL_MODE_SENSE

This command is used to get the Mode Sense Page/Subpage.

```
/**************************** GENERIC SCSI IOCTLS ****************************/
#define IOCTL_IBM_BASE          (('IBM' << 8) | FILE_DEVICE_SCSI)

#define DEFINE_IBM_IOCTL(x)    CTL_CODE(IOCTL_IBM_BASE, x, METHOD_BUFFERED, \
    FILE_READ_ACCESS | FILE_WRITE_ACCESS)
#define IOCTL_MODE_SENSE        DEFINE_IBM_IOCTL(0x003)
```

The structure that is used for this IOCTL is

```
typedef struct _MODE_SENSE_PARAMETERS
{
   UCHAR page_code;         /* [IN]  mode sense page code      */
   UCHAR subpage_code;      /* [IN]  mode sense subpage code   */
   UCHAR reserved[6];
   UCHAR cmd_code;          /* [OUT] SCSI Command Code: this field is set with */
                                    /*   SCSI command code which the
                                         device responded.                   */
                                    /*   x'5A' = Mode Sense (10)             */
                                    /*   x'1A' = Mode Sense (6)              */
   CHAR data[MAX_MODESENSEPAGE]; /* [OUT] whole mode sense data include header,
   block descriptor and page */
} MODE_SENSE_PARAMETERS, *PMODE_SENSE_PARAMETERS;
```

An example of the **IOCTL_MODE_SENSE** command is

```
DWORD cb;
MODE_SENSE_PARAMETERS mode_sense;
...
DeviceIoControl(gp->ddHandle0,
IOCTL_MODE_SENSE,
&mode_sense,
sizeof(MODE_SENSE_PARAMETERS),
NULL,
0,
&cb,
(LPOVERLAPPED) NULL);
```

Variable and fixed block read/write processing

In Windows 200x, tape APIs can be configured to manipulate tapes that use either fixed block size or variable block size.

If variable block size is wanted, the block size must be set to zero. 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 wanted block size. Any block size other than 0 sets the media parameters to fixed block size. The size of the block is 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 that are 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 cannot exceed the maximum transfer length of the Host Bus Adapter. This length is the length of each transfer page (typically 4 K) times the number of transfer pages (if set to 16, the maximum transfer length for variable sized transfers is 64 K). This number can be modified by changing the *scatter-gather* variable in the system registry, but this action is not recommended because it uses up scarce system resources.

Reading a tape that contains 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 is read without any errors. If the buffer is larger than a block, then only data in a single block is read and the tape is advanced to the next block.

The size of the block is returned by the read operation in the **`*pBytesRead`** parameter. If a data buffer is too small to contain all of the data in a block, then a couple of things occur. First, the data buffer contains data from the tape, but the read operation fails and **GetLastError** returns **ERROR_MORE_DATA**. This error value indicates that more data is 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 wanted block and a larger buffer must be specified. It is best to specify as large a buffer as possible so that this issue does not occur.

If a tape contains 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. The sizes 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 fails and **GetLastError** returns **ERROR_INVALID_BLOCK_LENGTH**. The only exception 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 that indicates that a tapemark was encountered.) The tape is positioned past the tapemark, and an application can call **ReadFile** again to continue reading.

Event log

The Magstar or **ibmtpxxx**, **ibmcgxxx**, and **Magchgr** device drivers log certain data to the Event Log when exceptions are encountered.

To interpret this event data, the user must be familiar with the following components:

- Microsoft Event Viewer

- The SDK and DDK components from the Microsoft Development Network (MSDN)
- Magstar and Magstar MP hardware terminology
- SCSI terminology

Several bytes of "Event Detail" data are logged under Source = Magstar or **Magchgr** (for Windows NT), or under Source = **ibmtpxxx** or **ibmcgxxx** (for Windows 2000; Windows Server 2003, Windows Server 2008, and Windows Server 2012).

The following description texts are expected:

- The description for Event ID (0) in Source (MagStar **or ibmtpxxx**) was not found. It contains the following insertion strings: `\Device\Tapex`.
- The description for Event ID (x) in Source (MagChgr) was not found.

The user must view the event data in Word format to properly decode the data.

Table 6 and Table 7 on page 334 indicate the hexadecimal offsets, names, and definitions for Magstar or **ibmtpxxx** and **ibmcgxxx** event data. **Magchgr** event data has a unique format that appears later in this chapter.

Table 6. Magstar ibmtpxxx, and ibmcgxxx event data

Offset	**Name**	**Definition**
0x00-0x01	DumpDataSize	Indicates the size in bytes required for any DumpData the driver places in the packet.
0x02	RetryCount	Indicates how many times the driver tried 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-0x0F	ErrorCode	For the Magstar device driver, it is 0. For the **Magchgr** device driver, it is always 0xC00400B (IO_ERR_CONTROLLER_ERROR, from NTIOLOGC.H).
0x10-0x13	UniqueErrorValue	Reserved
0x14-0x17	FinalStatus	Indicates the value that is 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-0x1F	IoControlCode	For the Magstar device driver, 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 is 0. For the **Magchgr** 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.

Table 6. Magstar ibmtpxxx, and ibmcgxxx event data (continued)

Offset	Name	Definition
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, **ibmcgxxx** logs the following error 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 00000000 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 7. Magstar ibmtpxxx, and ibmcgxxx event data

Field	Value	Definition
DumpDataSize	0x006C	6C hex (108 dec) bytes of dump data, beginning at byte 28 hex.
RetryCount	0x00	The first time that the operation is 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).
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 8 on page 335 and Table 9 on page 336 contain definitions for event data that is logged under **Magchgr**.

Table 8. Magchgr event data

Offset	Name	Definition
0x00-0x01	DumpDataSize	Indicates the size in bytes required for any DumpData the driver places in the packet.
0x02	RetryCount	Indicates how many times the driver tried 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-0x0F	ErrorCode	For the Magstar device driver, it is 0. For the **Magchgr** device driver, it is always 0xC00400B (IO_ERR_CONTROLLER_ERR) (from **NTIOLOGC.H**).
0x10-0x13	UniqueErrorValue	Reserved
0x14-0x17	FinalStatus	Indicates the value that is 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-0x1F	IoControlCode	For the Magstar device driver, it indicates the I/O control code from the driver I/O stack location in the current IRP if the MajorFunctionCode is IRP_MJ_DEVICE_CONTROL. Otherwise, this value is 0. For the **Magchgr** device driver, 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.
0x30-0x33	Timeout Value	For the Magstar device driver, this value is always 0. For the **Magchgr** device driver, this value is the command timeout value in seconds.
0x38	FRU or Sense Byte 14	For the Magstar device driver, this value is the Field Replaceable Unit Code. For the **Magchgr** device driver, 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) is displayed. Otherwise, the CDB length is displayed. See offset 0x3D to determine whether valid sense data is returned.

Table 8. Magchgr event data (continued)

Offset	Name	Definition
0x3B	SenseKeySpecific[2] or CDB[0]	If valid sense data was returned, SenseKeySpecific[2] (Sense Byte 17) is displayed. Otherwise, the CDB operation code is displayed. See offset 0x3D to determine whether valid sense data is 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 the byte 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 is given here. See offset 0x3D to determine whether valid sense data is returned.
0x3F	ASCQ or SCSI_STATUS	Indicates Sense Byte 13, if there was valid sense information. Otherwise, the SCSI status value is given here. See offset 0x3D to determine whether valid sense data is returned.

For example,

```
0000: 0018000f 006c0001 00000000 00000000
0010: 00000000 c0000185 00000000 00000000
0020: 00000000 00000000 00000300 0015c402
0030: 00000000 00000000 f50ac607 700b4b00
```

Table 9. Magchgr event data

Field	Value	Definition
DumpDataSize	0x0018	-
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 - 17	0x0AC607	-

Table 9. Magchgr event data (continued)

Field	Value	Definition
Sense Byte 0	0x70	-
Sense Key Sense Byte 2	0xb4	-
ASC	0x4B	-
ACSQ	0x00	-

Chapter 7. 3494 Enterprise tape library driver

AIX 3494 Enterprise tape library driver

After the driver is installed and a library manager control point (LMCP) is configured and made available for use, access is provided through the special files. These special files, which are the standard AIX special files for the character device driver, are in the **dev** directory. Each instance of an LMCP has exactly one special file that is associated with it.

Opening the Special File for I/O

The LMCP special file is opened for access by the standard AIX **open** command. The device driver ignores any flags that are associated with the open call (although the calling convention specifies that the flags parameter must be present). The **open** command is

```
fd = open("/dev/lmcp0", O_RDONLY);
```

Header definitions and structure

The input/output control (IOCTL) request has the following header definition and structure.

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

The syntax of the IOCTL request is

```
int ioctl(int fildes, int request, void *arg);
```

Parameters

You can set some of the parameters for the header definitions and structure as follows:

fildes Specifies the file descriptor that is returned from an **open** system call.

request
Specifies the command that is completed on the device.

arg Specifies the individual operation.

Reading and writing the Special File

The read and write entry points are not available in the library device driver. Any call that is made to the **read** or **write** subroutine results in a return code of ENODEV.

Closing the Special File

The file descriptor that is returned by the **open** command is used as the parameter for the close routine:

```
rc = close(fd);
```

The *errno* value that is set during a close operation indicates whether a problem occurred while the special file is closed. In the case of the LMCP device, the only value of *errno* is ENXIO (error occurs from internal code bug).

See “3494 Enterprise tape library system calls” on page 348 for information.

HP-UX 3494 Enterprise tape library driver

After the HP-UX 3494 Enterprise tape library driver is installed and started, an application might use subroutines that are provided with the software to access an Enterprise tape library.

