

## 22. Naming of Files

A Lisp Machine generally has access to many file systems. While it may have its own file system on its own disks, usually a community of Lisp Machine users want to have a shared file system accessible by any of the Lisp Machines over a network. These shared file systems can be implemented by any computer that is capable of providing file system service. A file server computer may be a special-purpose computer that does nothing but service file system requests from computers on a network, or it may be a time-sharing system.

Programs need to use names to designate files within these file systems. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, in the ITS file system, a typical name looks like:

```
DSK: GEORGE; FOO QFASL
```

with DSK being a device name, GEORGE being a directory name, FOO being the first file name and QFASL being the second file name. However, in TOPS-20, a similar file name is expressed as:

```
PS:<GEORGE>FOO.QFASL
```

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists, or new formats that could get added in the future. However, existing programs should retain their abilities to manipulate the names.

The functions and flavors described in this chapter exist to solve this problem. They provide an interface through which a program can deal with names of files and manipulate them without depending on anything about their syntax. This lets a program deal with multiple remote file servers simultaneously, using a uniform set of conventions.

[On the other hand, changes in the standard pathname interface frequently make your old programs fail to work with any file systems. But that's excusable, right?]

### 22.1 Pathnames

All file systems dealt with by the Lisp Machine are mapped into a common model, in which files are named by something called a *pathname*. A pathname always has six components, each with a standard meaning. These components are the common interface that allows programs to work the same way with different file systems: the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. Pathname components are described in the following section, and the mappings between components and user syntax is described for each file system later in this chapter.

A pathname is an instance of a flavor (see chapter 20, page 321); exactly which flavor depends on what the host of the pathname is, but `fs:pathname` is always one of its component flavors. If *p* is a pathname, then `(typep p 'fs:pathname)` will return `t`. One of the messages handled by host objects is the `:pathname-flavor` operation, which returns the name of the flavor to use for pathnames on that host. And one of the differences between flavors of host is how they handle this operation.

There are functions for manipulating pathnames, and there are also messages that can be sent to them. These are described later in this chapter.

Two important operations of the pathname system are *parsing* and *merging*. Parsing is the conversion of a string—which might be something typed in by the user when asked to supply the name of a file—into a pathname object. This involves finding out what host the pathname is for, then using the file name syntax conventions of that host to parse the string into the standard pathname components. Merging is the operation that takes a pathname with missing components and supplies values for those components from a set of defaults.

The function `string`, applied to a pathname, converts it into a string that is in the file name syntax of its host's file system, except that the name of the host followed by a colon is inserted at the front. This is the inverse of parsing. `princ` of a pathname also does this, then prints the contents of the string. Flavor operations such as `:string-for-dired` exist which convert all or part of a pathname to a string in other fashions that are designed for specific applications. `prin1` of a pathname prints the pathname using the `#c` syntax so it can be read back in to produce an equivalent pathname (or the same pathname, if read in the same session).

Since each kind of file server can have its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and figuring out what each component is. The parsers all work differently. How can the parsing operation know which parser to use? The first thing that the parser does is to figure out which host this filename belongs to. A filename character string may specify a host explicitly by having the name of the host, followed by a colon, at either the beginning or the end of the string. For example, the following strings all specify hosts explicitly:

```
AI: COMMON; GEE WHIZ           ; This specifies host AI.
COMMON; GEE WHIZ AI:           ; So does this.
AI: ARC: USERS1; FOO BAR       ; So does this.
ARC: USERS1; FOO BAR AI:       ; So does this.
EE:PS:<COMMON>GEE.WHIZ.5       ; This specifies host EE.
PS:<COMMON>GEE.WHIZ.5 EE:       ; So does this.
```

If the string does not specify a host explicitly, the parser will assume some particular host is the one in question, and will use the parser for that host's file system. The optional arguments passed to the parsing function (`fs:parse-pathname`) tell it which host to assume. Note: the parser won't be confused by strings starting with "DSK:" or "PS:" because it knows that neither of those is a valid host name. (If some file system's syntax allowed file names that start with the name of a valid host followed by a colon, there could be problems.)

Pathnames are kept unique, like symbols, so that there is only one object with a given set of components. This is useful because a pathname object has a property list (see section 5.9, page 81) on which you can store properties describing the file or family of files that the pathname represents. The uniqueness implies that any time the same components are typed in, the program will get the same pathname object and find there the properties it ought to find.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; a pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with "newest" as its version will refer to the same file as a pathname with the same components but a certain number as the version. In systems with links, multiple file names, logical devices, etc. two pathnames that look

quite different may really turn out to address the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

When you want to store properties describing an individual file, use the pathname you get by sending `:truename` to a stream rather than the pathname you open. This avoids problems with different pathnames that refer to the same file.

To get a unique pathname object representing a family of files, send the message `:generic-pathname` to a pathname for any file in the family (see section 22.5, page 467).

## 22.2 Pathname Components

These are the components of a pathname. They will be clarified by an example below.

<i>host</i>	An object that represents the file system machine on which the file resides. A host object is an instance of a flavor one of whose components is <code>si:basic-host</code> . The precise flavor varies depending on the type of file system and how the files are to be accessed.
<i>device</i>	Corresponds to the "device" or "file structure" concept in many host file systems.
<i>directory</i>	The name of a group of related files belonging to a single user or project. Corresponds to the "directory" concept in many host file systems.
<i>name</i>	The name of a group of files that can be thought of as conceptually the "same" file. Many host file systems have a concept of "name" which maps directly into this component.
<i>type</i>	Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is; such as, a Lisp source file, a QFASL file, etc.
<i>version</i>	Corresponds to the "version number" concept in many host file systems. This is a number that increments every time the file is modified.

As an example, consider a Lisp program named **CONCH**. If it belongs to **GEORGE**, who uses the **FISH** machine, the host would be the host-object for the machine **FISH**, the device would probably be the default and the directory would be **GEORGE**. On this directory would be a number of files related to the **CONCH** program. The source code for this program would live in a set of files with name **CONCH**, type **LISP**, and versions **1**, **2**, **3**, etc. The compiled form of the program would live in files named **CONCH** with type **QFASL**; each would have the same version number as the source file that it came from. If the program had a documentation file, it would have type **INFO**.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is `nil`. Before a file server can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in from defaults. But pathnames with missing components are often handed around inside the machine, since almost all pathnames typed by users do not specify all the components explicitly. The host is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to know what its host is. All pathnames have host attributes, even if the string being parsed does not specify one explicitly.

A component of a pathname can also be the special symbol `:unspecific`. `:unspecific` means, explicitly, "this component has been specified as missing", whereas `nil` means that the component was not specified and should default. In merging, `:unspecific` counts as a specified component and is not replaced by a default. `:unspecific` does *not* mean "unspecified"; it is unfortunate that those two words are similar.

`:unspecific` is used in *generic* pathnames, which refer not to a file but to a whole family of files. The version, and usually the type, of a generic pathname are `:unspecific`. Another way `:unspecific` is used has to do with mapping of pathnames into file systems such as ITS that do not have all six components. A component that is really not there will be `:unspecific` in the pathname. When a pathname is converted to a string, `nil` and `:unspecific` both cause the component not to appear in the string.

A component of a pathname can also be the special symbol `:wild`. This is useful only when the pathname is being used with a directory primitive such as `fs:directory-list` (see page 442), where it means that this pathname component matches anything. The printed representation of a pathname usually designates `:wild` with an asterisk; however, this is host-dependent.

What values are allowed for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually just treated as something supplied by the user that only needs to be remembered.

