FTT Design

Chuck Debaun, Stephan Lammel, Marc Mengel, Gene Oleynik, Marilyn Schweitzer, Margaret Votava

Fermilab Computing Division

ABSTRACT

FTT is the underlying tape I/O package that all other higher level packages at Fermilab use. It hides explicit knowledge about tape related operating system dependencies as well as specific drive type pecularities from the higher level package.

Contents

1 Goals

As required in the *FTT Requirements* this design attempts to resolve the various OS and tape drive dependencies found in various UNIX flavors and elsewhere in dealing with tape devices, including:

- Differing constant names -- We address this by providing ftt specific constants for various activities that are consistent on all platforms
- Differing errno values -- We address this by translating errno into ftt_errno where necessary to provide a consistent error number for the same error on different platforms
- Differing close behavior -- We address this by performing ensuring a correct number of filemarks, etc. are written to the tape, and making sure that we prepare for whatever system specific actions will happen at device close.
- Special actions to cross file marks -- We will hide from the user any open/close sequences, etc. required to get past filemarks, and ensure consistent tape position of being just past the filemark when an EOF is returned.
- Differing mechanisms to sense tape status -- we provide a consistent interface on all systems, which allows for “I don’t know” responses. This allows configurations with rich available status information to be taken advantage of, while allowing for configurations where little or no status information is available.
- System dependencies should be determined from a table or tables of possibilities and checked at runtime, rather than being #ifdef-ed in the code.

2 Rationale

Use of the **ftt** library revolves around the concept of a **descriptor**, which is a data structure which describes a given tape drive. The descriptor contains information about what operating system devices are associated with the drive, which of those devices are associated with what modes of operation, and what system specific variations are required to perform which operations. The descriptor can be used to obtain information about the drive without performing any I/O to the tape drive, or can be used to request that I/O be performed at various density and compression modes. The library will open and close the appropriate operating system devices to perform the requested I/O as needed.

2.1 Creating descriptors

There are two ways to create an **ftt** descriptor. The usual method is to call **ftt_open** which takes as an argument a **basename**, which is a string describing the tape drive wanted, (usually the common prefix between the devices associated with a given tape drive). The **ftt_open** call verifies that the indicated device exists on the system, determines what sort of device it is, and creates a descriptor. The descriptor is actually created by a routine called **ftt_open_logical**, which manufactures the appropriate descriptor for a device of a given type on a given operating system.

For convenience, an actual device pathname can be used for a basename, and will be trimmed to the basename internally.

2.2 Performing tape motion, input, and output

Once a descriptor has been obtained, motion, input, and output can be performed with straightforward calls. All of the basics of input, output, forward and backwards skipping of records, filemarks, etc. as well as rewind, retension, erase tape, etc. are provided with simple **ftt** calls.

All system specific operations like opening the operating system device, close/reopen on read to write transition, forward skipping over filemarks when an EOF is read, etc. are performed implicitly as needed by the **ftt** library.

2.3 Changing Modes

Tape drives provide a variety of modes, usually some combination of density (e.g. bits per inch written to tape), and compression. **ftt** calls provide several ways to select what mode(s) should be used in performing subsequent operations, and to determine what modes are currently selected. For example **ftt_get_mode** will tell you about the currently selected mode, while **ftt_set_mode** will select a specific density and compression pair. Calls **ftt_set_mode_dev** and **ftt_get_mode_dev** let you select or examine modes associated with a particular device name. Also there are calls like **ftt_avail_mode** to let you check if a given mode could be set on a given drive. Note that the **ftt** library does not *reset* modes to their previous values on close, so on systems like AIX, this will leave the device in whatever mode you last had set when **ftt_open_dev** was called.

Operating systems also provide several fictitious (that is not provided by the drive) modes, like rewind-on-close, retension-on-open, swap-bytes, etc. The **ftt** library will tell you about those modes if you ask it about particular devices, but only **ftt_set_mode_dev** with a force bit set will set them as the mode to be used to do I/O.

