25. Accessing Files

The Lisp Machine can access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Lisp Machine itself, if the machine has its own file system. You are not allowed to refer to files without first logging in, and you may also need to specify a username and password for the host on which the file is stored; see page 801.

The way to read or write a file's contents is to *open* the file to get an input or output stream, use the standard stream I/O functions or operations described in chapters 22 and 23, and then close the stream. The first section of this chapter tells how to open and close the stream. The rest of the chapter describes things specific to files such as deleting and renaming, finding out the true name of the file that has been opened, and listing a directory.

Files are named with *pathnames*. There is much to know about pathnames aside from accessing files with them; all this is described in the previous chapter.

Many functions in this chapter take an argument called *file* which is intended to specify a file to be operated on. This argument may be given as a pathname (which is defaulted), a namestring (which is parsed into a pathname and then defaulted), or a stream open to a file (the same file is used).

25.1 Opening and Closing File Streams

with-open-file (*stream file options...*) *body...* *Macro*

Evaluates the *body* forms with the variable *stream* bound to a stream that reads or writes the file named by the value of *file*. The *options* forms evaluate to the file-opening options to be used; see page 582.

When control leaves the body, either normally or abnormally (via throw), the file is closed. If a new output file is being written and control leaves abnormally, the file is aborted and it is as if it were never written. Because it always closes the file, even when an error exit is taken, with-open-file is preferred over open. Opening a large number of files and forgetting to close them tends to break some remote file servers, ITS's for example.

If an error occurs in opening the file, the result depends on the values of the *error* option, the *if-exists* option, and the *if-does-not-exist* option. An error may be signaled (and possibly corrected with a new pathname), or *stream* may be bound to a condition object or even nil.

with-open-file-case (*stream file options...*) *clauses...* *Macro*

This opens and closes the file like with-open-file, but what happens afterward is determined by *clauses* that are like the clauses of a condition-case (page 702). Each clause begins with a condition name or a list of condition names and is executed if open signals a condition that possesses any of those names. A clause beginning with the symbol :no-error is executed if the file is opened successfully. This would be where the reading

or writing of the file would be done.
Example:

```
        (with-open-file-case (stream (send generic-pathname
                                           :source-pathname))
          (sys:remote-network-error (format t "~&Host down."))
          (fs:file-not-found (format t "~&(New file)"))
          (:no-error (setq list (read stream))))
```

file-retry-new-pathname (*pathname-var condition-names...*) *body...* *Macro*
file-retry-new-pathname-if *Macro*
cond-form (*pathname-var condition-names...*) *body...*

file-retry-new-pathname executes *body*. If *body* does not signal any of the conditions in *condition-names*, *body*'s values are simply returned. If any of *condition-names* is signaled, file-retry-new-pathname reads a new pathname, setq's *pathname-var* to it, and executes *body* again.

The user can type End instead of a pathname if he wishes to let the condition be handled by the debugger.

file-retry-new-pathname-if is similar, but the conditions are handled only if *cond-form*'s value is non-nil.

For an example, see the example of the following macro.

with-open-file-retry *Macro*
(*stream* (*pathname-var condition-names...*) *options...*) *body...*

Like with-open-file inside of a file-retry-new-pathname. If an error occurs while opening the file and it has one of the specified *condition-names*, a new pathname is read, the variable *pathname-var* is setq'd to it, and another attempt is made to open a file with the newly specified name. Example:

```
        (with-open-file-retry (instream (infile fs:file-not-found))
          ...)
```

infile should be a variable whose value is a pathname or namestring. The example is equivalent to

```
        (file-retry-new-pathname (infile fs:file-not-found)
          (with-open-file (instream infile)
            ...))
```

with-open-file-search *Macro*

Opens a file, trying various pathnames until one of them succeeds. The pathnames tried differ only in their type components. For example, load uses this macro to search for either a compiled file or a source file. The calling sequence looks like

```
        (with-open-file-search
           (streamvar (operation defaults auto-retry)
                      types-and-pathname options...)
          body...)
```

with-open-file-search tries opening various files until one succeeds; then binds *streamvar* to the stream and executes *body*, closing the stream on exit. The values of *body* are returned.

types-and-pathname specifies which files to open. It should be a form which evaluates to two values, the first being a list of types to try and the second being a pathname called the base pathname. Each pathname to try is made by merging the base pathname with the defaults *defaults* and one of the types. The types may be strings or canonical type keywords (see section 24.2.3, page 551).

options are forms whose values should be alternating to keywords and values, which are passed to open each time.

If all the names to be tried fail, a fs:multiple-file-not-found error is signaled. *operation* is provided just so that the :operation operation on the condition object can return it. Usually the value given for *operation* should be the user-level function for which the with-open-file-search is being done.

If *auto-retry* is non-nil, an error causes the user to be prompted for a new base pathname. The entire set of types specified is tried anew with the new pathname.

open *file* &rest *options*
Returns a stream that is connected to the specified file. Unlike Maclisp, the open function creates streams only for *files*; streams of other kinds are created by other functions. The *file* and *options* arguments are the same as in with-open-file; see above.

When the caller is finished with the stream, it should close the file by using the :close operation or the close function. The with-open-file special form does this automatically and so is usually preferred. open should only be used when the control structure of the program necessitates opening and closing of a file in some way more complex than the simple way provided by with-open-file. Any program that uses open should set up unwind-protect handlers (see page 82) to close its files in the event of an abnormal exit.

close *stream* &optional *option*
The close function simply sends the :close message to *stream*. If *option* is :abort for a file output stream, the file is discarded.

cli:close *stream* &key *abort*
The Common Lisp version of close is the same as close except for its calling convention. If *abort* is non-nil for a file output stream, the file is discarded.

fs:close-all-files
Closes all open files. This is useful when a program has run wild opening files and not closing them. It closes all the files in :abort mode (see page 464), which means that files open for output are deleted. Using this function is dangerous, because you may close files out from under various programs like Zmacs and ZMail; only use it if you have to and if you feel that you know what you're doing.

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. In addition, for compatibility with the Maclisp **open** function, if only a single option is specified it is either a keyword or a list of keywords (not alternating with values).

The file-opening options control things like whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, etc.

The following keyword arguments are standardly recognized; additional keywords can be implemented by particular file system hosts.

direction
: Controls which direction of I/O can be done on the resulting stream. The possible values are **:input** (the default), **:output**, **nil**, **:probe**, **:probe-directory** and **:probe-link**. The first two should be self-explanatory. **nil** or **:probe** means that this is a "probe" opening; no data are to be transferred, the file is being opened only to verify its existence or access its properties. The stream created in this case does not permit any I/O. **nil** and **:probe** differ in causing different defaults for the argument *if-does-not-exist*. If that argument is specified explicitly, **nil** and **:probe** are equivalent.

: **:probe-directory** is used to see whether a directory exists. If the directory specified for the file to be opened is found, then the **open** completes (returning a non-I/O stream) as if the specified file existed whether it really exists or not.