In general, programs can interpret the components of a pathname independent of the file system; and a certain minimum set of possible values of each component will be supported on all file systems. The same pathname component value may have very different representations when the pathname is made into a string, depending on the file system. This does not affect programs that operate on the components. The user, when asked to type a pathname, always uses the system-dependent string representation. This is convenient for the user who moves between using the Lisp machine on files stored on another host and making direct use of that host. However, when the mapping between string form and components is complicated, the components may not be obvious from what you type.

The type is always a string, or one of the special symbols `nil`, `:unspecific`, and `:wild`. Certain hosts impose a limit on the size of string allowed, often very small. Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are usually "LISP", compiled Lisp programs are "QFASL", etc. However, these file type conventions are host-specific, for the important reason that some hosts do not allow a string five characters long to be used as the type. Therefore, programs should use a *canonical type* rather than an actual string to specify their conventional default file types. Canonical types are described below.

For the version, it is always legitimate to use a positive fixnum, or certain special symbols. `nil`, `:unspecific`, and `:wild` have been explained above. The other standardly allowed symbols are `:newest` and `:oldest`. `:newest` refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. `:oldest` refers to the smallest version number that exists. Some file systems may define other special version symbols, such as `:installed` for

example, or may allow negative numbers. Some do not support versions at all. Then a pathname may still contain any of the standard version components, but it does not matter what the value is.

The device, directory, and name are more system-dependent. These can be strings (with host-dependent rules on allowed characters and length) or they can be *structured*. A structured component is a list of strings. This is used for file system features such as hierarchical directories. The system is arranged so that programs do not need to know about structured components unless they do host-dependent operations. Giving a string as a pathname component to a host that wants a structured value converts the string to the appropriate form. Giving a structured component to a host that does not understand them converts it to a string by taking the first element and ignoring the rest.

Some host file systems have features that do not fit into this pathname model. For instance, directories might be accessible as files, there might be complicated structure in the directories or names, or there might be relative directories, such as "<" in Multics. These features appear in the parsing of strings into pathnames, which is one reason why the strings are written in host-dependent syntax. Pathnames for hosts with these features are also likely to handle additional messages besides the common ones documented in this chapter, for the benefit of host-dependent programs that want to access those features. However, note that once your program depends on any such features, it will only work for certain file servers and not others; in general, it is a good idea to make your program work just as well no matter what file server is being used.

### 22.2.1 Raw Components and Interchange Components

On some host file systems it is conventional to use lower-case letters in file names, while in others upper case is customary, or possibly required. When pathname components are moved from pathnames of one file system to pathnames of another file system, it is useful to convert the case if necessary so that you get the right case convention for the latter file system as a default. This is especially useful when copying files from one file system to another.

The Lisp machine system defines two representations for each of several pathname components (the device, directory, name and type). There is the *raw* form, which is what actually appears in the filename on the host file system, and there is the *interchange* form, which may differ in alphabetic case from the raw form. The raw form is what is stored inside the pathname object itself, but programs nearly always operate on the interchange form. The `:name`, `:type`, etc., operations return the interchange form, and the `:new-name`, etc., operations expect the interchange form. Additional operations `:raw-name`, etc., are provided for working with the raw components, but these are rarely needed.

The interchange form is defined so that it is always customarily in upper case. If upper case is customary on the host file system, then the interchange form of a component is the same as the raw form. If lower case is customary on the host file system, as on Unix, then the interchange form has case inverted. More precisely, an all-upper-case component is changed to all-lower-case, an all-lower-case component is changed to all-upper-case, and a mixed-case component is not changed. (This is a one-to-one mapping). Thus, a Unix pathname with a name component of "foo" will have an interchange-format name of "FOO", and vice versa.

For host file systems which record case when files are created but ignore case when comparing filenames, the interchange form is always upper case.

The host component is not really a name, and case is always ignored in host names, so there is no need for two forms of host component. The version component does not need them either, because it is never a string.

### 22.2.2 Pathname Component Operations

<b>:host</b>	<i>Operation on fs:pathname</i>
<b>:device</b>	<i>Operation on fs:pathname</i>
<b>:directory</b>	<i>Operation on fs:pathname</i>
<b>:name</b>	<i>Operation on fs:pathname</i>
<b>:type</b>	<i>Operation on fs:pathname</i>
<b>:version</b>	<i>Operation on fs:pathname</i>

These return the components of the pathname, in interchange form. The returned values can be strings, special symbols, or lists of strings in the case of structured components. The type will always be a string or a symbol. The version will always be a number or a symbol.

<b>:raw-device</b>	<i>Operation on fs:pathname</i>
<b>:raw-directory</b>	<i>Operation on fs:pathname</i>
<b>:raw-name</b>	<i>Operation on fs:pathname</i>
<b>:raw-type</b>	<i>Operation on fs:pathname</i>

These return the components of the pathname, in raw form.

<b>:new-device</b> <i>dev</i>	<i>Operation on fs:pathname</i>
<b>:new-directory</b> <i>dir</i>	<i>Operation on fs:pathname</i>
<b>:new-name</b> <i>name</i>	<i>Operation on fs:pathname</i>
<b>:new-type</b> <i>type</i>	<i>Operation on fs:pathname</i>
<b>:new-version</b> <i>version</i>	<i>Operation on fs:pathname</i>

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as being in interchange form, which means its case may be converted. The **:new-device**, **:new-directory** and **:new-name** operations accept a string (or a special symbol) or a list that is a structured name. If the host does not define structured components, and you specify a list, its first element is used.

<b>:new-raw-device</b> <i>dev</i>	<i>Operation on fs:pathname</i>
<b>:new-raw-directory</b> <i>dir</i>	<i>Operation on fs:pathname</i>
<b>:new-raw-name</b> <i>name</i>	<i>Operation on fs:pathname</i>
<b>:new-raw-type</b> <i>type</i>	<i>Operation on fs:pathname</i>

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as raw.

**:new-suggested-name** *name* *Operation on fs:pathname*

**:new-suggested-directory** *dir* *Operation on fs:pathname*

These differ from the `:new-name` and `:new-directory` operations in that the new pathname constructed has a name or directory based on the suggestion, but not necessarily identical to it. It tries, in a system-dependent manner, to adapt the suggested name or directory to the usual customs of the file system in use.

For example, on a TOPS-20 system, these operations would convert *name* or *dir* to upper case, because while lower-case letters *may* appear in TOPS-20 pathnames, it is not customary to generate such pathnames by default.

**:new-pathname** &rest *options* *Operation on fs:pathname*

This returns a new pathname that is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are `:host`, `:device`, `:directory`, `:name`, `:type`, and `:version`. Alternatively, the keywords `:raw-device`, `:raw-directory`, `:raw-name` and `:raw-type` may be used to specify a component in raw form.

Two additional keywords, `:canonical-type` and `:original-type`, allow the type field to be specified as a canonical type. See the following section for a description of canonical types. Also, the value specified for the keyword `:type` may be a canonical type symbol.

The operations `:new-name`, etc., are equivalent to `:new-pathname` specifying only one component to be changed; in fact, that is how those operations are implemented.

### 22.2.3 Canonical Types

*Canonical types* are a way of specifying a pathname type component using host-dependent conventions without making the program itself explicitly host dependent. For example, the function `qc-file` normally provides a default type of "LISP", but on VMS systems the default must be "LSP" instead, and on Unix systems it is "I". What `qc-file` actually does is to use a canonical type, the keyword `:lisp`, as the default. This keyword is given a definition as a canonical type, which specifies what it maps into on various file systems.

A single canonical type may have more than one mapping on a particular file system. For example, on TOPS-20 systems the canonical type `:LISP` maps into either "LISP" or "LSP". One of the possibilities is marked as "preferred"; in this case, it is "LISP". The effect of this is that either `FOO.LISP` or `FOO.LSP` would be acceptable as having canonical type `:lisp`, but merging will yield "LISP" as the type when defaulting from `:lisp`.