2.4 Statistics

Various device statistics are extractable from most SCSI tape drives, which the standard operating system drivers do not provide to the user. **ftt** includes code to send and receive the appropriate SCSI command frames to the drive to obtain this information and report it to the user. The call **ftt_get_stats** obtains a status block from the drive, and **ftt_extract_stats** is used to pull individual statistics from the block. In this manner, the library provides whichever statistics are available in a given situation in a consistent, device independent fashion. **ftt_extract_stats** is allowed to return a null string to indicate "I don't know" as a response to any given request.

2.5 Error handling

Errors are handled in a consistent fashion throughout **ftt**, where an error number and error string can be obtained if any of the library calls return an unsuccessful response. The error strings have specific information about the specific error which just occurred, not just a stock expansion of the error number.

2.6 Informational calls

The **ftt** library provides various informational calls to find out what devices are associated with a given drive, whether particular modes are available on a given drive, what error occurred in the preceding call, what sort of tape label a block represents.

2.7 Higher level calls

The **ftt** library also provides some higher level calls which perform multiple low level calls; calls to verify the appropriate tape is mounted with the right permissions, calls to maintain tape usage and error statistics across a series of operations, and routines to print various sorts of information to files.

3 Error Returns

All of the ftt integer functions return -1, set an internal error number, and call ftt_eprintf (which prints into a static buffer) with details if an error occurs.

zero is failure for functions returning a character pointer, ftt_descriptor, or ftt_stat_buf. These also set an error number and call ftt_eprintf (which prints into a static buffer) with details if an error occurs.

The ftt_eprintf() result string and numeric error code can be obtained by calling ftt_get_error().

ftt_get_error() does not affect the error string

The error string will include mode sense data if it is available and appropriate for the error. The error string will include the device name involved in the error, if any.

The ftt library will translate errno according to this table before returning from an ftt call, and place the result where ftt_get_error() can find it.

Successful returns also eprintf() a message, usually “Ok”.

TABLE 1. Error numbers returned in ftt_errno by ftt

ftt_errno	Description
FTT_SUCCESS	no error has occurred
FTT_EPERM	you do not have permission to perform the specified operation on the device.
FTT_ENOTAPE	there is no tape in the drive
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)
FTT_EBUSY	Asynchronous I/Ois going on this device, or another process has it open
FTT_ENODEV	System does not have drive configured
FTT_EINVAL	invalid compression flag value, or bad statbuf pointer, etc.
FTT_ENXIO	Attempt to read blank tape
FTT_ENFILE	No system file descriptors available
FTT_ENOTTAPE	What we thought was a tape device was not.
FTT_EROFS	Tape is write protected
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 skippin g records
FTT_ERWFS	Tape is not write protected (and should be)
FTT_EPIPE	Setuid bypass died
FTT_ENOMEM	Unable to allocate memory required to perform call
FTT_ERANGE	Record too large for buffer
FTT_ENOTSUPPORTED	Unsupported device
FTT_ENOTAPE	No tape in tapedrive
FTT_EUNRECOVERED	previous error was nrecoverable, tape can only be unloaded/rewound (to protect data)
FTT_EPARTIALSTAT	unable to obtain some or all of the information requested

.

4 Low Level Entry Point Overview

The FTT product will need several entry points to provide the required services. This section of the document describes the entry points and arguments. Many of the requested functions are provided as a (short) sequence of ftt calls, short wrappers functions can be provided if needed; for example:

```
ftt_descriptor d;
d = ftt_open("/dev/rmt/tps0d3",0);
ftt_rewind(d);
ftt_unload(d)
ftt_close(d);
```

would rewind and unload a tape drive by basename.

The preceding section specifies error returns.

This section specifies internal routines in the library, many of which are not called by users (e.g. ftt_eprintf), whereas the next section specifies higher level routines

4.1 Header file ftt.h

The ftt.h header file will include definitions of preprocessor symbols for all ftt_status flag bits, label type numbers, error codes, and density definitions. Internal data structures will not be visible in ftt.h, and will instead be hidden in private header files in the product sources.

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

The **ftt_open** entry takes 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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.