: **:probe-link** is used to find out the truename of a link. If the file specified exists as a link, then the **open** completes returning a non-I/O stream which describes the link itself rather than the file linked to. If the file exists and is not a link, the **open** also completes for it as with any probe.

: Common Lisp defines the value **:io** for this argument, requesting a stream that can do input and output, but no file system supported by the Lisp Machine has this capability.

characters
: The possible values are **t** (the default), **nil**, which means that the file is a binary file, and **:default**, which means that the file system should decide whether the file contains characters or binary data and open it in the appropriate mode.

byte-size
: The possible values are **nil** (the default), a number, which is the number of bits per byte, and **:default**, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, **nil** selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, **nil** selects the default byte size of 16 bits.

element-type
: This is the Common Lisp way to specify what kind of objects the stream wants to read or write. This combines the effect of the *characters* and *byte-size* arguments. The value is a type specifier; it must be one of the following:

: **string-char** Read or write characters as usual. The default.

: **character** Read or write characters, dealing with characters that are more than 8 bits. You can succeed in writing out any sequence of character objects and reading it back, but the file does not look anything like a text file.

(unsigned-byte *n*)	Read or write *n*-bit bytes. Like *characters* = nil, *byte-size* = *n*.
unsigned-byte	Similar, but uses the byte size that the file was originally written with. This is the same as *characters* = nil, *byte-size* = :default.
(signed-byte *n*)	Read or write *n*-bit bytes, sign-extending on input. Each byte read from the file is sign-extended so that its most significant bit serves as a sign bit.
signed-byte	Similar, but uses the byte size that the file was originally written with.
(mod *n*)	Like unsigned-byte for a big enough byte size to hold all numbers less than *n*. bit is also accepted, and means (mod 2).
:default	Is allowed, even though it is not a type specifier. It is the same as using :default as the value of *characters*.

if-exists For output opens, *if-exists* specifies what to do if a file with the specified name already exists. There are several values you can use:

:new-version	Create a new version. This makes sense only when the pathname has :newest as its version, and it is the default in that case.
:supersede	Make a new file which, when closed, replaces the old one.
:overwrite	Write over the data of the existing file, starting at the beginning, and set the file's length to the length of the newly written data.
:truncate	Like :overwrite except that it discards the old contents of the file immediately, making it empty except for what is written into it this time.
:append	Add new data onto the existing file at the end.
:rename	Rename the existing file and then create a new one.
:rename-and-delete	Rename the existing file, create a new one, and delete the old file when the new one is closed.
:error	Signal an error (fs:file-already-exists). This is the default when the pathname's version is not :newest. The further handling of the error is controlled by the *error* argument.
nil	Return nil from open in this case. The *error* argument is irrelevant in this case.

if-does-not-exist
Specifies what to do when the file requested does not exist. There are three allowed values:

:create	Create a file. This is the default for output opens, except when *if-exists* is :append, :overwrite or :truncate. This silly exception is part of the Common Lisp specifications.
:error	Signal an error. This is the default for input opens, and also for output opens when *if-exists* is :append, :overwrite or :truncate. The further handling of the error is controlled by the *error* argument.
nil	Return nil from open. This is the default for :probe opens. The *error* argument is irrelevant in this case.

error Specifies what to do if an error is signaled for any reason. (Note that the values of the *if-exists* and *if-does-not-exist* arguments control whether an error is signaled in certain circumstances.) The possible values are t (the default), :reprompt and nil. t means that nothing special is done, so the error invokes the debugger if the caller does not handle it. nil means that the condition object should be returned as the value of open. :reprompt means that a new file name should be read and opened.

Any caller which need not know reliably which file was ultimately opened might as well specify :reprompt for this argument. Callers which need to know if a different file is substituted should never specify :reprompt; they may use with-open-file-retry or file-retry-new-pathname (see page 581) if they wish to permit an alternative file name to be substituted.

:submit If specified as t when opening a file for output, the file is submitted as a batch job if it is closed normally. The default is nil. You must specify :direction :output as well.

deleted The default is nil. If t is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.

temporary If t is specified, the file is marked as temporary, if the file system has that concept. The default is nil.

preserve-dates If t is specified, the file's reference and modification dates are not updated. The default is nil.

flavor This controls the kind of file to be opened. The default is nil, a normal file. Other possible values are :directory and :link. Only certain file systems recognize this keyword.

link-to When creating a file with *flavor* :link, this argument must be specified; its value is a pathname or namestring that becomes the target of the link.

submit The value can be either nil (the default) or t. If the value is t, and the :direction is :output, the resulting file will be submitted as a batch job. Currently, this option is implemented only for Twenex and VMS.

estimated-size The value may be nil (the default), which means there is no estimated size, or a number of bytes. Some file systems use this to optimize disk allocation.

physical-volume	The value may be nil (the default), or a string that is the name of a physical volume on which the file is to be stored. This is not meaningful for all file systems.
logical-volume	The value may be nil (the default), or a string that is the name of a logical volume on which the file is to be stored. This is not meaningful for all file systems.
super-image	The value may be nil (the default), or t, which disables the special treatment of rubout in ASCII files. Normally, rubout is an escape which causes the following character to be interpreted specially, allowing all characters from 0 through 376 (octal) to be stored. This applies to ASCII file servers only.
raw	The value may be nil (the default), or t, which disables all character set translation in ASCII files. This applies to ASCII file servers only.

In the Maclisp compatibility mode, there is only one *option*, and it is either a symbol or a list of symbols. These symbols are recognized no matter what package they are in, since Maclisp does not have packages. The following symbols are recognized:

in, read	Select opening for input (the default).
out, write, print	Select opening for output; a new file is to be created.
binary, fixnum	Select binary mode; otherwise character mode is used. Note that fixnum mode uses 16-bit binary words and is not compatible with Maclisp fixnum mode, which uses 36-bit words. On the PDP-10, fixnum files are stored with two 16-bit words per PDP-10 word, left-justified and in PDP-10 byte order.
character, ascii	The opposite of fixnum. This is the default.
single, block	Ignored for compatibility with the Maclisp open function.
byte-size	Must be followed by a number in the options list, and must be used in combination with fixnum. The number is the number of bits per byte, which can be from 1 to 16. On a PDP-10 file server these bytes will be packed into words in the standard way defined by the ILDB instruction. The :tyi stream operation will (of course) return the bytes one at a time.
probe, error, noerror, raw, super-image, deleted, temporary	These are not available in Maclisp. The corresponding keywords in the normal form of file-opening options are preferred over these.

25.2 File Stream Operations

The following functions and operations may be used on file streams, in addition to the normal I/O operations which work on all streams. Note that several of these operations are useful with file streams that have been closed. Some operations use pathnames; refer to chapter 24, page 545 for an explanation of pathnames.

file-length *file-stream*

Returns the length of the file open on *file-stream*, in terms of the units in which I/O is being done on that stream. (A stream is needed, rather than just a pathname, in order to specify the units.)

file-position *file-stream* &optional *new-position*

With one argument, returns the current position in the file of *file-stream*, using the :read-pointer stream operation. It may return nil meaning that the position cannot be determined. In fact, it always returns nil for a stream open in character mode and not at the beginning of the file.

