User-mode API for Tape Libraries

Aram Khalili

Computer Science Department
University of Maryland Baltimore County
1000 Hilltop Circle
Baltimore, MD 21250
akhali1@umbc.edu

Abstract

Tape libraries are becoming more commonplace in various installations, whether they are used for automatic backups, archive or on-line storage. This work represents a freely distributable user-mode C-library that implements various SCSI commands of the Exabyte EXB-210 unit.

1 Introduction

Tape libraries are becoming more and more common in installations where large amounts of data need to be stored but also retrieved often for analysis. Up until recently mass storage systems were used only at scientific facilities but due to the development of new applications such as digital libraries, and electronic commerce the demands for storing enormous amounts of information while at the same time providing efficient access to this information is becoming commonplace. The original project at the Laboratory for Information Systems Technology called for development of a file system that integrates tape libraries with disk storage in a transparent manner. The project has been stopped, but our results are freely available. We chose Linux as the operating system to support this project since access to the source code, internals documentation, and a large support group were necessary requirements [2, 5].

In order to understand the behavior, in terms of performance, of the robotic arm that performs the mount operations in our Exabyte EXB-210 tape library we developed a user mode API for access to the SCSI medium changer command set [3]. Our interface is built on top of the generic SCSI interface (sg driver) presently available in the Linux kernel. Based on our knowledge there is currently no other driver or utility available for any of the public domain Unix operating systems for managing automated tape libraries.

2 The library's SCSI Interface

The SCSI-2 Interface standard[1] is a standard that enables a host computer and peripheral devices, such as this tape library and its tape drives, to communicate. The library and tape drives are independent SCSI devices, and each supports an independent set of SCSI commands.

2.1 Architecture of the SCSI Interface

The physical components of a SCSI system consist of the following:

host adapter: A device which connects between a host system and the SCSI bus. The device usually performs the lowest layers of the SCSI protocol and normally operates in the initiator role.

initiator: A computer equipped with a SCSI host adapter card enabling it to send commands, messages and data across the SCSI bus to targets. The initiator can also receive data, messages and status information from the targets.

targets: Devices capable of receiving commands from an initiator. The tape library and the tape drives are independent targets. The library is the target for cartridge inventory and movement commands, while the tape drive(s) is the target for read and write operations.

SCSI bus: The SCSI cables that connect initiators and targets.

2.2 Architecture of the tape library

The tape library includes the following types of components:

medium changer element: This is the robotic arm that moves the tapes. There is exactly one medium changer element in this unit. The medium changer is also called the cartridge handling mechanism (CHM) and the transport element.

data transfer element: This is a tape drive, of which there can be one or two in our unit.

storage element: This is a cartridge holder. There are either 11 or 21 storage elements in a EXB-210 tape library. It is also called storage location. The structure that contains 10 of the storage elements is called a magazine or a cartridge magazine.

cartridge: Tapes. There can be 0 to 21 tapes in the library (plus 3 for two tape drives and one medium changer, theoretically).

bar code scanner: A bar code scanner that may be mounted on the medium changer. This component is optional, i.e. there can be 0 or 1 bar code scanner in the unit. It scans bar code labels (if installed) on the cartridges.

2.3 SCSI commands supported by the library

The EXB-210 unit has two kinds of SCSI devices, the tape drive(s) and the medium changer, each supporting a different SCSI command set. Out of the list of commands that the tape library supports (see [4]) we have implemented this subset of commands: INQUIRY, MODE SENSE, MODE SELECT, INITIALIZE ELEMENT STATUS, MOVE MEDIUM, POSITION TO ELEMENT and READ ELEMENT STATUS.

2.4 INQUIRY

The INQUIRY command requests that the library sends information regarding its static parameters to the initiator. INQUIRY is used to obtain information such as vendor and product ID, firmware code revision levels, serial number, availability of an optional barcode scanner in the library and support of various SCSI-2 and other options.