4.7 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.

ftt_read will call **ftt_open_dev** (which sets modes and does a device open) if needed; also it performs the obvious tape motion.

4.8 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.

ftt_write will call **ftt_open_dev** (which sets modes and does a device open) if needed; also it performs the obvious tape motion.

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.

4.9 int **ftt_writefm**(ftt_descriptor **d)**

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

Appropriate system dependent kludges for file offsets are performed.

The indicated tape motion is performed, and **ftt_open_dev** will be called if needed.

4.10 int **ftt_status**(ftt_descriptor **d,** int **time-out)**

The ftt_status entry returns a flags word with the following bits set:

TABLE 3. ftt_status flag bits

Bit setting	**Description**
FTT_ABOT	At beginning of tape
FTT_AFM	Just After a FileMark
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 time-out 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 is to be avoided.

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

The ftt_guess_label examines a buffer (ostensibly returned by ftt_read), 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 4. 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
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.

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

This function will format a label into a buffer, 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**.

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.

4.13 int **ftt_skip_to_double_fm**(ftt_descriptor **d)**

The tape is skipped forward to the next double filemark. At return from the call the tape is positioned between the two filemarks.

This is intended to be used for appending a new file to a tape using double-file-mark termination.

In the case of an error (i.e. there are not two filemarks at the end of the tape), an error number of FTT_UNRECOVERED will be returned for this call, and all subsequent calls until one of **ftt_rewind,f ftt_unload, ftt_close_dev,** or **ftt_close** is called.

4.14 int **ftt_skip_fm**(ftt_descriptor **d,** int **n**)

If **n** is positive, the tape is moved forward to just after the nth next filemark.

If **n** is negative, the tape is moved backward to just before the nth previous filemark.

If **n** is zero, nothing is done.

If the requested operation would involve going past BOT or EOT, we return FTT_ENXIO.

When skipping backwards, this routine performs the necessary operations to ensure that the tape has two end-of-file marks at the end of recorded data, (see the discussion in **ftt_close**).

In the case of an 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.

4.15 int **ftt_skip_rec**(ftt_descriptor **d,** int **n)**

If **n** is positive, the tape is moved forward to just after the nth next filemark

If **n** is negative, the tape is moved backward to just before the nth previous filemark.

If **n** is zero, nothing is done.

If the request would involve going past a filemark , BOT, or EOT, we return FTT_ENXIO.

When skipping backwards, this routine performs the necessary operations to ensure that the tape has two end-of-file marks at the end of recorded data, assuming this condition held when ftt_open was called. (see the discussion in **ftt_close.)**

In the case of an 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

4.16 int **ftt_rewind**(ftt_descriptor **d)**

The tape is rewound

This routine performs the necessary operations to ensure that the tape has two end-of-file marks at the end of recorded data, assuming this condition held when ftt_open was called. (see the discussion in **ftt_close.)**

4.17 int **ftt_erase(ftt_descriptor d)**

The tape is erased and rewound

4.18 int **ftt_unload**(ftt_descriptor **d)**

The tape is rewound and unloaded from the drive.

This routine performs the necessary operations to ensure that the tape has two end-of-file marks at the end of recorded data, assuming this condition held when ftt_open was called. (see the discussion in **ftt_close.**)

4.19 int **ftt_retension**(ftt_descriptor **d)**

The tape is retensioned and rewound.

This routine performs the necessary operations to ensure that the tape has two end-of-file marks at the end of recorded data, assuming this condition held when ftt_open was called. (see the discussion in **ftt_close.)**.

4.20 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:

```
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);
}
```

4.21 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.

4.22 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.

4.23 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.

4.24 ftt_stat_buf **ftt_alloc_stat();** int **ftt_free_stat**(ftt_stat_buf **b)**

These routines allocate and free statistics buffers for use with the following statistics calls, so that the type and size of a statistics buffer is opaque to the user.

ftt_alloc_stat, and **ftt_free_stat** perform no tape operations.

4.25 int **ftt_get_stats(ftt_descriptor d,ftt_stat_buf)**

This routine 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. If necessary, an external, setuid program will be executed in a subshell to obtain the statistics for the drive.