With two arguments, sets the position using the :set-pointer stream operation, if possible, and returns t if the setting was possible and nil if not. You can specify :start as the *new-position* to position to the beginning of the file, or :end to position to the end.

:pathname *Operation on file streams*

Returns the pathname that was opened to get this stream. This may not be identical to the argument to open, since missing components will have been filled in from defaults. The pathname may have been replaced wholesale if an error occurred in the attempt to open the original pathname.

:truename *Operation on file streams*

Returns the pathname of the file actually open on this stream. This can be different from what :pathname returns because of file links, logical devices, mapping of version :newest to a particular version number, etc. For an output stream the truename is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:generic-pathname *Operation on file streams*

Returns the generic pathname of the pathname that was opened to get this stream. Normally this is the same as the result of sending the :generic-pathname message to the value of the :pathname operation on the stream; however, it does special things when the Lisp system is bootstrapping itself.

:qfaslp *Operation on file streams*

Returns t if the file has a magic flag at the front that says it is a QFASL file, nil if it is an ordinary file.

:length *Operation on file streams*

Returns the length of the file, in bytes or characters. For text files on ASCII file servers, this is the number of ASCII characters, not Lisp Machine characters. The numbers are different because of character-set translation; see section 25.8, page 607 for a full explanation. For an output stream the length is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:creation-date *Operation on file streams*
Returns the creation date of the file, as a number that is a universal time. See the chapter on the time package (chapter 34, page 776).

:info *Operation on file streams*
Returns a cons of the file's truename and its creation date. This can be used to tell if the file has been modified between two open's. For an output stream the information is not guaranteed to be correct until after the stream has been closed.

:properties &optional (*error-p* t) *Operation on file streams*
This returns two values: a property list (like an element of the list returned by fs:directory-list), and a list of the settable properties. See the section on standard file properties (section 25.6, page 598) for a description of the ones that may possible found in the list.

:set-byte-size *new-byte-size* *Operation on file streams*
This is only allowed on binary file streams. The byte size can be changed to any number of bits from 1 to 16.

:delete &optional (*error-p* t) *Operation on file streams*
Deletes the file open on this stream. For the meaning of *error-p*, see the deletef function. The file doesn't really go away until the stream is closed.

:undelete &optional (*error-p* t) *Operation on file streams*
If you have used the :deleted option in open to open a deleted file, this operation undeletes the file.

:rename *new-name* &optional (*error-p* t) *Operation on file streams*
Renames the file open on this stream. For the meaning of *error-p*, see the renamef function.

File output streams implement the :finish and :force-output operations.

25.3 Manipulating Files

This section describes functions for doing things to files aside from reading or writing their contents.

truename *object*
Returns the truename of the file specified somehow by *object*. If *object* is a plausible stream, it is asked for the truename with the :truename operation. Otherwise, *object* is converted to a pathname and that pathname is opened to get its file's truename.

delete-file *file* &key (*error-p* t) *query?*
deletef *file* &optional (*error-p* t) *query?*

Both delete the specified file. The two functions differ in accepting keyword arguments versus positional arguments. *file* may contain wildcard characters, in which case multiple files are deleted.

If *query?* is non-nil, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually deleted.

If *error-p* is t, then if an error occurs it is signaled as a Lisp error. If *error-p* is nil and an error occurs, the error message is returned as a condition object. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the truename of the file, and the cadr is non-nil if the file was actually deleted (it is always t unless querying was done).

undelete-file *file* &key (*error-p* t) *query?*
undeletef *file* &optional (*error-p* t) *query?*

Both undelete the specified file. Wildcards are allowed, just as in deletef. The rest of the calling conventions are the same as well. The two functions differ in taking keyword arguments versus positional arguments.

Not all file systems support undeletion, and if it is not supported on the one you are using, it gets an error or returns a string according to *error-p*. To find out whether a particular file system supports this, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion.

rename-file *file new-name* &key (*error-p* t) *query?*
renamef *file new-name* &optional (*error-p* t) *query?*

Both rename the specified file to *new-name* (a pathname or string). The two functions differ in taking keyword arguments versus positional arguments. *file* may contain wildcards, in which case multiple files are renamed. Each file's new name is produced by passing *new-name* to merge-pathname-defaults with the file's truename as the defaults. Therefore, *new-name* should be a string in this case.

If *query?* is non-nil, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually renamed.

If *error-p* is t, then if an error occurs it is signaled as a Lisp error. If *error-p* is nil and an error occurs, the error message is returned as a condition object. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the original truename of the file, the cadr is the name it was to be renamed to, and the caddr is non-nil if the file was renamed. The caddr is nil if the user was queried and said no.

copy-file *file new-name* &key (*error* t) (*copy-creation-date* t) (*copy-author* t) *report-stream* (*create-directories* :query) (*characters* :default) (*byte-size* :default)

Copies the file specified by *file* to the name *new-name*.

characters and *byte-size* specify what mode of I/O to use to transfer the data. *characters* can be

t	to specify character input and output.
nil	for binary input and output,
:ask	meaning ask the user which one
:maybe-ask	meaning ask if it is not possible to tell with certainty which method is best,
:default	meaning to guess as well as possible automatically.

If binary transfer is done, *byte-size* specifies the byte size to use; :default means to ask the file system for the byte size that the old file is stored in, just as it does in **open**.

copy-author and *copy-creation-date* say whether to set those properties of the new file to be the same as those of the old file. If a property is not copied, it is set to your login name or the current date and time.

report-stream, if non-nil, is a stream on which a message should be printed describing the file copied, where it is copied to, and which mode was used.

create-directories says what to do if the output filename specifies a directory that does not exist. It can be t meaning create the directory, nil meaning treat it as an error, or :query meaning ask the user which one to do. The default is :query.

error, if nil, means that if an error happens then this function should just return an error indication.

If the pathname to copy from contains wildcards, multiple files are copied. The new name for each file is obtained by merging *new-name* (parsed into a pathname) with that file's truename as a default. The mode of copy is determined for each file individually, and each copy is reported on the *report-stream* if there is one. If *error* is nil, an error in copying one file does not prevent the others from being copied.

There are four values. If wildcards were used, each value is a list with one element describing each file that matched; otherwise, each value describes the single file specified (though the value may be a list anyway). The values, for each file, are:

output-file	The defaulted pathname to be opened for output in copying this file.
truename	The truename of the file copied
outcome	The truename of the new file, If the file was successfully copied. A condition object, if there was an error and *error* was nil. nil if the user was asked whether to copy this file and said no.
mode	A Common Lisp type descriptor such as string-char or (unsigned-byte 8) saying how the file was copied.

probe-file *file*
probef *file*

Returns **nil** if there is no file named *file*; otherwise returns a pathname that is the true name of the file, which can be different from *file* because of file links, version numbers, etc. If *file* is a stream, this function cannot return **nil**.