Opening the library device

Before you can issue commands to the library, you must first use the **open_ibmatl** subroutine to open it. This subroutine call is similar in structure to the **open** system call. The syntax of the command is

```
int open_ibmatl(char *lib_name);
```

The `lib_name` is a symbolic name for a library that is defined in the **/etc/ibmatl.conf** file. If it is successful, the subroutine returns a positive integer that is used as the file descriptor for future library operations. If it is not successful, the subroutine returns **-1** and sets *errno* to one of the following values:

ENODEV
The library that is specified by the **lib_name** parameter is not known to the **lmcpd**.

EIO The **lmcpd** is not running or a socket error occurred while it was communicating with the **lmcpd**.

Closing the library device

In the same manner that you close a file with the UNIX **close** system call, close the library file descriptor when you are finished issuing commands to the library. The syntax of the **close_ibmatl** command is

```
int close_ibmatl(int ld);
```

The `ld` is the library file descriptor that is returned for the **open_ibmatl** command. If it is successful, the **close_ibmatl** command returns 0. If it is not successful, this command returns **-1**, and the *errno* variable is set to EBADF. (The library file descriptor that is passed to the **close_ibmatl** is not valid.)

Issuing the library commands

To issue commands to the library, use the **ioctl_ibmatl** command. The format of the command is the same as the UNIX input/output control (IOCTL) system call. The syntax of the command is

```
int ioctl_ibmatl (
  int ld,
  int request,
  void *arg);
```

Parameters

Certain parameters are set for the library commands, as follows.

ld
: Specifies the library file descriptor that is returned from an **open_ibmatl** call.

request
: Specifies the command that is completed on the device.

arg
: Specifies the pointer to the data that is associated with the particular command.

Building and linking applications with the library subroutines

An application that uses HP-UX Tape Library Driver commands and functions must include the driver interface definition header file that is provided with the **lmcpd** package and installed in the **/usr/include/sys** subdirectory. Include this header file in the application as follows,

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

An application that uses the HP-UX 3494 Enterprise Tape Library Driver commands and functions must also be linked with either the 32-bit (**/usr/lib/libibm.o**) or the 64-bit (**/usr/lib/libibm64.o**) driver interface C object module that is provided with the **lmcpd** package, depending on whether the application is a 32-bit or a 64-bit application. Link a 32-bit application program with the 3494 object module as follows,

```
cc -c -o myapp.o myapp.c
cc -o myapp myapp.o /usr/lib/libibm.o
```

The first **cc** command compiles the user application but suppresses the link operation, producing the **myapp.o** object module. The second **cc** command links the **libibm.o** library object module to the **myapp.o** object module to create the executable **myapp** file.

A 64-bit application program is built by following the instructions for a 32-bit application, except it uses **/usr/lib/libibm64.o** instead of **/usr/lib/libibm.o** when linking.

The two 3494 driver interface C object modules that contain position independent code (PIC) are also created with the +z or +Z compiler option. An application can use either the 32 bit (**usr/lib/libibmz.o**) or the 64 bit (**usr/lib/libibm64z.o**) in the **lmcpd** package. Which one is used to make a shared library with its own PIC object files depend on whether the application is a 32-bit or a 64-bit application. Create a 32-bit shared library with the 3494 PIC object module as follows.

```
ld -b -o lib3494.sl myappz1.o myappz2.o /usr/lib/libibmz.o
```

The **ld** command combines the **libibmz.o** library PIC object module with the **myappz1.o** and **myappz2.o** PIC object modules to build the shared library named **lib3494.sl**.

A 64-bit shared library is created by following the instructions for a 32-bit shared library, except it uses **/usr/lib/libibm64z.o** instead of **/usr/lib/libibmz.o**.

Linux 3494 Enterprise tape library driver

After the Linux 3494 Enterprise Tape Library Driver is installed and started, an application might use subroutines that are provided with the software to access an Enterprise tape library.

Opening the library device

Before you can issue commands to the library, you must first use the **open_ibmatl** subroutine to open it. This subroutine call is similar in structure to the **open** system call. The syntax of the command is

```
int open_ibmatl(char *lib_name);
```

The `lib_name` is a symbolic name for a library that is defined in the **/etc/ibmatl.conf** file. If it is successful, the subroutine returns a positive integer that is used as the file descriptor for future library operations. If it is not successful, then the subroutine returns **-1** and sets *errno* to one of the following values.

ENODEV
: The library that is specified by the **`lib_name`** parameter is not known to the **lmcpd**.

EIO
: The **lmcpd** is not running or a socket error occurred while it was communicating with the **lmcpd**. An input/output error.

Closing the library device

In the same manner that you close a file with the Linux **close** system call, close the library file descriptor when you are finished issuing commands to the library. The syntax of the **`close_ibmatl`** command is

```
int close_ibmatl(int ld);
```

The `ld` is the library file descriptor that is returned for the **`open_ibmatl`** command. If it is successful, the **`close_ibmatl`** command returns 0. If it is not successful, this command returns **`-1`**, and the *errno* variable is set to EBADF. (The library file descriptor that is passed to the **close_ibmatl** is not valid.)

Issuing the library commands

To issue commands to the library, use the **`ioctl_ibmatl`** command. The format of the command is the same as the UNIX input/output control (IOCTL) system call. The syntax of the command is

```
int ioctl_ibmatl(
  int ld,
  int request,
  void *arg);
```

Parameters

You can set some parameters on the library commands.

`ld`
: Specifies the library file descriptor that is returned from an **open_ibmatl** call.

`request`
: Specifies the command that is completed on the device.

`arg`
: Specifies the pointer to the data associated with the particular command.

Building and linking applications with the library subroutines

An application that uses Linux Tape Library Driver commands and functions must include the driver interface definition header file that is provided with the **lmcpd** package and installed in the **/usr/include/sys** subdirectory. Include this header file in the application as follows,

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

A 32- or 64-bit application that uses the library driver commands and functions must be linked with the **/usr/lib/libibm.o** driver interface C object module provided with the **ibmatl** package. Link a 32- or 64-bit application program with the 3494 object module as follows,

```
cc -c -o myapp.o myapp.c
cc -o myapp myapp.o /usr/lib/libibm.o
```

Note: libibm.o is a 64-bit object file for Intel IA64 and 64-bit zSeries architectures, but is a 32-bit object file for the other architectures.

The first **cc** command compiles the user application but suppresses the link operation, producing the **myapp.o** object module. The second **cc** command links the **libibm.o** library object module to the **myapp.o** object module to create the executable **myapp** file.

SGI IRIX 3494 Enterprise tape library

The following software development files are installed with the IBM automated tape library software.

```
/usr/include/sys/mtlibio.h
/usr/lib/libibm.o
```

If you are developing software applications for the IBM Enterprise tape library, you must include the **mtlibio.h** header file in your source program by adding the following line.

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

In addition, you must include the **libibm.o** object file when you compile and link your program. For example:

```
cc -o myprogram myprogram.c /usr/lib/libibm.o
```

The **libibm.o** object file provides the **open_ibmatl**, **ioctl_ibmatl**, and **close_ibmatl** functions for interfacing with the IBM Enterprise tape library. The function prototypes are defined **mtlibio.h**. These functions use the same system call conventions as **open**, **ioctl**, and **close**. If the function fails, **-1** is returned and the global *errno* value is set to indicate the error. Otherwise, a nonnegative value is returned.

The following example uses these functions.

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

int myfunction(char *libname)
{

  int rc,fd;
  struct mtdevinfo devices;
  /* open a library defined in the ibmatl.conf file */
  fd=open_ibmatl(libname);
```

```
  if(fd<0) return errno;

  /* query devices */
  rc=ioctl_ibmatl(fd,MTIOCLDEVINFO,&devices);
  if(rc<0) rc=errno;

  /* close library */
  close_ibmatl(fd);
  return rc;
}
```

Solaris 3494 Enterprise tape library driver

After the Solaris 3494 Enterprise tape library driver is installed and started, an application can access an Enterprise tape library by using subroutines that are provided with the software installation.

Opening the library device

Before you can issue commands to the library, you must first open it by using the **open_ibmatl** subroutine. This subroutine call is similar in structure to the **open** system call. The syntax of the command is

```
int open_ibmatl(char *lib_name);
```

The `lib_name` is a symbolic name for a library that is defined in the **/etc/ibmatl.conf** file. If it is successful, the subroutine returns a positive integer that is used as the file descriptor for future library operations. If it is not successful, the subroutine returns **-1** and sets *errno* to one of the following values:

ENODEV
: The library that is specified by the **lib_name** parameter is not known to the **lmcpd**.

EIO
: The **lmcpd** is not running or a socket error occurred while it was communicating with the **lmcpd**. An input/output error.

Closing the library device

In the same manner that you close a file with the UNIX close system call, close the library file descriptor when you are finished issuing commands to the library. The syntax of the **close_ibmatl** command is

```
int close_ibmatl(int ld);
```

The `ld` is the library file descriptor that was returned for the **open_ibmatl** command. If it is successful, the **close_ibmatl** command returns 0. If it is not successful, this command returns **-1**, and the *errno* variable is set to EBADF. (The library file descriptor that is passed to the **close_ibmatl** is not valid.)

Issuing the library commands

To issue commands to the library, use the **ioctl_ibmatl** command. The format of the command is the same as the UNIX input/output control (IOCTL) system call. The syntax of the command is