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**) .

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

This routine extracts a string describing a named piece of tape data, including:

TABLE 5.

stat	**Description**	**Notes**
FTT_VENDOR_ID	SCSI Inquiry vendor id data	string
FTT_PRODUCT_ID	SCSI Inquiry product id	string
FTT_FIRMWARE	SCSI Inquiry firmware rev level	string
FTT_SERIAL_NUM	Device serial number	string
FTT_HOURS_ON	Number of hours device has been on	counter
FTT_CLEANING_BIT	Device thinks it needs cleaning	flag
FTT_READ_COUNT	Decimal number of blocks read since rewind/ reset	counter
FTT_WRITE_COUNT	Decimal number of blocks written since rewind/reset	counter
FTT_READ_ERRORS	Read errors	counter
FTT_WRITE_ERRORS	Write errors	counter
FTT_DENSITY	Mode Sense Tape density	numeric
FTT_READ_COMP	Compression factor for reads	numeric
FTT_WRITE_COMP	Compression factor for writes	numeric
FTT_FILE_NUMBER	File number from BOT	numeric
FTT_BLOCK_NUMBER	Block number in current file	numeric
FTT_BOT	at beginning of tape	flag
FTT_READY	Device ready status	flag
FTT_WRITE_PROT	Device write protect status	flag

TABLE 5.

stat	Description	Notes
FTT_FMK	At filemark	flag
FTT_EOM	At end of media (early warning)	flag
FTT_PEOT	At physical end of tape (really at end)	flag
FTT_MEDIA_TYPE	SCSI Media type	numeric
FTT_BLOCK_SIZE	Tape block size	numeric
FTT_BLOCK_TOTAL	Number of blocks on tape	numeric
FTT_TRANS_DENSITY	Translated density ala ftt_get_mode	numeric
FTT_TRANS_COMPRESS	Translated compression ala ftt_get_mode	numeric
FTT_REMAIN_TAPE	Remaining KB of tape	counter
FTT_USER_READ	KB read since open/unload	counter
FTT_USER_WRITE	KB written since open/unload	counter
FTT_CONTROLLER	Controller type	string
FTT_ILI	SCSI ILI bit from Req Sense	flag
FTT_SCSI_ASC	SCSI Additional Sense Code	numeric
FTT_SCSI_ASCQ	SCSI Additional Sense Code Qualifier	numeric
FTT_PF	SCSI PF flaog	flag
FTT_CLEANDED_BIT	SCSI cleaned bit	flag
FTT_TRACK_RETRY	Track retry errors	numeric
FTT_UNDERRUN	stop/start counter from EXB 8505s	numeric
FTT_MOTION_HOURS	hours tape drive is in motion	counter
FTT_TUR_STATUS	SCSI Test Unit Ready response	numeric
FTT_COUNT_ORIGIN	Origin of read/write kb counters	string
FTT_N_READS	Number of read calls	counter
FTT_N_WRITES	Number of write and writefm calls	counter
FTT_TNP	Exabyte Tape Not Present bit	flag
FTT_SENSE_KEY	SCSI Request Sense Sense Key	numeric
FTT_TRANS_SENSE_KEY	Above translated to strings	string

ftt_extract_stats performs no tape operations.

The routine will return a null pointer (0) if a given statistic is not available in the status block.

4.27 void **ftt_add_stats**(ftt_stat_buf **b1**, ftt_stat_buf **b2**, ftt_stat_buf **res**), **ftt_sub_stats**(ftt_stat_buf **b1**, ftt_stat_buf **b2**, ftt_stat_buf **res**)

Numeric values in the stat buffer **b1** are added to/subtracted 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 adding them or subtracting them is meaningless.

ftt_add_stats and **ftt_sub_stats** perform no tape operations.

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

Returns the current file and block position on the tape.

Fails 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.

4.29 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.

4.30 char ***ftt_get_basename(**ftt_descriptor **d)**

This routine 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.

4.31 char ****ftt_list_all(**ftt_descriptor **d)**

A pointer to a static list of device pathnames related to this drive is returned.