2.5 MODE SENSE

The MODE SENSE command is used to discover the current operating mode parameters of the library. It can return useful information such as the number and addresses of tape storage locations (11,21), the number of medium changers (1), and the number of tape drives (1-2). It gives information about the parameters that are valid for the MOVE MEDIUM command, i.e. which types of moves the library supports, whether LCD security mode (access restrictions) is enabled, the LCD, and whether SCSI bus parity is enabled. It can also be used to determine default settings, and currently saved (in non-volatile memory) settings.

2.6 MODE SELECT

MODE SELECT allows one to define the current operating parameters for the library. It allows one to set the parameters which can be discovered by MODE SENSE. Note that parameters are constant and cannot be changed.

2.7 INITIALIZE ELEMENT STATUS

This command causes the library to check all elements for tapes, and the tape's barcodes, if a barcode reader is installed. This includes the tape drive and the medium changer, although no action is taken on an INITIALIZE ELEMENT STATUS targeting the medium changer, since this information is always maintained and assumed to be accurate. Checking is done in increasing element address order.

2.8 POSITION TO ELEMENT

This command requests the medium changer to position itself vertically (for a standalone unit; horizontally for rack mounted systems) in front of the requested address. If the medium changer's element address is given as the destination, the changer positions itself in a park position, giving (manual) access to the tape drive(s) and tape magazine(s). This command can also be used to reduce travel time if the next move command's source address can be predicted, or to position the changer at a place where the average distance to any location is minimized.

2.9 MOVE MEDIUM

The MOVE MEDIUM command causes the medium changer, if possible, to move the cartridge from the source to the destination address. Normally this is between the tape magazine and the tape drive(s), but one could also exchange the magazine location of a tape, to minimize the travel distance for frequently used tapes. If the destination address is one of a tape drive, the library will insert the cartridge. Valid source and destination element combinations can be discovered by the MODE SENSE command.

2.10 READ ELEMENT STATUS

This command allows one to discover the status of the elements of the library. Elements are the tape drive, the medium changer, and the storage locations. Information that can be obtained about the elements include whether they are in a normal state, the location of the transport element, whether a storage slot is filled with a tape and whether a tape drive or the medium changer contains a tape.

3 Linux's SCSI Interface

Probably ever since its inception Linux included support for SCSI devices. The main functionality used in the command library (mid-level) lies in */usr/src/linux/drivers/scsi/scsi.c* and (higher level) in */usr/src/linux/drivers/scsi/sg.c* It was not necessary to explicitly call hardware-level code (adapter drivers). The purpose of this interface is to provide a simple and consistent abstraction of a SCSI device that allows user-process control and follows file-system command syntax. *Scsi.c* facilitates the kernel with the mechanisms to scan the SCSI bus for devices (using INQUIRY), issue and queue commands to these devices and maintains the kernel data structures about them. It also implements exception handlers, such as SCSI time-out, abort and reset routines, and re-entrance prevention for the low-level driver. It also supplies functions for registering and removing removing drivers and modules.

On top of *scsi.c* sit *sg.c, sd.c* and *st.c*. *Sg.c* provides the file-system interface for generic SCSI devices utilizing the *scsi.c* functions. It implements the *open, close, read, write* and *ioctl* calls that abstract the SCSI devices as files. Linux uses this to create the special device files */dev/sg**, where a letter corresponding to the device's discovery during intial scan follows the *sg*, e.g. */dev/sga* for the first discovered device (the one with the lowest SCSI ID). These device files can be accessed through the normal file operations. An *open* system call will return the usual file descriptor, which can be passed to *write* to issue commands to the device. The data for the command is the actual SCSI command appended to an *sg_header* structure, which the user has to provide. The structure includes information about the command, a return value field and a buffer with the command descriptor block. After the *write* finishes, a *read* is used to read back the results.

Sd.c and *st.c* provide similar abstraction for disk and tape devices, respectively. They were not used for this part of the project which only deals with the medium changer device.