```
int ioctl_ibmatl(
  int ld,
  int request,
  void *arg);
```

Parameters

Some parameters can be set for the library commands, as follows.

ld Specifies the library file descriptor that is returned from an **open_ibmatl** call.

request
Specifies the command that is completed on the device.

arg Specifies the pointer to the data that is associated with the particular command.

Building and linking applications with the library subroutines

An application that uses the Solaris 3494 Enterprise tape library driver commands and functions must include the driver interface definition header file that is provided with the **lmcpd** package. It is installed in the **/usr/include/sys** subdirectory. Include this header file in the application as follows:

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

An application that uses the library driver commands and functions must be linked with either the 32-bit (**/usr/lib/libibm.o**) or the 64-bit (**/usr/lib/libibm64.o**) driver interface C object module. It is provided with the **ibmatl** package. Which one is used depends on whether the application is a 32-bit or a 64-bit application. Link a 32-bit or a 64-bit application program with the 3494 object module as follows:

```
cc -c -o myapp.o myapp.c
cc -o myapp myapp.o /usr/lib/libibm.o
```

For 64-bit IBM zSeries systems only, link the application program with the 3494 object module as follows:

```
cc -c -o myapp64.o myapp64.c
cc -o myapp64 myapp64.o /usr/lib/libibm64.o
```

The first **cc** command compiles the user application but suppresses the link operation, producing the **myapp.o** object module. The second **cc** command links the **libibm.o** library object module to the **myapp.o** object module to create the executable **myapp** file.

Windows 3494 Enterprise tape library service

After all of the software is installed on the system and the library service is started, access to the library is accomplished by using the subroutines provided in the **libibm** module installed in the **c:\winnt\system32** directory.

Opening the library device

Before you can issue commands to the library, you must first open it using the **open_ibmatl** subroutine. This subroutine call is similar in structure to the **open** system call. The syntax of the command is

```
int open_ibmatl(char *lib_name);
```

The `lib_name` is a symbolic name for a library that is defined in the **/etc/ibmatl.conf** file. If it is successful, the subroutine returns a positive integer that is used as the file descriptor for future library operations. If it is not successful, the subroutine returns **-1** and sets *errno* to one of the following values:

ENODEV
The library that is specified by the **lib_name** parameter is not known to the library service.

EIO The library service is not running or a socket error occurred during communication with the library service. An input/output error.

Closing the library device

In the same manner that you close a file with the UNIX **close** system call, close the library file descriptor when you are finished issuing commands to the library. The syntax of the **close_ibmatl** command is

```
int close_ibmatl(int ld);
```

The `ld` is the library file descriptor that was returned for the **open_ibmatl** command. If it is successful, the **close_ibmatl** command returns zero. If it is not successful, this command returns **-1** and the *errno* variable is set to EBADF. (The library file descriptor that is passed to the **close_ibmatl** is not valid.)

Issuing the library commands

To issue commands to the library, use the **ioctl_ibmatl** command. The format of the command is the same as the UNIX input/output control (IOCTL) system call. The syntax of the command is

```
int ioctl_ibmatl(
  int ld,
  int request,
  void *arg);
```

Parameters

You can specify some parameters for library commands, as follows.

ld Specifies the library file descriptor that is returned from an **open_ibmatl** call.

request
Specifies the command that is completed on the device.

See “3494 Enterprise tape library system calls” on page 348 for commands that can be issued to the library.

arg Specifies the pointer to the data that is associated with the particular command.

Building and linking applications with the library subroutines

An application that uses the library service commands and functions includes the **mtlibio.h** driver interface definition header file that is provided with the package. If you used the default installation directory, it is now at `C:\Program Files\IBM Automated Tape Library` on 32-bit Windows system or `C:\Program Files (x86)\IBM Automated Tape Library` on 64-bit Windows system. Ensure that the installation directory is included in the compiler path for included files, and reference the file as follows.

```
#include <mtlibio.h>
```

A 32 or 64–bit application can statically link its application with the **libibm.lib** or **libibm64.lib** driver interface object library during application build time. Or, it can dynamically link to the **libibm.dll** or **libibm64.dll** driver interface DLL at run time.

The default directory location for **libibm.lib** or **libibm64.lib** is

On 32-bit Windows systems:
: `C:\Program Files\IBM Automated Tape Library`.

On 64-bit Windows systems:
: `C:\Program Files (x86)\IBM Automated Tape Library`.

The DLLs (**libibm.dll** and **libibm64.dll**) are stored in these locations.

On Windows NT and 2000:
: `C:\WINNT\system32`.

On 32-bit Windows 2003:
: `C:\Windows\system32`.

On 64-bit Windows 2003:
: `C:\Windows\SysWOW64` for 32-bit **libibm.dll**.

: `C:\Windows\System32` for 64-bit **libibm64.dll**.

To link the interface DLL at run time dynamically, locate the executable file of the application in the same directory of the DLL file. To link the driver interface object library statically, specify the driver interface object library during the final link of the application. The following sample can be used as a starting point for an application that wants to dynamically link to the subroutines in the DLL. The subroutines must be called through the pointer, rather than their name. For example:

```
fd = t_open_ibmatlP("3494b");
static int dynload_lib( );

#define  T_INTERFACE_MODULE "LIBIBM"
#define T_OPEN_IBMATL   "open_ibmatl"
#define T_CLOSE_IBMATL   "close_ibmatl"
#define T_IOCTL_IBMATL   "ioctl_ibmatl"
HINSTANCE  t_mod_handle = NULL;
typedef int (* t_open_ibmatlF )( char *devNameP );
typedef int (* t_close_ibmatlF )( int fd );
typedef int (* t_ioctl_ibmatlF )( int fc,
                        int function,
                        void *parmsP );
t_open_ibmatlF t_open_ibmatlP = NULL;
t_close_ibmatlF t_close_ibmatlP = NULL;
t_ioctl_ibmatlF t_ioctl_ibmatlP = NULL;

static int dynload_lib( )
{
 t_mod_handle = LoadLibrary( T_INTERFACE_MODULE );
 if ( t_mod_handle == NULL ) /* Handle error */

 t_open_ibmatlP = (t_open_ibmatlF)
  GetProcAddress( t_mod_handle, T_OPEN_IBMATL );
 if ( t_open_ibmatlP == NULL ) /* Handle error */

 t_close_ibmatlP = (t_close_ibmatlF)
  GetProcAddress( t_mod_handle, T_CLOSE_IBMATL );
 if ( t_close_ibmatlP == NULL ) /* Handle error */

 t_ioctl_ibmatlP = (t_ioctl_ibmatlF)
```

```
  GetProcAddress( t_mod_handle, T_IOCTL_IBMATL );
 if ( t_ioctl_ibmatlP == NULL ) /* Handle error */

 return 0;  /* Good return */
}
```

3494 Enterprise tape library system calls

The system calls are provided to control the operation of the tape library device.

The set of library commands available with the base operating system is provided for compatibility with existing applications. In addition, a set of expanded library function commands gives applications access to more features of the tape drives.

The following library system calls are accepted by the library device driver only if the special file that is opened by the calling program is a Library Manager Control Point.

The following library commands are supported.

MTIOCLM
: Mount a volume on a specified drive.

MTIOCLDM
: Demount a volume on a specified drive.

MTIOCLQ
: Return information about the tape library and its contents.

MTIOCLSVC
: Change the category of a specified volume.

MTIOCLQMID
: Query the status of the operation for a specified message ID.

MTIOCLA
: Verify that a specified volume is in the library.

MTIOCLC
: Cancel the queued operations of a specified class.

MTIOCLSDC
: Assign a category to the automatic cartridge loader for a specified device.

MTIOCLRC
: Release a category that is previously assigned to a specified host.

MTIOCLRSC
: Reserve one or more categories for a specified host.

MTIOCLCSA
: Set the category attributes for a specified category.

MTIOCLDEVINFO
: Return a list of all devices currently available in the library.

MTIOCLDEVLIST
: Return an expanded list of all devices currently available in the library.

MTIOCLADDR
: Return the library address, configuration information, and the current online/offline status of the library.

MTIOCLEW
: Wait until an event occurs that requires the tape device driver to notify the Library Manager.

Library device number

The device number that is used for library system calls consists of the control unit serial number with a one-digit device number that is appended to it. For example, a device number for the second device in a library with the control unit serial number of 51582 is 515821. The control unit serial number is a hexadecimal number, where 0123456789ABCDEF are the valid digits. The valid one-digit device numbers are also hexadecimal. For the IBM 3494 Enterprise tape library, the drives are numbered from left to right, starting with 0.

For the Library Mount (**MTIOCLM**), Library Demount (**MTIOCLDM**), Library Cancel (**MTIOCLC**), and Library Set Device Category (**MTIOCLSDC**) library system calls, the device number must be a valid device number that is obtained by the **MTDEVICE** system call or supplied as described in the previous paragraph.

The remaining library system calls are designed for a user supplied device number or a zero. If the user supplies a zero, the library support selects a device to complete the operation that is requested.

The device number can be determined by issuing an OS-specific IOCTL to the drive. For AIX and Linux, use the **MTDEVICE** IOCTL. For HP-UX and Oracle Solaris, use the **STIOC_DEVICE_SN** IOCTL. For Windows, use the **IOCTL_TAPE_OBTAIN_MTDEVICE** vendor-specific device IOCTL. The **`mtlib`** command option **`-D`** also displays the device numbers.

MTIOCLM (Library Mount)

This library system call mounts a volume on a specified drive. Passed to this call are the device number of the device on which the volume is mounted, the VOLSER of the volume to be mounted, a target category to which the VOLSER is assigned at the time of the mount, and a source category from which a volume is mounted. If the target category field in the input argument to this call is specified, the volume is assigned to the category specified at the time of the mount. If the target category field in the input argument to this call is not specified, the volume is not assigned to a category at the time of this library system call. If the VOLSER parameter is not specified, the next available VOLSER from the category (which is specified in the **source_category** input parameter) is mounted.

If the `wait_flg` in the input argument indicates that the calling process waits until the mount is completed, the calling process is put to sleep after the call that initiates the mount command. The subsystem generates an operation completion notification to indicate the completion status of the mount. The return information argument is updated to include the completion status of the mount and the calling process is awakened.

If the `wait_flg` in the input argument does not indicate that the calling process waits until the mount is complete, the initial status is updated in the return information argument and control is returned to the calling process. If the mount command is initiated successfully, the completion code in the return information argument indicates success. If it is not successfully initiated, the completion code indicates the reason for the failure. After the mount completes, the driver determines which process, if any, is waiting for the status through the **MTIOCLEW** library system call. The process, if any, is notified of the completion status of the mount.

Passed to this library system call is a return information argument structure. After the completion of the call and before control is returned to the calling process, the return information structure is updated to indicate the completion status of the mount request.

Description

arg Points to the **mtlmarg** structure.

The **mtlmarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlmarg {
  int     resvd          /* reserved */
  int     versn          /* version number field */
  int     device;        /* device number */
  int     wait_flg;      /* indicates requester will wait or not wait */
  ushort target_cat;     /* category to which the VOLSER is assigned */
  ushort source_cat;     /* category from which a volume is mounted */
  char    volser[8];     /* specific VOLSER number to mount */
  struct mtlmret mtlmret; /* return information structure */
};
struct mtlmret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 uint    req_id;         /* message ID for an asynchronous operation */
 int     number_sense;   /* number of valid sense bytes */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read from device */
};
```