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.

4.32 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**.

4.33 char *ftt_avail_mode(ftt_descriptor **d,** int **density,** int **mode,** int **blocksize)**

Returns a character pointer to a device name if the density, mode (compression, etc.), and blocksize combination requested is available on this tape drive, where density is an integer, 0 for lowest density for this media type, 1 for the next highest, etc. and compression is 0 or 1 for no compression or compression, respectively.

Returns 0 if there is no device matching the request.

The value FTT_MAX_DENSITY will give the highest density available on that drive.

#define values for EXB_2G, EXB_5G, DLT10G, etc. will be provided, to allow the user to request specific tape densities.

A blocksize of 0 indicates variable blocking.

ftt_avail_mode performs no tape operations.

4.34 char *ftt_set_mode(ftt_descriptor **d,** int **density,** int **mode,** int **blocksize)**

Sets the default device for this tape to be the given density number, sets modes(compression, etc.). on or off, fixed block size (0 for variable block), and rewind on close mode on or off, where mode, fixed, and rewind flags are set to 1 to turn them on, and returns the name of the device.

Setting the density, etc. can only be done reliably at BOT.

A blocksize of 0 indicates variable blocking

ftt_set_mode performs no tape operations, it merely sets flags, etc. for **ftt_open_dev.**

4.35 char *ftt_get_mode(ftt_descriptor **d,** int ***density,** int ***mode,** int ***blocksize)**

Returns the name of the current default device for this tape drive, and puts the density, mode (compression, etc.), etc. values for the device into the integers whose addresses are passed into the routine.

A blocksize of 0 indicates variable blocking, a non-zero blocksize indicates the current fixed block size.

ftt_get_mode performs no tape operations.

4.36 int ftt_get_mode_dev(ftt_descriptor **d,** char ***devname,** int ***density,** int ***mode,** int ***blocksize,** int ***rewind)**

Returns the density, mode, , etc. values for the device **devname** into the integers whose addresses are passed into the routine.

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

ftt_get_mode_dev performs no tape operations.

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

Sets the mode, 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.**

4.38 int **ftt_eprintf**(char ***format, ...)**

Prints the format and arguments on the error block for ftt. making it the result of the next ftt_get_error call.

Behaves basically like sprintf(), except the buffer length is checked for overflow, etc.

ftt_eprintf performs no tape operations.

5 High Level Calls

The medium level calls provide combinations of the low level calls to perform various more complicated tasks. They are also used to provide higher level calls, and for the **ftt_test** program.

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

This 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.

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.**

The tape is left positioned after the volume label.

This routine performs numerous tape operations.

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

This rewinds the tape, formats a label 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.

5.3 int **ftt_describe_dev**(ftt_descriptor **d,** char ***dev,** FILE *pf)

This routine prints a description of the given device, 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.

5.4 ftt_stat_buf ***ftt_init_stats**(ftt_descriptor **d)**

This routine a pair of statisics buffers, and returns a pointer to them, and initializes the first 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.

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

This routine 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

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

This routine 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.

6 SCSI Pass Through calls

These calls are used internally by some upper level ftt calls, and have a separate open/close interface. They are also used by the juke product, and others which need a direct SCSI CDB send/receive interface.

Note that a scsi_handle is distinct from a regular ftt handle.

The ftt_scsi routines are callable without the use of the rest of ftt, for use by the juke product, etc.

6.1 char ***ftt_get_scsi_devname(handle)**

This routine returns the SCSI pass-thru device name of a given tape basename, suitable for calling ftt_scsi_open(). It also ensures any other device routes to the device are closed if necessary.

6.2 scsi_handle **ftt_scsi_open(devname)**

Returns a scsi_handle for other ftt_scsi calls. Opens the device with whatever special flags are needed for this platform.

6.3 int **ftt_scsi_close**(scsi_handle **s)**

Closes the file handle, no further calls will be made. The user should be sure to close the scsi_handle before other calls using the regular ftt handle are used.

