Symbolics File System

August 1981

Table of Contents

Table of Figures

1. Symbolics File System

1.1 Introduction

The Symbolics File System provides a file system that runs on a Lisp Machine and stores information permanently and reliably on the Lisp Machine's disk. It can be used locally (from that Lisp Machine itself) or remotely (over the network). This document describes the file system, its abilities and conventions, how to use it from programs, and how the associated File System Maintainance program works.

Some of this document is directed toward users of the file system; other parts are directed only to system maintainance personnel who will be responsible for performing dumping, reloading, salvaging, and configuring. It does not describe the internal program logic or organizational details of the file system. It assumes that the reader is familiar with the way that pathnames work on the Lisp Machine (see chapter 22 of *The Lisp Machine Manual*), and the general model of the way the Lisp Machine deals with all file systems; it is important for you to understand these concepts if you want to fully understand the rest of this document.

1.2 Concepts

There are *files*. Files contain data. There are character files and binary files. Character files consist of a certain number (the *byte count*) of characters in the Lisp Machine character set. Binary files consist of a number (their byte count) of binary data bytes, which are unsigned binary numbers up to sixteen bits in length.

A file has a *name*, a *type*, and a *version*. The name is a character string of any length. The type is a character string of up to fourteen characters in length, and the version a positive integer up to 16777215. Names and types may consist of upper and lower case characters. However, searching for file names is not sensitive to case. This means that if you create a file whose name is "MyFile", the file will have that name and appear that way in directory listings, but if you ask for "myfile" or "MYFILE" or "MYfIle", the file will be found. The characters ">" and "return" may not appear in names and types.

The name is an arbitrary user-chosen string describing the file. The type is supposed to indicate what type of data the file contains; a type of "lisp" is the system convention for files containing Lisp source programs, "qfasl" for compiled Lisp programs, and so forth. The version number distinguishes successive generations of a file; to change a file; you normally read the latest version of the file into the Lisp Machine, modify it, and write out a new version with a the next highest version number.

Files reside in *directories*. The combination of name, type, and version of any file is unique in the directory in which it is contained. With the exception of a single directory (the *root*), directories also reside in other directories. The directory in which a file or directory resides is called its *parent* directory, and these files and directories are said to be *inferior* to their parent directory. Directories and files thus form a strict tree (hierarchy); the root directory is the root of this tree. Directories have neither type nor version. Thus, the name of a directory alone identifies it in its containing (*superior*) directory.

Links are the third (and last) kind of object that can live in a directory. A link contains the character-string representation of the pathname of something else in the file system, called the *target* of the link. This pathname specifies a directory, a name and a type, and it may or may not specify a version. A link is an indirect pointer to something else; when certain operations are done on a link, the operation really gets done to the target instead of the link itself.

The file system also stores and maintains *properties* of objects. For example, for each file it stores the creation date, the author, whether the file has been backed up, and so on. Users can also create their own properties; each file has a property list that lets you store arbitrary associated information with the file. See section 1.4, page 3 for more details.

The *File System Maintainance* program is an interactive subsystem that lets you manipulate the file system. Some of its commands are of general interest; others are useful only to file system maintainers and should not be used by non-experts. The File System Editor is part of this program; you use the File System Editor to do basic file system operations. You can invoke the program by typing `System F`. It is described in more detail on section 1.10, page 17.

Before you use the file system on a machine, you must log into that machine (using the `login` function; see *The Lisp Machine Manual*). If you are using the file system locally, it is desirable to log out of the machine before you cold-boot it or otherwise abandon it. This is especially desirable if you have created files on the file system or expunged directories (see below) while using it. If you do not observe this recommendation, you will run out free disk records at the rate of about 30 to 50 records per cold boot (in which files were created), and the free record salvager will have to be run (see section 1.10.3, page 21).