On Request

The field usage is defined as follows:

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field contains the device number of the device on which the operation is run. See “Library device number” on page 349 for all device fields.

wait_flg
This field indicates whether the process waits for the completion status of the operation. A value of zero indicates that the process does not wait for the completion status. A value other than zero indicates that the process waits for the completion status of the operation.

target_cat
If this field is *non_zero*, then it specifies a category to which the VOLSER is assigned.

source_cat
If the field VOLSER contains all blanks, this field specifies the category from which a volume is mounted. Otherwise, this field is ignored.

volser This field contains the ASCII value of the specific volume serial number to be mounted. The field is left-aligned and padded with blanks. If this field is all blanks, the **source_cat** field is used to identify a volume to be mounted. In this case, the next volume in the category that is specified is mounted.

On Return

The field usage of **struct mtlmret** is defined as follows:

cc This field contains the completion code for the operation. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

req_id If the **mount** operation is completed asynchronously (that is, the requester does not wait until completion of the command processing), this field contains the message ID corresponding to the mount request issued. The calling process can use this request ID to query the status of the mount. The caller must use the Query Message ID library system call to run this function.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

Return Value

When a process does not wait until the mount is complete, the completion code is set to indicate that the request was accepted for processing. The request ID indicates the message ID associated with the mount request. This request ID can be used to query the status of the mount operation.

See Table 13 on page 376 for possible return values.

MTIOCLDM (Library Demount)

This library system call demounts a volume from a specified drive. If the target category field in the **mtldarg** structure is specified, the volume is assigned to this category. If the target category field in the **mtldarg** structure is not specified, the volume is not assigned to this category.

Description

arg Points to the **mtldarg** structure.

The **mtldarg** structure is defined in **mtlibio.h** as follows:

```
struct mtldarg {
  int     resvd           /* reserved */
  int     versn           /* version number field */
  int     device;         /* device number */
  int     wait_flg;       /* indicates requester will wait or not wait */
  ushort  target_cat;     /* category to which the VOLSER is assigned */
  ushort  pad;            /* pad to maintain alignment */
  char    volser[8];      /* specific VOLSER number to demount */
  struct mtldret mtldret; /* return information structure */
};

struct mtldret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 uint    req_id;         /* message ID for an asynchronous operation */
 int     number_sense;   /* number of valid sense bytes */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read from device */
};
```

On Request

The field usage is defined as follows:

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field contains the device number of the device on which the operation is completed. See "Library device number" on page 349 for all device fields.

wait_flg
This field indicates whether the process waits for the completion status of the operation. A value of zero indicates that the process does not wait for the completion status. A value other than zero indicates that the process waits for the completion status of the operation.

target_cat
If this field is *non_zero*, it specifies a category to which the VOLSER is assigned when the **demount** operation begins. If this field is 0x0000, the volume category assignment is unchanged.

pad This field contains the pad to maintain alignment.

volser This field contains the ASCII value of the specific volume serial number to be demounted. The field is left-aligned and padded with blanks. If this field is all blanks, the volume is demounted. If a target category is specified, the category assignment of the volume is updated.

On Return

The field usage of **struct mtldret** is defined as follows:

cc This field contains the completion code for the operation. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

req_id If the **demount** operation is completed asynchronously (that is, the requester does not wait until completion of the command processing), this field contains the message ID corresponding to the **demount** request that is issued.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

Return Value

See Table 13 on page 376 for possible return values.

When the **`demount`** command is run asynchronously, the completion code is set to indicate the request was accepted for processing. The request ID indicates the message ID associated with the demount request. This request ID can be used to query the status of the demount operation.

MTIOCLQ (Library Query)

This library system call returns information about the Library Manager and its contents. Depending on the value of the subcommand that is passed to this call, the following information is returned.

Volume Data
: Information about a specific volume.

Library Data
: Configuration data.

Device Data
: Information about a specific drive.

Library Statistics
: Performance statistics.

Inventory Data
: Inventory report for up to 100 volumes.

Category Inventory Data
: Category information for up to 100 volumes.

Inventory Volume Count Data
: Total number of volumes in the library or the number of volumes in a specified category.

Expanded Volume Data
: Status of commands for the volume that was accepted by the library, but not completed.

Reserved Category List
: List of categories that are reserved for a specific host.

Category Attribute List
: List of category attributes.

Description

arg Points to the **mtlqarg** structure.

The **mtlqarg** structure is defined in **mtlibio.h** as follows:

```
struct mtlqarg {
  int     resvd          /* reserved */
  int     versn          /* version number field */
  int     device;        /* device number */
  int     cat_seqno;     /* category sequence number */
  int     subcmd;        /* subcommand field */
  ushort  source_cat;    /* source category */
  ushort  cat_to_read;
  char hostid[8]         /* host identifier */
  char    volser[8];     /* VOLSER number */
  struct mtlqret mtlqret; /* return information from query system call */
};
```

```
struct mtlqret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 int     device;         /* device number */
 int     number_sense;   /* number of valid sense bytes   */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read  */
 struct  lib_query_info info;         /* query information */
};
```

See **mtlibio.h** for **struct lib_query_info**.

On Request

The field usage is defined as follows:

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field contains the device number of the device on which to run the **Query Device Data** operation. It is ignored for all other query commands. See “Library device number” on page 349 for all device fields.

cat_seqno
This field contains the category sequence number. This field is used only for the **Category Inventory Data** subcommand. The inventory records are provided from the specified source category after this category sequence number. If X'0000' or the number is beyond the last volume in the category, the inventory records start with the first VOLSER in the category. This number is represented in hexadecimal.

subcmd
This field contains the subcommand that directs the device driver action. The possible values are

MT_QVD
Query Volume Data. Request information about the presence and use of the specific volume and its affinity to the subsystem in the library. The volume subsystem affinity is a prioritized list of subsystems closest to the physical storage location of the specified VOLSER.

MT_QLD
Query Library Data. Request information about the current operational status of the library.

MT_QSD
Query Statistical Data. Request information about the workload and performance characteristics of the library.

MT_QID
Query Inventory Data. Request information about up to 100 inventory data records for the library. The end of the list is indicated with a returned VOLSER name of “ ” all blanks. If the list contains 100 records, the next set is obtained by setting the **VOLSER** field in the input/output control (IOCTL) to the last volume name in the list (number 100). If the **VOLSER** field in the IOCTL is set to 0, the first set is returned.

MT_QCID
Query Category Inventory Data. Request information about up to 100 inventory data records for the VOLSERs assigned to the category specified. The end of the list is indicated with a returned category of 0. If the list contains 100 records, the next set is obtained by setting the **cat_seqno** in the IOCTL to the last category sequence number in the list (number 100). If the **cat_seqno** in the IOCTL is set to 0, the first set is returned.

MT_QDD
Query Device Data. Request information about the device that is specified in the **device** field.

MT_QIVCD
: Query Inventory Volume Count Data. Request either the total number of volumes in the library or the number of volumes in a specified category.

MT_QEVD
: Query Expanded Volume Data. Request expanding information about the specified VOLSER in the library.

MT_QRCL
: Query Reserved Category List. Request a list of categories that are reserved for the specified host identifier.