Any problem in opening the file except for **fs:file-not-found** signals an error.

probef is the Maclisp name; **probe-file** is the Common Lisp name.

file-write-date *file*

Returns the creation date/time of *file*, as a universal time.

file-author *file*

Returns the name of the author of *file* (the user who wrote it), as a string.

viewf *file* &optional (*output-stream* ***standard-output***) *leader*

Copies the contents of the specified file, opened in character mode, onto **output-stream**. Normally this has the effect of printing the file on the terminal. *leader* is passed along to **stream-copy-until-eof** (see page 457).

fs:create-link *link-name link-to* &key (*error* **t**)

Creates a link named *link-name* which points to a file named *link-to*. An error happens if the host specified in *link-name* does not support links, or for any of the usual problems that can happen in creating a file.

25.3.1 Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as **defun** and **defvar** which define the functions and variables of the program.

Loading a compiled (or QFASL) file is similar, except that the file does not contain text but rather pre-digested expressions created by the compiler which can be loaded more quickly.

These functions are for loading single files. There is a system for keeping track of programs which consist of more than one file; for further information refer to chapter 28, page 660.

load *file* &key *verbose print* (*if-does-not-exist* **t**) *set-default-pathname package*

Loads the specified file into the Lisp environment. If *file* is a stream, **load** reads from it; otherwise *file* is defaulted from the default pathname defaults and the result specifies a file to be opened. If the file is a QFASL file, **fasload** is used; otherwise **readfile** is used. If *file* specifies a name but no type, **load** looks first for the canonical type **:qfasl** and then for the canonical type **:lisp**.

Normally the file is read into the package specified in its attribute list, but if *package* is supplied then the file is read in that package. If *package* is **nil** and *verbose* is **nil**, **load** prints a message saying what file is being loaded and what package is being used. *verbose* defaults to the value of ***load-verbose***.

If *if-does-not-exist* is nil, load just returns nil if no file with the specified name exists. Error conditions other than fs:file-not-found are not handled by this option.

If a file is loaded, load returns the file's truename.

If *print* is non-nil, the value of each expression evaluated from the file is printed on *standard-output*.

pathname is defaulted from the default pathname defaults. If *set-default-pathname* is non-nil, the pathname defaults are set to the name of the file loaded. The default for *set-default-pathname* is t.

load used to be called with a different calling sequence:

(load *pathname pkg nonexistent-ok*
dont-set-default)

This calling sequence is detected and still works, but it is obsolete.

load-verbose *Variable*

Is the default value for the *verbose* argument to load.

readfile *file* &optional *pkg no-msg-p*

readfile is the version of load for text files. It reads and evaluates each expression in the file. As with load, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package.

fasload *file* &optional *pkg no-msg-p*

fasload is the version of load for QFASL files. It defines functions and performs other actions as directed by the specifications inserted in the file by the compiler. As with load, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package.

25.4 Pathname Operations That Access Files

Here are the operations that access files. Many accept an argument *error* or *error-p* which specifies whether to signal an error or to return a condition instance, if the file cannot be accessed. For these arguments, nil and non-nil are the only significant values. :reprompt has no special meaning as a value. That value when passed to one of the file accessing functions (open, deletef, etc.) has its special significance at a higher level.

:truename *Operation on* pathname

Returns a pathname object describing the exact name of the file specified by the pathname the object is sent to.

This may be different from the original pathname. For example, the original pathname may have :newest as the version, but the truename always has a number as the version if the file system supports versions.

:open *pathname* &rest *options* *Operation on* pathname

Opens a stream for the file named by the pathname. The argument *pathname* is what the :pathname operation on the resulting stream should return. When a logical pathname is opened, *pathname* is that logical pathname, but self is its translated pathname.

options is a list of alternating keywords and values, as would be passed to open. The old style of open keywords are not allowed; when they are used with open, open converts them to the new style before sending the :open message.

:delete &optional (*error-p* t) *Operation on* pathname
:undelete &optional (*error-p* t) *Operation on* pathname

Respectively delete or undelete the file specified by the pathname.

All file systems support :delete but not all support :undelete.

If *error-p* is nil, problems such as nonexistent files cause a string describing the problem to be returned. Otherwise, they signal an error.

:undeletable-p *Operation on* pathname

Returns t if this pathname is for a file system which allows deletion to be undone. Such pathnames support the :undelete and :expunge operations.

:rename *new-name* &optional (*error-p* t) *Operation on* pathname

Renames the file specified by the pathname. *new-name*, a string or pathname, specifies the name to rename to. If it is a string, it is parsed using self as the defaults.

If *error-p* is nil, problems such as nonexistent files cause a string describing the problem to be returned. Otherwise, they signal an error.

:complete-string *string options* *Operation on* pathname

Attempts to complete the filename *string*, returning the results. This operation is used by the function fs:complete-pathname (see page 602). The pathname the message is sent to is used for defaults. *options* is a list whose elements may include :deleted, :read (file is for input), :write (it's for output), :old (only existing files allowed), or :new-ok (new files are allowed too).

There are two values: a string, which is the completion as far as possible, and a flag, which can be :old, :new or nil. :old says that the returned string names an existing file, :new says that the returned string is no file but some completion was done, nil says that no completion was possible.

:change-properties *error-p* &rest *properties* *Operation on* pathname

Changes the properties of the file specified by the pathname. *properties* should be an alternating list of property names and values.

:directory-list *options* *Operation on* pathname

Performs the work of (fs:directory-list *this-pathname options*...).

:properties *Operation on* pathname

Returns a property list (in the form of a directory-list element) and a list of settable properties. See section 25.6, page 598 for more information on file properties.

:wildcard-map *function plistp dir-list-options* &rest *args* *Operation on* pathname

Maps *function* over all the files specified by this pathname (which may contain wildcards). Each time *function* is called, its first argument is a pathname with no wildcards, or else a directory-list element (whose car is a pathname and whose cdr contains property names and values). The elements of *args* are given to *function* as additional arguments.

plistp says whether *function*'s first argument should be a directory-list element or just a pathname. t specifies a directory-list element. That provides more information, but it makes it necessary to do extra work if the specified pathname does *not* contain wildcards.

dir-list-options is passed to fs:directory-list. You can use this to get deleted files mentioned in the list, for example.

The remaining file-access operations are defined only on certain file systems.

:expunge &key (*error* t) *Operation on* pathname

Expunges the directory specified by the host, device and directory components of the pathname.

The argument *error* says whether to signal an error if the directory does not exist. nil means just return a string instead.

:create-directory &key (*error* t) *Operation on* pathname

Creates the directory specified in this pathname.

:remote-connect &key (*error* t) *access* *Operation on* pathname

Performs the work of fs:remote-connect with the same arguments on this pathname's host.