1.3 Pathnames

You can access the file system *locally* or *remotely*. Local access means that your program is running on the same Lisp Machine as the file system, and the files are located on your own disk. Remote access means that the file system is on some other machine, and you are connected to that machine by the Chaosnet. You choose which kind of access to use by specifying the appropriate host name in pathnames. The host name `local` always means the file system on your own machine. To use a remote file system, use the Chaosnet host name of the machine that has the file system as the host name in pathnames. (Host names are not case-sensitive.)

To specify a pathname for a file, you have to give the names of all of the ancestor directories, starting at the root directory, and going down to the superior directory of the file; then you give the name, type, and version. Directory names are separated by ">" characters, and the components of the pathname are separated by "." characters. For example, the pathname

```
>Smith>new-stuff>utilities.lisp.4
```

refers to a file with name `utilities`, type `lisp`, and version 4. It is contained in a directory namd `new-stuff`, which is contained in a directory called `Smith`, which is contained in the root directory.

The pathname of a directory is like the pathname of a file, except that there is no type and no version. The pathname of the directory that contains the `utilities.lisp.4` above is `>Smith>new-stuff`. The pathname of its superior directory is `>Smith`. The pathname of `>Smith`'s superior directory is the root directory, whose pathname is >.

You can leave out components of a pathname; the components that you leave out will usually be supplied from some set of defaults. When you omit a component you should also omit the "." that preceeds it, if any, but if you are omitting all three components, do not omit the trailing ">"

(or else the pathname will be the pathname of the containing directory). To explicitly specify that a component should be "wild" (i.e., match anything in a program that searches directories), use "*" as the component.

Here are some examples:

`local:>Smith>work.text.12`
: This specifies a file on the local file system with directory `>Smith`, name `work`, type `text`, and version 12.

`local:>Smith>work.text`
: This is the same except that the version number is unspecified. Some programs would default it to `:newest` to refer to the latest version of the file; others would default it to `:wild`, which would match any version of the file.

`george:>Jones>`
: This specifies a file on host `george` with all three components of the file name left unspecified. If the program were to default all components to `:wild`, this would match any file in the `>Jones` directory.

`>Jones>*.text.*`
: This specifies a file name on the default host whose name and version components are `:wild`. In a program that searches directories, this name will match any file whose type is `text`.

1.4 Properties

Files, directories, and links have various ***properties***. There are ***system*** properties, which are defined and maintained by the file system itself, and ***user*** properties, which are defined and maintained by programs and people that use the file system. Every property has a ***name***, which is a keyword symbol, and a ***value***, which is a Lisp object. The names of all of the system properties are listed below. (These properties should not be confused with the ***file property list*** (the -*- line in the beginning of the file); see ***The Lisp Machine Manual*** for information about file property lists.)

Users examine the values of properties by using the `View Properties` command in the File System Editor, and alter the values by using the `Edit Properties` command (see section 1.8, page 10). Programs access the values of properties by using the `fs:directory-list` and `fs:file-properties` functions, and alter the values by using the `fs:change-file-properties` function (see ***The Lisp Machine Manual***, section 21.10).

Some system properties apply to files, directories, and links alike; for example, all of these objects have an ***author*** and a ***creation time***. Other system properties aren't defined for all kinds of object; for example, only files have a ***length in bytes***, only directories have an ***auto-expunge interval***, and only links have a ***link-to***. The table of system properties tells you which kinds of objects each property applies to. User properties can always apply to any object.

The values of some system properties are determined by the file system and cannot be set by the user; for example, you can't set the ***length in bytes*** nor the ***byte size***. The values of other properties can be changed arbitrarily; for example, you can set the ***generation retention count*** or the *don't delete* property whenever you want to. The properties of the latter set are called *changeable* properties. The reason for the distinction is that the properties in the first group reflect facts about the file, whereas those in the second group represent the current state of user-settable options regarding the file.

There is also a third group, called the *secretly changeable* properties; these can be changed by programs (that is, by using `fs:change-file-properties`), but the `Edit Properties` command in the File System Editor doesn't show them. These properties are things that reflect facts about the file, but that the user is allowed to change anyway. For example, the *author* property is secretly changeable, so that you can change things to make it look as if someone else were the author even though the file system thinks he wasn't. This is occasionally useful.