MT_QCAL
: Query Category List. Request a list of categories with their attributes that are reserved by the specified host identifier.

source_cat
: This field contains a category number. It is used in the **`Category Inventory Data`**, **`Volume Count Data`**, **`Reserved Category List`**, and **`Category Attribute List`** subcommands. The effect on each subcommand is as follows:

- **Category Inventory Data**. The **`source_cat`** parameter specifies the category from which to return the inventory records. See the **`cat_seqno`** parameter for related information.
- **Inventory Volume Count Data.** If the **`source_cat`** parameter contains 0000, a count of all volumes in the library is returned. If this parameter is not zero, a count of all volumes in the category is returned.
- **Reserved Category List**. If the **`source_cat`** parameter is not zero, the categories after this value are returned in the response. If this parameter is 0000 or beyond the last category that is reserved for the specified host identifier, the returned data starts with the first category reserved for the host identifier.
- **Category Attribute List**. If the **`source_cat`** parameter is not zero, the categories after this value are returned in the response. If this parameter is 0000 or beyond the last category that is reserved for the specified host identifier, the list of attributes for the categories starts with the first category reserved for the host identifier. See the **`cat_to_read`** parameter for information.

cat_to_read
: If this field is not zero, the category is read and returned in the response. If this field is zero, then the **source_cat** field is used to determine which data to return.

hostid
: This field indicates which reserved category list or category attribute list is returned to the caller. A process can request a reserved category or category attribute list for any host that is connected to the Dataserver if the correct host identifier is passed in this parameter. If the **`hostid`** parameter is NULL, the data is returned for the host that issued the command.

volser
: This field contains the volume serial number. The field is left-aligned and padded with blanks. This field is ignored when the **`subcmd`** parameter specifies **MT_QLD**, **MT_QSD**, **MT_QCID**, **MT_QIVCD**, **MT_QDD**, **MT_QRCL**, and **MT_QCAL**.

On Return

The field usage of **struct mtlqret** is defined as follows:

cc This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

device This field is ignored.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

info This field contains the query information requested based on the **subcmd** parameter. The possible values are shown in the following table.

Table 10. Subcmd parameter values.

Value	Description
MT_QVD **Query Volume Data.**	Provides detailed information about the VOLSER specified in the tape library. This information includes • the current state of the specified VOLSER • the class of the volume (for example, IBM 3480 1/2-inch cartridge tape) • the volume type (for example, 160 m nominal length tape • the ASCII VOLSER • the category to which the VOLSER is assigned • the subsystem affinity list (which is a prioritized list of up to 32 subsystems closest to the physical storage location of the specified VOLSER)
MT_QLD **Query Library Data**	Provides information about • current library operational state • number of input/output stations that are installed in the library • status of the input/output stations in the library • library machine type • library sequence number • total number of cells in the library • number of cells available for inserting new volumes into the library • number of subsystem IDs in the library • number of cartridge positions in each convenience station • configuration type of the accessor • accessor status • status of the optional components in the library

Table 10. Subcmd parameter values. (continued)

Value	Description
MT_QSD **Query Statistical Data**	Provides detailed information about the workload and performance characteristics of the tape library. The statistical information that is returned includes • device • mount • demount • eject • audit • input
MT_QID **Query Inventory Data**	Provides up to 100 inventory data records for the tape library. The information that is returned includes • library sequence number • number of VOLSERs in the library • volume inventory data records The individual volume data records include • category value • ASCII physical VOLSER name • state of the volume • type or class of the volume
MT_QCID **Query Category Inventory Data**	Provides up to 100 inventory data records for the VOLSERs that are assigned to a specified category. The information that is returned is identical to the information from a **Query Inventory Data** call. In addition to this information, the category sequence number is returned, which can be used to obtain the next 100 inventory data records in the category.
MT_QDD **Query Device Data**	Provides information about the device to which the command was issued. The information that is returned includes • mounted VOLSER if it is available • mounted category if a VOLSER is mounted • assigned device category if the device is assigned • device states • device class
MT_QIVCD **Query Inventory Volume Count Data**	Provides either the total number of volumes in the library or the number of volumes in a specified category.

Table 10. Subcmd parameter values. (continued)

Value	Description
MT_QEVD **Query Expanded Volume Data**	Provides expanded information about a specific VOLSER in the tape library. The information that is returned includes • volume states • volume class • volume type • VOLSER • category to which the VOLSER is assigned
MT_QRCL **Query Reserved Category List**	Provides a list of categories that are reserved for the host that is specified in the **hostid** parameter. The total number of categories is returned with a list of the categories that are reserved.
MT_QCAL **Query Category Attribute List**	Provides a list of category attributes for the categories that are reserved for the host identifier that is specified in the **hostid** parameter. The total number of categories that are reserved for the host and a list of reserved categories and their attributes are returned to the calling process.

Return Value

See Table 13 on page 376 for possible return values.

MTIOCLSVC (Library Set Volume Category)

This library system call changes the category of a specified volume in the tape library. This process includes assigning a volume to the EJECT category or BULK EJECT category so it can be removed from the tape library. If the EJECT category or BULK EJECT category is specified, the command is executed asynchronously. Otherwise, the command is executed synchronously.

Description

arg Points to the **mtlsvcarg** structure.

The **mtlsvcarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlsvcarg {
  int     resvd          /* reserved */
  int     versn          /* version number field */
  int     device;        /* device number */
  int     wait_flg;      /* indicates requester will wait or not wait */
  ushort target_cat;     /* category to which the VOLSER is assigned  */
  ushort source_cat;     /* source category of the VOLSER */
  char    volser[8];     /* VOLSER number assigned to a category */
  struct mtlsvcret mtlsvcret;   /* return information structure */
};
struct mtlsvcret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 uint    req_id;         /* message ID for an asynchronous operation */
```

```
 int     device;          /* device number */
 int     number_sense;    /* number of valid sense bytes */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field is ignored.

wait_flg
This field indicates whether the process waits for the completion status of the operation. A value of zero indicates that the process does not wait for the completion status. A value other than zero indicates that the process waits for the completion status of the operation. This field is ignored unless the target category specifies the eject category.

target_cat
This field contains the target category to which the VOLSER is assigned.

source_cat
This field contains the category to which the volume is assigned. This field must contain X'FF00' if the volume is in the insert category. If this field contains X'0000', it is ignored.

volser This field contains the volume serial number to be assigned to a category. The field is left-aligned and padded with blanks.

On Return

The field usage of **struct mtlsvcret** is defined as follows.

cc This field contains the completion code for the operation. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

req_id If the operation is completed asynchronously (that is, the requester does not wait until the command processing completes), then this field contains the message ID corresponding to the operation issued. This field is defined only when the target category specified is an eject category.

device This field is ignored.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

Return Value

See Table 13 on page 376 for possible return values.

MTIOCLQMID (Library Query Message ID)

This library system call queries the status of a specified message ID. The two types of status responses are

- **Delayed Response Message Status**. The Library Manager keeps a list of the last 600 delayed response messages for **mount**, **demount**, **audit**, and **eject** commands. If the message ID is for a command with a delayed response message, all the delayed response information is returned to the calling application.
- **Unknown or Pending Status**. If the message ID supplied to the Library Manager is pending execution or is no longer in the 600 item delayed response message list, a single status byte is returned as a response to this command.

Description

arg Points to the **mtlqmidarg** structure.

The **mtlqmidarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlqmidarg {
  int     resvd         /* reserved */
  int     versn         /* version number field */
  int     device;       /* device number */
  uint    req_id;       /* message ID for an asynchronous operation */
  struct mtlqmidret mtlqmidret; /* return information structure */
};
struct mtlqmidret {
    int       cc;              /* completion code */
    int       up_comp;         /* reserved */
    int       device;          /* device number the operation was performed on */
    int       number_sense;    /* number of valid sense bytes */
    char      sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
    struct qmid_info info;     /* information about queried message id */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field is ignored.

req_id This field contains the ID of a request that was previously initiated.

On Return

The field usage of **struct mtlqmidret** is defined as follows.

cc This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

device This field is ignored.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

info See **mtlibio.h** for a description of the **qmid_info** structure.

Return Value

See Table 13 on page 376 for possible return values.

MTIOCLA (Library Audit)

This library system call verifies that a specified volume is in the library. The specified VOLSER is physically verified as being in the tape library. The operation is asynchronous and completes when the volume is audited.

Description

arg Points to the **mtlaarg** structure.

The **mtlaarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlaarg {
  int     resvd           /* reserved */
  int     versn           /* version number field */
  int     device;         /* device number */
  int     wait_flg;       /* indicates requester will wait or not wait */
  int     audit_type;     /* audit type */
  char    volser[8];      /* specific VOLSER number to audit */
  struct mtlaret mtlaret; /* return information structure */
};
struct mtlaret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 uint    req_id;         /* message ID for an asynchronous operation */
 int     device;         /* device number */
 int     number_sense;   /* number of valid sense bytes */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field is ignored.

wait_flg
This field indicates whether the process waits for the completion status of the operation. A value of zero indicates that the process does not wait for the completion status. A value other than zero indicates that the process waits for the completion status of the operation.

audit_type
This field contains the type of audit. The only possible value is **VOL_AUDIT**.

volser This field contains the volume serial number to be audited. The field is left-aligned and padded with blanks.

On Return

The field usage of **struct mtlaret** is defined as follows.

cc This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

req_id If the operation is completed asynchronously (that is, the requester does not wait until completion of the command processing), then this field contains the message ID corresponding to the operation issued.

device This field is ignored.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

Return Value

See Table 13 on page 376 for possible return values.

MTIOCLC (Library Cancel)

This library system call cancels all queued operations of a specified class. The caller can request this function for a specific device or a specific asynchronous operation. If an operation completion notification was owed for any operation that is canceled before execution, a notification indicates that the operation was canceled at the program's request. Any operation that began or completed execution is not canceled.

Description

arg Points to the **mtlcarg** structure.

The **mtlcarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlcarg {
  int     resvd          /* reserved */
  int     versn          /* version number field */
  int     device;        /* device number */
  uint    req_id;        /* message ID for an asynchronous operation */
  int     cancel_type    /* type of cancel requested */
  struct mtlcret mtlcret; /* return information structure */
};
struct mtlcret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 int     device;         /* device number */
 int     number_sense;   /* number of valid sense bytes */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field is ignored unless the **cancel_type** field specifies CDLA. This field contains the device number. See "Library device number" on page 349 for all device fields.

req_id This field contains the message ID of the queued operation to cancel. This field is ignored unless the cancel type that is specified in the **cancel_type** field is **Message ID Cancel (MIDC)**.

cancel_type
This field defines the type of cancel. The possible values are:

CDLA Cancel Drive Library Activity. All library mount operations queued for the specified drive are canceled.