25.5 File Attribute Lists

Any text file can contain an *attribute list* that specifies several attributes of the file. The above loading functions, the compiler, and the editor look at this property list. Attribute lists are especially useful in program source files, i.e. a file that is intended to be loaded (or compiled and then loaded). QFASL files also contain attribute lists, copied from their source files.

If the first non-blank line in a text file contains the three characters '-*-', some text, and '-*-' again, the text is recognized as the file's attribute list. Each attribute consists of the attribute name, a colon, and the attribute value. If there is more than one attribute they are separated by semicolons. An example of such an attribute list is:

```
        ; -*- Mode:Lisp; Package:Cellophane; Base:10 -*-
```

This defines three attributes: mode, package, and base. The initial semicolon makes the line look like a comment rather than a Lisp expression. Another example is:

```
    .c Part of the Lisp Machine manual.   -*- Mode:Bolio -*-
```

An attribute name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. An attribute value can be such a name, or a decimal number, or several such items separated by commas. Spaces may be used freely to separate tokens. Upper and lower-case letters are not distinguished. There is *no* quoting convention for special characters such as colons and semicolons.

If the attribute list text contains no colons, it is an old Emacs format, containing only the value of the **Mode** attribute.

The file attribute list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret. The Emacs editor on the PDP-10 knows how to interpret these attribute lists (primarily in order to look at the **Mode** attribute).

The Lisp Machine handles the attribute list stored in the file by parsing it into a Lisp data structure, a property list. Attribute names are interpreted as Lisp symbols and are interned on the keyword package. Numbers are interpreted as Lisp fixnums and are read in decimal. If a attribute value contains any commas, then the commas separate several expressions that are formed into a list.

When a file is compiled, its attribute list data structure is stored in the QFASL file. It can be loaded back from the QFASL file as well. The representation in the QFASL file resembles nothing described here, but when the attribute list is extracted from there, the same Lisp data structure described above is obtained.

When a file is edited, loaded, or compiled, its file attribute list is read in and the properties are stored on the property list of the generic pathname (see section 24.5, page 561) for that file, where they can be retrieved with the **:get** and **:plist** messages. This is done using the function **fs:read-attribute-list**, below. So the way you examine the properties of a file is usually to use messages to a pathname object that represents the generic pathname of a file. Note that there are other properties there, too.

Here the attribute names with standard meanings:

Mode The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are **Lisp** and **Text**.

Package This attribute specifies the package in which symbols in the file should be interned. The attribute may be either the name of a package, or a list that specifies both the package name and how to create the package if it does not exist. If it is a list, it should look like (*name superpackage initial-size ...options...*). See chapter 27, page 636 for more information about packages.

Base The number base in which the file is written (remember, it is always parsed in decimal). This affects both ***read-base*** and ***print-base***, since it is confusing to have the input and output bases be different. The most common values are 8 and 10.

Readtable The value specifies the syntax (that is, the choice of readtable) to use for reading Lisp objects from this file. The defined values are **t** or **traditional** for traditional Lisp Machine syntax, and **cl** or **common-lisp** for Common Lisp syntax. If you do not specify this option, the objects in the file are read using whatever readtable is current in the program that reads them.

Lowercase If the attribute value is not **nil**, the file is written in lower-case letters and the editor does not translate to upper case. (The editor does not translate to upper case by default unless the user enables Electric Shift Lock mode.)

Fonts The attribute value is a list of font names, separated by commas. The editor uses this for files that are to be displayed in a specific font, or contain multiple fonts. If this attribute is present, the file is actually stored in the file system with font-change indicators. A font-change indicator is an epsilon (ε) followed by a digit or *. ε*n* means to enter font *n*. The previous font is saved on a stack and ε* means to pop the stack, returning to the previous font. If the file includes an epsilon as part of its contents, it is stored as εε.

When expressions are read from such files, font-change indicators are ignored, and εε is treated as a single ε.

Backspace If the attribute value is not **nil**, **Overstrike** characters in the file should cause characters to overprint on each other. The default is to disallow overprinting and display **Overstrike** the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.

Patch-File If the attribute value is not **nil**, the file is a *patch file*. When it is loaded the system will not complain about function redefinitions. In a patch file, the **defvar** special-form turns into **defconst**; thus patch files always reinitialize variables. Patch files are usually created by special editor commands described in section 28.8, page 672.

Cold-Load A non-**nil** value for this attribute identifies files that are part of the cold load, the core from which a new system version is built. Certain features that do not work in the cold load check this flag to give an error or a compiler warning if used in such files, so that the problem can be detected sooner.

You are free to define additional file attributes of your own. However, to avoid accidental name conflicts, you should choose names that are different from all the names above, and from any names likely to be defined by anybody else's programs.

The following functions are used to examine file attribute lists:

fs:file-attribute-list *pathname*
Returns the attribute list of the file specified by the pathname. This works on both text files and QFASL files.

fs:extract-attribute-list *stream*

Returns the attribute list read from the specified stream, which should be pointing to the beginning of a file. This works on both text streams and QFASL file binary streams. After the attribute list is read, the stream's pointer is set back to the beginning of the file using the :set-pointer file stream operation (see page 468).

fs:read-attribute-list *pathname stream*

pathname should be a pathname object (*not* a string or namelist, but an actual pathname); usually it is a generic pathname (see section 24.5, page 561). *stream* should be a stream that has been opened and is pointing to the beginning of the file whose file attribute list is to be parsed. The attribute list is read from the stream and then corresponding properties are placed on the specified *pathname*. The attribute list is also returned.

The fundamental way that programs in the Lisp Machine notice the presence of properties on a file's attribute list is by examining the property list in the generic pathname. However, there is another way that is more convenient for some applications. File attributes can cause special variables to be bound whenever Lisp expressions are being read from the file—when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base attributes work. You can also deal with attributes this way, by using the following function:

fs:file-attribute-bindings *pathname*

Returns values describing the special variables that should be bound before reading expressions from file *pathname*. It examines the property list of *pathname* and finds all those property names that have fs:file-attribute-bindings properties. Each such property name specifies a set of variables to bind and a set of values to which to bind them. This function returns two values, a list of all the variables and a list of all the corresponding values. Usually you use this function by calling it on a generic pathname that has had fs:read-attribute-list done on it, and then you use the two returned values as the first two arguments of a progv special form (see page 32). Inside the body of the progv the specified bindings will be in effect.

pathname may be anything acceptable as the first argument of get. Usually it is a generic pathname.

Of the standard attribute names, the following ones have fs:file-attribute-bindings, with the following effects. Package binds the variable package (see page 637) to the package. Base binds the variables *print-base* (see page 514) and *read-base* (see page 517) to the value. Readtable binds the variable readtable to a value computed from the specified attribute. Patch-file binds fs:this-is-a-patch-file to the value. Cold-load binds si:file-in-cold-load to the value. Fonts binds si:read-discard-font-changes to t.

Any properties whose names do not have fs:file-attribute-bindings properties are ignored completely.

