The Fermi Tape Tools (ftt) Porting Guide

by Marc W. Mengel, Fermilab

.

ISBN

CHAPTER 1

Overview

The Fermi Tape Tools (**ftt**) product is a software interface to tape drives which is independant of drive type and operatng system. It is designed to be portable to numerous platforms and tape drive types, although most of the existing work is for SCSI connected tape drives on UNIX[1] systems. This book is a guide to getting ftt running on new platforms and new drive types. Folks who want to learn more about using ftt should refer to the **Fermi Tape Tools (ftt) Library User's Guide**.

The portability claims of **ftt** are based on the fact that the information regarding system and drive differences currently reside in a very few system specific source files, and a central table whiich allows **ftt** to be aware of things like device naming conventions and supported tape densities for all of the systems it currently supports.

The remainder of the book will concentrate on the platform specific parts of **ftt**, as well as the table structure which describes supported drives and operating modes on different platforms.

1.UNIX is a trademark of Netware Inc.

CHAPTER 2

Platform Specific Sources

The Fermi Tape Tools product has several platform specific source files for each supported platform. They are in the *`$FTT_DIR/ftt_lib/PLATFORM`* directory for platform PLATFORM. Each such directory contains several source files:

- *`ftt_scsi.c`* contains definitions of routines to allow SCSI command frames to be sent to SCSI connected drives. It defines the routines:
    1. `ftt_scsi_open()`
    2. `ftt_scsi_close()`
    3. `ftt_scsi_command()`

- `ftt_machdep.c` contains subroutine definitions for several internal **ftt** calls, including:
    1. `ftt_status()`
    2. `ftt_set_hwdens()`
    3. `ftt_set_compression()`
    4. `ftt_set_blocksize()`
    5. `ftt_get_hwdens()`
- `ftt_mtio.h` contains definitions of preprocessor symbols to allow consistent use of `mtio(7)` style calls to perform rewinds, etc. on the various platforms.

- `libs` contains special compiler flags to allow the code to be linked on this platform.

These 4 files comprise the entirety of the platform specific code in **ftt** for each platform. We will examine each of these files and their corresponding routines in this chapter.

SCSI I/O routines

The various SCSI Input/Output routines are kept in `ftt_scsi.c` in the platform dependant routines directory. The main routine that needs work to define on any given platform is `ftt_scsi_command()` which sends a SCSI command to a target device, and returns any data requested. On some platforms there are clearly documented `ioctl()` calls to normal tape SCSI device nodes to perform this sort of i/o, while on others a separate device node is called for.

The `ftt_scsi_command()` routine take several arguments, describing the data tobe sent, where to put received data, timeouts, etc. They are:

- A file handle for the scsi device in question. This may be a regular UNIX file descriptor, a pointer to a structure, etc. depending what sort of information you need to keep around to make SCSI calls. This will be a value you returned from `ftt_scsi_open()`
- A string that the upper level code passes in that you should print in any debugging messages to indicate what is going on.
- a character pointer and length for the SCSI CDB -- the command descriptor bytes that tell the device what to do.
- a character pointer and length for a read/write buffer to put either data being sent to the device, or data being received.
- a timeout value (in integer seconds) of how long to wait for the command to complete.
- A flag that is 1 if this command has write data in the read/write buffer.

The `ftt_scsi_command()` call should take this information, and make the appropriate call or calls to send the scsi frame to the target device, and receive any data to be returned. It should then perform some error translation, as described in the next paragraph.

After the scsi command has been sent, you should call the routine `ftt_scsi_check()` with the following parameters:

- The SCSI descriptor you were given
- The descriptive string you were given (these are the same first two arguments as were passed in to this routine)
- the SCSI status code (0 == success, 2 == device not ready, etc.)
- The amount of data actually read/written into the read/write buffer, in bytes, or a suitable lie. (Note: if anyone figures out how to actually get this number on Solaris, send me a patch!)

The `ftt_scsi_check()` routine wil return a standard error code for the action which your routine should then return to its caller. The ftt_scsi_check() routine that comes with **ftt** mainly prints a detailed error message with ftt_eprintf() and sets ftt_errno, and returns -1 on errors.

As mentioned earlier, `ftt_scsi_command()` will be called with some sort of descriptor for a SCSI device. As the programmer for this portion of the code, you get to decide what this descriptor should be, (either a long integer value, or a pointer to some data structure type-cast to a long integer). You manufacture the descriptor for a given device in the `ftt_scsi_open()` routine. In the case where you simply open an operating system device, this routine will simply call open() with the right arguments and return a file descriptor. If you need to make a series of calls and fill out a data structure, etc. you should allocate the structure, fill it in, and return a pointer to it. We will discuss how these descriptors get cleaned back up in the next section

After SCSI i/o is completed on a given device, **ftt** will call your routine `ftt_scsi_close()`, whose job is to close any system devices that are opened, free up any allocated memory, etc. **ftt** passes it the argument of the scsi descriptor created by `ftt_scsi_open()`, so that your code can clean back up after it.

