Fermi Tape Tools (ftt) Library User's Guide

Chuck Debaun, **Marc W. Mengel**
Operating Systems Support Department

Marilyn Schweitzer
High Performance and Parallel Computing Department

Dorota Genser, Gene Oleynik, **Margaret Votava**
Online Systems Department
Fermilab Computing Division

ABSTRACT

The Fermi Tape Tools library is a C subroutine package that provides consistent, platform-independent tape I/O for a variety of tape drives on a variety of platforms. It provides basic tape input and output, support for obtaining and tracking available tape statistics, and various informational and support routines for tools which need to allocate and control tape usage.

This users guide will provide an overview of the **ftt** library calls, take the reader through the process of writing a simple **ftt** based tape copy program, and also provide a complete call reference to the routines provided for tape input, output, and information.

1 Getting Started

1.1 Installation

The **ftt** product is distributed as a Fermilab **ups** product, and should be installed as such, as root if possible. There is an **INSTALL** file in the product giving details of non-**ups** installation. Important notes on prerequisites for various platforms are in Chapter 14 of this document.

1.2 Basic Concepts

The **ftt** library is used by creating a **descriptor** for each tape drive being used and making various **ftt** calls using that descriptor. The **descriptor** itself contains information about what system **devices** are associated with the tape drive, what densities and formats are available, etc. To get a **descriptor** for a particular tape drive, you need to know the **basename** of the drive, which is basically a common substring of the various system devices associated with the tape drive. Another important data structure is the **ftt_stat_buf** which is used to hold tape drive statistics obtained from the tape drive. These types, as well as all of the function prototypes, preprocessor definitions, etc. are found in the **ftt.h** header file.

1.3 Starting and Finishing

The **ftt** library can be used either to do tape input and output, to manage tape drives, to get information about tape devices, or all of the above. All of these activities start with a call to **ftt_open** to get an **ftt** descriptor.

```
#include <ftt.h>
ftt_descriptor d;
d = ftt_open("/dev/rmt0", FTT_RDONLY);
/* do some stuff here */
ftt_close(d);
```

There are several itemuser_guide_maintext1.mifs that should be mentioned here.:

- The header file **ftt.h** is included. This file contains the various preprocessor symbols and function declarations for users of the **ftt** library.
- We declare the **descriptor** variable with a type named **ftt_descriptor**.
- We pass in a read-only/read-write flag to the open call.
- **ftt_close** frees up all the memory associated with the **descriptor** and closes any open system devices associated with the drive.

If your program needs to deal with multiple tape drives, it can call **ftt_open** multiple times, to get a unique **descriptor** for each drive. It should then issue an **ftt_close** on each descriptor it got from **ftt_open** before exiting

1.4 Doing Input and Output

Once you have a **descriptor**, very often you want to read data from the associated drive, or write data to it. The **ftt** library calls to read and write data look very much like the UNIX system calls **read** and **write** and are called (you guessed it) **ftt_read** and **ftt_write**. Both take as arguments the **descriptor** for the tape drive involved, a character pointer to the data buffer, and the number of bytes requested to be read or written. Both return the number of bytes actually transferred, or -1 to indicate an error. The calls look like:

```
length = ftt_read(d, buffer, buffer_length);
```

or;

```
length = ftt_write(d, buffer, data_length);
```

2 Finding out what went wrong

If your **ftt** calls return an error (a value of -1 for **ftt_read** or **ftt_write** or a value of 0 from **ftt_open** for example), you may want to know specifically what went wrong. To find out the specific error condition, you can use the routine called **ftt_get_error** which gives you an error string and (optionally) an error number. If you simply want to tell the user what went wrong, you can do something as simple as:

```
fprintf(stderr, "%s\n", ftt_get_error(0));
```

which will print the error string on standard error. If your program wants to attempt to do something to recover from errors, you will need to pass a pointer to an integer into the first argument of **ftt_get_error** which will give you back an error number. Then you can attempt to take appropriate action based on the error. This looks like:

```
char *p;
int n;
...
p = ftt_get_error(&n);
if (n == FTT_ENOTAPE) {
      request_tape_mount();
}
```

The error numbers you get from **ftt_get_error** will be one of the following:

TABLE 1. ftt error codes

ftt_errno	Description
FTT_SUCCESS	no error has occurred
FTT_EPARTIALSTAT	unable to obtain some or all of the information requested
FTT_EUNRECOVERED	previous error was unrecoverable, tape can only be unloaded/rewound (to protect data)
FTT_ENOTAPE	there is no tape in the drive
FTT_ENOTSUPPORTED	Unsupported device
FTT_EPERM	you do not have permission to perform the specified operation on the device.
FTT_EFAULT	you gave a buffer address range whose access yields a memory fault
FTT_ENOSPC	A write attempted to cross end of media
FTT_ENOENT	User attempted to set density to a nonexistent one
FTT_EIO	Physical read or write error.0
FTT_EBLKSIZE	Attempt to write/read unsupported block size s
FTT_ENOEXEC	Attempt to run setuid executable for security bypass failed -- not executable
FTT_EBLANK	Encountered blank tape (i.e. passed logical EOT)

TABLE 1. ftt error codes

ftt_errno	Description
FTT_EBUSY	Asynchronous I/O is going on this device, or another process has it open
FTT_ENODEV	System does not have drive configured
FTT_ENXIO	Tried to contact a nonexistent process
FTT_ENFILE	No system file descriptors available
FTT_EROFS	Tape is write protected
FTT_EPIPE	Setuid bypass died
FTT_ERANGE	Record too large for buffer
FTT_ENOMEM	Unable to allocate memory required to perform call
FTT_ENOTTAPE	What we thought was a tape device was not.
FTT_2SMALL	The block size issued is smaller then the device can handle
FTT_ERWFS	Tape is not write protected (and should be)
FTT_EWRONGVOL	Wrong volume label on tape for verification
FTT_EWRONGVOLTYP	Wrong type of label on tape for verification
FTT_ELEADER	hit tape leader while skipping backwards
FTT_EFILEMARK	hit file mark while skipping records
FTT_ELOST	The tape postion is unknown
FTT_ENOTBOT	The tape is not at BOT

Your code can compare the error number (**n** in our example) to the values in this table to decide what went wrong in the preceding **ftt** library call.

3 Opening/closing Device

In general, to access a device, the user need simply call **ftt_open()**. He can then perform normal tape operations (e.g., read, write, rewind). **ftt_open()** doesn't actually open this device with the unix device driver. It simply creates/initializes the **ftt** file descriptor. Once the descriptor is created, the user can explicitly open a thread to the device driver with **ftt_open_dev()**, although this is not necessary. All **ftt** routines will internally call **ftt_open_dev()** if the device has not yet been opened.

A utility routine, **ftt_open_logical()**, is provided to create an **ftt** descriptor for a possibly fictitious device. It provides the opportunity to see what the various device names would be for specific modes/densities on a different operating system platform. The average user would not use this routine to open the device, but would use **ftt_open()** instead.

The pair of close routines, **ftt_close()** and f**tt_close_dev()**, are similar in function to the pair of open routines, f**tt_open()** and **ftt_open_dev()**. In general, the user need simply call **ftt_close()** when finished with the device. This routine will internally call **ftt_close_dev()** if needed. **ftt_close_dev()** will ensure that the proper number of filemarks have been written at the end of tape (if writing) and that the tape is correctly positioned. **ftt_close()** will free the **ftt** file descriptor.

It's important to note that some **ftt** routines will internally call **ftt_close_dev()**, which may write filemarks, if they need to close the normal unix device driver and open the raw scsi device. The documentation for the individual subroutines will indicate if a specific routine exhibits this behavior.

Note that several f**tt_open_dev()** and f**tt_close_dev()** pairs can be called within a single **ftt_open()** and **ftt_close()** pair.

3.1 ftt_descriptor **ftt_open**(char ***basename,** int **rdonly**)

opens a *basename* for a device and returns a *descriptor,* which can be used to perform other operations on the device. The *basename* argument can also be a full device name for a tape device, and will internally be trimmed to a basename.

The *rdonly* argument indicates

TABLE 2. rdonly flag values

Value	Description
FTT_RDONLY	Device will not be written on
FTT_RDWR	Device is read and writable.

Note that FTT_RDONLY does not check that the device is physically write protected, nor does FTT_RDWR check that is not physically write protected; a call to **ftt_status** is required to obtain that information.

This routine performs no positioning of the device.

Note that **ftt_open** and **ftt_close** do not need to be called to switch between reading and writing, to cross filemarks, etc. The ftt software will perform operating system opens and closes as needed to allow those transitions.

ftt_open does not perform any operations on the tape drive or tape devices, it simply does bookkeeping to prepare for such operations.

Example:

```
#include "ftt.h"
ftt_descriptor d;
d = ftt_open("/dev/rmt/tps2d4", FTT_RDWR);
```

3.2 ftt_descriptor **ftt_open_logical** (char ***basename,** char ***os,** char ***driveid,** int **rdonly**)

Create a (possibly mythical) ftt_descriptor for a device of type *driveid* on operating system *os*, using base device name *basename*, with *rdonly* as in **ftt_open**. This call will be used internally to construct a descriptor, but can also be used in combination with other **ftt** calls to see what the name of the high density EXB-8500 tape device would be on IRIX from an IBM system. Performing any actual tape-io from this descriptor will probably fail unless the *os* and *driveid* and *basename* are valid for the current system.