CAHA
Cancel all host-related activity. All queued commands that are issued by this host are canceled.

MIDC Message ID Cancel. The queued operation that is identified by the **req_id** field is canceled.

On Return

The field usage of **struct mtlcret** is defined as follows.

cc This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility (which is zero).

device This field is ignored.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes read from the device.

Return Value

See Table 13 on page 376 for possible return values.

MTIOCLSDC (Library Set Device Category)

This library system call assigns a category to a device in the IBM 3494 Enterprise tape library. This command also specifies how and when cartridges are mounted on the device when the assignment takes place. The following parameters can be set with this command.

- **Enable Category Order**

  When active, the Library Manager selects volumes to mount based on the order in which they were assigned to the category, starting with the first volume assigned. After the end of the category is reached, the subsequent requests receive a **Category Empty** error.

  In addition, when this parameter is active, only one device can be assigned to this category. Therefore, multiple devices can be assigned the same category when this parameter is not active. If multiple devices are assigned to the same category, the volumes are picked in the order in which they were assigned. There is no method to determine which volumes are mounted on a particular device.

  If the specified category is in use by another device and the enable category bit is set, the operation fails and the command is presented unit check status with associated sense data that indicates ERA X'7F'.

- **Clear Out ICL (integrated cartridge loader)**

  When active, the category assignment that is previously set on the specified device is removed. All other parameters that are specified in the **Library Set Device Category** command are ignored when this parameter is active. Any cartridge in the specified drive is unloaded and returned to a storage cell.
- **Generate First Mount**

  When active, the Library Manager queues a mount for the first volume in the category that is specified in the **category** parameter. A delayed response message is not generated for this mount. If the mount fails, an unsolicited attention interrupt is generated and sent to the host. This command can be used with the **Enable Auto Mount** command.
- **Enable Auto Mount**

  When the device is issued an unload command, the Library Manager queues a demount for the volume that is mounted in device. Additionally, a mount command is queued for the next volume in the category. This mount command does not generate a delayed response message. If the mount fails, an unsolicited attention interrupt is generated and sent to the host. When **Enable Auto Mount** is cleared, an unload command is sent to the device. This parameter can be used with the **Generate First Mount** command.

Description

arg Points to the **mtlsdcarg** structure.

The **mtlsdcarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlsdcarg {
  int     resvd          /* reserved */
  int     versn          /* version number field */
  int     device;        /* device number */
  int     fill_parm;     /* fill parameters */
  ushort category;       /* category to be assigned to the device */
  ushort  demount_cat;
  struct mtlsdcret mtlsdcret;      /* return information structure */
};
struct mtlsdcret {
 int     cc              /* completion code */
 int     up_comp         /* reserved */
 int     number_sense;   /* number of valid sense bytes  */
 char sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field contains the device number of the device on which the operation is run. See “Library device number” on page 349 for all device fields.

fill_parm
This field contains the following fill parameters.

MT_ECO(0x40)
Category Order. When it is active, the Library Manager fills the loader index stack by selecting volumes from the specified category that is based on how they were assigned to the category.

MT_CACL(0x20)
: Clear Automatic Cartridge Loader. The Library Manager resets the category assignment to the specified device. If this value is specified, then all other parameter values that are sent with this command are ignored.

MT_GFM (0x10)
: Generate First Mount. The Library Manager queues a mount request for the first volume in the category. No delayed response message is generated.

MT_EAM (0x08)
: Enable Auto Mount. The Library Manager queues the mount requests for the next volume in the category when the device receives a rewind/unload command. If this field is cleared, then the Library Manager issues a rewind/unload command to the specified device.

category
: This field contains the category to be assigned to the device. If this field contains X'0000', then it causes the Library Manager to remove all volumes from the cartridge loader. This operation has the same effect as specifying **MT_CACE_ACL** in the **`fill_parm`** parameter.

demount_cat
: This field specifies the category in which to place the volume when it is demounted from the device. If this field is X'00', then the category is not changed for the demount operation.

On Return

The field usage of **struct mtlsdcret** is defined as follows.

cc
: This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
: This field is reserved for upward compatibility (which is zero).

number_sense
: This field contains the number of valid sense bytes.

sense_bytes
: This field contains the sense bytes read from the device.

Return Value

See Table 13 on page 376 for possible return values.

MTIOCLRC (Library Release Category)

This library system call releases a category that was assigned to the specified host with the **`MTIOCLRSC`** command. Passed to this command are the category identifier to be released and the host identifier. The category identifier was reserved when a **`Library Reserve Category`** command was issued for the specified host identifier. The category must not contain any volumes when this command is issued. If the category contains any tape volumes, the command fails. The host ID specifies the host for which the category was reserved.

Description

arg Points to the **mtlrcarg** structure.

The **mtlrcarg** structure is defined as follows.

```
struct mtlrcarg {
 int    resvd;
 int    versn;
 int    device;
 ushort release_cat;        /* category to release */
 ushort pad;                /* maintain alignment */
 char hostid [8];           /* host identifier */
 struct mtlrcret mtlrcret;
};

struct mtlrcret {
 int     cc;                /* completion code */
 int     up_comp;           /* reserved */
 int     number_sense;      /* number of valid sense bytes */
 cha     sense_bytes[MT_SENSE_LENGTH];   /* sense bytes */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

device This field contains the device number of the device on which the operation is run. See "Library device number" on page 349 for all device fields.

pad This field contains the pad to maintain alignment.

release_cat
This field contains the category to be released.

hostid This field specifies the host identifier that reserved the category that is released. Only the same host identifier that reserved the category can release it.

On Return

The field usage of **struct mtlrcret** is defined as follows.

cc This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
This field contains the sense bytes.

MTIOCLRSC (Library Reserve Category)

This library system call reserves one or more categories for the host that issues this command. The host that issues this command either chooses the category to reserve or allows the Library Manager to choose the categories to reserve. If the host chooses the category, only one category at a time can be reserved. If the host allows the Library Manager to choose the categories, more than one category at a time can be reserved.

Description

arg Points to the **mtlrscarg** structure.

The **mtlrscarg** structure is defined as follows.

```
struct mtlrscarg {
 int    resvd             /* reserved, must be zero */
 int    versn             /* version number */
 int    device            /* device number */
 ushort num_cat           /* number of categories to reserve */
 ushort category          /* category to reserve if num_cat == 1 */
 char hostid [8]
 struct mtlrscret mtlrscret   /* return information structure */
};
struct mtlrscret {
 int       cc;            /* completion code */
 int       up_comp;       /* reserved */
 int       number_sense;  /* number of valid sense bytes */
 char      sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
 struct reserve_info info;
};

struct reserve_info
{
 char   atl_seqno[3];     /* library sequence number */
 char   ident_token[8];   /* token for which categories are reserved */
 char   count[2];         /* total number of categories in list */
 uchar  cat[256][2]       /* reserved category records */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains zero.

device This field is ignored.

num_cat
The number of categories to reserve.

category
If the **num_cat** field = 1, the library attempts to reserve the specified category.

hostid Eight character host identifier for which the category is reserved.

On Return

The field usage of **struct mtlrscret** is defined as follows.

cc This field contains the completion code. See Table 13 on page 376 for possible values.

up_comp
This field is reserved for upward compatibility.

number_sense
This field contains the number of valid sense bytes.

sense_bytes
: This field contains the sense bytes.

reserve_info
: This structure contains a list of categories that are reserved with the **Library Reserve Category** command.

MTIOCLSCA (Library Set Category Attribute)

This library system call allows the host to specify the attributes for a category that is previously reserved for this host with the **MTIOCLRSC** library system call. The only attribute that can be set is **category name**. The name is a 10 character string, which does not have to end with a null character. The following naming conventions are allowed.

- Uppercase letters A-Z
- Numbers 0-9
- Blank, underscore (_), or asterisk (*)
- Blanks in any position

Description

arg Points to the **mtlscaarg** structure.

The **mtlscaarg** structure is defined as follows.

```
struct mtlscaarg {
 int    resvd                  /* reserved, must be zero */
 int    versn                  /* version number */
 int    device                 /* device number */
 ushort attr                   /* attribute description */
 ushort category               /* category whose attribute to set */
 char   attr_data[ATTR_MAXLN]  /* data to assign to the category */
 struct mtlscaret mtlscaret    /* return information structure */
};

struct mtlscaret {
int      cc;                   /* completion code */
int      up_comp;              /* reserved */
int      number_sense;         /* number of valid sense bytes */
char     sense_bytes[MT_SENSE_LENGTH];   /* sense bytes read */
};
```

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains zero.

device This field is ignored.

attr This field describes the attribute. It contains the following value.

```
MT_SCM (0x01)  Set Category Name
```

category
: This field specifies the category.

attr_data
: This field contains the 10 character category name.

MTIOCLDEVINFO (Device List)

This library system call returns a list of all devices currently available in the library and their associated device numbers. See “Library device number” on page 349 for a description of device numbers. The **MTIOCLDEVLIST** library system call returns the same device list in an expanded format.