Examples of several variations on these routines are in the **ftt** distribution. For example, the AIX version does a (slightly modified) open() call, and uses a regular AIX file descriptor for the SCSI descriptor. The IRIX version uses a vendor provided library which allocates a descriptor with dsopen(), and cleans it back up with dsclose(), and performs operations using that pointer as a descriptor. The OSF1 implmentation shares access to a global /dev/cam device, and keeps a descriptor which identifies SCSI bus number, target id, etc. and a reference count on the shared system device so we know when to close it.

The OSF1 implementation also points out an important tactic for platforms which do not have separate device nodes for different SCSI target id's -- namely making up pseudo-device-names so that you can specify in the string passed into `ftt_scsi_open()` a name for the device, even if there is no such name in the filesystem. This is important for other non-tape packages which use this code, like the **juke** package.

Making mtio(7) style i/o work

Most UNIX style platforms have some variation on the original mtio(7) interface avaliable for tape operations like rewinds, file skips, etc. **ftt** takes advantage of the similarities between these implementations to use common code for these operations by having a set of **ftt**-specific symbols defined, so that the following code fragment should to work rewind the tape drive attached to file descriptor `fd`.

```
struct tapeop buf;
buf.tape_op = FTT_TAPE_REW;
buf.tape_count = 1;
ioctl(fd,FTT_TAPE_OP,&buf);
```

The various operation codes like `FTT_TAPE_REW` are defined as follows:

TABLE 1. ftt_mtio.h tape operation names

Name	Description
FTT_TAPE_NOP	Does nothing
FTT_TAPE_RETEN	Retension tape
FTT_TAPE_ERASE	Erase tape
FTT_TAPE_REW	Rewind tape
FTT_TAPE_RSF	Reverse skip filemark
FTT_TAPE_FSF	Forward skip filemark
FTT_TAPE_RSR	Reverse skip record
FTT_TAPE_FSR	Forward skip record
FTT_TAPE_WEOF	write filemark
FTT_TAPE_UNLOAD	unload tape

There are example ftt_mtio.h files for each of the various platforms. You should define all of these symbols to some integer value, even if you are not using them because you are doing operations via scsi pass-through. Most of the existing implementations define the value as -1 if the option doesn't exist or is not used.

Miscelleneous Platform Dependant Routines

There are a few other routines clumped into `ftt_machdep.c` which are used to perform various operations that **ftt** is called upon to perform, for example:

- ftt_status() returns a byte with several status bits bitwise-or-ed together to say whether the tape drive is in various states. Most implementations of tape drivers provide a status ioctl call which gives this data in some format or other, on other platforms (like AIX) this information can only be obtained via SCSI pass-through. This should set the following bits if appropriate:

TABLE 2. ftt_status return code bits

Name	Description
FTT_BUSY	Tape in use by another process
FTT_ABOT	Tape is at rewound, or at BOT
FTT_AEOT	Tape is at end of tape
FTT_EW	Tape is at ANSI early warning mark
FTT_ONLINE	Drive has a tape loaded
FTT_WRITE_PROT	Tape is write protected

- ftt_set_hwdens() Set the drive density code with a SCSI mode_select or equivalent. This is a separate call because on some platforms you set a property of the device driver, so that future opens will use that density. This routine is a no-op on platforms where the density is selected entirely by choosing a device node.
- ftt_set_compression() Similar to set_hwdens but specifically set compression on or off. This routine is a no-op on platforms where compression is selected entirely by selecting a device node.
- ftt_set_blocksize() Set the blocksize to be used on this device. Setting the blocksize to zero means to do variable-block i/o. This routine is a no-op on platforms where the blocksize is selected entirely by the device node.

- ftt_get_hwdens() Get the hardware density associated with the current default **ftt** device node. On platforms where there is only one density associated with a device node, this routine simply returns the density associated with that device node.

Note that the only platform currently requiring the last 4 calls to do much of anything is AIX, or an implementation that does everything by explicit raw SCSI calls.

CHAPTER 3

Drive Description Tables

Overview

The **ftt** product is largely table driven. While there are some code differences betweeen platforms, as described in the previous chapter, the majority of the code is common, and controlled by tables. The **ftt** tables reside in `ftt_tables.c` and can be described as follows:

- Error translation tables: These tables map an *(operation, errno)* pair to an **ftt** error number, allowing the variances in various platforms error returns to be corrected.
- Device node descriptions: For each (*platform, driveid)* pair, this table gives behaviour flags, a generic basename format, a drive identification command, and a list of device node formats with corresponding behavior information. This table also has various forms of "generic" entries describing how to handle unkown *driveid* and *platform* values.
- Drive statistics tables: For each *driveid* this table gives a description of what statistics commands (usually SCSI Mode Select/Mode Sense commands) are available on the drive.