6.4 int **ftt_scsi_command**(scsi_handle **s,** char ***pcOp,** char ***pcCmd,** int **nCmd,** char ***pcRdWr,** int **nRdWr,** int **timeout,** int **writeflag)**

Performs a SCSI command, whose CDB is pointed to by pcCmd, and is of length nCmd, and uses a data buffer pointed to by pcRdWr, of length nRdWr.

It returns the SCSI sense key status word of 0 for success, other values indicating various sorts of failure messages as defined by the SCSI II spec.

A timeout equal to **timeout** will be set (in seconds) for the command to return.

Writeflag indicates wether the pcRdWr buffer is for sending or receiving.

If request sense information is desired, an extra call to ftt_scsi_command with a request-sense CDB must be made; if the local system auto-senses, the last auto-sense data will be saved and returned if a request sense call is made immediately following its attainment.

7 ftt descriptor data structure

The ftt descriptor data structure is opaque to the user, but internally needs information to operate the device, maintain status of tape position, etc.

7.1 dev_info[] table

The dev_info table will be an array of structures, with each element describing a device associated with this tape device:

TABLE 6.

Field	Description
char *device_name	Pathname to open to access device
char density	Density code, 0 is lowest density...
char mode	Flag to indicate if compression is used
char rewind	flag to indicate device is rewind on close
char passthru	flag to indicate the device can be used for SCSI pass-thru

The descriptor will have room for up to 32 such entries.

7.2 which_is_open flag

This will contain the subscript of the dev_info table for the device that is currently open, if any. If no device is currently open, it will be set to -1.

7.3 file_descriptor

If the which_is_open flag is non-negative, this contains the file descriptor for the open device.

7.4 which_is_default

This will contain the subscript of the dev_info table which is the default device to use if the user does not call ftt_set_mode().

7.5 open_read_only

This stores the readonly argument to ftt_open(), indicating that device opens for this device should be made read_only, and that ftt_write() and ftt_writefm() calls should fail.

7.6 current_block, current_file, current_valid

The current_block and current_file fields state where we believe we are on the tape based on calls to ftt_write, ftt_read, ftt_writefm, ftt_rewind.

current_block must be a 64 bit integer value

The current_valid flag will be set if we have confidence on the estimates of position due to having performed an ftt_rewind, or an ftt_status which said we were at BOT.

7.7 async_pf

The **FILE *** for the pipe to read the result string, etc. from an asynchronous operation.

7.8 last_operation

This flag says whether we are reading, or writing, or neither one. This lets us do appropriate things on systems that need special stuff on a read-to-write transition.

7.9 unrecovered_error

This flag will indicate that FTT_EUNRECOVERED has been returned, indicating that successive operations other than rewind will fail.

8 Test Design

8.1 Overview

The test software for ftt must be able to test each command or a series of commands in a user defined order. This user defined order can either be in a script file or typed in interactively at the command line. Like ftt itself, this command interpreter needs to be lightweight and easy to port; therefore, it cannot be very sophisticated. Errors will simply be reported to stderr. Only ftt commands themselves will be supported: there will be no flow control (if-then-else, for, e.t.c.) constructs.

We can then use this interpreter as the fundamental building block for a series of scripts to verify both an ftt port to a particular platform and drive type and to test that a particular drive is behaving as expected. These series of scripts will be provided as part of the testing portion of ftt.

8.2 ftt interpreter (ftt_test)

In the ftt interpreter, ftt_test, only a single device can be open at any given time. This greatly simplies the user interface. ftt_test will support the following commands either entered via a script or the command line. If no corresponding library routine exists, the routine is described here.

All commands will write any errors to stderr. If a particular routine returns any data as well (e.g.,get_status) that information is also written to stderr unless otherwise indicated. If a command invokes a corresponding ftt library routine, refer to the library section for a detailed description of what the specific command does.

For commands that operate on a base filename (e.g., ftt_open), the base name can be supplied in one of three ways in descending order of priority:

1. on the individual command line, e.g., ftt_open /dev/rmt/tps0d3
2. on the command line invoking the test routine itself, e.g., ftt_test -f /dev/rmt/tps0d3
3. by the environment variable FTT_TAPE, e.g., setenv FTT_TAPE /dev/rmt/tps0d3

For testing purposes, it is sometimes 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 commad will NOT be executed, but a usage messasge will simply be printed to stderr.