Example:

```
#include "ftt.h"
ftt_descriptor d;
d = ftt_open_logical("/dev/rmt/tps2d4", "IRIX", "EXB-8500",
FTT_RDWR);
```

3.3 int **ftt_open_dev** (ftt_descriptor **d**)

Explicitly perform a UNIX open of the device for the currently selected density, etc. Routines which need the device open to work will call **ftt_open_dev** if they need to.

If the density and mode have not been set by **ftt_set_mode(), ftt_set_mode_dev(),** etc., the default density and compression mode of the highest density the device supports and no compression will be used.

The routine may sets appropriate modes and/or flags on the underlying device to allow other ftt operations to proceed properly.

ftt_open_dev performs antecessor operations to set the device density, etc. and actually performs an operating system open on the device. No tape motion is performed.

Example:

```
#include "ftt.h"
ftt_descriptor d;
int status;
status = ftt_open_dev(d);
```

3.4 int **ftt_close** (ftt_descriptor **d**)

The **ftt_close** entry releases a descriptor, indicating that no further operations will be performed on that descriptor. If a device has been opened not closed, an **ftt_close_dev** will be performed.

ftt_close will invoke **ftt_close_dev** if needed, but performs no other tape or device operations.

Example:

```
#include "ftt.h"
ftt_descriptor d;
int status;
status = ftt_close(d);
```

3.5 int **ftt_close_dev**(ftt_descriptor **d)**

This call closes the system device which was opened by **ftt_open_dev**, after ensuring proper double file mark termination of written data. To ensure this, if writes have been successfully performed since the last rewind or other tape motion, it writes two filemarks and backspaces over them before closing the device. Note that this means that if you are writing, perform an **ftt_close_dev** followed by an **ftt_open_dev**, and continue writing, you will overwrite the filemarks that **ftt_close_dev** wrote.

Note that **ftt_close_dev** can fail, due to insufficient room to write tape filemarks, etc.

ftt_close_dev performs a close on the operating system device. It will only perform tape motion (writing filemarks and backspacing) if the preceding operation was a write or write filemark. No other operations or tape motions are performed.

Example:

```
#include "ftt.h"
ftt_descriptor d;
int status;
status = ftt_close_dev(d);
```

4 Reading/Writing

All of these routines return -1 on failure, for which more information can be obtained as described in the section on *Finding out what went wrong.*

4.1 int **ftt_read** (ftt_descriptor **d,** char ***buf,** int **length**)

If called with the tape positioned before a data block, the **ftt_read** entry reads a block of length at most *length* from the tape into buffer *buf*, and returns the length of the block actually read. If the next block on the tape will not fit into the buffer, or if any other error occurs, -1 will be returned

If called with the tape positioned before an end-of-file mark, 0 is returned, and the position is moved to just after the end-of-file mark.

Example:

```
#include "ftt.h"
ftt_descriptor d;
char data[32768]
int status;
status = ftt_read(d,data,32768);
```

4.2 int **ftt_write** (ftt_descriptor **d, char *buf,** int **length**)

The ftt_write entry transfers as much as possible of the block in buffer *buf* of *length* to the tape, and returns the length written, unless an error occurs, in which case -1 is returned. The tape is positioned after the written block.

In the case of an unrecoverable error, an error number of FTT_UNRECOVERED will be returned for this call, and all subsequent calls made until one of **ftt_rewind, ftt_unload, ftt_close_dev,** or **ftt_close** is called.

Example:

```
#include "ftt.h"
ftt_descriptor d;
char data[32768]
int status;
status = ftt_write(d,data,32768);
```

4.3 int **ftt_writefm** (ftt_descriptor **d**)

The ftt_writefm entry writes a single filemark to the tape. After the call, the tape position is after the written filemark.

Example:

```
#include "ftt.h"
ftt_descriptor d;
int status;
status = ftt_writefm(d);
```

4.4 int **ftt_write2fm** (ftt_descriptor **d)**

The ftt_write2fm call writes 2 filemarks to the tape. After the call, the tape position is after the second filemark so written. After making this call, **ftt** "forgets" that it has been writing, since the guarantee that there is a double file mark at the end of tape has been established.

4.5 **ftt_retry (**ftt_descriptor **d**, int **tries,** (***op**) (ftt_descriptor, char *, int), char ***buf**, int **len**);

The **ftt_retry** call calls the routine passed in as **op**, with arguments **d, buf,** and **len**, and if the call fails, retries it up to **tries** more times. Before retrying, **ftt_retry** will skip backwards to the nearest filemark, then skip forward to just after the last successfully written block, and retry the operation. The recovery sequence is correspondingly sluggish on some tape drives.

The **op** passed in should be either **ftt_read** or **ftt_write**.

Note that if retrying **ftt_write** calls, the reverse skip will attempt to write 2 filemarks, making the recovery even slower.

This call will fail if:

- any of the file or record skips fail
- the number of retires is exceeded without a successful write

The return code will be the result of the last operation, either **ftt_skip_fm, ftt_skip_rec,** or the return code from the passed in **op**.

Example:

```
char buf[2048]
res = ftt_retry(d, 5, ftt_read, buf, 2048);
```

will make up to a total of 6 read calls, interspersed with file and record skips.

5 Moving around on the tape

Tape drives allow you to do more then just read and write, they also allow you to move around the tape, and perform other operations. The **ftt** library contains several calls for tape motion.

All of these routines return -1 on failure, for which more information can be obtained as described in the section on *Finding out what went wrong.*

5.1 int **ftt_skip_fm** (ftt_descriptor **d,** int **count)**

skips forwards to just after a filemark, or backwards to just before a filemark. *count* indicates the number of filemarks to skip. *count* should be negative to skip backwards.

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
status = ftt_skip_fm(d, 3);
```

to skip forward 3 filemarks, or:

```
status = ftt_skip_fm(d, -2);
```

to skip backwards 2 filemarks.

Note that skipping backwards will write two filemarks and skip back over them first if the preceding operation was an **ftt_write** or **ftt_writefm** call.

5.2 int **ftt_skip_rec** (ftt_descriptor **d,** int **count)**

skips forward just past records, or backwards just before records. *count* indicates the number of records to skip. *count* should be negative to skip backwards.

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
status = ftt_skip_rec(d, 3);
```

to skip forward 3 records, or:

```
status = ftt_skip_rec(d, -2);
```

to skip backwards 2 records.

Note that skipping backwards will write two filemarks and skip back over them first if the preceding operation was an **ftt_write** or **ftt_writefm** call.

5.3 int **ftt_skip_to_double_fm** (ftt_descriptor **d)**

skips forward until it finds two filemarks in a row, and positions the tape between the two filemarks. This is often useful for appending to unlabeled tapes.

Example:

```
#include "ftt.h"
```

```
int status;
ftt_descriptor d;
status = ftt_skip_to_double_fm(d);
```

5.4 int **ftt_rewind** (ftt_descriptor **d)**

skips backwards to the beginning of tape.

Example:

```
#include “ftt.h”
int status;
ftt_descriptor d;
ftt_rewind(d);
```

5.5 int **ftt_retension** (ftt_descriptor **d)**

retensions and rewinds tape.

Example:

```
#include “ftt.h”
int status;
ftt_descriptor d;
ftt_rewind(d);
```

5.6 int **ftt_unload** (ftt_descriptor **d)**

ejects the tape from the tape drive (if possible).

Example:

```
#include “ftt.h”
int status;
ftt_descriptor d;
ftt_unload(d);
```

5.7 int **ftt_erase** (ftt_descriptor **d)**

erases and rewinds the tape in the tape drive (if possible).

Example:

```
#include “ftt.h”
int status;
ftt_descriptor d;
ftt_erase(d);
```

6 Densities, Compression, and other Modes

The various tape drives generally read and write at a particular **density** (which refers to the amount of data per length or area of tape), and often have some densities they can read and write, some they can only read, etc. In addition drives can use **compression** to write less data on the tape. Also, many operating systems provide devices with various software-generated modes, such as having the drive rewound when the system device is closed, or retensioned when it is opened. The **ftt** library provides several calls for dealing with these various sorts of mode settings.

All of these routines normally return a character pointer to a string containing the name of whatever system device will be used to perform I/O in this mode. The routines all return 0 if anything goes wrong, see the section *Finding out what went wrong* for a discussion of errors.

6.1 char ***ftt_set_mode (**ftt_descriptor **d,** int **density,** int **cmp,** int **blocksize)**

lets you pick a specific density number in *density*, turn compression on or off in *cmp*, and set variable blocking, or a given fixed blocksize in *blocksize*. This mainly has effect for writing tapes, since most drives will automatically switch densities when reading to match the density at which the tape was written. *blocksize* is specified in bytes, with a *blocksize* of zero indicating variable block mode. The routine returns the name of the system device it will use for operations at that density, etc.