When the `fs:change-file-properties` function is called for a changeable or secretly changeable system property, the property is changed. When it is called for a non-changeable system property, an error is signalled. When it is called for any property name that is not the name of one of the system properties (listed below), it assumes that it is the name of a user property, and the property is set or changed.

When the `fs:file-properties` function is called for a file in this file system, it returns a second value: a list of the names of all the properties of the file that are changeable (this does not include the secretly changeable ones). This function lists all of the system properties and all of the user properties for the object it is given.

The names of user properties must be symbols on the keyword package, and must not be the same as any of the system property names. The value associated with a user property must be a string. The combined length of the name of the property and its value must not not exceed 512 characters. To remove a user property from a file, you set the value of the property to `nil`. `fs:file-properties` returns all of the user properties of a file, but `fs:directory-list` does not return any of them. A user can create new user properties with the `New Property` command in the File System Editor; after they are created, users can edit them with `Edit Properties`. Programs create and change user properties by using `fs:change-file-properties`.

The following is a list of all the system properties. All of these properties are displayed by the `View Properties` command, returned by `fs:file-properties`, and returned in directory listings produced by `fs:directory-list`, with exceptions as noted. The changeable properties will be displayed by the `Edit Properties` command. The changeable properties and the secretly changeable properties will be accepted by `fs:change-file-properties`.

`:length-in-bytes` *(Files)*
: For a character file, the number of characters in the file; a binary file, the number of bytes in the file. A non-negative fixnum.

`:byte-size` *(Files)*
: For a character file, 8. For a binary file, the byte size of the file (the number of bits in each byte), a fixnum between 1 and 16, inclusive.

`:length-in-blocks` *(Files, directories, links)*
: The number of blocks occupied by this object. A *block* is 1024 32-bit words; this is the basic allocation unit of the file system. [This number may not be meaningful for directories.]

`:creation-date` *(Files, directories, links) (Secretly changeable)*
: The time at which this object was created, expressed in Universal Time (see *The Lisp Machine Manual*, chapter 30). A positive bignum.

`:reference-date` *(Files, directories)*
: The most recent time at which this object was used, expressed in Universal Time. A positive bignum. If the object has never been referenced, this property is not present.

`:modification-date` *(Files)*
: The most recent time at which this file was modified, expressed in Universal Time. This is the same as the creation date unless the file has been opened for appending. [In an early file system release, this property was accidentally inaccessible; this bug has been fixed.]

`:author` *(Files, directories, links) (Secretly changeable)*
: The name of the person who created this object. A string.

`:not-backed-up` *(Files, directories, links)*
: `t` if the file has never been copied onto a backup tape; `nil` if it has. (Backup is discussed in section 1.7, page 9.)

`:offline` *(Files, directories, links)*
: If `t`, this file is not really present in the file system, bu⁺ has been *migrated* to an *archival dump tape*. `nil` otherwise. [Migration to archival dump tapes is not yet implemented; the value of this property will always be `nil` for now.]

`:deleted` *(Files, directories, links) (Changeable)*
: If `t`, object has been marked for deletion, and will cease to exist when its superior directory is expunged. (See the discussion of deletion in section 1.5, page 6.)

`:dont-delete` *(Files, directories, links) (Changeable)*
: If `t`, do not allow this object to be deleted. The purpose of this attribute is to prevent the accidental deletion of important files. [This is not y‘ implemented.]

`:generation-retention-count` *(Files, links) (Changeable)*
: The generation retention count of the object (see the discussıon of deletion in section 1.5, page 6). `nil` or a non-negative fixnum.

`:default-generation-retention-count` *(Directories) (Changeable)*
: The default value for the `:generation-retention-count` property of new objects created in thie directory (see the discussion of deletion in section 1.5, page 6). `nil` or a non-negative fixnum.

`:directory` *(Directories)*
: `t` for directories. This property is not present for other types of objects.

`:last-expunge-time` *(Directories)*
: The most recent time that the directory was expunged (see the discussion of deletion in section 1.5, page 6), expressed in Universal Time. A positive bignum, or `0` if the directory has never been expunged.