Note that the canonical type of a pathname is not a distinct component. It is another way of describing or specifying the type component.

A canonical type must be defined before it is used.

**fs:define-canonical-type** *symbol standard-mapping* *Special Form*  
*system-dependent-mappings..*

Defines *symbol* as a canonical type. *standard-mapping* is the actual type component that it maps into (a string), with exceptions as specified by *system-dependent-mappings*. Each element of *system-dependent-mappings* (that is, each additional argument) is a list of the form

*(system-type preferred-mapping other-mappings. . .)*

*system-type* is one of the system-type keywords the `:system-type` operation on a host object can return, such as `:unix`, `:tops20`, and `:lispm` (see page 481). The argument describes how to map this canonical type on that type of file system. *preferred-map* (a string) is the preferred mapping of the canonical type, and *other-mappings* are additional strings that are accepted as matching the canonical type.

*system-type* may also be a list of system types. Then the argument applies to all of those types of file systems.

All of the mapping strings are in interchange form.

For example, the canonical type `:lisp` is defined as follows:

```
(fs:define-canonical-type :lisp "LISP"
  (:unix "L" "LISP")
  (:vms "LSP")
  ((:tops20 :tenex) "LISP" "LSP"))
```

Other canonical types defined by the system include `:qfasl`, `:text`, `:press`, `:qwabl`, `:babyl`, `:mail`, `:xmail`, `:init`, `:patch-directory`, `:midas`, `:palx`, `:unfasl`, `:kst`, `:widths`, and `:output`. The standard mapping for each is the symbol's pname.

To match a pathname against a canonical type, use the `:canonical-type` operation.

**:canonical-type** *Operation on fs:pathname*

Returns two values which describe whether and how this pathname's type component matches any canonical type.

If the type component is one of the possible mappings of some canonical type, the first value is that canonical type (the symbol). The second value is `nil` if the type component is the preferred mapping of the canonical type; otherwise it is the actual type component, in interchange form. The second value is called the *original type* of the pathname.

If the type component does not match a canonical type, the first value is the type component in interchange form (a string), and the second value is `nil`.

This operation is useful in matching a pathname against a canonical type; the first value is `eq` to the canonical type if the pathname matches it. The operation is also useful for transferring a type field from one file system to another while preserving canonical type; this is described below.

A new pathname may also be constructed by specifying a canonical type.



**:new-canonical-type** *canonical-type* &optional *original-type*      *Operation on fs:pathname*

Returns a pathname different from this one in having a type component that matches *canonical-type*.

If *original-type* is a possible mapping for *canonical-type* on this pathname's host, then it is used as the type component. Otherwise, the preferred mapping for *canonical-type* is used. If *original-type* is not specified, it defaults to this pathname's type component. If it is specified as nil, the preferred mapping of the canonical type is always used. If *canonical-type* is a string rather than an actual canonical type, it is used directly as the type component, and the *original-type* does not matter.

The **:new-pathname** operation accepts the keywords **:canonical-type** and **:original-type**. The **:new-canonical-type** operation is equivalent to **:new-pathname** with those keywords.

Suppose you wish to copy the file named *old-pathname* to a directory named *target-directory-pathname*, possibly on another host, while preserving the name, version and canonical type. That is, if the original file has a name acceptable for a QFASL file, the new file should also. Here is how to compute the new pathname:

```
(multiple-value-bind (canonical original)
  (send old-pathname ':canonical-type)
  (send target-directory-pathname ':new-pathname
    ':name (send old-pathname ':version)
    ':version (send old-pathname ':version)
    ':canonical-type canonical
    ':original-type original))
```

Suppose that *old-pathname* is OZ:<FOO>A.LISP.5, where OZ is a TOPS-20, and the target directory is on a VMS host. *canonical* will be **:lisp** and *original* will be "LISP". Since "LISP" is not an acceptable mapping for **:lisp** on a VMS system, the resulting pathname will have as its type component the preferred mapping for **:lisp** on VMS, namely, "LSP".

But if the target host is a Unix host, the new file's type will be "LISP", since that is an acceptable (though not preferred) mapping for **:lisp** on Unix hosts. If you would rather that the preferred mapping always be used for the new file's type, omit the **:original-type** argument to the **:new-pathname** operation. This would result in a type component of "L" in interchange form, or "l" in raw form, in the new file's pathname.

The function **qc-file** actually does something cleverer than using the canonical type as a default. Doing that, and opening the resulting pathname, would look only for the preferred mapping of the canonical type. **qc-file** actually tries to open *each* possible mapping, trying the preferred mapping first. Here is how it does so:

**:open-canonical-default-type** *canonical-type* *Operation on fs:pathname*  
*&rest options*

If this pathname's type component is non-nil, the pathname is simply opened, passing the *options* to the :open operation. If the type component is nil, each mapping of *canonical-type* is tried as a type component, in the order the mappings appear in the canonical type definition. If an open succeeds, a stream is returned. The possibilities continue to be tried as long as **fs:file-not-found** errors happen; other errors are not handled. If all the possibilities fail, a **fs:file-not-found** error is signaled for the caller, with a pathname that contains the preferred mapping as its type component.

## 22.3 Defaults and Merging

When the user is asked to type in a pathname, it is of course unreasonable to require the user to type a complete pathname, containing all components. Instead there are defaults, so that components not specified by the user can be supplied automatically by the system. Each program that deals with pathnames typically has its own set of defaults.

The system defines an object called a *defaults alist*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults alist can remember more than one default pathname if defaults are being kept separately for each host; this is controlled by the variable **fs:\*defaults-are-per-host\***. The main primitive for using defaults is the function **fs:merge-pathname-defaults** (see page 465).

In place of a defaults alist, you may use just a pathname. Defaulting one pathname from another is useful for cases such as a program that has an input file and an output file, and asks the user for the name of both, letting the unsupplied components of one name default from the other. Unspecified components of the output pathname will come from the input pathname, except that the type should default not to the type of the input but to the appropriate default type for output from this program.

The implementation of a defaults alist is an association list of host names and default pathnames. The host name nil is special and holds the defaults for all hosts, when defaults are not per-host.

The *merging* operation takes as input a pathname, a defaults alist (or another pathname), a default type, and a default version, and returns a pathname. Basically, the missing components in the pathname are filled in from the defaults alist. However, if a name is specified but the type or version is not, then the type or version is treated specially.

The full details of the merging rules are as follows. First, if no host is specified, the host is taken from the defaults. If the pathname explicitly specifies a host and does not supply a device, then the device will be the default file device for that host.

If the pathname specifies a device named DSK, that is replaced with the *working device* for the pathname's host, and the directory defaults to the *working directory* for the host if it is not specified. See **fs:set-host-working-directory**, below.

Next, if the pathname does not specify a host, device, directory, or name, that component comes from the defaults.

If the value of `fs:*always-merge-type-and-version*` is non-nil, the type and version are merged just like the other components.

If `fs:*always-merge-type-and-version*` is nil, as it normally is, the merging rules for the type and version are more complicated and depend on whether the pathname specifies a name. If the pathname doesn't specify a name, then the type and version, if not provided, will come from the defaults, just like the other components. However, if the pathname does specify a name, then the type and version come from the *default-type* and *default-version* arguments to `merge-pathname-defaults`. If those arguments were omitted, the value of `fs:*name-specified-default-type*` (initially, `:lisp`) is used as the default type, and `:newest` is used as the default version.

The reason for this is that the type and version "belong to" some other filename, and are thought to be unlikely to have anything to do with the new filename you are typing in.