Example:

```
#include "ftt.h"
char *name;
ftt_descriptor d;
int density;
int cmp = TRUE;          /* compression on */
int blocksize = 0;       /* variable block */
name = ftt_set_mode(d, density, cmp, blocksize);
```

6.2 char ***ftt_get_mode (**ftt_descriptor **d,** int ***density,** int ***cmp,** int ***blocksize)**

tells you what density in *density*, compression in *cmp*, and blocksize in blocksize were last set by **ftt_get_mode**. This is mainly useful to find out what defaults were set, which should be the highest density, uncompressed format for the drive you are using. You pass in pointers to the integers you want back, so that the routine can return multiple values.

Example:

```
#include "ftt.h"
char *name;
ftt_descriptor d;
int density;
int cmp;
int blocksize;
name = ftt_get_mode(d, &density, &cmp, &blocksize);
```

6.3 char ***ftt_avail_mode** (ftt_descriptor **d,** int **density,** int **cmp,** int **fixed)**

tells you whether a call to **ftt_set_mode** with a given set of arguments should succeed, although it only checks whether the last argument is zero or nonzero, to indicate fixed blocksize or variable, respectively. It similarly returns the system device name that would be used when doing input/output at that density, etc.

Example:

```
#include "ftt.h"
char *name;
ftt_descriptor d;
int density;
int cmp = TRUE;          /* compression on */
int fixed = FALSE;       /* variable block */
name = ftt_avail_mode(d, density, cmp, fixed);
```

6.4 char ***ftt_density_to_name (**ftt_descriptor **d,** int **density)**

Translates an integer density number to a descriptive string or name.

6.5 int **ftt_name_to_density (**ftt_descriptor **d,** char ***name)**

Translates a name to an integer density number. The name can be any prefix of a density string which would be returned by **ftt_density_to_name()** for that device.

6.6 int **ftt_get_max_blocksize (**ftt_descriptor **d)**

Returns the largest blocksize usable on this drive type and platform, suitable for getting a block size to allocate for a read/write or copy buffer.

6.7 int **ftt_set_data_direction(**ftt_descriptor **d**, int **n)**

This sets the "data direction" flag, which is used on drives like Exabyte 8200's that do not provide real read and write counters, to decide whether i/o estimates based on remaining tape counters should be treated as read or write counts. Making this call is important if collecting tape statistics for tape I/O done by another process.

7 Dealing with System Device Names

The various operating systems have multiple device names which are associated with the same tape drive. For example, the IRIX operating system currently has 38 device nodes for a single 8500 tape drive.

7.1 char ****ftt_list_all** (ftt_descriptor **d)**

returns a pointer to a static list of device pathnames related to this drive.

ftt_list_all performs no tape operations. It returns the list of devices we think should be present, which is determined when **ftt_open_logical** is called.

Example:

```
#include "ftt.h"
char **all_names;
ftt_descriptor d;
all_names = ftt_list_all(d);
```

7.2 int **ftt_chall (**ftt_descriptor **d,** int **uid,** int **gid,** int **mode)**

All character special devices relating to the file named by descriptor have their ownership changed to the pass-ed in uid, gid. and are set to the given mode. This call will only succeed if called with the appropriate permissions, it will not be handled by a setuid executable.

ftt_chall performs no tape operations, it merely changes permissions on the devices in **/dev**.

7.3 int **ftt_get_mode_dev(**ftt_descriptor **d,** char ***devname,** int ***density,** int ***cmp,** int ***fixed,** int ***rewind)**

Returns the density in *density*, compression in *cmp*, fixed block flag, *fixed*, and rewind/retension status in *rewind* for the device *devname*.

The *rewind* value returned will have a value of **FTT_RWOC, FTT_RTOO**, or bitwise combinations of these values, to indicate ReWind On Close and ReTension On Open, respectively.

A *fixed* value of 0 indicates variable blocking, a *fixed* of 1 indicates fixed block operation, **but does not provide the current blocksize.**

ftt_get_mode_dev performs no tape operations.

Example:

```
 #include "ftt.h"
int status;
ftt_descriptor d;
char *devname = "/dev/my_favorite_device";
int density;
int cmp;
int fixed;
int rewind
```

```
status = ftt_get_mode_dev(d, devname, &density, &cmp, &fixed,
&rewind);
```

7.4 int **ftt_set_mode_dev** (ftt_descriptor **d,** char * **devname,** int **force**)

Sets the compression, density, etc. to match that of the device name passed in. The argument must be a name associated with the device opened by **ftt_open**.

The *force* flag (if set to one) forces the exact device name specified to be used, allowing access to devices which are otherwise unsupported.

ftt_set_mode_dev performs no tape operations, it merely sets flags, etc. which will be used by **ftt_open_dev.**

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
char *devname = "/dev/my_favorite_device";
int force = FALSE;
status = ftt_set_mode_dev (d, devname, force);
```

7.5 int **ftt_describe_dev**(ftt_descriptor **d,** char ***devname,** FILE *pf)

prints a description of the given device indicated in *devname*, listing the density, compression options, rewind modes, etc. on the file *pf*. It returns the number of bytes written on the file. This routine performs no tape operations.

Example:

```
#include <stdio.h>
#include "ftt.h"
int status;
ftt_descriptor d;
char *devname = "/dev/my_favorite_device";
status = ftt_describe_dev (d, devname, stdout);
```

7.6 char ***ftt_get_basename**(ftt_descriptor **d**)

returns the basename for the device; thereby the basename for a tape device can be found by doing an **ftt_open** followed by an **ftt_get_basename.**

ftt_get_basename performs no tape operations.

Example:

```
#include "ftt.h"
char *basename;
ftt_descriptor d;
basename = ftt_get_basename(d);
```

7.7 int **ftt_list_supported** (FILE ***pf)**

Prints a description of the device types and operating systems that ftt officially supports on the file descriptor **pf**. No ftt descriptors need to be open to make this call.

8 Tape Statistics

Tape drives (especially newer ones) maintain a wealth of information about how much data they have written, how many errors they encountered, etc. The following table lists the various statistics that ftt keeps track of and what type of statistic it is: string, numeric, counter, or flag. Note that all drives do not support all statistics.

TABLE 3. ftt statistics

stat	Description	Notes	Origin
FTT_VENDOR_ID	vendor id data	string	Drive:SCSI Inquiry
FTT_PRODUCT_ID	product id	string	Drive:SCSI Inquiry
FTT_FIRMWARE	firmware rev level	string	Drive:SCSI Inquiry
FTT_SERIAL_NUM	Device serial number	string	Drive:SCSI2 Inquiry
FTT_CLEANING_BIT	Device needs cleaning	flag	Drive: SCSI ReqSense
FTT_READ_COUNT	Decimal number of KBread since rewind/reset	counter	Drive: SCSI ReqSense (EXB specific) **or** SCSI2 Log Sense
FTT_WRITE_COUNT	Decimal number of KB written since rewind/reset	counter	(see above)
FTT_READ_ERRORS	Read errors - drive corrected	counter	(see above)
FTT_WRITE_ERRORS	Write errors - drive corrected	counter	(see above)
FTT_READ_COMP	Compression factor for reads	numeric	Drive: SCSI2 Log Sense
FTT_WRITE_COMP	Compression factor for writes	numeric	Drive: SCSI2 Log Sense
FTT_FILE_NUMBER	File number from BOT	numeric	Ftt: d->current_file
FTT_BLOCK_NUMBER	Block number in current file	numeric	Ftt: d->current_block
FTT_BOT	at beginning of tape	flag	Ftt: bookkeeping (above), **or** SCSI Request Sense, **or** SCSI Read Position
FTT_READY	Device ready status	flag	Drive: SCSI ReqSense **or** SCSI Test Unit Ready
FTT_WRITE_PROT	Device write protect status	flag	Drive: SCSI Mode Sense
FTT_FMK	At filemark	flag	Drive: SCSI ReqSense
FTT_EOM	At end of media (early warning)	flag	Drive: SCSI ReqSense
FTT_PEOT	At physical end of tape (really at end)	flag	Drive: SCSI ReqSense (EXB-Specific)
FTT_MEDIA_TYPE	Media type	numeric	Drive: SCSI ModeSense

TABLE 3. ftt statistics

stat	Description	Notes	Origin
FTT_BLOCK_SIZE	Tape block size	numeric	Drive: SCSI Mode Sense
FTT_BLOCK_TOTAL	Tape length in KB	numeric	Drive: SCSI Mode Sense
FTT_TRANS_DENSITY	Translated density ala ftt_get_mode	numeric	Ftt: table and FTT_DENSITY
FTT_TRANS_COMPRESS	Translated compression ala ftt_get_mode	numeric	(see above) **and** SCSI2 Mode Sense page 0x10
FTT_REMAIN_TAPE	Remaining KB of tape	counter	Drive: SCSI ReqSense
FTT_USER_READ	KB read since open/ unload	counter	Ftt: d->readkb
FTT_USER_WRITE	KB written since open/ unload	counter	Ftt: d->writekb
FTT_CONTROLLER	Controller type	string	Ftt: table
FTT_DENSITY	Tape density	numeric	Drive:SCSI Mode Sense
FTT_ILI	Invalid Length Indicator	flag	Drive: SCSI ReqSense
FTT_SCSI_ASC	Additional Sense Code	numeric	Drive: SCSI ReqSense
FTT_SCSI_ASCQ	Additional Sense Code Qualifier	numeric	Drive: SCSI ReqSense
FTT_PF	Power Failed!	flag	Drive: SCSI ReqSense (EXB specific)
FTT_CLEANDED_BIT	Cleaning performed per FTT_CLEANING_BITt	flag	Drive: SCSI ReqSense
FTT_TRACK_RETRY	Track retry errors	numeric	Drive:SCSI ReqSense (EXB 8x05 specific)
FTT_UNDERRUN	stop/start counter from	numeric	(see above)
FTT_MOTION_HOURS	hours tape drive is in motion	counter	Drive SCSI ReqSense (DLT specific)
FTT_POWER_HOURS	Number of hours device has been on	counter	Drive:SCSI ReqSense (DLT specific)
FTT_TUR_STATUS	Status	numeric	Drive:ScsiTestUnit Ready
FTT_COUNT_ORIGIN	Origin of read/write kb	string	Ftt
FTT_N_READS	Number of read calls	counter	Ftt: d->nreads
FTT_N_WRITES	Number of write and writefm calls	counter	Ftt: d->nwrites
FTT_TNP	Tape Not Present bit	flag	Drive: SCSI ReqSense (EXB specific)
FTT_SENSE_KEY	Sense Key	numeric	Drive:SCSI ReqSense
FTT_TRANS_SENSE_KEY	Above translated to strings	string	Ftt:
FTT_RETRIES	total ftt_retry() repeats	counter	Ftt:

TABLE 3. ftt statistics

stat	Description	Notes	Origin
FTT_FAIL_RETRIES	total ftt_retry() failures	counter	Ftt
FTT_RESETS	Unexpected BOT discoveries	counter	Ftt
FTT_HARD_ERRORS	Counter of recovered errors	counter	Ftt

8.1 ftt_stat_buf **ftt_alloc_stat (void)**

allocates a statistic buffer for use with the other ftt statistics routines.

Example:

```
#include "ftt.h"
ftt_stat_buf stbuff;
stbuff = ftt_alloc_stat();
```

8.2 int **ftt_free_stat**(ftt_stat_buf **stbuff)**

frees a statistic buffer indicated by *stbuff*.

Example:

```
#include "ftt.h"
int status;
ftt_stat_buf stbuff;
status = ftt_free_stat(stbuff);
```

8.3 int **ftt_get_stats**(ftt_descriptor **d,** ftt_stat_buf **stbuff)**

extracts current statistics for tape usage, including data written, data read, write errors, read errors, etc. and packs it into an ftt_stat_buf structure, *stbuff*. Note that because on some platforms we must close and reopen devices to implement this call, ftt_get_stats will always put filemarks on your tape if you have been writing (see **ftt_close_dev**).

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
ftt_stat_buf stbuff;
stbuff = ftt_alloc_stat();
status = ftt_get_stats(d,stbuff);
```

8.4 char ***ftt_extract_stats (**ftt_stat_buf **b,** int **stat)**

extracts a string describing a named piece of tape data, indicated by stat. Acceptable values for stat are listed in Table 2. The routine will return a null pointer (0) if a given statistic is not available in the status block.

ftt_extract_stats performs no tape operations.

Example:

```
#include <stdio.h>
#include "ftt.h"
char *value;                  /* value of extracted stat */
ftt_stat_buf stbuff;          /* statistics buffer */
int status, ftt_status;       /* status of ftt calls */
char *estring;                /* error string */
```

```
stbuff = ftt_alloc_stat();
status = ftt_get_stats(d,stbuff);
if (status == -1)
   {
   estring = ftt_get_error(&ftt_status);
   fprintf (stderr, "Error:%s Error String:%s \n",
      ftt_ascii_error[ftt_status],estring);
}
value = ftt_extract_stats(stbuff,FTT_MOTION_HOURS);
ftt_free_stat(stbuff);
```

8.5 void **ftt_sub_stats (**ftt_stat_buf **b1**, ftt_stat_buf **b2**, ftt_stat_buf **res)**

subtracts numeric values in the stat buffer *b1* from the values in *b2* and the result is put in *res*, to keep running totals, and/or compute deltas. String values are taken from *b1*, unless the entry in *b1* is null and the one in *b2* is not. Flag values are taken from *b1*.

Note that media type and compression rates, while technically numeric, are not treated as such by these routines, since subtracting them is meaningless.

ftt_sub_stats performs no tape operations.

Example:

```
#include <stdio.h>
#include "ftt.h"
int status, ftt_status;       /* status of ftt calls */
char *estring;                /* error string */
ftt_stat_buf b1,b2,res;       /* statistics buffers */

b1 = ftt_alloc_stat();              /* get buffers */
b2 = ftt_alloc_stat();
res = ftt_alloc_stat();
status = ftt_get_stats(d,b1);       /* get tape statistics */
if (status == -1)
   {
   estring = ftt_get_error(&ftt_status);
   fprintf (stderr, "Error:%s Error String:%s \n",
      ftt_ascii_error[ftt_status],estring);
}

/* do some tape i/o here */

status = ftt_get_stats(d,b2);       /* get tape statistics again
*/
if (status == -1)
   {
   estring = ftt_get_error(&ftt_status);
   fprintf (stderr, "Error:%s Error String:%s \n",
      ftt_ascii_error[ftt_status],estring);
}
ftt_sub_stats(b1, b2, res);         /* get deltas and store in res
*/
```

```
ftt_free_stat(b1);                  /* free buffers */
ftt_free_stat(b2);
ftt_free_stat (res);
```

8.6 void **ftt_add_stats** (ftt_stat_buf **b1**, ftt_stat_buf **b2**, ftt_stat_buf **res**)

adds numeric values in the stat buffer *b1* to the values in *b2* and the result is put in *res*, to keep running totals, and/or compute deltas. String values are taken from *b1*, unless the entry in *b1* is null and the one in *b2* is not. Flag values are taken from *b1*.

Note that media type and compression rates, while technically numeric, are not treated as such by these routines, since adding them is meaningless.

ftt_add_stats performs no tape operations.

Example: see ftt_sub_stats example.

8.7 ftt_stat_buf ***ftt_init_stats** (ftt_descriptor **d**)

returns pair of statistics buffers. The first is initialized with zeros and the second with current statistics for the drive.

This routine performs a **ftt_get_stats** which in turn calls **ftt_close_dev** which will write file marks if the preceding operation was a write or write file mark.

Example:

```
#include "ftt.h"
ftt_stat_buf *bp;
ftt_descriptor d;
bp = ftt_init_stats(d);
```

8.8 int **ftt_update_stats** (ftt_descriptor **d,** ftt_stat_buf ***bp**)

updates the status pair *bp***,** keeping running totals in *bp[0]* and latest statistics in *bp[1]*.This routine performs a **ftt_get_stats** which in turn calls **ftt_close_dev** which will write file marks if the preceding operation was a write or write file mark.

Example:

```
#include "ftt.h"
ftt_stat_buf *bp;              /* status buffer pair */
ftt_descriptor d;              /* ftt file descriptor */
int status, ftt_status;        /* status of ftt calls */
char *estring;                 /* error string */

bp = ftt_init_stats(d);
/* do some tape i/o here */
status = ftt_update_stats(d,bp);    /* update statistics again */
if (status == -1)
   {
   estring = ftt_get_error(&ftt_status);
```

```
   fprintf (stderr, "Error:%s Error String:%s \n",
      ftt_ascii_error[ftt_status],estring);
}
```

8.9 int **ftt_dump_stats (**ftt_stat_buf **b,** FILE ***pf)**

prints important statistics from the statistics buffer *b* on file descriptor *pf*. **ftt_dump_stats** performs no tape I/O. It returns the number of bytes written on the file.

Example:

```
#include <stdio.h>
#include "ftt.h"
int status;
ftt_descriptor d;
ftt_stat_buf stbuff;
stbuff = ftt_alloc_stat();
status = ftt_get_stats(d,stbuff);
status = ftt_dump_stats (stbuff, stderr);
ftt_free_stat(stbuff);
```

9 Status Routines

9.1 int **ftt_status**(ftt_descriptor **d,** int **timeout**)

returns a flags word with the following bits set:

TABLE 4. ftt_status flag bits

Bit setting	**Description**
FTT_ABOT	At beginning of tape
FTT_AEOT	At Physical end of tape
FTT_AEW	At Early Warning mark near EOT
FTT_PROT	Write Protected tape
FTT_ONLINE	A tape is loaded and online
FTT_BUSY	The tape is rewinding, unloading, etc. and not responding.

indicating the current tape position. This routine shall block up to *timeout* seconds if there is a tape in the drive, waiting for the tape to come online, or to become not busy. If we cannot determine the presence of a tape, we will simply block until the timeout expires.

An external setuid executable will be run to obtain this data if necessary (e.g. AIX), which means that file marks will be written if the last operation was a **ftt_write** or **ftt_writefm,** (see **ftt_close_dev**()). Therefore, calling **ftt_status** after every write should be avoided.