`:auto-expunge-interval` *(Directories) (Changeable)*
: The auto-expunge interval (see the discussion of deletion in section 1.5, page 6); either `nil` or a positive fixnum expressing a time interval in units of seconds.

`:default-link-transparencies` *(Directories) (Changeable)*
: The initial value for the `:link-transparencies` attribute of links created in this directory (see the discussion of links in section 1.6, page 7). To set this property, use the `Link Transparencies` command in the File System Editor rather than `Edit Properties`.

`:link-transparencies` *(Links) (Changeable)*
The transparencies of this link (see the discussion of links in section 1.6, page 7). To set this property, use the `Edit Link Transparencies` command in the File System Editor rather than `Edit Properties`.

`:link-to` *(Link)*
The pathname of the target of this link. A string.

`:complete-dump-date` *(Files, directories, links)*
The most recent time at which this object was dumped on a complete dump tape (see the discussion of dumping in section 1.7, page 9), expressed in Universal Time. A positive bignum. If this object has never been dumped on a complete dump tape, this property is not present. This property does not appear in directory listings.

`:complete-dump-tape` *(Files, directories, links)*
The tape reel ID of the complete dump tape on which this object was most recently dumped. A string. If this object has never been dumped on a complete dump tape, this property is not present. This property does not appear in directory listings.

`:incremental-dump-date` *(Files, directories, links)*
The most recent time at which this object was dumped on an incremental dump tape, expressed in Universal Time. A positive bignum. If this object has never been dumped on an incremental dump tape, this property is not present. This property does not appear in directory listings.

`:incremental-dump-tape` *(Files, directories, links)*
The tape reel ID of the incremental dump tape on which this object was most recently dumped. A string. If this object has never been dumped on an incremental dump tape, this property is not present. This property does not appear in directory listings.

1.5 Deletion, Expunging, and Versions

When an object in the file system is deleted, it does not really cease to exist. Instead, it is marked as "deleted" and continues to reside in the directory. If you change your mind about whether the file should be deleted, you can *undelete* the file, which will bring it back. The deleted objects in a directory actually go away when the directory is *expunged*; this can happen by explicit user command or by means of the auto-expunge feature (see below). When a directory is expunged, the objects in it really disappear, and cannot be brought back (except from backup tapes; see section 1.7, page 9).

When a file is deleted, any attempts to open the file will fail as if the file did not exist. It is possible to open a deleted file by supplying the `:deleted` keyword to open, but this is rare.

Users normally delete and undelete objects with the `Delete` and `Undelete` commands in the File System editor (see section 1.8, page 10), and they expuge directories with the `Expunge` command. (The DIRED command in the Zmacs editor can also be used to delete and undelete objects.)

Programs normally delete files using the `deletef` function (see *The Lisp Machine Manual*). Whether a file is deleted or not also appears as the `:deleted` property of the file, and programs can delete or undelete files by using `fs:change-file-properties` to set this property to `t` or nil.

Directories may optionally be automatically expunged. Every directory has an :auto-expunge-interval property, whose value is a time interval. If any file system operation is done on a directory and the time since the last expunging of the directory is greater than this interval, the directory is immediately expunged. The default value for this property is nil, meaning that the directory should never be automatically expunged.

[Currently if you delete a directory and expunge its superior, the objects inferior to the deleted directory will continue to exist and will become orphans (see page 22). This will be fixed.]

The way file are normally used in the Lisp Machine is that when you write a file, you create a new version of the file. When you're editing something with Zmacs, for example, every so often you use the Save File command, and a new version is written out. After a while, you end up with a lot of versions of the same file, which clutters your directory, and uses up disk space. Zmacs has some convenient commands that make it easy for you to delete the old versions. The file system also has a feature that lets the old versions get deleted automatically.