You can also add your own attribute names that affect bindings. If an indicator symbol has an fs:file-attribute-bindings property, the value of that property is a function that is called when a file with a file attribute of that name is going to be read from. The function is given three arguments: the file pathname, the attribute name, and the attribute value. It must return two values: a list of variables to be bound and a list of values to bind them to. The function for the **Base** keyword could have been defined by:

```
(defun (:base file-attribute-bindings) (file ignore bse)
  (if (not (and (typep bse 'fixnum)
                (> bse 1)
                (< bse 37.)))
      (ferror 'fs:invalid-file-attrbute
              "File ~A has an illegal -*- Base:~D -*-"
              file bse))
  (values (list 'base 'ibase) (list bse bse)))
```

fs:extract-attribute-bindings *stream*
Returns two values: a list of variables, and a corresponding list of values to bind them to, giving the attribute bindings of the attribute list found on *stream*

fs:invalid-file-attribute (error) *Condition*
An attribute in the file attribute list had a bad value. This is detected within **fs:file-attribute-bindings.**

25.6 Accessing Directories

To understand the functions in this section, it is vital to have read the chapter on *pathnames*. The *filespec* argument in many of these functions may be a pathname or a namestring; its name, type and version default to **:wild.**

listf *filespec*
Prints on ***standard-output*** the names of the files that match *filespec*, and their sizes, creation dates, and other information that comes in the directory listing.

fs:directory-list *filespec* &rest *options*
Finds all the files that match *filespec* and returns a list with one element for each file. Each element is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a disembodied property list and **get** may be used to access the file's properties. The car of one element is **nil**; the properties in this element are properties of the file system as a whole rather than of a specific file.

filespec normally contains wildcards, and the data returned describe all existing files that match it. If it contains no wildcards, it specifies a single file and only that file is described in the data that are returned.

The *options* are keywords which modify the operation. The following options are currently defined:

:noerror If a file-system error (such as no such directory) occurs during the operation, normally an error is signaled and the user is asked to supply a new pathname. However, if **:noerror** is specified then, in the event of an error, a condition object describing the error is returned as the result of **fs:directory-list.** This is identical to the **:noerror** option to **open.**

:deleted	This is for file servers on which deletion is not permanent. It specifies that deleted (but not yet expunged) files are to be included in the directory listing.
:sorted	This requests that the directory list be sorted by filenames before it is returned.

The properties that may appear in the list of property lists returned by fs:directory-list are host-dependent to some extent. The following properties are those that are defined for both ITS and TOPS-20 file servers. This set of properties is likely to be extended or changed in the future.

:length-in-bytes	The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file).
:byte-size	The number of bits in one of those units.
:length-in-blocks	The length of the file in terms of the file system's unit of storage allocation.
:block-size	The number of bits in one of those units.
:creation-date	The date the file was created, as a universal time. See chapter 34, page 776.
:reference-date	The most recent date on which the file was used, as a universal time or nil, meaning the file was never referenced.
:modification-date	The most recent date on which the file's contents were changed, as a universal time.
:author	The name of the person who created the file, as a string.
:reader	The name of the person who last read the file, as a string.
:not-backed-up	t if the file exists only on disk, nil if it has been backed up on magnetic tape.
:directory	t if this file is actually a directory.
:temporary	t if this file is temporary.
:deleted	t if this file is deleted. Deleted files are included in the directory list only if you specify the :deleted option.
:dont-delete	t indicates that the file is not allowed to be deleted.
:dont-supersede	t indicates that the file may not be superseded; that is, a file with the same name and higher version may not be created.
:dont-reap	t indicates that this file is not supposed to be deleted automatically for lack of use.

:dont-dump	t indicates that this file is not supposed to be dumped onto magnetic tape for backup purposes.
:characters	t indicates that this file contains characters (that is, text). nil indicates that the file contains binary data. This property, rather than the file's byte size, should be used to decide whether it is a text file.
:link-to	If the file is a link, this property is a string containing the name that the link points to.
:offline	T if the file's contents are not online.
:incremental-dump-date	The last time this file was dumped during an incremental dump (a universal time).
:incremental-dump-tape	The tape on which the last was saved in that incremental dump (a string).
:complete-dump-date	The last time this file was dumped during an full dump (a universal time).
:complete-dump-tape	The tape on which the last was saved in that full dump (a string).
:generation-retention-count	The number of files differing in version that are kept around.
:default-generation-retention-count	The generation-retention-count that a file ordinarily gets when it is created in this directory.
:auto-expunge-interval	The interval at which files are expunged from this directory, in seconds.
:date-last-expunged	The last (universal) time this directory was expunged, or nil.
:account	The account to which the file belongs, a string.
:protection	A system-dependent description of the protection of this file as a string.
:physical-volume	A string naming the physical volume on which the file is found.
:volume-name	A string naming the logical volume on which the file is found.
:pack-number	A string describing the pack on which this file is found.
:disk-space-description	A system-dependent description of the space usage on the file system. This usually appears in the plist that applies to the entire directory list.

The element in the directory list that has nil instead of a file's pathname describes the directory as a whole.

:physical-volume-free-blocks
: This property is an alist in which each element maps a physical volume name (a string) into a number, that is the number of free blocks on that volume.

:settable-properties
: This property is a list of file property names that may be set. This information is provided in the directory list because it is different for different file systems.

:pathname
: This property is the pathname from which this directory list was made.

:block-size
: This is the number of words in a block in this directory. It can be used to interpret the numbers of free blocks.

fs:directory-list-stream *filespec* &rest *options*

This is like fs:directory-list but returns the information in a different form. Instead of returning the directory list all at once, it returns a special kind of stream which gives out one element of the directory list at a time.

The directory list stream supports two operations: :entry and :close. :entry asks for the next element of the directory stream. :close closes any connection to a remote file server.

The purpose of using fs:directory-list-stream instead of fs:directory-list is that, when communicating with a remote file server, the directory list stream can give you some of the information without waiting for it to all be transmitted and parsed. This is desirable if the directory is being printed on the console.

directory *filespec*

Returns a list of pathnames (truenames) of the files in the directory specified by *filespec*. Wildcards are allowed. This is the Common Lisp way to find the contents of a directory.

fs:expunge-directory *filespec* &key (*error* t)

Expunges the directory specified in *filespec*; that is, permanently eliminates any deleted files in that directory. If *error* is nil, there is no error if the directory does not exist.

Note that not all file systems support this function. To find out whether a particular one does, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion (and therefore expunging).

fs:create-directory *filespec* &key (*error* t)

Creates the directory specified in *filespec*. If *error* is nil, there is no error if the directory cannot be created; instead an error string is returned. Not all file servers support creation of directories.

fs:remote-connect *filespec* &key (*error* t) *access*

Performs the TOPS-20 "connect" or "access" function, or their equivalents, in a remote file server. Access is done if *access* is non-nil; otherwise, connect is done.