#### **fs:set-host-working-directory** *host* *pathname*

This sets the *working device* and *working directory* for *host* to those specified in *pathname*. *host* should be a host object or the name of a host. *pathname* may be a string or a pathname. The working device and working directory are used for defaulting pathnames in which the device is specified as `DSK`.

The editor command `Meta-X Set Working Directory` provides a convenient interface to this function.

The following special variables are parts of the pathname interface that are relevant to defaults.

#### **fs:\*defaults-are-per-host\***

*Variable*

This is a user customization option intended to be set by a user's `LISPM INIT` file (see section 32.7, page 648). The default value is nil, which means that each program's set of defaults contains only one default pathname. If you type in just a host name and a colon, the other components of the name will default from the previous host, with appropriate translation to the new host's pathname syntax. If `fs:*defaults-are-per-host*` is set to `t`, each program's set of defaults will maintain a separate default pathname for each host. If you type in just a host name and a colon, the last file that was referenced on that host will be used.

#### **fs:\*always-merge-type-and-version\***

*Variable*

If this variable is non-nil, then the type and version are defaulted only from the pathname defaults just like the other components.

#### **fs:\*name-specified-default-type\***

*Variable*

If `fs:*always-merge-type-and-version*` is nil, then when a name is specified but not a type, the type defaults from an argument to the merging function. If that argument is not specified, this variable's value is used. It may be a string or a canonical type keyword. The value is initially `:lisp`.

**fs:\*default-pathname-defaults\****Variable*

This is the default defaults alist; if the pathname primitives that need a set of defaults are not given one, they use this one. Most programs, however, should have their own defaults rather than using these.

**fs:load-pathname-defaults***Variable*

This is the defaults alist for the `load` and `qc-file` functions. Other functions may share these defaults if they deem this to be an appropriate user interface.

**fs:last-file-opened***Variable*

This is the pathname of the last file that was opened. Occasionally this is useful as a default. Since some programs deal with files without notifying the user, you must not expect the user to know what the value of this symbol is. Using this symbol as a default may cause unfortunate surprises, and so such use is discouraged.

These functions are used to manipulate defaults alists directly.

**fs:make-pathname-defaults**

Creates a defaults alist initially containing no defaults. If you ask this empty set of defaults for its default pathname before anything has been stored into it you will get the file FOO on the user's home directory on the host he logged in to.

**fs:copy-pathname-defaults** *defaults*

Creates a defaults alist initially a copy of *defaults*.

**fs:default-pathname** &optional *defaults host default-type default-version*

This is the primitive function for getting a default pathname out of a defaults alist. Specifying the optional arguments *host*, *default-type*, and *default-version* to be non-`nil` forces those fields of the returned pathname to contain those values.

If `fs:*defaults-are-per-host*` is `nil` (its default value), this gets the one relevant default from the alist. If it is `t`, this gets the default for *host* if one is specified, otherwise for the host most recently used.

If *defaults* is not specified, the default defaults are used.

This function has an additional optional argument *internal-p*, is obsolete.

**fs:default-host** *defaults*

Returns the default host object specified by the defaults-alist *defaults*. This is the host that will be used by pathname defaulting with the given defaults if no host is specified.

**fs:set-default-pathname** *pathname* &optional *defaults*

This is the primitive function for updating a set of defaults. It stores *pathname* into *defaults*. If *defaults* is not specified, the default defaults are used.

## 22.4 Pathname Functions

These functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

### **fs:parse-pathname** *thing* &optional *host defaults*

This turns *thing*, which can be a pathname, a string, a symbol, or a Maclisp-style name list, into a pathname. Most functions that are advertised to take a pathname argument call **fs:parse-pathname** on it so that they will accept anything that can be turned into a pathname.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host. If *thing* does not contain a manifest host name, then if *host* is non-*nil*, it is the host name to use, as a string. If *thing* is a string, a manifest host name may be at the beginning or the end, and consists of the name of a host followed by a colon. If *host* is *nil* then the host name is obtained from the default pathname in *defaults*. If *defaults* is not supplied, the default defaults (**fs:\*default-pathname-defaults\***) are used.

Note that if *host* is specified, and *thing* contains a host name, an error is signalled if they are not the same host.

### **fs:pathname-parse-error** (**fs:pathname-error** error)

*Condition*

This condition is signaled when **fs:parse-pathname** finds a syntax error in the string it is given.

**fs:parse-pathname** sets up a nonlocal proceed type **:new-pathname** for this condition. The proceed type expects one argument, a pathname, which is returned from **fs:parse-pathname**.

### **fs:merge-pathname-defaults** *pathname* &optional *defaults default-type default-version*

Fills in unspecified components of *pathname* from the defaults and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, a symbol, or a Maclisp namelist. The returned value will always be a pathname. The merging rules are documented on page 462.

If *defaults* is a pathname, rather than a defaults alist, then the defaults are taken from its components. This is how you merge two pathnames. (In Maclisp that operation is called **mergef**.)

*defaults* defaults to the value of **fs:\*default-pathname-defaults\*** if unsupplied. *default-type* defaults to the value of **fs:\*name-specified-default-type\***. *default-version* defaults to **:newest**.

**fs:merge-and-set-pathname-defaults** *pathname* &optional *defaults* *default-type*  
*default-version*

This is the same as `fs:merge-pathname-defaults` except that after it is done the defaults-list *defaults* is modified so that the merged pathname is the new default. This is handy for programs that have "sticky" defaults. (If *defaults* is a pathname rather than a defaults alist, then no storing back is done.) The optional arguments default the same way as in `fs:merge-pathname-defaults`.

This function yields a pathname given its components.

**fs:make-pathname** &rest *options*

The *options* are alternating keywords and values, which specify the components of the pathname. Missing components default to nil, except the host (all pathnames must have a host). The `:defaults` option specifies what defaults to get the host from if none is specified. The other options allowed are `:host`, `:device`, `:structured-device`, `:directory`, `:structured-directory`, `:name`, `:structured-name`, `:type`, and `:version`.

These functions return useful information.

**fs:user-homedir** &optional *host* *reset-p*

Returns the pathname of the logged-in user's home directory on *host*, which defaults to the host the user logged in to. Home directory is a somewhat system-dependent concept, but from the point of view of the Lisp Machine it is the directory where the user keeps personal files such as init files and mail. This function returns a pathname without any name, type, or version component (those components are all nil). If *reset-p* is specified non-nil, the machine the user is logged in to is changed to be *host*.

**fs:init-file-pathname** *program-name* &optional *host*

Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the `login` function.

These functions are useful for poking around.

**fs:describe-pathname** *pathname*

If *pathname* is a pathname object, this describes it, showing you its properties (if any) and information about files with that name that have been loaded into the machine. If *pathname* is a string, this describes all interned pathnames that match that string, ignoring components not specified in the string. One thing this is useful for is finding the directory of a file whose name you remember. Giving `describe` (see page 641) a pathname object will do the same thing as this function will.

**fs:pathname-plist** *pathname*

Parses and defaults *pathname*, then returns the list of properties of that pathname.

**fs:\*pathname-hash-table\****Variable*

This is the hash table in which pathname objects are interned. Applying the function `maphash` to this will extract all the pathnames in the world.

## 22.5 Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the `--` line (see section 21.9.2, page 438) of a source file in the family. Several types of files with that name, in that directory, belong together. They are different members of the same family; for example, they may be source code and compiled code. However, there may be several other types of files that form a logically distinct group even though they have this same name; `TEXT` and `PRESS` for example. The exact mapping is done on a per host basis since it can sometimes be affected by host naming conventions.