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
status = ftt_status (d, 0);
```

9.2 int **ftt_get_position**(ftt_descriptor **d,** int ***file,** int ***block**)

returns the current *file* and *block* position on the tape. It ails if no tape is loaded, or the position is unknown (i.e. the tape was just opened, has never been rewound and is not at BOT.) Note that both *file* and *block* are zero at BOT, the count is the number of file marks we are past BOT and the number of blocks past the last filemark, where BOT is treated as the 0-th filemark.

ftt_get_position performs no tape operations, it merely reports bookkeeping information.

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
int fileno, blockno
status = ftt_get_position (d, &fileno, &blockno);
```

9.3 char *ftt_get_error (int *pn)

returns a pointer to a string describing the result of the last call. If *pn* is not null, it is treated as an integer pointer and the current error number is stored there.

ftt_get_error performs no tape operations. Any mode sense data reported was obtained by the routine reporting the error.

Example:

```
#include <stdio.h>
#include "ftt.h"
int error;
char *estring;                  /* error string */

estring = ftt_get_error(&ftt_status);
fprintf (stderr, "Error:%s Error String:%s \n",
   ftt_ascii_error[error],estring);
```

10 Volume verification

10.1 int **ftt_write_vol_label (**ftt_descriptor **d,** int **type,** char * **vollabel)**

rewinds the tape, formats a label specified in *vollabel* into a buffer, and writes the label on the tape, then writes two filemarks and backs over one by doing an **ftt_close_dev** and forward skipping one filemark.*type* can be any of the types specified in **ftt_guess_label** (except FTT_DONTCHECK)**.**

This routine performs numerous tape operations.

```
#include "ftt.h"
int status;
ftt_descriptor d;
status = ftt_write_vol_label(d, FTT_ANSI_HEADER,"mydog");
```

10.2 int **ftt_verify_vol_label (**ftt_descriptor **d,** int **type,** char ***vollabel,** int **timeout,** int **rdonly)**

rewinds the tape, uses **ftt_status** to check if the tape is write protected, reads the first block, uses **ftt_guess_label** to check that the label is the indicated type and volume label specified in *vollabel*.*timeout* is the timeout for the status check for write protect status, in seconds. *type* can be any of the types specified in **ftt_guess_label**. *rdonly* is a boolean flag to indicate if the device should be opened readonly.

The tape is left positioned after the volume label.

Example:

```
#include "ftt.h"
int status;
ftt_descriptor d;
status = ftt_verify_vol_label(d, FTT_ANSI_HEADER,"mydog",0,TRUE);
```

10.3 int **ftt_guess_label (**char ***buf,** int **length,** char ****vol,** int ****vlen)**

examines a buffer in *buf* (ostensibly returned by ftt_read) with a length of *length*, and attempts to guess what sort of tape format this is, assuming this is the first block of the tape. It returns a value of the following type:

TABLE 5. ftt_guess_label label types

Value	Description
FTT_ANSI_HEADER	An ANSI X3.27 standard tape
FTT_FMB_HEADER	An FMB tape
FTT_TAR_HEADER	A tar tape (POSIX nnn)
FTT_CPIO_HEADER	A Cpio -c format tape
FTT_UNKNOWN_HEADER	No known label type

TABLE 5. ftt_guess_label label types

Value	Description
FTT_BLANK_HEADER	Verifies length is -1 and last error was FTT_BLANK
FTT_DONTCHECK_HEADER	Only used by ftt_verify_vol_label()

The *vol* and *vlen* arguments are taken to be the address of a character pointer and of an integer, and will be filled in with the address and length of the volume name in the tape header, if any.

ftt_guess_label performs no tape operations.

In most cases, the user could use **ftt_verify_vol_label** directly, which internally makes a **ftt_guess_label call**.

Example:

```
#include "ftt.h"
int type;
char *databuf;
int buflen;
char *vollabel;
int vollen;
type = ftt_guess_label(databuf, buflen, vollabel, vollen);
```

10.4 int **ftt_format_label (**char ***buf,** int **length,** char ***vol,** int **vlen,** int **type)**

This function will format a label pointed to by string *vol* and of given length (*vlen*) into a buffer (*buf*) of given length (*length*) which will return the same volume label and type in later calls to **ftt_guess_label**. Users can then write this label at the beginning of tape with **ftt_write**. In most cases, the user would call **ftt_write_label** directly. *type* can be any of the types specified in **ftt_guess_label** (except FTT_DONTCHECK_HEADER). It returns the length of the filled in volume header block.

ftt_format_label performs no tape operations.

Example:

```
#include "ftt.h"
int length;
char *databuf;
int buflen;
length = ftt_format_label(databuf, buflen, "mydog",
            strlen ("mydog"), FTT_ANSI_HEADER);
```

11 Issuing Commands Asynchronously

By default, all **ftt** commands are issued synchronously. There are several commands, however, that a user may wish to issue asynchronously, e.g. f**tt_rewind**(). With the exception of **ftt_open()** and **ftt_close()** themselves, all **ftt** commands can optionally be asynchronous.

When a user decides to issue a command asynchronously, he should issue an **ftt_fork()** command. The child then does the desired commands and returns the status vi **ftt_report**(). The parent process can continue doing whatever it wants and then rendezvous with the child via **ftt_wait()**. It can also periodically check if the child process has completed via **ftt_check**();

It is important to note that only one asynchronous operation at a time may be active on a given file descriptor.

11.1 int **ftt_fork** (ftt_descriptor **d**)

Creates an asynchronous process for performing ftt calls, which should report status via **ftt_report**() which will in turn be collected via **ftt_wait**(). Returns a process ID in the parent process, a 0 in the child process, and -1 in the case of failure. The call is generally used as follows:

Example:

```
#include "ftt.h"
char *switch (ftt_fork(d)){
case 0: /* child */
      ftt_skip_fm(d,2);
      ftt_rewind(d);
      ftt_report(d);
default: /* parent */
      do_stuff_during_rewind();
      ftt_wait(d);
      break
case -1: /* error */
      pc = ftt_get_error(&err);
      fprintf(stderr,pc);
      exit(0);
}
```

11.2 void **ftt_report** (ftt_descriptor **d**)

Collects up the status from the last **ftt** call, reports it to the parent process, so that it will be collected by **ftt_wait**(), and exits.

11.3 int **ftt_wait** (ftt_descriptor **d**)

Waits for a pending asynchronous operation to report its status with **ftt_report()**, and sets the error string and error code for ftt_get_error to the result of the completed operation.

ftt_wait performs no tape operations of its own.

11.4 int **ftt_check** (ftt_descriptor **d)**

The ftt_check routine succeeds if the asynchronous operation occurring on the device has completed, otherwise it returns a busy device error.

ftt_check performs no tape operations.

12 Expert Commands

As much as **ftt** would like hide protect the user from implementation specifics, there are times that the expert user would REALLY like to something detailed. Most of the routines were born when trying to recover data from tapes that are overwritten (see the Appendix at the end of this user's guide). As a result, **ftt** provides a few routines to facilitate some of these functions for the knowledgeable user. It is not recommended that the novice user issue these commands.

12.1 int **ftt_clear_unrecovered (**ftt_descriptor **d)**

clear the **ftt** internal unrecovered flag. If certain operations fail (e.g. ftt_skip) the tape is left in an unknown position. To protect the user from writing reading/writing in this area, subsequent i/o calls require the user to first reposition to a known location, i.e., the beginning of tape. ftt_clear_unrecoverd clears this internal **ftt** flag and will therefore allow additional i/o calls without rewinding. This is useful in conjunction with ftt_locate.

Example:

```
#include "ftt.h"
#include "ftt.h"
int status;
ftt_descriptor d;
ftt_clear_unrecovered (d);
```

12.2 int **ftt_all_scsi** (ftt_descriptor **d)**

all **ftt** commands will bypass the standard vendor device driver and will go through the raw scsi device driver. Normally, commands like read and write are not issued in raw scsi mode. This is useful when trying to access all of the request sense information for read and write commands (by setting the ftt_debug level so this information will be displayed). It is also useful if the vendor supplied device driver is buggy.

Example:

```
#include "ftt.h"
#include "ftt.h"
int status;
ftt_descriptor d;
ftt_all_scsi(d);
```

12.3 int **ftt_locate (**ftt_descriptor **d**, int **blockno)**

issues the SCSI locate command to position the tape to the specified logical block.

Example:

```
#include "ftt.h"
int status;
int blockno;
ftt_descriptor d;
status = ftt_locate(d,blockno);
```

13 Linking an Executable

Linking your code with the **ftt** library should be accomplished by adding the following link options to your `Makefile` or other build script:

```
-L$(FTT_DIR)/lib -lftt 'cat $(FTT_DIR)/lib/libs'
```

where the last portion in backquotes places the contents of the file `libs` in the **ftt** library directory on the link command line. This will include any platform-specific libraries needed on the current platform in the link without you having to know what they are, or tailor your build scripts to different platforms. For example, a complete makefile for a program using **ftt** would look like:

```
INC = -I$(FTT_DIR)/include
LIB = -L$(FTT_DIR)/lib -lftt `cat $(FTT_DIR)/lib/libs`
CFLAGS = $(INC)
LDFLAGS = $(LIB)

EXAMPLES = tapecopy

all: $(EXAMPLES)

tapecopy: tapecopy.c
        $(CC) $(CFLAGS) -o tapecopy tapecopy.c $(LIB)
```

14 ftt test

An binary is provided with ftt to verify that the test routines themselves work as intended as well as that an individual drive is behaving normally. ftt_test is a command line parser. It is very unsophisticated in that it doesn't support any flow control whatsoever: it simply executes one line at time. It can be run interactively, through a shell script, or by redirecting stdin.

ftt_test can only have one drive open at a time. The drive name is specified in one of the following fashions, in descending order of priority:

- on the ftt_open call itself (e.g., ftt_open /dev/rmt/tps2d4)
- on the invocation line (e.g., ftt_test -f /dev/rmt/tps2d4)
- from the environment variable FTT_TAPE (e.g. setenv FTT_TAPE /dev/rmt/tps2d4).