There is a file property called the *generation retention count* that says how many generations (i.e., new versions) of a file should be kept around. Suppose the generation retention count of a file is three, and versions 12, 13, and 14 exist. If you write out a new version of the file, then version 12 will be deleted, and now versions 13, 14, and 15 will exist. Actually, version 12 is only deleted and not expunged, so you can still get it back by undeleting it. If the generation retention count is zero, that means that no automatic deletion should take place.

The above explanation is simplified. You might wonder what would have happened if versions 2, 3, and 14 existed, and what might have happened if the different versions of the file had different generation retention counts. To be more exact: each file has its own generation retention count. When you create a new version of a file and there is already some other version of the file (that is, another file in the directory with the same name and type but some other version), then the new file's generation retention count is set to the generation retention count of the highest existing version of the file. If there isn't any other version of the file, it is set from the *default generation retention count* of the directory. (When a new directory is created, its default generation retention count is zero (no automatic deletion).) So if you want to change the generation retention count of a file, you should change the count of the highest-numbered version; new versions will inherit the new value. When the new file is closed, if the generation retention count is not zero, all versions of the file with a number less than or equal to the version number of the new file minus the generation retention count will be deleted.

1.6 Links

A link is a file system object that points to some other file system object. The idea is that if there is a file called >George>Sample.lisp and you want it to appear in the >Fred directory, named New.lisp, you can create a link by that name to the file. Then if you opens >Fred>New.lisp, you will really get >George>Sample.lisp. The object to which a link points is called the *target* of the link, and can be found from the :link-to property of the link.

The above explanation is simplified. You might wonder what happens if you try to, say, rename >Fred>New.lisp: does the link get renamed, or does the target? Each link has a property called its :link-transparencies. The value of this property is a list of keyword symbols. Each symbol specifies an operation to which the link is transparent. If the link is transparent to an operation, that means that if the operation is performed, it will really happen to the target. If the link is not transparent to the operation, then the operation will happen to the link itself. Here is a list of the keywords, and the operations to which they refer:

`:read`	Opening the file for `:read`.
`:write`	Opening the file for `:append`.
`:create`	Opening the file for `:write`.
`:rename`	Renaming the file.
`:delete`	Deleting the file.

Users can create new links with the `Create Link` command in the File System Editor (see section 1.8, page 10); programs can use the `:create-link` message to pathnames (see section 1.9.2, page 17). When a new link is created, its transparencies are set from the `:default-link-transparencies` property of its superior directory. When a new directory is created, its `:default-link-transparencies` property is set to `(:read :write)`.

When changing the value of either of these properties with `fs:change-file-properties`, you pass in a list of the form

`((`*keyword value*`) (`*keyword value*`) ...)`

For each keyword whose value you want to change, you put one element into the list with the keyword and the value. The value of the transparency for the keywords that you do not specify is unchanged.

When you create a new link with the `Create Link` command, you have to specify both the name and the type component of the new link; the version will default to being the newest version, as of the time when you create the link. When you specify the target, you have to give a complete pathname with the name and the type; the version may be left unspecified. Targets of links can have unspecified versions; whenever such a link is used, the version will be treated as `:newest`.

There is a subtle point regarding "create-through" links (links transparent to `:create`): what happens when you try to create a new version of `foo.lisp` when the highest version of `foo.lisp` is a create-through link? Does a new version of `foo` get created, or does a new version get created in the directory of the target of the existing link? Here is the rule. If a pathname is opened for `:write`, which means that it is being created, and the pathname has version `:newest` or a version number that is, in fact, the newest version, and the newest version is actually a create-through link, then the link is transparent and the operation happens in the target's directory. If the target pathname has a version, then it is as if that exact pathname were opened for `:write`; if the target has no version, then it is as if the target pathname with a version of `:newest` were opened.

[This transparency scheme is so general as to be unwieldy, and it may be changed in future releases. Also, links to files and links work, but links to directories are not yet implemented.]

1.7 Backup

A file system can be damaged or destroyed in any number of ways. To guard against such a disaster, it is wise to periodically *dump* the file system; that is, write out the contents of the files, their properties, and the directory information onto magnetic tapes. If the file system is destroyed, it can then be *reloaded* from the tapes. Individual files can also be *retrieved* from tapes, in case a single file is destroyed, or just accidentally deleted (and expunged). Dump tapes can also be used to save a copy all the files for archival storage.