The generic pathname of pathname *p* usually has the same host, device, directory, and name as *p* does. However, it has a version of `:unspecific`. The type of the generic pathname is obtained by sending a `:generic-base-type type-of-p` message to the host of *p*. The default response to this message is to return the associated type from `fs:*generic-base-type-alist*` if there is one, else `type-of-p`. Both the argument and the value are either strings, in interchange form, or canonical type symbols.

However, the ITS file system presents special problems. One cannot distinguish multiple generic base types in this same way since the type component does not exist as such; it is derived from the second filename, which unfortunately is also sometimes used as a version number. Thus, on ITS, the type of a generic pathname is always `:unspecific` if there is any association for the type of the pathname on `fs:*generic-base-type-alist*`.

Since generic pathnames are primarily useful for storing properties, it is important that they be as standardized and conceptualized as possible. For this reason, generic pathnames are defined to be "backtranslated", i.e. the generic pathname of a pathname that is (or could be) the result of a logical host translation has the host and directory of the logical pathname. For example, the generic pathname of `AI:LMWIN;STREAM>` would be `SYS:WINDOW;STREAM U U` if `AI` is the system host.

All version numbers of a particular pathname share the same identical generic pathname. If the values of particular properties have changed between versions, it is possible for confusion to result. One way to deal with this problem is to have the property be a list associating version number with the actual desired property. Then it is relatively easy to determine which versions have which values for the property in question and select one appropriately. But in the applications for which generic pathnames are typically used, this is not necessary.

The `:generic-pathname` operation on a pathname returns its corresponding generic pathname. See page 469. The `:source-pathname` operation on a pathname returns the actual or probable pathname of the corresponding source file (with `:newest` as the version). See page 469.

**fs:\*generic-base-type-alist\****Variable*

This is an association list of the file types and the type of the generic pathname used for the group of which that file type is a part. Constructing a generic pathname will replace the file type with the association from this list, if there is one (except that ITS hosts always replace with `:unspecific`). File types not in this list are really part of the name in some sense. The initial list is

```
( (:text . :text) ("DOC" . :text)
  (:press . :text) ("XGP" . :text)
  (:lisp . :unspecific) (:qfasl . :unspecific)
  (NIL . :unspecific))
```

The association of `:lisp` and `:unspecific` is unfortunately made necessary by the problems of ITS mentioned previously. This way makes the generic pathnames of logically mapped LISP files identical no matter whether the logical host is mapped to an ITS host or not.

The first entry in the list with a particular cdr is the entry for the type that source files have. Note how the first element whose cdr is `:unspecific` is the one for `:lisp`. This is how the `:source-pathname` operation knows what to do, by default.

Some users may need to add to this list.

The system records certain properties on generic pathnames automatically.

- `:warnings` This property is used to record compilation and other warnings for the file.
- `:definitions` This property records all the functions and other things defined in the file. The value has one element for each package into which the file has been loaded; the element's car is the package itself and the cdr is a list of definitions made.

Each definition is a cons whose car is the symbol or function spec defined and whose cdr is the type of definition (usually one of the symbols `defun`, `defvar`, `defflawor` and `defstruct`).

- `:systems` This property's value is a list of the names of all the systems (defined with `defsystem`, see page 520) of which this is a source file.

**:file-id-package-alist**

This property records what version of the file was most recently loaded. In case the file has been loaded into more than one package, as is sometimes necessary, the loaded version is remembered for each package separately. This is how `make-system` tells whether a file needs to be reloaded. The value is a list with an element for each package that the file has been loaded into; the elements look like

```
(package file-information)
```

*package* is the package object itself; *file-information* is the value returned by the `:info` operation on a file stream, and is usually a cons whose car is the truename (a pathname) and whose cdr is the file creation date (a universal time number).

Some additional properties are put on the generic pathname by reading the attribute list of the file (see page 440). It is not completely clear that this is the right place to store these properties, so it may change in the future. Any property name can appear in the attributes list and get onto the generic pathname; the standard ones are described in section 21.9.2, page 438.



## 22.6 Pathname Operations

This section documents the operations a user may send to a pathname object. Pathnames handle some additional operations that are only intended to be sent by the file system itself, and therefore are not documented here. Someone who wants to add a new host to the system would need to understand those internal operations. This section also does not document operations that are peculiar to pathnames of a particular host; those would be documented under that host.

### **:generic-pathname**

*Operation on fs:pathname*

Returns the generic pathname for the family of files of which this pathname is a member. See section 22.5, page 467 for documentation on generic pathnames.

### **:source-pathname**

*Operation on fs:pathname*

Returns the pathname for the source file in the family of files to which this pathname belongs. The returned pathname will have `:newest` as its version. If the file has been loaded in some fashion into the Lisp environment, then the pathname type is that which the user actually used. Otherwise, the conventional file type for source files is determined from the generic pathname.

### **:primary-device**

*Operation on fs:pathname*

Returns the default device name for the pathname's host. This is used in generating the initial default pathname for a host.

### **:wild-p**

*Operation on fs:pathname*

Returns `t` if this pathname contains any sort of wildcards.

Operations to get a path name string out of a pathname object:

### **:string-for-printing**

*Operation on fs:pathname*

Returns a string that is the printed representation of the path name. This is the same as what you get if you `princ` the pathname or take `string` of it.

### **:string-for-wholine *length***

*Operation on fs:pathname*

Returns a string like the `:string-for-printing`, but designed to fit in *length* characters. *length* is a suggestion; the actual returned string may be shorter or longer than that. However, the who-line updater will truncate the value to that length if it is longer.

### **:string-for-editor**

*Operation on fs:pathname*

Returns a string that is the pathname with its components rearranged so that the name is first. The editor uses this form to name its buffers.

### **:string-for-dired**

*Operation on fs:pathname*

Returns a string to be used by the directory editor. The string contains only the name, type, and version.

**:string-for-directory***Operation on fs:pathname*

Returns a string that contains only the device and directory of the pathname. It identifies one directory among all directories on the host.

**:string-for-host***Operation on fs:pathname*

Returns a string that is the pathname the way the host file system likes to see it.

Operations to move around through a hierarchy of directories:

**:pathname-as-directory***Operation on fs:pathname*

Assuming that the file described by the pathname is a directory, return another pathname specifying that *as* a directory. Thus, if sent to a pathname OZ:<RMS>FOO.DIRECTORY, it would return the pathname OZ:<RMS.FOO>. The name, type and version of the returned pathname are *:unspecific*.

**:directory-pathname-as-file***Operation on fs:pathname*

This is the inverse of the preceding operation. It returns a pathname specifying *as* a file the directory of the original pathname. The name, type and version of the original pathname are ignored.

The special symbol *:root* can be used as the directory component of a pathname on file systems that have a root directory.

Operations to manipulate the property list of a pathname:

**:get** *indicator**Operation on fs:pathname***:get1** *list-of-indicators**Operation on fs:pathname***:putprop** *value indicator**Operation on fs:pathname***:remprop** *indicator**Operation on fs:pathname***:plist***Operation on fs:pathname*

These manipulate the pathname's property list, and are used if you call the property list functions of the same names (see page 82) giving the pathname as the first argument. Please read the paragraph on page 454 explaining some care you must take in using property lists of pathnames.

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 fs:pathname*

Return 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* will have a number as the version.

**:open** *pathname &rest options* *Operation on fs:pathname*  
 This operation 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* will be that logical pathname, but `self` will be 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 fs:pathname*

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

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 fs:pathname*

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

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

Rename 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 fs:pathname*

Attempt to complete the filename *string*, returning the results. This operation is used by the function `fs:complete-pathname` (see page 446). 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 fs:pathname*

This operation 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 fs:pathname*

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

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

Map *function* over all the files specified by this pathname (which may contain wildcards). Each time *function* is called, its first argument will be 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* will be 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 &optional (*error t*) *Operation on fs:pathname*

Expunge 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 &optional (*error t*) *Operation on fs:pathname*

Creates the directory specified in this pathname.