14.1 ftt_test Commands

For testing purposes, it is desirable to issue commands that will generate a certain expected error. To avoid a lot of unnecessary clutter, this expected error messages to stderr can be suppressed. Each command can take a "-status [FTT_ERROR]" option which indicates the expected error for that command. An error message will be written to stderr only if this error code is not returned. The list of valid values for FTT_ERROR are all of the error returns listed earlier in this document (e.g., FTT_EIO, FTT_ENOENT, FTT_ENOTAPE).

All commands optionally take a "-usage" switch as well. This is to aid the interactive user in determining what are the valid options and parameters for a given command. If the "-usage" switch is used, the actual command will NOT be executed, but a usage message will simply be printed to stderr.

14.1.1 Open/close Commands

ftt_open [basename] [-readonly] [-status <error>] [-usage]

calls the corresponding ftt_open library routine using *basename* as the input filename. *-readonly* will open the file for read only access. Without this flag, the device is opened for read/write access. The ftt file descriptor is stored in a global variable for all other accesses on this device by the test routines.

ftt_open_logical [basename] [-flavor <flavor>] <-driveid <driveid>> [-readonly] [-status <error>] [-usage]

calls the corresponding ftt_open library routine using *basename* as the input filename. *flavor* indicates the operating system flavor to use. If not supplied, the test routine will use the current operating system flavor. *driveid* specifies the drive type. *-readonly* will open the file for read only access. Without this flag, the device is opened for read/write access. The ftt file descriptor is stored in a global variable for all other accesses on this device by the test routines.

ftt_open_dev [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_close [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_close_dev [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

14.1.2 Read/write Commands

ftt_write_tblock [-nblocks <n>] [-bsize <nbytes>] [-delta <ndbytes>] [-alignmask <nalign>] [-delay <nsec>] [-status <error>] [-usage]

write *n* (default = 1) test blocks of size *nbytes* (default = 32768). If *ndbytes* is specified, the writes will be of random size between *nbytes* and *nbytes* + *ndbytes*. If *alignment* is specified, random size records must will be adjusted to this byte alignment. Valid values for *nalign* are 0 (allows odd byte writes), 1 (allows odd word writes), and 3 (allows only longword multiples). If *delay* is specified, a random delay of 0 to *nsec* will occur between writes. The first longword in the test block is the number of bytes in the block, the second is the block number, the third is the filename. All remaining bytes are filled with ascending bytes starting with (file number + block number)% 256.

ftt_verify_tblock [-nblocks <n>] [-delay <nsec>] [-oddbyte] [-filemark] [-status <error>] [-maxbytes <readsize>] [-usage]

read and verify *n* (default = 1) test blocks If delay is specified, a random delay of 0 to *nsec* will occur between reads. If the *-oddbyte* switch is used, an odd number of bytes will be used for the input buffer. *-filemark* indicates that the record should be a filemark. *readsize* will indicate the maximum number of bytes to read. This value cannot be greater than 64kbytes.

ftt_dump [-nblocks n] [-filename <filename>] [-status <error>] [-usage]

read *n* blocks (defaults until end of tape) and then write them to stdout or f*ilename* if specified.

ftt_undump [-filename <filename>] [-status <error>] [-usage]

write until end of data from stdin or filename if specified.

ftt_writefm [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_write2fm [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_verify_vol_label [-timeout <nsecs>] [-type <type>] [-label <labelname>] [-readonly] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. Information is written to stderr. This command will wait *nsecs* for the drive to come online. The default value is 0. Valid *types* are:

- FTT_ANSI_HEADER

- FTT_FMB_HEADER
- FTT_TAR_HEADER
- FTT_CPIO_HEADER
- FTT_UNKNOWN_HEADER
- FTT_BLANK_HEADER
- FTT_DONTCHECK_HEADER

If not specified, a type of FTT_ANSI_HEADER is assumed.

ftt_write_vol_label [-type <type>] [-label <labelname>] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. The valid values for *type* are the same as in ftt_verify_vol_label.

If not specified, FTT_ANSI_HEADER is assumed.

14.1.3 Tape Positioning Commands

ftt_status [-timeout <nsecs>] [-status <error>]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor and prints the current drive status. The user can specify a timeout of *nsecs*. If no timeout is specified, the default value is 0.

ftt_test_status [-<flag1>... [-<flag>]] [-timeout <nsecs>] [-status <error>]

calls ftt_status using the global variable for the ftt file descriptor. Each flag is tested to see if the condition is true. Valid values for *flag* are: FTT_ABOT, FTT_AEOT, FTT_AEW, FTT_PROT, FTT_ONLINE, and FTT_BUSY. The user can specify a timeout of *nsecs*. If no timeout is specified, the default value is 0.

ftt_skip_to_double_fm [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously.

ftt_skip_fm <nfilemarks> [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously.

ftt_skip_rec <nrecords> [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously.

ftt_rewind [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously.

ftt_retension [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously

ftt_erase [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously.

ftt_unload [-async] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor where *async* will issue the command asynchronously.

ftt_get_position [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. The current file and block number is written to stderr.

ftt_verify_position <fileno> <blockno>

verifies that the position specified by fileno and blockno match those returned by ftt_get_position.

14.1.4 Asynchronous Support Commands

ftt_wait [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_check [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

14.1.5 Get/set Mode Commands and Other Commands That Care About Filenames

ftt_list_all [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. The list of device node pathnames is written to stderr.

ftt_chall <uid> <gid> <mode> [-status error] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. The user name is specified in *uid*, the group name in *gid*, and the mode in *mode*.

ftt_verify_exist

verifies that all of the filenames listed by ftt_list_all really do exist.

ftt_avail_mode <density> <mode> [-blocksize <b>] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. The density and mode must be specified in *density* and *mode* respectively. If the blocksize is not specified, variable block will be assumed. The device node is printed to stderr.

ftt_get_mode [-status <error>] [-usage]

calls the ftt_get_mode library routine using the global variable for the ftt file descriptor.

ftt_set_mode <density> <mode> [-blocksize <b>] [-status <error>] [-usage]

calls the ftt_set_mode library routine using the global variable for the ftt file descriptor. If no blocksize is specified, variable block mode is assumed.

ftt_get_mode_dev [device_name] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. Node information is written to stderr.

ftt_set_mode_dev <devname> [-blocksize <b>] [-force] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. If no blocksize is specified, variable block mode is assumed. *-force* will turn the force flag on. It is off by default.

ftt_verify_modes

verifies all modes of the device. This routines will loop through all modes of the device. It will set the mode, write, verify the mode is what we think it should be, rewind, and then verify the data.

ftt_get_basename [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_describe_dev [device_name] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

14.1.6 Statistics Commands

ftt_get_stats [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. Information is written to stderr.

ftt_extract_stats <statistic> [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. Information is written to stderr. Valid values for *statistic* are from the list described earlier in this document (e.g., FTT_PRODUCT_ID, FTT_VENDOR_ID, FTT_DENSITY, etc.).

ftt_init_stats [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_update_stats [-errors][-part_display] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. The accumulated totals are written to stderr. *-part_display* will only display the statistics that are counters, while *-errors* will display the read/write counts, errors, and percent error rates.

ftt_dump_stats [-filename <filename>] [-status <error>] [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. It will dump the stats to *filename* if specified, otherwise stderr.

ftt_list_supported [-filename <filename>] [-usage]

calls the corresponding ftt library routine. It will dump the stats to *filename* if specified, otherwise stderr.

14.1.7 Error Commands

ftt_eprintf <string> [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_get_error [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. Information is written to stderr.

ftt_max_error [<maxerror>]

Set/show the current maximum number of errors before ftt_test will abort.

14.1.8 Expert Commands

These commands are not meant for the novice user, but we have found them invaluable in salvaging overwritten tapes. Refer to the appendix on how to do this recovery.

ftt_clear_unrecovered [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_all_scsi [-usage]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor.

ftt_locate <blockno> [-usage] [-status <error>]

calls the corresponding ftt library routine using the global variable for the ftt file descriptor. It will locate the tape to the logical block number specified by *blockno*.

14.1.9 Miscellaneous Commands

ftt_date [-usage]

output current date to stderr.

ftt_echo [string] [-usage]

print *string* to stderr.

ftt_debug [level] [-test] [-usage]

Set/show the current debug level for the ftt library as well as the test routines. The debug level of the library will be set to *level*. If *-test* is specified, it will set the debug level for just the test routines. If level is not specified, the current debug level for both the library and the test routines are displayed.