The connect operation grants you full access to the specified directory. The access operation grants you whatever access to all files and directories you would have if logged in on the specified directory. Both operations affect access only, since the connected directory of the remote server is never used by the Lisp Machine in choosing which file to operate on.

This function may ask you for a password if one is required for the directory you specify. If the operation cannot be performed, then if *error* is **nil**, an error object is returned.

File Properties:

fs:change-file-properties *file error-p* &rest *properties*

Changes one or more properties of the file *file*. The *properties* arguments are alternating keywords and values. If an error occurs accessing the file or changing the properties, the *error-p* argument controls what is done; if it is **nil**, a condition object describing the error is returned; if it is **t** a Lisp error is signaled. If no error occurs, **fs:change-file-properties** returns **t**.

Only some of the properties of a file may be changed; for instance, its creation date or its author. Exactly which properties may be changed depends on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list** as explained above.

fs:file-properties *file* &optional (*error-p* **t**)

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. The *error-p* argument is the same as in **fs:change-file-properties**.

Filename Completion:

fs:complete-pathname *defaults string type version* &rest *options*

string is a partially-specified file name. (Presumably it was typed in by a user and terminated with the **Altmode** key or the **End** key to request completion.) **fs:complete-pathname** looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded out in a host-dependent fashion.

defaults, *type*, and *version* are the arguments to be given to **fs:merge-pathname-defaults** (see page 558) when the user's input is eventually parsed and defaulted.

options are keywords (without following values) that control how the completion is performed. The following option keywords are allowed:

:deleted	Looks for files which have been deleted but not yet expunged.
:read or **:in**	The file is going to be read. This is the default.
:print or **:write** or **:out**	The file is going to be written (i.e. a new version is going to be created).

:old	Looks only for files that already exist. This is the default.
:new-ok	Allows either a file that already exists or a file that does not yet exist. An example of the use of this is the C-X C-F (Find File) command in the editor.

The first value returned is always a string containing a file name, either the original string or a new, more specific string. The second value returned indicates the success or failure of the completion. It is nil if an error occurred. One possible error is that the file is on a file system that does not support completion, in which case the original string is returned unchanged. Other possible second values are :old, which means that the string completed to the name of a file that exists, :new, which means that the string completed to the name of a file that could be created, and nil again, which means that there is no possible completion.

Balance Directories:

fs:balance-directories *filespec1 filespec2* &rest *options*

fs:balance-directories is a function for maintaining multiple copies of a directory. Often it is useful to maintain copies of your files on more than one machine; this function provides a simple way of keeping those copies up to date.

The function first parses *filespec1*, filling in missing components with wildcards (except for the version, which is :newest). Then *filespec2* is parsed with *filespec1* as the default. The resulting pathnames are used to generate directory lists using fs:directory-list. Note that the resulting directory lists need not be entire directories; any subset of a directory that fs:directory-list can produce will do.

First the directory lists are matched up on the basis of file name and type. All of the files in either directory list which have both the same name and the same type are grouped together.

The directory lists are next analyzed to determine if the directories are consistent, meaning that two files with the same name and type have equal creation-dates when their versions match, and greater versions have later creation-dates. If any inconsistencies are found, a warning message is printed on the console.

If the version specified for both *filespec1* and *filespec2* was :newest (the default), then the newest version of each file in each directory is copied to the other directory if it is not already there. The result is that each directory has the newest copy of every file in either of the two directories.

If one or both of the specified versions is not :newest, then *every* version that appears in one directory list and not in the other is copied. This has the result that the two directories are completely the same. (Note that this is probably not the right thing to use to *copy* an entire directory. Use copy-file with a wildcard argument instead.)

The *options* are keywords arguments which modify the operation. The following options are currently defined:

:ignore	This option takes one argument, which is a list of file names to ignore when making the directory lists. The default value is **nil**.
:error	This option is identical to the :error option to **open**.
:query-mode	This option takes one argument, which indicates whether or not the user should be asked before files are transferred. If the argument is **nil**, no querying is done. If it is :1->2, then only files being transferred from *filespec2* to *filespec1* are queried, while if it is :2->1, then files transferred from *filespec1* to *filespec2* are queried. If the argument is :always, then the user is asked about all files.
:copy-mode	This option is identical to the :copy-mode option of **copy-file**, and is used to control whether files are treated as binary or textual data.
:direction	This option specifies transfer of files in one direction only. If the value is :1->2 then files are transfered only from *filespec1* to *filespec2*, never in the other direction. If the value is :2->1 then files are transferred only from *filespec2* to *filespec1*. **nil**, the default, means transfer in either direction as appropriate.

25.7 Errors in Accessing Files

fs:file-error (error) *Condition Flavor*

This flavor is the basis for all errors signaled by the file system.

It defines two special operations, :pathname and :operation. Usually, these return the pathname of the file being operated on, and the operation used. This operation was performed either on the pathname object itself, or on a stream.

It defines prompting for the proceed types :retry-file-operation and :new-pathname, both of which are provided for many file errors. :retry-file-operation tries the operation again exactly as it was requested by the program; :new-pathname expects on argument, a pathname, and tries the same operation on this pathname instead of the original one.

fs:file-operation-failure (fs:file-error) *Condition*

This condition name signifies a problem with the file operation requested. It is an alternative to fs:file-request-failure (page 609), which means that the file system was unable to consider the operation properly.

All the following conditions in this section are always accompanied by **fs:file-operation-failure**, **fs:file-error**, and **error**, so they will not be mentioned.

fs:file-open-for-output *Condition*

The request cannot be performed because the file is open for output.

fs:file-locked *Condition*

The file cannot be accessed because it is already being accessed. Just which kinds of simultaneous access are allowed depends on the file system.

fs:circular-link *Condition*

A link could not be opened because it pointed, directly or indirectly through other links, to itself. In fact, some systems report this condition whenever a chain of links exceeds a fixed length.

fs:invalid-byte-size *Condition*

In open, the specified byte size was not valid for the particular file server or file.

fs:no-more-room *Condition*

Processing a request requires resources not available, such as space in a directory, or free disk blocks.

fs:filepos-out-of-range *Condition*

The :set-pointer operation was used with a pointer value outside the bounds of the file.

fs:not-available *Condition*

A requested pack, file, etc. exists but is currently off line or not available to users.

fs:file-lookup-error *Condition*

This condition name categorizes all sorts of failure to find a specified file, for any operation.

fs:device-not-found (fs:file-lookup-error) *Condition*

The specified device does not exist.

fs:directory-not-found (fs:file-lookup-error) *Condition*

The specified directory does not exist.

fs:file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name, type and version. This implies that the device and directory do exist, or one of the errors described above would have been signaled.

fs:multiple-file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name and any of the specified types, in with-open-file-search. Three special operations are defined:

:operation	Returns the function which used with-open-file-search, such as load.
:pathname	The base pathname used.
:pathnames	A list of all the pathnames that were looked for.