The remainder of the chapter will describe these tables in detail, and describe where to find the information needed to fill them in.

Error Translation Tables

These tables are the simplest tables to describe, and the hardest ones to fill in properly. As mentioned eariler, they convert *(operation, errno)* pairs to **ftt** error numbers. The format is of arrays to arrays; which is to say for a given platform, there is an array of pointers to arrays of integers indexed by *operation*, where each element of that array points to an array of integers indexed by *errno*, and these integers are the **ftt** error numbers. This arrangement allows the same second level table to be shared by numerous *operations* on a various platforms, and drastically cuts down on the size of the tables.

Initial Error Translation Tables

For a new platform, it is probably simplest to copy the error table of an existing platform as a good first guess, and then correct it as testing and review of the platform's documentation point out any errors in the translation.

Updating Error Tables

The **ftt** test suite tests that various errors are reported correctly, especially things like attempts to skip backwards past BOT, forwards past EOT, or to skip records past a filemark. Some errors are not easily tested in a test suite, so a review of the platform's documentation of error codes returned by tape I/O, opening devices, etc. is also reccomended.

Device Node Descriptions

These tables take up the majority of the `ftt_tables.c` file, and are perhaps the hardest to read. The format of the table is an array of structures:

```
typedef struct {
    char *os;
    char *drivid;
    char *controller;
    long flags;
    long scsi_ops;
    int **errortrans;
    char **densitytrans;
    char *baseconv_in;
    char *baseconv_out;
    int nconv;
    char *drividcmd;
    ftt_devinfo devs[MAXDEVSLOTS];
} ftt_dev_entry;

extern ftt_dev_entry devtable[];
```

Where each such entry in the table describes the appearance of a given type of tape drive (identified the **drivid** string), on a given operating system (identified by the **os** string), attached to a given type of peripheral controller (identified by the **controller** string). We will shortly examine each of these remaining fields in more detail, but first let us discuss the meaning of these strings, and the signifigance of the order of entries in the table.

Operating System and Drive id strings

The **ftt** Users Guide describes that drive id's and operating system names are matched on a prefix basis -- that is if one string is a prefix of the other, they are considered to match. So if you are looking for "OSF1+V3.2D" in the table, and you find "OSF1", that is considered a match. Similarly if you are looking for the drive id string "EXB-8500NANQX" and you find "EXB-85" in the table, that would be considered a match.

The code searches the **devtable** array in order from lowest index to highest, and takes the first match. This means that if you have entries that are prefixes of one another, you **must** put the longer one first. It also means that the generic-system, generic-drive entry, which has empty strings for both drive-id and operating system, must be last in the table, since any entries that came after it would never be found.

The Controller string

The **controller** string is the controller name passed back to the user upon request. It is deduced from operating system, device basename, and drive-id matches.

The Flags word

The **flags** field is a bitmask describing various behavioral differences which can be allowed for in the **ftt** code, which are described below. If behavior changes are required for a new drive type or platform type, they should be encoded as bits into this field, rather than being #ifdef-ed into the code.

TABLE 3. ftt_devinfo flags field values

Name	Description
FTT_FLAG_CHK_BOT_AT_FMK	Check for being at BOT after writing a filemark, indicating unintentional rewinds.
FTT_FLAG_FSF_AT_EOF	A forward file skip is required to get past a filemark when read() returns 0
FTT_FLAGO_REOPEN_AT_EOF	A device open/close is required to get past a filemark when read() returns 0
FTT_FLAG_HOLD_SIGNALS	The sighold() and sigrelse() calls should be used around reads and writes
FTT_FLAG_REOPEN_R_W	The system device should be closed and reopened to switch from reading to writing
FTT_FLAG_SUID_SCSI	You must be root to do scsi pass-through
FTT_FLAG_BSIZE_AFTER	Blocksize must be set after open()ing the device, rather than before.