4 Tape Library API

The functions we developed for access to the medium changer command set are the following:

```
int inquiry(int fd, inquiry_t *inq,
            u_char type);

int mode_sense(int fd,
               __u16 pagecontrol,
               __u16 pagecode,
               elt_addr_assgn_t *eas,
               tgd_t *tgd,
                dev_cap_t *dcp,
                LCD_t *LCD,
                parity_t *par);

int mode_select(int fd, __u16 save,
                elt_addr_assgn_t *eas,
                LCD_t *LCD,
                parity_t *par);

int init_elt_status(int fd, __u18 nbl);

int move_medium(int fd, __u16 tea,
                __u16 src, __u16 dest);

int position_elt(int fd, __u16 tea,
                 __u16 dest);

int read_elt_stat(int fd,
             elt_stat_req_t *esr,
             elt_stat_data_t *esd,
             elt_stat_page_t **esp,
             stor_elt_desc_t **sed,
             data_transf_elt_desc_t **dted,
             med_transp_elt_desc_t **mted);
```

4.1 inquiry

Inquiry takes as arguments the file descriptor to the special device file, a pointer to a pre-allocated *inquiry_t* structure for the data that the command should return, and the type of inquiry requested. Valid values for the type are 0 for standard inquiry data, 1 for the Supported Vital Product Data page, and 2 for the Unit Serial Number page. It returns the number of bytes read from the SCSI bus.

4.2 mode_sense

Mode_sense takes as argument the file descriptor to the special device file, the control type, the page code, and pointers to *elt_addr_assg_t, tgd_t, dev_cap_t, LCD_t* and *parity_t*. Valid control type arguments are 0 for the current operating values of the library, 1 for a *changeable value* mask, which sets all bits of unchangeable values to 0 and all bits of changeable values to 1, and 2 for the default values. The page code holds information about the type of pages requested. The following values are valid:

`1Dh` - Element Address Assignment page.

`1Eh` - Transport Geometry Descriptor page.

`1Fh` - Device Capabilities page.

`22h` - LCD mode page.

`00h` - Parity page.

`3Fh` - All pages in above order.

The pointers to the structures need only be pre-allocated if the corresponding page is requested, otherwise `NULL` may be passed. *Mode_sense* returns the number of bytes read from the SCSI bus.

4.3 mode_select

Mode_select's arguments are similar to those for *mode_sense*, except that *mode_select* includes the option to save the sent values to non-volatile memory (1 to save, 0 otherwise), and that only structures that can be modified are passed. They need to be pre-allocated and set to the desired values if one wants to change their values, otherwise they may be `NULL`. *Mode_select* returns the number of bytes read from the SCSI bus.

4.4 init_elt_status

Init_elt_status takes the file descriptor and an indicator of whether the library should attempt to scan bar codes as arguments. Set *nbl* to 0 scan for bar codes labels, and to 1 to omit. It returns the number of bytes read from the SCSI bus.

4.5 move_medium

Move_medium takes the file descriptor and the element addresses of the transport element, and the source and destination addresses. It returns the number of bytes read from the SCSI bus.

4.6 position_elt

Position_elt takes as arguments the file descriptor and the element addresses of the transport element and its destination. It returns the number of bytes read from the SCSI bus.

4.7 read_elt_status

Read_elt_status takes the file descriptor, and pointers to various structures as arguments. *elt_stat_req_t* describes the type of request, i.e. whether bar code information should be returned (0 for no, 1 for yes), and which type of elements should be reported on, 0 for all, 1 for the transport element, 2 for the storage elements, and 4 for the data transfer elements. Upon successful completion, *elt_stat_data_t* will hold the element address of the first element, the number of elements, and the number of bytes returned. *Elt_stat_page_t* holds the type of element reported, the length in bytes of information per element, and the total number of bytes of information. Up to three records, one record per element type will, will be returned, hence the double pointer. The next structures hold information on a particular element. Multiple records may be returned. The structure pointers passed to *read_elt_status* should not be pre-allocated. It returns 1 for success and 0 for failure.