Usually a file system is shared between many Lisp Machines, and some central authority or person is responsible for periodically dumping the file system. So most users don't actually have to know how to perform dumps. However, it is useful to know what is going on with dumping in general; you might get involved with the process someday, especially if you need to have a file retrieved.

In a *complete dump*, all of the files, directories, and links in the file system are written out to tapes. This, obviously, saves all the information needed to reload the file system. However, it can take a long time and use a lot of tape to do a complete dump, especially if the file system is large. In order to make it practical and convenient to dump the file system at short intervals, a second kind of dump can be done, called an *incremental dump*. In an incremental dump, only those files and links that have been created or modified since the last dump (of either kind) are dumped; things that have stayed the same are not dumped. (All directories are always dumped in an incremental dump.) Now, if the file system is destroyed, the way you reload it is to first reload from the most recent complete dump, and then reload each of the incremental dump tapes made since that complete dump, in the same order as they were created in. Therefore, you don't need to retain incremental dump tapes made before the most recent complete dump was done; you can reuse those tapes for future dumps.

Since all incremental dumps done since a complete dump must be reloaded in order to restore the file system, doing a complete dump regularly makes recovery time faster. Doing complete dumps also lets you reuse incremental dump tapes, as described above. The more incremental dump tapes you must load at recovery time, the longer it will take to recover, and thus the more chance there is that something will go wrong. Thus, it is advantageous to take complete dumps regularly. On the other hand, complete dumps take a long time, and consume a large amount of tape at once, and are thus a substantially larger chore to perform. The tradeoff is yours: how often you take complete dumps should depend on how much tape you have, how heavily you use your file system, how many files are on it, and how often you create new files.

Dumping, reloading, and retrieving is done with the File System Maintainance program (see section 1.10.2, page 19). It is possible to dump just a certain sub-tree of the file system (a specified directory and all of its inferiors), but usually you just dump the whole thing.

[Two more kinds of dumps, *consolidated* and *archival* dumps, are planned but not yet implemented. The `:offline` attribute applies to the archival dumping system. Backing up of the properties of directories also is not yet implemented.]

1.8 File System Editor

The File System Editor is an interactive program that lets you examine and modify the contents of the file system. You can create directories and links, view and edit the properties of file system objects, delete objects and expunge directories. The File System Editor is part of the File System Maintainance program, and it is the only part that most users ever use. The rest of the File System Maintenance program is described in section 1.10, page 17.

To get the File System Maintainance program, type System F. The screen will look like Figure 1. At the top of the frame there is a menu of commands. The three commands whose names start with "Tree edit" invoke the File System Editor.