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

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

## 22.7 Host File Systems Supported

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncracies. More host types will no doubt be added in the future.

## 22.7.1 ITS

An ITS pathname looks like "*host: device: dir; name type-or-version*". The primary device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: may be used.

ITS does not exactly fit the virtual file system model, in that a file name has two components (FN1 and FN2) rather than three (name, type, and version). Consequently to map any virtual pathname into an ITS filename, it is necessary to choose whether the FN2 will be the type or the version. The rule is that usually the type goes in the FN2 and the version is ignored; however, certain types (LISP and TEXT) are ignored and instead the version goes in the FN2. Also if the type is :unspecific the FN2 is the version.

Given an ITS filename, it is converted into a pathname by making the FN2 the version if it is "<", ">", or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols :oldest and :newest, which correspond to "<" and ">" respectively.

In every ITS pathname either the version or the type is :unspecific or nil; sometimes both are. When you create a new ITS pathname, if you specify only the version or only the type, the one not specified becomes :unspecific. If both are specified, the version is :unspecific unless the type is a normally-ignored type (such as LISP) in which case the version is :newest and the type is :unspecific so that numeric FN2's are found.

Each component of an ITS pathname is mapped to upper case and truncated to six characters.

Special characters (space, colon, and semicolon) in a component of an ITS pathname can be quoted by prefixing them with right horseshoe (⤵) or equivalence sign (≐). Right horseshoe is the same character code in the Lisp Machine character set as control-Q in the ITS character set.

An ITS pathname can have a structured name, which is a list of two strings, the FN1 and the FN2. In this case there is neither a type nor a version.

An ITS pathname with an FN2 but no FN1 (i.e. a type and/or version but no name) is represented with the placeholder FN1 "↔", because ITS pathname syntax provides no way to write an FN2 without an FN1 before it.

The ITS init file naming convention is "*homedir; user program*".

### **fs:\*its-uninteresting-types\***

*Variable*

The ITS file system does not have separate file types and version numbers; both components are stored in the "FN2". This variable is a list of the file types that are "not important"; files with these types use the FN2 for a version number. Files with other types use the FN2 for the type and do not have a version number. The initial list is

```
("LISP" "TEXT" nil :unspecific)
```

Some users may need to add to this list.

**:fn1** *Operation on its-pathname*  
**:fn2** *Operation on its-pathname*

These two operations return a string that is the FN1 or FN2 host-dependent component of the pathname.

**:type-and-version** *Operation on fs:pathname*  
**:new-type-and-version** *new-type new-version* *Operation on fs:pathname*

These two operations provide a way of pretending that ITS pathnames can have both a type and a version. It uses the first three characters of the FN2 to store a type and the last three to store a version number.

On an ITS-pathname, **:type-and-version** returns the type and version thus extracted (not the same as the type and version of the pathname). **:new-type-and-version** returns a new pathname constructed from the specified new type and new version.

On any other type of pathname, these operations simply return or set both the type component and the version component.

### 22.7.2 TOPS-20 (Twenex), Tenex, and VMS.

A pathname on TOPS-20 (better known as Twenex) looks like "*host:device:<directory>name.type.version*". The primary device is **PS:**.

TOPS-20 pathnames are mapped to upper case. Special characters (including lower-case letters) are quoted with the circle-X (⊗) character, which has the same character code in the Lisp Machine character set as Control-V in the ASCII character set.

If you specify a period after the name, but nothing after that, then the type is **:unspecific**, which translates into an empty extension on the TOPS-20 system. If you omit the period, you have allowed the type to be defaulted.

TOPS-20 pathnames allow the special version symbols **:oldest** and **:newest**. In the string form of a pathname, these are expressed as **.-2**, and as an omitted version.

The directory component of a TOPS-20 pathname may be structured. The directory **<FOO.BAR>** is represented as the list ("**FOO**" "**BAR**").

The characters **\*** and **%** are wildcards that match any sequence of characters and any single character (within one pathname component), respectively. To specify a filename that actually contains a **\*** or **%** character, quote the character with ⊗. When a component is specified with just a single **\***, the symbol **:wild** appears in the pathname object. When wildcards are embedded in pathname components, the pathname object represents them by using the character **#\break** to stand for **\*** and **#\quote** to stand for **%**. If a **\*** or **%** character appears in a component of a pathname object, it stands for a quoted character, since the convention is that quote characters are *not* present in the components stored in the pathname object.

The TOPS-20 init file naming convention is "**<user>program.INIT**".

When there is an attempt to display a TOPS-20 file name in the who-line and there isn't enough room to show the entire name, the name is truncated and followed by a center-dot character to indicate that there is more to the name than can be displayed.

Tenex pathnames are almost the same as TOPS-20 pathnames, except that the version is preceded by a semi-colon instead of a period, the default device is DSK instead of PS, and the quoting requirements are slightly different.

VMS pathnames are basically like TOPS-20 pathnames, with a few complexities. The primary device is SYS\$SYSDISK.

First of all, only alphanumeric characters are allowed in filenames (though \$ and underscore can appear in device names).

Secondly, a version number is preceded by ";" rather than by ".".

Thirdly, file types ("extensions") are limited to three characters. Each of the system's canonical types has a special mapping for VMS pathnames, which is three characters long:

:lisp → LSP	:text → TXT	:qfasl → QFS	:midas → MID
:press → PRS	:widths → WID	:patch-directory → PDR	
:qwabl → QWB	:babyl → BAB	:mail → MAI	:xmail → XML
:init → INI	:unfasl → UNF	:output → OUT	

### 22.7.3 Unix pathnames

A Unix pathname is a sequence of directory or file names separated by slashes. The last name is the filename; preceding ones are directory names (but directories are files anyway). There are no devices or versions. Alphabetic case is significant in Unix pathnames, no case conversion is normally done, and lower case is the default. Therefore, components of solid upper or lower case are inverted in case when going between interchange form and raw form. (What the user types in a pathname string is the raw form.)

Unix allows you to specify a pathname relative to your default directory by using just a filename, or starting with the first subdirectory name; you can specify it starting from the root directory by starting with a slash. In addition, you can start with ".." as a directory name one or more times, to refer upward in the hierarchy from the default directory.

Unix pathnames on the Lisp Machine provide all these features too, but the canonicalization to a simple descending list of directory names starting from the root is done on the Lisp Machine itself when you merge the specified pathname with the defaults.

If a pathname string starts with a slash, the pathname object that results from parsing it is called "absolute". Otherwise the pathname object is called "relative".

In an absolute pathname object, the directory component is either a symbol (nil, :unspecific or :root), a string, or a list of strings. A single string is used when there is only one level of directory in the pathname.

A relative pathname has a directory that is a list of the symbol `:relative` followed by some strings. When the pathname is merged with defaults, the strings in the list are appended to the strings in the default directory. The result of merging is always an absolute pathname.

In a relative pathname's string form, the string `".."` can be used as a directory name. It is translated to the symbol `:up` when the string is parsed. That symbol is processed when the relative pathname is merged with the defaults.

Restrictions on the length of Unix pathnames require abbreviations for the standard Zetalisp pathname types, just as for VMS. On Unix the preferred mappings of all canonical types are one or two characters long. We give here the mappings in raw form; they are actually specified in interchange form.

```
:lisp → l      :text → tx      :qfasl → qf      :midas → md
:press → pr    :widths → wd    :patch-directory → pd
:qwabl → qw    :babyl → bb    :mail → ma      :xmail → xm
:init → in     :unfasl → uf   :output → ot
```

The Multics file system is much like the Unix one; there are absolute and relative pathnames, absolute ones start with a directory delimiter, and there are no devices or versions. Alphabetic case is significant.