The scsi_ops Field

The **scsi_ops** field is a bitmask describing which **ftt** operations should be performed by SCSI pass-through, rather than system calls, so turning all the bits on indicates that all operations on this device should be done via pass-through. The operation numbers, and corresponding bitfield values are described below:

TABLE 4. Operation codes in ftt

Name	Description
FTT_OP_READ	A block is read from the tape
FTT_OP_WRITE	A block is written to the tape
FTT_OP_WRITEFM	A filemark is written on the tape
FTT_OP_SKIPREC	Forward movement just past blocks
FTT_OP_SKIPFM	Forward movement just past filemarks
FTT_OP_REWIND	The tape is moved backwards to BOT
FTT_OP_UNLOAD	The tape is ejected from the drive
FTT_OP_RETENSION	The tape is rewound and skipped forward repeatedly
FTT_OP_ERASE	The tape is erased and rewound
FTT_OP_STATUS	Drive status is obtained
FTT_OP_ASYNC	An asynchronous task is created to do I/O on the device
FTT_OP_PASSTHRU	A raw SCSI command is sent to the drive
FTT_OP_CHALL	Permissions on the device nodes are changed
FTT_OP_OPEN	A device node is opened
FTT_OP_RSKIPREC	Backward movement just before blocks
FTT_OP_RSKIPFM	Backward movement just before filemarks

The errortrans field

The **errortrans** field points to an error translation table as described in the previous section.

The densitytrans field

The **densitytrans** field points to an array of strings which are the names for the various drive densities this drive has. For example, the DLT 4000 has 5 different density settings, The table is indexed by the **ftt** density plus one, so that the name for a density value of -1 is specified in the table (usualy "unknown"), followed by a string name for each of the valid densities, and a null pointer which terminates the list.

Drive name conversion strings

baseconv_in and **baseconv_out** are `sscanf()` and `sprintf()` format strings used to recognize and generate **ftt** drive basenames. The **nconv** field describes the expected value sscanf() should return when **baseconv_in** is used on a matching device string.   Thus doing a sprintf(out,**baseconv_in,...**) with the values returned from sscanf(in,**baseconv_in,...**,) strips a device to its basename.

Drive identifcation command

The **drividcmd** specifies the text of a shell script as a sprintf() format string that should be executed to determine the drive-id string, based on the values extracted with scanf(in, **baseconv_in,**...).

Device node descriptions:

ftt_devinfo describes each device node associated with the instance, where each item in the **ftt_devinfo** table has several members.

- **device_name** is a sprintf() format string, which is given the values obtained using sccanf(in, **baseconv_in,...**) and then run with **popen()** to obtain a device-id string, which is read from the command's standard output. This command is run by the **ftt_suid** executable, and therefore should reset the environment varibles **IFS** and **PATH** before doing anything else.
- **density** is an **ftt** density number, where the drive families lowest density is numbered 0, and higher densities are given the next higher integer value.
- **mode** is a compression flag, with 1 indicating compression turned on, and 0 indicating it turned off. At some point, this may allow other values to describe multiple compression types.

- **hwdens** is the density number actually reported by the drive, as by a SCSI Mode Sense command.
- **passthru** is a flag which says that this device can be used for SCSI passthrough calls.
- **fixed** is a flag which says this device node does fixed-block I/O.
- **rewind** is a flag which says that closing this device causes a device rewind.
- **first** is a flag which says that this entry is the first one with this **device_name** string. It is only used in calls like **ftt_list_all**() which deal with lists of device names.
- **max_blocksize** is the size of the largest blocksize supported by this drive on this platform. It should be the minimum of the maximum blocksize supported by the drive, and the maximum blocksize supported by the O.S. Writes with block-sizes larger than this amount should be expected to fail, as should reads larger than this amount.

It should be noted that if a given device node supports both fixed and variable mode operation, two entries are requried here to describe it. Furthermore, if it supports both fixed and variable, and compressed and uncompressed mode, **four** entries will be requrired, etc. In such an instance, only the first such entry should have the **first** flag set, this allows the code to avoid gratuitous string compares.

*** Examples!!! ***

Drive Statistics Tables

The **ftt_stat_op_tab** array has pairs of drive-id strings and flags which indicate SCSI commands which are supported for obtaining statistics.

TABLE 5. Statistics operation table bitfields

Name	Description
FTT_DO_TUR	Perform a Test Unit Ready command
FTT_DO_INQ	Perform an Inquiry command
FTT_DO_MS	Perform a plain Mode Sense command
FTT_DO_RS	Perform a plain Request Sense command
FTT_DO_EXBRS	Look for Exabyte vendor specific statistics
FTT_DO_SN	Perform an Inquiry with the serial number page

TABLE 5. Statistics operation table bitfields

Name	Description
FTT_DO_O5RS	look for Exabyte 8x05 status fields
FTT_DO_LSRW	Perform Log Sense for Read/Write op/error counts
FTT_DO_RP	Perform Read Position
FTT_DO_RP_SOMETIMES	Perform Read Position, but its okay if it fails -- this is for EXB-85xx drives, which support Read Position only in high density mode...
FTT_DO_MS_Px10	Perform a Mode Sense with page code 0x10 for compression information.
FTT_DO_LSC	Perform a Log Sense for Compression information