The **mtdevinfo** structure is defined in **mtlibio.h** as follows.

```
struct mtdevinfo {
  struct {
    int      device;            /* device number */
    char     name[32];          /* device name   */
  } dev[MAXDEVICES];
```

On Return

The field usage of **struct mtdevinfo** is defined as follows.

device This field contains the device number. The end of the list is indicated with a device number equal to **-1**.

name This field is the name of the device. It consists of 6 bytes for the device type, 3 bytes for the model number, and 2 bytes for the decimal index number in the device list array.

Return Value

See Table 13 on page 376 for possible return values.

Example:

The following code is used in the **mtlib** utility for the `-D` option:

```
struct mtdevinfo dinfo;

int devices(int lib_fd)
  {
  int rc;
  int i;

  rc = ioctl(lib_fd, MTIOCLDEVINFO, &dinfo);
  if (rc)
    {
    printf("Operation Failed - %s\n", strerror(errno));
    return errno;
    }

  for (i=0; i < MAXDEVICES; i++)
    {
    if (dinfo.dev[i].device == -1) break;
      printf("%3d, %08X %s\n",i, dinfo.dev[i].device, dinfo.dev[i].name);
    }

  return(0);
  }
```

MTIOCLDEVLIST (Expanded Device List)

This library system call returns a list of all devices currently available in the library and their associated device numbers in an expanded format. See “Library device number" on page 349 for a description of device numbers. The **MTIOCLDEVINFO** library system call returns the same device list in a different format.

The **mtdevlist** structure is defined in **mtlibio.h** as follows.

```
struct mtdevlist {
  struct {
    char           type[6];
    char           model[3];
    char           serial_num[8];
    unsigned char  cuid;
    unsigned char  dev;
    int            dev_number;
    int            vts_library;
  } device[MAXDEVICES];
};
```

On Return

The field usage of **struct mtdevinfo** is defined as follows.

dev_number
: This field contains the device number. The end of the list is indicated with a device number equal to **-1**.

type
: This field contains the device type.

model
: This field contains the model number of the device.

serial_num
: This field contains the serial number of the device.

cuid and dev
: These fields contain the library subsystem ID (cuid) and device (dev) within the subsystem for this device in the library.

vts_library
: This field indicates whether the device is in a VTS library, and if so, which logical VTS library. A value of 0 indicates that the device is not in a VTS library.

Return Value

See Table 13 on page 376 for possible return values.

Example:

The following code is used in the **mtlib** utility for the `-DE` option:

```
struct mtdevlist dlist;

int device_list(int lib_fd)
  {
  int rc;
  int i;
  char type[7];
  char model[4];
  char sn[9];
  int pass = 1;

  rc = ioctl(lib_fd, MTIOCLDEVLIST, &dlist);
  if (rc)
    {
    printf("Operation Failed - %s\n", strerror(errno));
```

```
    return errno;
    }

  for (i=0; i <MAXDEVICES; i++)
    {
    if (dlist.device[i].dev_number == -1) break;
    strncpy(type, dlist.device[i].type,6);
    type[6] = '\0';
    strncpy(model, dlist.device[i].model,3);
    model[3] = '\0';
    strncpy(sn, dlist.device[i].serial_num,8);
    sn[8] = '\0';
    if (pass == 1)
      {
      printf(" Type   Mod  Serial #   Devnum   Cuid  Device  VTS Library\n");
      pass++;
      }
    if (dlist.device[i].vts_library)
      {
      printf("%s  %s  %s  %08X   %2d     %2d      %2d\n", type,model,sn,
             dlist.device[i].dev_number, dlist.device[i].cuid,
             dlist.device[i].dev,
             dlist.device[i].vts_library);
      }
    else
      {
      printf("%s  %s  %s  %08X   %2d     %2d         \n", type,model,sn,
             dlist.device[i].dev_number, dlist.device[i].cuid,
             dlist.device[i].dev);
      }
    }

  return(0);
  }
```

MTIOCLADDR (Library Address Information)

This library system call returns the library address and configuration information from the **ibmatl.conf** config file and the current online or offline status of the library. A 3494 Enterprise Model HA1 (High Availability) has two addresses that are configured, but only one address is online at a time.

The **mtlibaddr** structure is defined in **mtlibio.h** as follows.

```
#define MT_LIBADDR_INVALID   0      /* Address not configured             */
#define MT_LIBADDR_OFFLINE   1      /* Library is offline with this address  */
#define MT_LIBADDR_ONLINE    2      /* Library is online with this address   */

struct mtlibaddr {
      char library_name[32];        /* Logical name of library              */
      char host_ident[8];           /* Host identification for library      */
      char primary_addr[16];        /* Primary address of library           */
      char primary_status;          /* Primary status as defined above      */
      char alternate_addr[16];      /* Alternate address of library         */
      char alternate_status;        /* Alternate status as defined above    */
      char reserved[32];
   };
```

On Return

The field usage of **struct mtlibaddr** is defined as follows.

library_name
: This field contains the logical name of the library that is defined in the **ibmatl.conf** file.

host_ident
This field contains the host identification for the logical library.

primary_addr
This field contains the primary address for the logical library, either a tty serial port connection or an IP address.

primary_status
This field contains the status of the primary address connection as defined in the **primary_addr** field and is always either online or offline.

alternate_address
This field contains the alternate address for the logical library if configured in the **ibmatl.conf** file. If an alternate address is not configured, the **alternate_status** field is set to **MT_LIBADDR_INVALID**.

alternate_status
This field contains the status of the alternate address connection as defined in the **alternate_address** field: either **online**, **offline**, or **not configured**.

Return Value

See Table 13 on page 376 for possible return values.

Example:

The following code is used in the **mtlib** utility for the `-A` option:

```
struct mtlibaddr addrlist;

int libaddr(int lib_fd)
  {
  int rc;

  rc = ioctl(lib_fd, MTIOCLADDR, &addrlist);
  if (rc)
    {
    printf("Operation Failed - %s\n", strerror(errno));
    return errno;
    }

  printf("Library Address Information: \n");
  printf("  library name...........%0.32s\n",addrlist.library_name);
  printf("  host identification....%0.8s\n",addrlist.host_ident);
  printf("  primary address........%s\n",addrlist.primary_addr);
  if (addrlist.primary_status == MT_LIBADDR_ONLINE)
    printf("  primary status.........Online\n");
  else
    printf("  primary status.........Offline\n");

  if (addrlist.alternate_status == MT_LIBADDR_ONLINE)
    {
    printf("  alternate address......%s\n",addrlist.alternate_addr);
    printf("  alternate status.......Online\n");
    }
  else if (addrlist.alternate_status == MT_LIBADDR_OFFLINE)
    {
    printf("  alternate address......%s\n",addrlist.alternate_addr);
    printf("  alternate status.......Offline\n");
    }
  else
    printf("  alternate address......Not configured\n");

  return(0);
  }
```

MTIOCLEW (Library Event Wait)

This library system call reads the state information that is associated with a logical library device entry. Then, it optionally waits for a state change to occur before the state information is returned.

Description

arg Points to the **mtlewarg** structure.

The **mtlewarg** structure is defined in **mtlibio.h** as follows.

```
struct mtlewarg {
  int     resvd            /* reserved */
  int     versn            /* version number field */
  int     subcmd;          /* subcommand field */
  int     timeout;         /* timeout in seconds *
                           /* if set to zero, no timeout is performed */
  struct mtlewret mtlewret; /* return information structure */
};
struct mtlewret {
 int     up_comp           /* reserved */
 int     cc                /* completion code */
 int     lib_event         /* detected library event */
 int     msg_type          /* type of message */
 struct msg_info msg_info;  /* operation completion or unsolicited */
};
```

See **mtlibio.h** for **struct msg_info**.

On Request

The field usage is defined as follows.

resvd This field contains zero.

versn This field contains the version number (zero) of the block structure.

subcmd
: This field contains the **LEWTIME** subcommand. It is returned only when an error or exception condition is detected or after a timeout occurs (whichever happens first).

timeout
: This field contains the timeout time in seconds. If it is set to zero, no timeout is completed.

On Return

The field usage of **struct mtlewret** is defined as follows.

up_comp
: This field is reserved for upward compatibility (which is zero).

cc This field contains the completion code. See Table 13 on page 376 for possible values.

lib_event
: This field contains the detected event. The possible values are shown in Table 11 on page 374.

msg_type
: This field contains the type of message if it is reported. The possible values are:

NO_MSG
No message.

UNSOL_ATTN_MSG
Unsolicited notification.

DELAYED_RESP_MSG
Operation completion notification.

msg_info
This field contains the operation completion or unsolicited notification.

alternate_status
This field contains the status of the alternate address connection as defined in the **alternate_address** field: either **online**, **offline**, or **not configured**.