4.8 Additional functionality

In addition to the above implementation of the meadium changer's SCSI commands we found it necessary or useful to write the following routines and programs. The routines are for the control of the tape drive of the library, and the programs encapsulate some of these functions to be more useable. The routines are

```
int load(int fd, int luflag);

long get_position(int fd);

long set_position(int fd, long pos);

int rewind_tape(int fd);

int set_block_length(int fd, int blocksize);
```

All functions take as their first argument the file descriptor for the special device file. *Load* takes a load/unload flag as an additional argument. Setting the flag to 1 causes the tape drive to load a tape inside of it, 0 causes it to unload. If tape drive is not in auto-load mode, one needs to load a tape, once it has been inserted into the drive, before one can perform I/O on it. An unload will eject the tape. The command returns the number of bytes read from the SCSI bus. *Get_position* takes no additional arguments and returns the current offset in blocks of the tape. *Set_position* takes an offset as another argument and seeks the tape to the desired location. It returns the number of bytes returned by the *ioctl* command. *Set_block_length* takes the integer blocksize as an additional argument and tries to set the current blocsize to the desired value. Valid values depend on the blocksizes supported by the tape drive. It also returns the number of bytes returned by the *ioctl* call.

At this point the following programs exist:

```
move <device> <src> <dest>

load <device> [load/unload flag]

eject <device>

position <device> <dest>

read_stat <device>
```

All the programs take the device file as their first argument. Additonal arguments passed to *move* are the source and destination element addresses. It assumes the default transport element address of 86. *Load* takes an optional load/unload flag. If it is ommitted a load is performed on the device, if it is given, it is passed to the *load* function, which will load on a 1 and unload on a 0. *Eject* is just a more familiar form of *load* ⟨*device*⟩ *0*. *Position* takes the block offset as another parameter and seek to the specified block. *Read_stat* takes no additional arguments and implements the READ ELEMENT STATUS command for all elements.

For a more detailed discussion of the functions and their data structures we direct the reader to the source code and accompanying material, which will be available at *ftp://ftp.cs.umbc.edu/pub/exb210/*.

4.9 Use of the API

We used the API functions to collect performance measurements of the unit. We generated a workload of file accesses using a uniform distribution over the interval [0,8] in seconds for interarrival time of successive requests, a uniform distribution over the interval [1,40] in kilobytes for file sizes, uniform distribution over [1,10] of tape IDs. One process read the simulated workload and issued requests to another process, which interpreted the request and, using the API calls, fetched the right tape, loaded it, seeked to the given location, read or wrote the specified number of blocks, and collected timing information. Reads and writes were done using Linux's *st* interface, seeks with our *set_position* function. When we ran it, one thing we immediately noticed is that the library cannot service requests coming in at such a high rate, so in our next workload we changed the interarrival time to a uniform distribution over [0,40], which the device could handle. This indicates that this device by itself will not give adequate interactive on-line performance, but will work well as an automatic backup device. The most expensive operation time-wise was an eject. The manual does not give information about the device's caching scheme, and other things relevant to the performance of the individual commands, so we are left guessing as to why the unit behaves as it does. Another observation we made is that in sequential reads, if a 32 block boundary is crossed, the read command will take 2 orders of magnitude longer complete. A similar phenomenon happens during writing. Otherwise read and write completion time vary by a small amount.

5 Conclusion

We have developed a freely distributable library of functions that implement SCSI commands on a EXB-210 tape library and used it to collect performance measurements of the device. We see our library as a helpful tool to other programmers desiring to write programs that access the functionality of the EXB-210 unit for application development, and we have included some command encapsulation for use in such things as automatic backup scripts.

References

[1] ANSI, Small Computer Systems Interface-2 (SCSI-2), X3T9/89-042

[2] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, "Linux Kernel Internals", Addison-Wesley, 1996.

[3] Exabyte Corporation, "Installation and Operation", 1994.

[4] Exabyte Corporation, "EXB-210 and EXB-220 8mm libraries: SCSI Reference", 1996.

[5] David Rusling, "The Linux Kernel", The Linux Documentation Project, 1996.

[6] Friedhelm Schmidt, "The SCSI Bus and IDE Interface: Protocols, Applications, and Programming", Addison-Wesley, 1996.