Debug levels for the ftt library are:

- 0 = no debug prints
- 1 = print “Entering routine...” in each routine
- 2 = miscellaneous arguments “got this far” prints
- 3 = debug prints on each pass through loops, etc. More debug prints than you can shake a stick at.

Debug levels for the test routines are:

- 1 = display commands entered by user.

14.2 ftt_test Examples

14.2.1 Interactive Example:

This example will rewind a tape specified on the open command itself:

```
$FTT_DIR/bin/ftt_test
ftt_test> ftt_open /dev/rmt/tps2d4
ftt_test> ftt_rewind
ftt_test> ftt_close
```

14.2.2 Shell Script Example:

This is an example ksh script that will rewind a tape. The device name can be specified as the first argument to the script, or from the environment variable FTT_TAPE.

```
#!/bin/ksh
#=========================
# This will rewind the tape
# =========================
# Get the tape device by looking in $1.
# If that’s not set, try FTT_TAPE.
# If that’s not set either, then exit.
```

```
export FTT_TAPE
FTT_TAPE=${1:-${FTT_TAPE:-""}}
FTT_TAPE=${FTT_TAPE:?"No device specified"}
#==========================================
$FTT_DIR/bin/ftt_test << EOD
ftt_open
ftt_rewind
ftt_close
EOD
```

14.2.3 Redirection Example:

This example will rewind a tape drive based on an ftt_test script. This device is specified as a switch to the ftt_test command itself.

```
cat my.ftt
      ftt_open
      ftt_rewind
      ftt_close
$FTT_DIR/bin/ftt_test -f /dev/rmt/tps2d4 <my.ftt
```

14.3 ftt_test scripts

A series of test scripts are also provided to verify the ftt library as well as verify the behavior of a particular drive. Since the ftt_test program is very simple in nature (e.g., no flow control support, no variable support), the test scripts themselves will be ksh scripts. There are both small building block tests that will verify a particular aspect of the library as well as all inclusive tests that verify the entire package.

All tests can either rely on the device name being specified in the environment variable FTT_TAPE or passed in as argument 0 on the command line.

14.3.1 Building block scripts

Following scripts are set of the **ftt_test** commands that can be used as a redirect input.

Name and location: [1]

`$FTT_DIR/ftt_test/scripts/<script_name>.dat` [2]

Usage:

```
ftt_test -f <drive_name> < $FTT_DIR/ftt_test/script/<script_name>.dat
```

or:

```
ftt_run_test <script_name> <drive_name>
```

1. The name and usage was changed for version v2_0 to have the same behavior on UNIX and Windows NT

2. Scripts rate and describe are not ported to WIndows NT and they should executed directly or using ftt_run_test shell script

2fm

This test will test to make sure that two and only filemarks are written at the end of tape with the ftt_write2fm call. If two individual calls to ftt_writefm are called, four filemarks are actually written.

align

This test will write a bunch of records, rewind, and then verify. It will do this for block sizes that are of longword multiples, word multiples, and byte multiples.

async

This script will verify the asynchronous capability in ftt. It assumes that a test tape is inserted that has been written with the “write” test. It will then skip to the double file mark and then display a time stamp around an asynchronously issued rewind.

This output will need to be also checked manually to verify that the change in time is small. It will then check ftt_wait. An asynchronous command will also be issued that is expected to fail in order to verify that ftt_wait will return a failure.

close

This test will verify that a close command will leave the tape in a known position so that when it’s opened again, the user knows where he is.

delay

This script will test write a bunch of blocks and reading a bunch of blocks with a random delay between i/o operations. This tends to uncover strange behavior with devices and device drivers. It is a lengthy test and may take a few hours to run.

describe[1]

Prints all device files corresponding to the one tape drive.

erase

This script will write a volume label, erase the tape, and then verify that the volume label is blank.

full

This script will write to end of tape to verify appropriate errors reporting. It will take FOREVER to run.

label

This script will write all of the various label types and verify them.

1. This shell script is not ported to Windows NT since the is only one tape drive device

mode

This script will test the various ftt_mode routines. It will be a nested loop which for all densities, all compressions, and all block sizes (variable and two different fixed block sizes) will check if the mode is available. If so, it will rewind the device, set the mode, write some data, verify the mode, rewind, and finally verify the data.

notape

This script will unload the tape and then verify that the correct error is returned when trying to access the drive.

position

This script will verify all the positioning commands using the a tape written by write.ftt. It will verify spacing both records and filemarks both forwards and backwards. It will verify that the correct error code is returned if the operation can't be performed. It will check that ftt_status returns the correct positioning information and that ftt_skip_to_double_fm works.

rate [-n nblocks] [-s blocksize] [device][1]

This script will rewind the device, and then write the specified number of blocks with the specified block size. It will time the time for the data transfer and write this information to stdout. The default block size is 32768 and the default number of blocks to write is 1000.

read_only

This test verifies that all ftt_write functions will return an error if the drive is opened as read only. It should also verify that ftt_status determines that it's a read only drive.

root

This script needs to run as root to work successfully. It does all of the operations that require root privileges. Currently this is on ftt_chall.

stats

This script will verify the statistics information. The output of the script will need to be checked manually. It will do a few writes and display the delta stats and then do a few reads and display the delta stats.

verify

This script verifies a tape written with write.ftt. It verifies that the volume label, the number of blocks, the contents of the blocks themselves, the number of file marks, and that the tape is blank at the end. It also verifies that FTT_ERANGE is returned if the number of bytes to read is less than the number of bytes in the record.

1. This script wasn't ported to Windows NT

twrite

This script writes a test tape beginning with an ANSI label. It will write a few files with a few thousand blocks. The tape written by the script can be used by other scripts for additional testing. This test should leave the tape with only two filemarks at the end.

random

This is a special script that is actually generated by a c program. The c program will generate this ftt script to write a random number of files of random block size. It then randomly picks a position, skips to it, and verifies a random number of blocks.

14.3.2 All inclusive tests

A few test scripts are provided as a convenience that call the fundamental building scripts mentioned in the previous section. There are

ftt_fast [-a] [-u] [device]

This script tests all of the reasonable quick tests to verify that the ftt library is behaving the way that it should. It may take on the order of an hour or two to complete. If -a is specified, the alignment tests are performed as well. Note that this may cause SCSI resets. If the -u switch is specified, the unload tests are performed at the end. Note that the tape will have to be reloaded before any other tests can be run.

ftt_slow [-a] [-u] [device]

This tests is all inclusive test suite for the ftt routines. It will call ftt_fast to perform all of the reasonable quick tests. In addition, it will test of the slow routines (e.g., erase, write to end of tape). It may take several hours to run.

ftt_exercise [device]

This test is provided as a convenience for the user who wants to exercise the device to make sure that it is ok. For example, if a drive is replaced, the user would want to verify it with this script.

ftt_ansi_prelabel [device] [label]

This script writes ansi label and rewinds tape.

ftt_dump [drive]

This will dump a tape - used in recovering overwritten data tapes (see See “ftt_dump/undump” on page 48.).

Usage:

```
ftt_dump <input drive name> | ftt_test -f <output drive name>
```

ftt_run_test [script] [drive]

This script is used to execute **ftt_test** with data file located in `$FTT_DIR/ftt_test/scripts` directory.

Usage:

```
ftt_run_test [script] [drive]
```

is evaluated to:

```
ftt_test -f [drive] < $FTT_DIR/ftt_test/scripts/[script].dat[1]
```

ftt_multitest [-s -p] [script] [drive_1] [drive_2]....

Runs the requested script to run sequentially (**-s**) or parallel (**-p**) on listed drives.

1. with the exemption of **describe** and **rate** scripts

15 ftt_suid

15.1 Command invocation

The **ftt_suid** program which comes with the **ftt** distribution is a program which runs to allow various functions which must be run as root on various platforms to occur. It us generally invoked by the library, but can be invoked from the command line with the following options:

- **ftt_suid** *opts* **-c** *basename* -- clear error statistics on the drive
- **ftt_suid** *opts* **-s** *basename* --print statistics for the drive
- **ftt_suid** *opts* **-b** *n basename* -- set blocksize to **n** for the drive
- **ftt_suid** *opts* **-C** *n basename* -- set compression on (**n** == 1) or off (**n** == 0)
- **ftt_suid** *opts* **-d** *n basename* -- set density to **n**.

Only one of the above forms may be used per command invocation.

In each of the above *opts* may be **-w** to open the drive for writing, (which really only makes a difference for getting statistics on 8200's) and/or **-x** which turns on **ftt** debugging. The command performs the minimum number of **ftt** calls to perform the task, as well as **ftt_open**, and **ftt_close**. These are all operations which require superuser privilege on one or more systems. Currently the only platform where **ftt_suid** does not need to be setuid root at all is on IRIX systems from Silicon Graphics.

In all cases, after performing the requested operation, the values returned by **ftt_get_error()** are printed to file descriptor 0.

Note that for the **ftt** suite to work properly on those systems which require **ftt_suid** to be setuid to root, the **ftt_suid** executable must be in your command search path when the library is run, and installed with the correct permissions.