There are differences in details. Directory names are terminated, and absolute pathnames begun, with the character `">"`. The containing directory is referred to by the character `"<"`, which is complete in itself. It does not require a delimiter. Thus, `<<FOO>BAR` refers to subdirectory FOO, file BAR in the superdirectory of the superdirectory of the default directory.

The limits on filename sizes are very large, so the system canonical types all use their standard mappings. Since the mappings are specified as upper case, and then interpreted as being in interchange form, the actual file names on Multics will contain lower case.

## 22.7.4 Lisp Machine File Systems

There are two file systems that run in the MIT Lisp Machine system. They have different pathname syntax. Both can be accessed either remotely like any other file server, or locally.

The Local-File system uses host name LM for the machine you are on. A Local-File system on another machine can be accessed using the name of that machine as a host name, provided that machine is known as a file server.

The remainder of the pathname for the Local-File system looks like `"directory; name.type # version"`. There is no restriction on the length of names; letters are converted to upper case. Subdirectories are allowed and are specified by putting periods between the directory components, as in `RMS.SUBDIR;`

The TOPS-20 pathname syntax is also accepted. In addition, if the flag `fs:*lmfs-use-twenex-syntax*` is non-nil, Local-File pathnames will print out using TOPS-20 syntax. Note that since the printed representation of a pathname is cached, changing this flag's value will not change the printing of pathnames with existing representations.



The Local-File system on the filecomputer at MIT has the host name FS.

The LMFILE system is primarily for use as a file server, unless you have 512k of memory. At MIT it runs on the filecomputer and is accessed remotely with host name FC.

The remainder of an LMFILE pathname looks like "*directory; name type # version*". However, the directory and name can be composed of any number of subnames, separated by backslashes. This is how subdirectories are specified. FOO;BAR\X refers to the same file as FOO\BAR;X, but the two ways of specifying the file have different consequences in defaulting, getting directory listings, etc.

Case is significant in LMFILE pathnames; however, when you open a file, the LMFILE system ignores the case when it matches your pathname against the existing files. As a result, the case you use matters when you create or rename a file, and appears in directory listings, but it is ignored when you refer to an existing file, and you cannot have two files whose names differ only in case. When components are accessed in interchange form, they are always converted to upper case.

### 22.7.5 Logical Pathnames

There is another kind of pathname that doesn't correspond to any particular file server. It is called a "logical" pathname, and its host is called a "logical" host. Every logical pathname can be translated into a corresponding "physical" pathname; there is a mapping from logical hosts into physical hosts used to effect this translation.

The reason for having logical pathnames is to make it easy to keep bodies of software on more than one file system. An important example is the body of software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites may store the sources on an ITS file system, while others may store them on a TOPS-20. However, there is software that wants to deal with the pathnames of these files in such a way that the software will work correctly no matter which site it is run at. The way this is accomplished is that there is a logical host called SYS; all pathnames for system software files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translation will work differently at one site than at another. At a site where the sources are stored on a certain TOPS-20, for example, pathnames of the SYS host will be translated into pathnames for that TOPS-20.

Here is how translation is done. For each logical host, there is a mapping that takes the name of a directory on the logical host and produces a device and a directory for the corresponding physical host. To translate a logical pathname, the system finds the mapping for that pathname's host and looks up that pathname's directory in the mapping. If the directory is found, a new pathname is created whose host is the physical host, and whose device and directory come from the mapping. The other components of the new pathname are left the same. There is also, for each logical host, a "default device" (it is an instance variable of the host object). The "default device" is used if a translation exists but specifies nil as the device. If there is no translation for the specified directory, an error is signaled.

This means that when you invent a new logical host for a certain set of files, you also make up a set of logical directory names, one for each of the directories that the set of files is stored in. Now when you create the mappings at particular sites, you can choose any physical host for the files to reside on, and for each of your logical directory names, you can specify the actual directory name to use on the physical host. This gives you flexibility in setting up your directory names; if you used a logical directory name called FRED and you want to move your set of files to a new file server that already has a directory called FRED, being used by someone else, you can translate FRED to some other name and so avoid getting in the way of the existing directory. Furthermore, you can set up your directories on each host to conform to the local naming conventions of that host.

**fs:add-logical-pathname-host** *logical-host physical-host translations*

This creates a new logical host named *logical-host*. Its corresponding "physical" host (that is, the host to which it will forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications. Each translation specification should be a list of two strings. The first is the name of a directory on the logical host. The second is a pathname whose device component and directory component are the translation of that directory. A translation for logical directory nil specifies the default device for the logical host; if there is none, the primary device of the physical host is used. Example:

```
(fs:add-logical-pathname-host "MUSIC" "MUSIC-10-A"
  '(("MELODY" "SS:<MELODY>")
    ("DOC" "PS:<MUSIC-DOCUMENTATION>")))
```

This creates a new logical host called MUSIC. If you try to read the file MUSIC:DOC;MANUAL TEXT 2, you will be re-directed to the file MUSIC-10-A:PS:<MUSIC-DOCUMENTATION>MANUAL.TEXT.2 (assuming that the host MUSIC-10-A is a TOPS-20 system).

**:translated-pathname**

*Operation on fs:logical-pathname*

This converts a logical pathname to a physical pathname. It returns the translated pathname of this instance, a pathname whose :host component is the physical host that corresponds to this instance's logical host.

If this operation is applied to a physical pathname, it simply returns itself.

**:back-translated-pathname** *pathname*

*Operation on fs:logical-pathname*

This converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*.

Here is an example of how this would be used in connection with truenames. Given a stream that was obtained by opening a logical pathname,

```
(funcall stream ':pathname)
```

returns the logical pathname that was opened.

```
(funcall stream ':truenam)
```

returns the true name of the file that is open, which of course is a pathname on the physical host. To get this in the form of a logical pathname, one would do

```
(funcall (funcall stream ':pathname)
         ':back-translated-pathname
         (funcall stream ':truename))
```

If this operation is applied to a physical pathname, it simply returns its argument. Thus the above example will work no matter what kind of pathname was opened to create the stream.

**fs:unknown-logical-directory** (fs:pathname-error error)

*Condition*

This is signaled when a logical pathname's directory has no translation. The condition instance supports the `:logical-directory` operation, which returns the directory that was specified.

Two proceed types are supported, `:new-directory` and `:new-translation`. Each expects a single argument, a pathname or a string to be parsed into one. In either case the device and directory of the pathname are used for translation. `:new-directory` specifies the translation for this time only. `:new-translation` records a translation permanently for the logical directory that was used.

A logical pathname looks like "*host: directory; name type version*". There is no way to specify a device; parsing a logical pathname always returns a pathname whose device component is `:unspecific`. This is because devices don't have any meaning in logical pathnames.

The equivalence-sign character ( $\equiv$ ) can be used for quoting special characters such as spaces and semicolons. The double-arrow character ( $\Leftrightarrow$ ) can be used as a place-holder for components that are nil, and the up-horseshoe ( $\cup$ ) indicates `:unspecific` (generic pathnames typically have `:unspecific` as the type and the version). All letters are mapped to upper case unless quoted. The `:newest`, `:oldest`, and `:wild` values for versions are specified with the strings ">", "<", and "\*" respectively.

There isn't any init file naming convention for logical hosts; you can't log into them. The `:string-for-host`, `:string-for-wholine`, `:string-for-dired`, and `:string-for-editor` messages are all passed on to the translated pathname, but the `:string-for-printing` is handled by the `fs:logical-pathname` flavor itself and shows the logical name.