Table 11. Unsolicited Attention Interrupts

Event	ERA Code	Description
None	0x27	Command reject
MT_NTF_ERA60	0x60	Library attachment facility equipment check
MT_NTF_ERA62	0x62	Library Manager offline to subsystem
MT_NTF_ERA63	0x63	Control unit and Library Manager incompatible
MT_NTF_ERA64	0x64	Library VOLSER in use
MT_NTF_ERA65	0x65	Library volume reserved
MT_NTF_ERA66	0x66	Library VOLSER not in library
MT_NTF_ERA67	0x67	Library category empty
MT_NTF_ERA68	0x68	Library order sequence check
MT_NTF_ERA69	0x69	Library output stations full
MT_NTF_ERA6B	0x6B	Library volume misplaced
MT_NTF_ERA6C	0x6C	Library misplaced volume found
MT_NTF_ERA6D	0x6D	Library drive not unloaded
MT_NTF_ERA6E	0x6E	Library inaccessible volume restored
MT_NTF_ERA6F	0x6F	Library vision failure
MT_NTF_ERA70	0x70	Library Manager equipment check
MT_NTF_ERA71	0x71	Library equipment check
MT_NTF_ERA72	0x72	Library not capable – Manual mode
MT_NTF_ERA73	0x73	Library intervention required
MT_NTF_ERA74	0x74	Library informational data
MT_NTF_ERA75	0x75	Library volume inaccessible
MT_NTF_ERA76	0x76	Library all cells full
MT_NTF_ERA77	0x77	Library duplicate VOLSER ejected
MT_NTF_ERA78	0x78	Library duplicate VOLSER in input station
MT_NTF_ERA79	0x79	Library unreadable or invalid VOLSER in input station
MT_NTF_ERA7A	0x7A	Read library statistics
MT_NTF_ERA7B	0x7B	Library volume ejected manually
MT_NTF_ERA7C	0x7C	Library out of cleaner volumes
MT_NTF_ERA7F	0x7F	Library category in use
MT_NTF_ERA80	0x80	Library unexpected volume ejected

Table 11. Unsolicited Attention Interrupts (continued)

Event	ERA Code	Description
MT_NTF_ERA81	0x81	Library I/O station door open
MT_NTF_ERA82	0x82	Library Manager program exception
MT_NTF_ERA83	0x83	Library drive exception
MT_NTF_ERA84	0x84	Library drive failure
MT_NTF_ERA85	0x85	Library environmental alert
MT_NTF_ERA86	0x86	Library all categories reserved
MT_NTF_ERA87	0x87	Duplicate volume add requested
MT_NTF_ERA88	0x88	Damaged volume ejected
MT_NTF_ATTN_CSC	None	Category state change
MT_NTF_ATTN_LMOM	None	Library Manager operator message
MT_NTF_ATTN_IOSSC	None	I/O station state change
MT_NTF_ATTN_OSC	None	Operational state change
MT_NTF_ATTN_DAC	None	Device availability change
MT_NTF_ATTN_DCC	None	Device category change
MT_NTF_ATTN_VE	None	Volume exception
MT_NTF_DEL_MC	None	Mount complete
MT_NTF_DEL_DC	None	Demount complete
MT_NTF_DEL_AC	None	Audit complete
MT_NTF_DEL_EC	None	Eject complete
MT_NTF_TIMEOUT	None	Timeout

Return Value

If a library system call is successful, the return code is set to zero. If the library system call is not successful, the return code is set to **-1**. If the library system call is not successful, the *errno* variable is set to indicate the cause of the failure. The values in Table 12 are returned in the *errno* variable.

Table 12. MTIOCLEW errors

Return Code	errno	cc	Value	Description
0	ESUCCESS	0	0 X'0'	Completed successfully.
-1	ENOMEM	Undefined	-	Memory allocation failure.
-1	EFAULT	Undefined	-	Memory copy function failure.
-1	EIO	MTCC_NO_LMCP	32 X'20'	The Library Manager Control Point is not configured.
-1	EINVAL	MTCC_INVALID_SUBCMD	41 X'29'	An invalid subcommand is specified.
-1	EIO	MTCC_LIB_NOT_CONFIG	42 X'2A'	No library devices are configured.

Table 12. MTIOCLEW errors (continued)

Return Code	errno	cc	Value	Description
-1	EIO	MTCC_INTERNAL_ERROR	43 X'2B'	Internal error.

Error description for the library I/O control requests

If a library system call is successful, the return code is set to zero. If the library system call is not successful, the return code is set to **-1**. If the library system call is not successful, the *errno* variable is set to indicate the cause of the failure. The completion code in the return structure of the library system call is set with a value that indicates the result of the library system call.

Table 13 shows the return codes, the *errno* variables, and the completion codes for the library I/O control requests. See **mtlibio.h** for the code values.

Table 13. Error description for the library I/O control requests

Code	errno	Value	cc	Value	Description
0	ESUCCESS	0	MTCC_COMPLETE	0 X'0'	Completed successfully.
-1	EIO	5	MTCC_COMPLETE_VISION	1 X'1'	Completed. Vision system not operational.
-1	EIO	5	MTCC_COMPLETE_NOTREAD	2 X'2'	Completed. VOLSER not readable.
-1	EIO	5	MTCC_COMPLETE_CAT	3 X'3'	Completed. Category assignment that is not changed.
-1	EIO	5	MTCC_CANCEL_PROGREQ	4 X'4'	Canceled program requested.
-1	EIO	5	MTCC_CANCEL_ORDERSEQ	5 X'5'	Canceled order sequence.
-1	EIO	5	MTCC_CANCEL_MANMODE	6 X'6'	Canceled manual mode.
-1	EIO	5	MTCC_FAILED_HARDWARE	7 X'7'	Failed. Unexpected hardware failure.
-1	EIO	5	MTCC_FAILED_VISION	8 X'8'	Failed. Vision system not operational.
-1	EIO	5	MTCC_FAILED_NOTREAD	9 X'9'	Failed. VOLSER not readable.
-1	EIO	5	MTCC_FAILED_INACC	10 X'A'	Failed. VOLSER inaccessible.

Table 13. Error description for the library I/O control requests (continued)

Code	errno	Value	cc	Value	Description
-1	EIO	5	MTCC_FAILED_MISPLACED	11 X'B'	Failed. VOLSER misplaced in library.
-1	EIO	5	MTCC_FAILED_CATEMPTY	12 X'C'	Failed. Category empty.
-1	EIO	5	MTCC_FAILED_MANEJECT	13 X'D'	Failed. Volume that is ejected manually.
-1	EIO	5	MTCC_FAILED_INVENTORY	14 X'E'	Failed. Volume not in inventory.
-1	EIO	5	MTCC_FAILED_NOTAVAIL	15 X'F'	Failed. Device not available.
-1	EIO	5	MTCC_FAILED_LOADFAIL	16 X'10'	Failed. Irrecoverable load failure.
-1	EIO	5	MTCC_FAILED_DAMAGED	17 X'11'	Failed. Cartridge damaged and queued for eject.
-1	EIO	5	MTCC_COMPLETE_DEMOUNT	18 X'12'	Completed. Demount signaled before execution.
-1	EIO	5	MTCC_NO_LMCP	32 X'20'	Failed. LMCP not configured.
-1	EINVAL	22	MTCC_NOT_CMDPORT_LMCP	33 X'21'	Failed. Device not command-port LMCP.
-1	EIO	5	MTCC_NO_DEV	34 X'22'	Failed. Device not configured.
-1	EIO	5	MTCC_NO_DEVLIB	35 X'23'	Failed. Device not in library.
-1	ENOMEM	12	MTCC_NO_MEM	36 X'24'	Failed. Memory failure.
-1	EIO	5	MTCC_DEVINUSE	37 X'25'	Failed. Device in use.
-1	EIO	5	MTCC_IO_FAILED	38 X'26'	Failed. Unexpected I/O failure.
-1	EIO	5	MTCC_DEV_INVALID	39 X'27'	Failed. Invalid device.
-1	EIO	5	MTCC_NOT_NTFPORT_LMCP	40 X'28'	Failed. Device not notification-port LMCP.

Table 13. Error description for the library I/O control requests (continued)

Code	errno	Value	cc	Value	Description
-1	EIO	5	MTCC_INVALID_SUBCMD	41 X'29'	Failed. Invalid subcommand parameter.
-1	EIO	5	MTCC_LIB_NOT_CONFIG	42 X'2A'	Failed. No library device configured.
-1	EIO	5	MTCC_INTERNAL_ERROR	43 X'2B'	Failed. Internal error.
-1	EIO	5	MTCC_INVALID_CANCELTYPE	44 X'2C'	Failed. Invalid cancel type.
-1	EIO	5	MTCC_NOT_LMCP	45 X'2D'	Failed. Not LMCP device.
-1	EIO	5	MTCC_LIB_OFFLINE	46 X'2E'	Failed. Library is offline to host.
-1	EIO	5	MTCC_DRIVE_UNLOAD	47 X'2F'	Failed. Volume is still loaded in drive.
-1	ETIMEDOUT	78	MTCC_COMMAND_TIMEOUT	48 X'30'	Failed. Command that is timed out by the device driver.
-1	EIO	5	MTCC_UNDEFINED	-1 X'FF'	Failed. Undefined completion code.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries (or 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 may send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country (or region) 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 (or regions) do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement cannot apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes are incorporated in new editions of the publication. IBM may make improvements and/or changes in the products and/or programs 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.

The ITDT-SE and ITDT-GE software uses Henry Spencer's regular expression library that is subject to the following copyright notice:

Trademarks

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

AIX	IBMLink	RS/6000®	System z®
AIX 5L™	Magstar	S/390®	Tivoli®
FICON®	Micro Channel	StorageSmart	TotalStorage
HyperFactor®	Netfinity	System i®	Virtualization Engine
i5/OS™	POWER5	System p	xSeries
iSeries	ProtecTIER®	System Storage	z9
IBM	pSeries	System x	zSeries

Adobe and Acrobat are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks of Intel Corporation in the United States, other countries (or regions), or both.

Java™ and all Java-based trademarks are trademarks of Oracle, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and Windows 2000 are trademarks of Microsoft Corporation in the United States, other countries (or regions), or both.

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

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

Index

Numerics

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

V

W

Readers' Comments — We'd Like to Hear from You

IBM Tape Device Drivers
Programming Reference

Publication No. GA32-0566-09

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Send your comments to the address on the reverse side of this form.

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
GA32-0566-09

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Building 9032-2 Department GZW
Information Development
9000 South Rita Road
Tucson, AZ
USA 85744-0002

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in USA

GA32-0566-09