fs:link-target-not-found (fs:file-lookup-error) *Condition*

The file specified was a link, but the link's target filename fails to be found.

fs:access-error *Condition*
The operation is possible, but the file server is insubordinate and refuses to obey you.

fs:incorrect-access-to-file (access-error). *Condition*
fs:incorrect-access-to-directory (access-error). *Condition*
The file server refuses to obey you because of protection attached to the file (or, the directory).

fs:invalid-wildcard *Condition*
A pathname had a wildcard in a place where the particular file server does not support them. Such pathnames are not created by pathname parsing, but they can be created with the :new-pathname operation.

fs:wildcard-not-allowed *Condition*
A pathname with a wildcard was used in an operation that does not support it. For example, opening a file with a wildcard in its name.

fs:wrong-kind-of-file *Condition*
An operation was done on the wrong kind of file. If files and directories share one name space and it is an error to open a directory, the error possesses this condition name.

fs:creation-failure *Condition*
An attempt to create a file or directory failed for a reason specifically connected with creation.

fs:file-already-exists (fs:creation-failure) *Condition*
The file or directory to be created already exists.

fs:superior-not-directory (fs:creation-failure fs:wrong-kind-of-file) *Condition*
In file systems where directories and files share one name space, this error results from an attempt to create a file using a filename specifying a directory whose name exists in the file system but is not a directory.

fs:delete-failure *Condition*
A file to be deleted exists, but for some reason cannot be deleted.

fs:directory-not-empty (fs:delete-failure) *Condition*
A file could not be deleted because it is a directory and has files in it.

fs:dont-delete-flag-set (fs:delete-failure) *Condition*
A file could not be deleted because its "don't delete" flag is set.

fs:rename-failure *Condition*
A file to be renamed exists, but the renaming could not be done. The :new-pathname operation on the condition instance returns the specified new pathname, which may be a pathname or a string.

fs:rename-to-existing-file (fs:rename-failure) *Condition*
Renaming cannot be done because there is already a file with the specified new name.

fs:rename-across-directories (fs:rename-failure) *Condition*
Renaming cannot be done because the new pathname contains a different device or directory from the one the file is on. This may not always be an error—some file systems support it in certain cases—but when it is an error, it has this condition name.

fs:unknown-property (fs:change-property-failure) *Condition*
A property name specified in a :change-properties operation is not supported by the file server. (Some file servers support only a fixed set of property names.) The :property operation on the condition instance returns the problematical property name.

fs:invalid-property-value (fs:change-property-failure) *Condition*
In a :change-properties operation, some property was given a value that is not valid for it. The :property operation on the condition instance returns the property name, and the :value operation returns the specified value.

fs:invalid-property-name (fs:change-property-failure) *Condition*
In a :change-properties operation, a syntactically invalid property name was specified. This may be because it is too long to be stored. The :property operation on the condition instance returns the property name.

25.8 File Servers

Files on remote file servers are accessed using *file servers* over the Chaosnet. Normally connections to servers are established automatically when you try to use them, but there are a few ways you can interact with them explicitly.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Lisp Machine characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding; the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, is not the same measured in original Lisp Machine characters as it is measured in the encoded ASCII characters. In the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (Delete), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a quoting character for these codes. Codes 210 (Overstrike), 211 (Tab), 212 (Line), and 214 (Page), are converted to their ASCII cognates, namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (Return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

When a file server is first created for you on a particular host, you must tell the server how to log in on that host. This involves specifying a *username*, and, if the obstructionists are in control of your site, a password. The Lisp Machine prompts you for these on the terminal when they are needed.

Logging in a file server is not the same thing as logging in on the Lisp Machine (see **login**, page 801). The latter identifies you as a user in general and involves specifying one host, your login host. The former identifies you to a particular file server host and must be done for each host on which you access files. However, logging in on the Lisp Machine does specify the username for your login host and logs in a file server there.

The Lisp Machine uses your username (or the part that follows the last period) as a first guess for your password (this happens to take no extra time). If that does not work, you are asked to type a password, or else a username and a password, on the keyboard. You do not have to give the same user name that you are logged in as, since you may have or use different user names on different machines.

Once a password is recorded for one host, the system uses that password as the guess if you connect to a file server on another host.

fs:user-unames *Variable*

This is an alist matching host names with the usernames you have specified on those hosts. Each element is the cons of a host object and the username, as a string.

For hosts running ITS, the symbol **fs:its** is used instead of a host object. This is because every user has the same username on all ITS hosts.

fs:user-host-password-alist *Variable*

Once you have specified a password for a given username and host, it is remembered for the duration of the session in this variable. The value is a list of elements, each of the form

((*username hostname*) *password*)

All three data are strings.

The remembered passwords are used if more than one file server is needed on the same host, or if the connection is broken and a new file server needs to be created.

If you are very scared of your password being known, you can turn off the recording by setting this variable:

fs:record-passwords-flag *Variable*

Passwords are recorded when typed in if this variable is non-**nil**.

You should set the variable at the front of your init file, and also set **fs:user-host-password-alist** to **nil**, since it will already have recorded your password when you logged in.

If you do not use a file server for a period of time, it is killed to save resources on the server host.

fs:host-unit-lifetime *Variable*

This is the length of time after which an idle file server connection should be closed, in 60ths of a second. The default is 20 minutes.

Some hosts have a caste system in which all users are not equal. It is sometimes necessary to enable one's privileges in order to exercise them. This is done with these functions:

fs:enable-capabilities *host* &rest *capabilities*

Enables the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is nil, a default list of capabilities is enabled; the default is also dependent on the operating system type.

fs:disable-capabilities *host* &rest *capabilities*

Disables the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is nil, a default list of capabilities is disabled; the default is also dependent on the operating system type.

The PEEK utility has a mode that displays the status of all your file connections, and of the *host unit* data structures that record them. Clicking on a connection with the mouse gets a menu of operations, of which the most interesting is reset. Resetting a host unit may be useful if the connection becomes hung.

25.8.1 Errors in Communication with File Servers

fs:file-request-failure (fs:file-error error) *Condition*

This condition name categorizes errors that prevent the file system from processing the request made by the program.

The following condition names are always accompanied by the more general classifications fs:file-request-failure, fs:file-error, and error.

fs:data-error *Condition*

This condition signifies inconsistent data found in the file system, indicating a failure in the file system software or hardware.

fs:host-not-available *Condition*

This condition signifies that the file server host is up, but refusing connections for file servers.

fs:network-lossage *Condition*

This condition signifies certain problems in the use of the Chaosnet by a file server, such as failure to open a data connection when it is expected.

fs:not-enough-resources *Condition*

This condition signifies a shortage of resources needed to consider processing a request, as opposed to resources used up by the request itself. This may include running out of network connections or job slots on the server host. It does not include running out of space in a directory or running out of disk space, because these are resources whose requirements come from processing the request.

fs:unknown-operation *Condition*

This condition signifies that the particular file system fails to implement a standardly defined operation; such as, expunging or undeletion on ITS.