## 22.7.6 Editor Buffer Pathnames

The hosts `ED`, `ED-BUFFER` and `ED-FILE` are used in pathnames which refer to buffers in the editor. If you open such a pathname, you get a stream that reads or writes the contents of an editor buffer. The three host names differ only in the syntax of the pathname, and in how it is interpreted.

The host `ED` is followed by an abbreviation that should complete to the name of an existing editor buffer. For example, the pathname `ED:FOO` could refer to the buffer `FOO.LISP PS:<ME> OZ:.`

The host `ED-BUFFER` is followed by an exact buffer name. If there is no buffer with that name, one is created. This is most useful for creating a buffer.

The host ED-FILE is followed by an arbitrary pathname, including a host name. An ED-FILE pathname refers to a buffer visiting that file. If necessary, the file is read into the editor. For example, ED-FILE: OZ: PS:<ME>FOO.LISP would refer to the same buffer as ED: FOO. The current default defaults are used in processing the pathname that follows ED-FILE, when the pathname is parsed.

## 22.8 Hosts

Each host known to the Lisp Machine is represented by a flavor instance known as a host object. The host object records such things as the name(s) of the host, its operating system type, and its network address(es).

Not all hosts support file access. Those that do support it appear on the list `fs:*pathname-host-list*` and can be the host component of pathnames. A host object is also used as an argument when you make a chaosnet connection for any purpose.

The hosts that you can use for making network connections appear in the value of `si:host-alist`. Most of the hosts you can use for pathnames are among these; but some, such as logical hosts, are not.

### 22.8.1 Parsing Hostnames

**si:parse-host** *namestring* &optional *no-error-p* *unknown-ok*

Returns a host object that recognizes the specified name. If the name is not recognized, it is an error, unless *no-error-p* is non-`nil`; in that case, `nil` is returned.

If *unknown-ok* is non-`nil`, an unrecognized string is used to construct a new host object. However, that host object will not have a known operating system type or network addresses.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

**sys:unknown-host-name** (sys:local-network-error  
sys:network-error error)

*Condition*

This condition is signaled by `si:parse-host` when the host is not recognized, if that is an error.

The `:name` operation on the condition instance returns the string given to `si:parse-host`.

**si:get-host-from-address** *address* *network*

Returns a host object given an address and the name of the network which that address is for. Usually the symbol `:chaos` is used as the network name.

`nil` is returned if there is no known host with that address.

**fs:get-pathname-host** *name* &optional *no-error-p*

Returns a host object that can be used in pathnames. If the name is not recognized, it is an error, unless *no-error-p* is non-*nil*; in that case, *nil* is returned.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

**fs:unknown-pathname-host** (fs:pathname-error error) *Condition*

This condition is signaled by **fs:get-pathname-host** when the host is not recognized, if that is an error.

The *:name* operation on the condition instance returns the string given to **fs:get-pathname-host**.

**si:parse-host** and **fs:get-pathname-host** differ in the set of hosts searched.

**fs:\*pathname-host-list\*** *Variable*

This is a list of all the host objects that support file access.

**si:host-alist** *Variable*

This variable is a list of one element for each known network host. The element looks like this:

```
(full-name host-object (nickname nickname2 ... full-name)
 system-type machine-type site
 network list-of-addresses network2 list-of-addresses2 ...)
```

The *full name* is the host's official name. The *:name* operation on the host object returns this.

The *host-object* is a flavor instance that represents this host. It may be *nil* if none has been created yet; **si:parse-host** creates them when they are referred to.

The *nicknames* are alternate names that **si:parse-host** will recognize for this host, but which are not its official name.

The *system-type* is a symbol that tells what software the host runs. This is used to decide what flavor of host object to construct. Symbols now used include *:lisp*, *:its*, *:tops-20*, *:tenex*, *:vms*, *:unix*, *:multics*, *:minits*, *:waits*, *:chaos-gateway*, *:dos*, *:rsx*, *:magicsix*, and others. Not all of these are specifically understood in any way by the Lisp machine. If none of these applies to a host you wish to add, use a new symbol.

The *machine-type* is a symbol that describes the hardware of the host. Symbols in use include *:lisp*, *:pdp10*, *:pdp11*, *:vax*, *:pe3230*. (*nil*) has also been observed to appear here. Note that these machine types attempt to have wide meanings, lumping together as various brands, models, etc.

The *site* does not describe anything about the host. Instead it serves to say what the Lisp Machine's site name was when the host was defined. This is so that, when a Lisp Machine system is moved to a different institution that has a disjoint set of hosts, all the old site's hosts can be deleted from the host alist by site reinitialization.

The *networks* and lists of addresses describe how to reach the host. Usually there will be only one network and only one address in the list. The generality is so that hosts with multiple addresses on multiple networks can be recorded. Networks include `:chaos` and `:arpa`. The address is meaningful only to code for a specific network.

## 22.8.2 Host Object Operations

- :name** *Operation on host objects*  
Returns the full, official name of the host.
- :name-as-file-computer** *Operation on host objects*  
Returns the name to print in pathnames on this host (assuming it supports files). This is likely to be a short nickname of the host.
- :short-name** *Operation on host objects*  
Returns the shortest known nickname for this host.
- :pathname-host-namep** *string* *Operation on host objects*  
Returns `t` if *string* is recognized as a name for this host for purposes of pathname parsing.
- :system-type** *Operation on host objects*  
Returns the operating system type symbol for this host. See page 657.
- :network-type** *Operation on host objects*  
Returns the symbol for one network that this host is connected to, or `nil` if it is not connected to any. `:chaos` is preferred if it is one of the possible values.
- :network-typep** *network* *Operation on host objects*  
Returns `t` if the host is connected to the specified network.
- :sample-pathname** *Operation on host objects*  
Returns a pathname for this host, whose device, directory, name, type and version components are all `nil`. Sample pathnames are often useful because many file-system-dependent pathname operations depend only on the pathname's host.
- :open-streams** *Operation on host objects*  
Returns a list of all the open file streams for files on this host.
- :close-all-files** *Operation on host objects*  
Closes all file streams open for files on this host.
- :generic-base-type** *type-component* *Operation on host objects*  
Returns the type component for a generic pathname assuming it is being made from a pathname whose type component is the one specified.

## 22.9 Maclisp Conversion

This section briefly discusses how to convert from Maclisp I/O and filename functions to the corresponding but often more general Lisp Machine ones.

The functions `load`, `open`, `probef`, `renamef`, and `deletef` are upward compatible. Most of them take optional additional arguments to do additional things, usually connected with error handling. Where Maclisp wants to see a file name in the form of a symbol or a list, the Lisp Machine will accept those or a string or a pathname object. `probef` returns a pathname or `nil` rather than a namelist or `nil`.

`load` keeps defaults, which it updates from the file name it is given.

The old-I/O functions `uread`, `crunit`, etc. do not exist in the Lisp Machine. `fasload` exists but is a function rather than a special form.

There is a special form, `with-open-file`, which should replace most calls to `open`. See page 431.

The functions for manipulating file names themselves are different. The system will accept a namelist as a pathname, but will never create a namelist. `mergef` is replaced by `fs:merge-pathname-defaults`. `defaultf` is replaced by `fs:default-pathname` or `fs:set-default-pathname`, depending on whether it is given an argument. `namestring` is replaced by the `:string-for-printing` message to a pathname, or the `string` function. `namelist` is approximately replaced by `fs:parse-pathname`. `(status udir)` and `(status homedir)` are approximately replaced by `fs:user-nomedir`. The `truename` function is replaced by the `:truename` stream operation, which returns a pathname containing the actual name of the file open on that stream. The `directory` and `allfiles` functions are replaced by `fs:directory-list`.