8.2.1 Open/close Commands

8.2.1.1 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.

8.2.1.2 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.

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

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

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

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

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

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

8.2.2 Read/write Commands

8.2.2.1 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 of *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 londword 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.

8.2.2.2 ftt_verify_tblock [-nblocks <n>] [-delay <nsec>] [-oddbyte] [-filemark] [-status <error>] [-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.

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

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

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

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

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

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

8.2.2.6 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 writted 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.

8.2.2.7 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 value for *type* are the same as in ftt_verify_vol_label.

If not specified, FTT_ANSI_HEADER is assumed.

8.2.3 Tape Positioning Commands

8.2.3.1 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.

8.2.3.2 ftt_test_status [-<flag*1*> ... [-<flag*n*>]] [-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_AFM, 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.

8.2.3.3 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 asyncronously.

8.2.3.4 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 asyncronously.

8.2.3.5 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 asyncronously.

8.2.3.6 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 asyncronously.

8.2.3.7 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 asyncronously

8.2.3.8 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 asyncronously.

8.2.3.9 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 asyncronously.

8.2.3.10 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.

8.2.3.11 ftt_verify_position <fileno> <blockno>

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

8.2.4 Asynchronous Support Commands

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

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

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

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

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

8.2.5.1 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.

8.2.5.2 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*.

8.2.5.3 ftt_verify_exist

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

8.2.5.4 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.

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

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

8.2.5.6 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.

8.2.5.7 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.

8.2.5.8 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.

8.2.5.9 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.

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

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

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

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

8.2.6 Statistic Commands

8.2.6.1 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.

8.2.6.2 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).

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

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

8.2.6.4 ftt_update_stats [-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.

8.2.6.5 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* is specified, otherwise stderr.

8.2.7 Error Commands

8.2.7.1 ftt_eprintf <string> [-usage]

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

8.2.7.2 ftt_get_error [-usage]

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

8.2.7.3 ftt_max_error [<maxerror>]

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

8.2.8 Miscellaneous Commands

8.2.8.1 ftt_date [-usage]

output current date to stderr.

8.2.8.2 ftt_echo [string] [-usage]

print string to stderr.

8.2.8.3 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 test routines are:

- 1 = display commands entered by user.

8.3 Test Scripts

A series of test scripts will be provided to test the functionality of ftt as well as that 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 suite themselves will be shell scripts. These shell scripts will be built upon smaller shell scripts to do specific tasks. All scripts whether they be building blocks or higher level will operate on the device name that is (in descending order of priority):

- passed on the command line
- specified in the environment variable FTT_TEST

8.3.1 ftt building block scripts

8.3.1.1 align [device]

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.

8.3.1.2 asynch [device]

This script will verify the asynchronous routines in ftt. It will first write a bunch of records and then display a time stamp around an asyncronously 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 fai in order to verify that ftt_wait will return a failure.[device]

8.3.1.3 delay [device]

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.

8.3.1.4 erase [device]

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

8.3.1.5 full [device]

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

8.3.1.6 label [device]

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

8.3.1.7 mode [device]

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 if available. If so, it will rewind the device, set the mode, write some data, verify the mode, rewind, and finally verify the data.

8.3.1.8 notape [device]

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

8.3.1.9 position [device]

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.

8.3.1.10 read_only [device]

This test will verify 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.

8.3.1.11 root [device]

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

8.3.1.12 stats [device]

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.

8.3.1.13 verify [device]

This script will verify 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 the the tape is blank at the end.

8.3.1.14 write [device]

This script will write 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.

8.3.1.15 random [device]

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 will then randomly pick a position, skip to it, and verify a random number of blocks.

8.3.2 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

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

This script will test 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 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 specifed, the unload tests are performed at the end. Note that the tape will have to be reloaded before any other tests can be run.

8.3.2.2 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.

8.3.2.3 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.