15.2 Example of use

To get the current statistics from drive **/dev/rmt0** the command:

```
ftt_suid -s /dev/rmt0
```

would yield the output

```
FTT_VENDOR_ID is EXABYTE
FTT_PRODUCT_ID is EXB-8200
FTT_FIRMWARE is 2600
FTT_CLEANING_BIT is 0
...
FTT_SENSE_KEY is 0
FTT_TRANS_SENSE_KEY is NO_SENSE
- is -
0
Ok
```

showing the statistics, then the error code and error string. In this case, the error counters on the Exabyte 8200 would be interpreted as read errors, since **-w** was not specified.

Appendix A: System Configuration Issues

ftt makes very few assumptions about system configuration, however they are listed below, by platform. ftt attempts to continue and operate as best it can even when these options are not met, however it may be unable to determine tape drive type, etc. if these conditions are not met.

A.1 AIX issues

The major assumption for **AIX** systems is that the **ftt_suid** executable is installed setuid root. If this is not done, we will be unable to set density and/or compression, or get statistics.

A.2 IRIX issues

The major assumption under **IRIX** is that the devices under **/dev/scsi** exist, those entries under **/dev/scsi** which correspond to tape drives are writable by tape drive users, and that the *ds* module is built into the kernel to allow SCSI pass-through.

On **IRIX** we cannot currently distinguish between Exabyte 8500's and 8505's, or between DLT4000's and DLT2000's, since the **hinv** command lists them identically.

For **IRIX** we also recommend the following system patches be installed:

- 528 - scsi roll up patch for IRIX 5.3 It fixes no direct problems, but we think it's safer. Challenge and Onyx only.
- 736 - AN ABOSOLUTE MUST. fixes erroneous request sense data. for wd9 3 machines only. It was formerly patch 308.
- 742 - Added support for 8505XL on Challenge and Onyx machines
- 763 - Patch to work around Exabyte bugs. This helps an odd case with 8200s that are starting to go belly up. If it failed during the "close" command when writing a filemark, the driver still tried to reposition the drive. There was goofy 8200 firmware that would get lost at this point, and in an attempt to recover, the driver would reset the bus.
- 1283 - This fixes the ghost filemarks that we saw on dlt devices.

A.3 OSF1 / DEC UNIX issues

The major assumption for **DEC UNIX (OSF1)** port is that the **ftt_suid** executable is installed setuid root, because the **scu** command is used to determine what tape drives are on the system. Even though it would work to tell **ftt** that it did *not* need to be setuid root to operate on **/dev/cam**, and to make **/dev/cam** world writable, this would raise nasty security issues, since this would also allow all users direct pass-through access to *all* SCSI devices on the system.

Note that the software will work without the setuid binary as far as doing tape I/O, but the software will be unable to identify the drive type or do SCSI passthrough, so information about tape density names will be generic, and most statistics will be unavailable.

A.4 Solaris Issues

There are a couple of assumptions hidden in the **Solaris** port, underlying the fact that we use **dmesg** to discover what tape drives are on the system.[1]

- The **st** driver is configured to force-load, so that tape drives are listed at boot. This consists of adding a line to the **/etc/system** file like:

  ```
  forceload: /drv/st
  ```

- Tape devices **/dev/rmt*n*** in order correspond to the sequence tape devices are listed by **dmesg.**

1. It is hoped that in the future a better solution to this problem will be found

A.5 Windows NT

This system is very different then any UNIX system with the way the tape device are handled.

- there is only one tape device like \\.\tape***n***
- the type of the tape is unknown
- there is no way to set density
- only a few drive statistics are provided

A.6 General Security issues

On some platforms, it may not be possible to get the **ftt_suid** executable installed as with sufficient permissions. In those situations, you can:

```
mv ftt_suid ftt_suid.real
cp /bin/true ftt_suid
```

in the **bin** directory of the ftt product, and you will not get errors about the setuid binary -- of course functionality will be missing, as described in the preceding sections.

Appendix B: Recovering Data from Overwritten 8mm Tapes

B.1 Disclaimer

This appendix is to document personal experience in recovering data from tapes that have been overwritten. It is not meant to guarantee that this capability will always be available both in software and hardware.

B.2 850x Recovery

The actual problem was that a few valid data tapes were overwritten with an ANSI volume label at the beginning of the tape. The tapes in question were 8mm tapes written on an 850x drive in non-compressed, variable block mode.

When data is written to an 850x drive, a special end of data mark (EOD) is written after the last data record. When the tapes were overwritten, they contained two EOD marks: one at the end of the original data, and one at the end of the volume label. We wanted to skip past this first EOD and into the original data. Exabyte drives do NOT allow you to skip past an EOD mark UNLESS the drive is configured to support a directory structure. This requires special firmware from Exabyte. Note that the 850x documentation implies that with directory structure support, you can only skip past one EOD mark. We never tried more than one.

We loaded the special firmware into an 8505 drive and then loaded one of the overwritten tapes in this drive. We needed a few specialized commands in the ftt toolkit, so we added ftt_all_scsi[1], ftt_clear_unrecovered[2], and ftt_locate.

Using the ftt_test program, we were able to use ftt_locate to position the tape to a specific logical block in the original data set: For example:

```
ftt_test
ftt_test> ftt_open -readonly /dev/rmt/tps0d4
ftt_test> ftt_rewind
ftt_test> ftt_locate 1000
```

1. Some device drivers (e.g., IRIX) won't even issue commands on the SCSI bus if it thinks they will fail. For example, if you read until end of tape, the driver remembers this and won't even issue future reads to the drive. We originally thought that we might need to read even though the device driver thought we shouldn't, so we added ftt_all_scsi. This will force all drive operations to use the raw SCSI device driver, guaranteeing that the command would actually be sent to the drive. As it turned out, we didn't need this feature.

2. If ftt is ever confused about tape position (e.g., a skip filemark command fails), then it sets an internal flag that prevents any future tape movement. We believed that while we were looking for good data in the recovery process, we might need this feature. It's not clear that it was really needed.

```
command failed with FTT_EBLANK, expected FTT_SUCCESS
ftt_test> ftt_locate 1200
command failed with FTT_EIO, expected FTT_SUCCESS
ftt_test> ftt_locate 1500
ftt_test> ftt_locate 1300
ftt_test> ftt_rewind
ftt_test> ftt_locate 1300
ftt_test. ftt_close
```

Special Observations:

- As a precautionary measure, be sure to write protect the tape!
- The actual logical block number is dependent on both the original block size and the amount of data that was overwritten. This is something that you find by trial and error.
- The logical block number is relative to the beginning of tape, ftt_locate 1200 followed by ftt_locate 1500 will position the tape to the 1500th block from the beginning of tape, not the 2700th.
- ftt_locate will fail with a mix of FTT_EIO and FTT_EBLANK errors until it can sync with the original data.
- Sometimes an ftt_locate command would appear to have worked, but the drive reset itself and rewound. This was particularly true when backspacing records. Be sure to keep an eye on the drive to see what's happening. You can always rewind and start again.
- As another precautionary measure, once you believe you have found the start of good data, rewind the tape and use a single ftt_locate to move there. We found that the drive would sometimes let you backspace with ftt_locate a little further than forward spacing.
- Sometimes, the ftt_locate command would report an FTT_EBUSY error. I never quite understood this, but if you wait a little while and try again, it worked. It's as if the raw scsi device driver was already in use.

Once the input tape has been position into the start of good data, the tape can be copied via your favorite copy process. We tried two ways, both of which worked: ftt_dump/undump and a simple c program.

B.2.1 ftt_dump/undump

A ftt test program was written to dump the tape and this output was piped into the output tape:

ftt_dump /dev/rmt/tps0d4 | ftt_test /dev/rmt/tps0d5

B.2.2 simple copy program

ftt provides a VERY simple c program to do the copy. Note that it does nothing to select the mode of the output tape. It simply uses the default device, but it does rewind it first. The -i and -o switches define the input and output devices respectively.

ftt_copy -i /dev/rmt/tps0d4 -o /dev/rmt/tps0d5

B.3 8200 Recovery

Unlike newer Exabyte models, Exabyte 8200s do not write this EOD mark. We have recovered data from 8200s, but that was several years ago (1991?). One could skip to the first filemark in the original data set and copy from that point on. However, it was significantly more difficult to recover data from the first file.

Jim Meadows (meadows@fnal.gov) and Margaret Votava (votava@fnal.gov) had a test stand setup for such recoveries. It consisted of an mvme133 in a VME crate running pSOS and a Ciprico Rimfire board as the VME->SCSI adaptor. A PC was connected to the terminal port of the drive with the overwritten data. Jim would be able to overwrite some memory location in the drive to convince the heads to spin forward to some arbitrary point. We then attempted to read. This was procedure was repeated until good data was found. The Fermi written rimfire routines were used to control the drives.

B.4 Other Exabyte Tapes

According to Exabyte sales representatives, the techniques used to recover 850x tapes will be supported in newer models, e.g. the mammoth drives. This has not been tested at Fermi.

B.5 DLT 2000/4000

At the time of this writing (March 1997), Quantum representatives claim that there is currently no way to recover overwritten data tapes. They are working on special firmware to support this, but it is not anticipated that this firmware will be available to the general public.