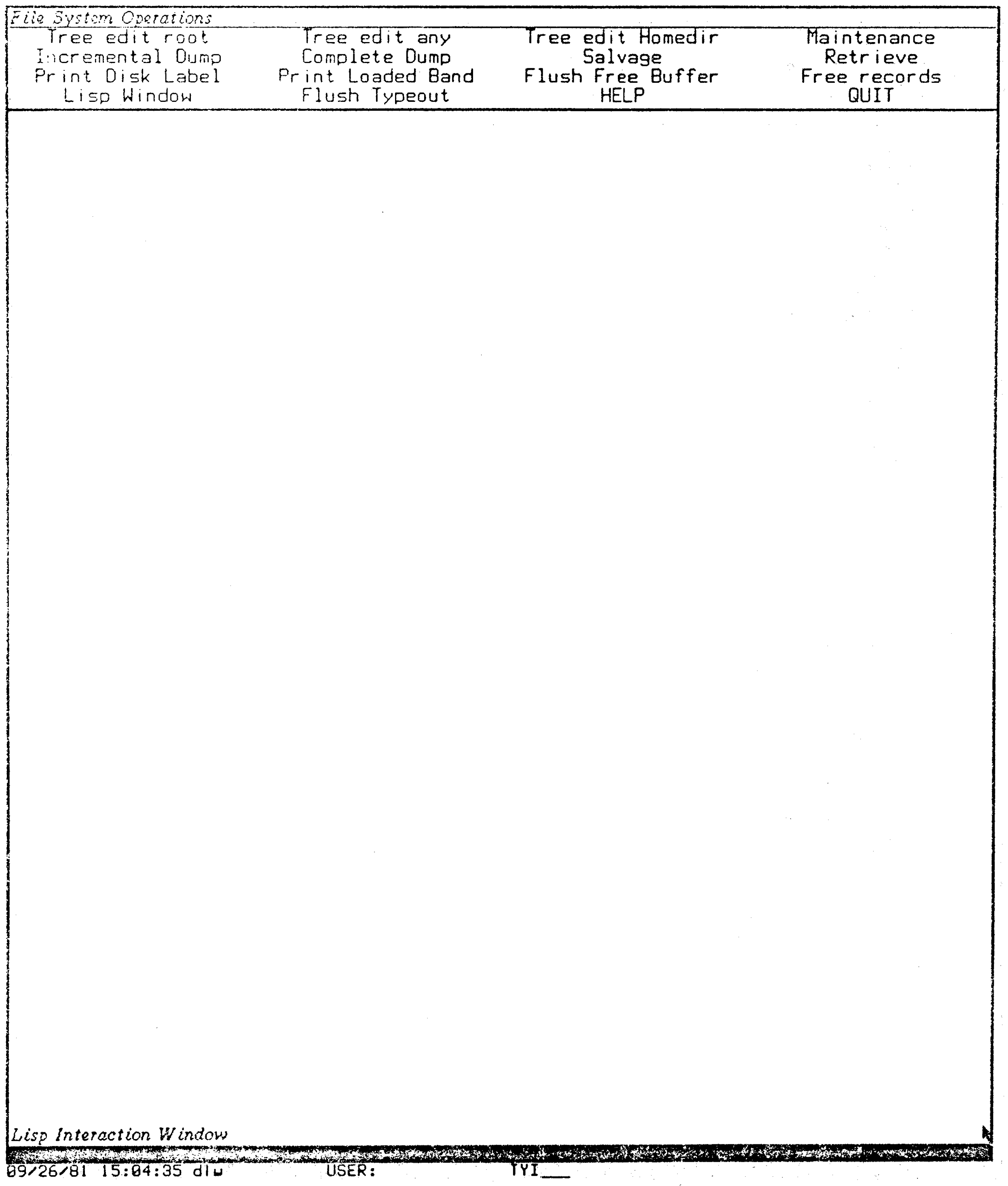
When you use the File System Editor, the big window in the frame displays a particular tree of the file system; that is, it displays a certain directory (the *base* directory) and some of the objects under that directory. If you click on the Tree Edit Root command, the base directory will be the root directory of the file system; this lets you get at any file in the entire file system. The Tree Edit any command lets you specify the base directory by typing in its pathname; after you click on this command you are prompted for a pathname. The Tree Edit Homedir command makes your home directory be the base directory. If you just want to try out the file system editor, try Tree Edit Root.

After you click on one of these commands (and answer the prompt, if the command is Tree Edit any), you are in the File System Editor. You never have to get "out" of it; if you want to do something unrelated to the file system, just select the window you want to use, and if you want to do something else with the File System Maintenance program, you just click on the appropriate command in the command menu (at the top).

Now you are in the File System Editor. If you did Tree Edit Root, the screen would look something like the one in Figure 2. (Of course, the exact appearance of the screen depends on the contents of your file system.) At the top of the main window is a line reading >*.*.*. This line represents the root directory. Below it is a set of indented lines, one representing each object in the root directory. Usually the root directory contains only directories, and this is the case in our example. Now, suppose we move the mouse over the line that represents the >LMIO directory, and click left. That line will change to read >LMIO>*.*.*, and several lines will be inserted just underneath it, one for each object in the >LMIO directory. We have just *opened* the >LMIO directory, and the screen now looks like Figure 3.

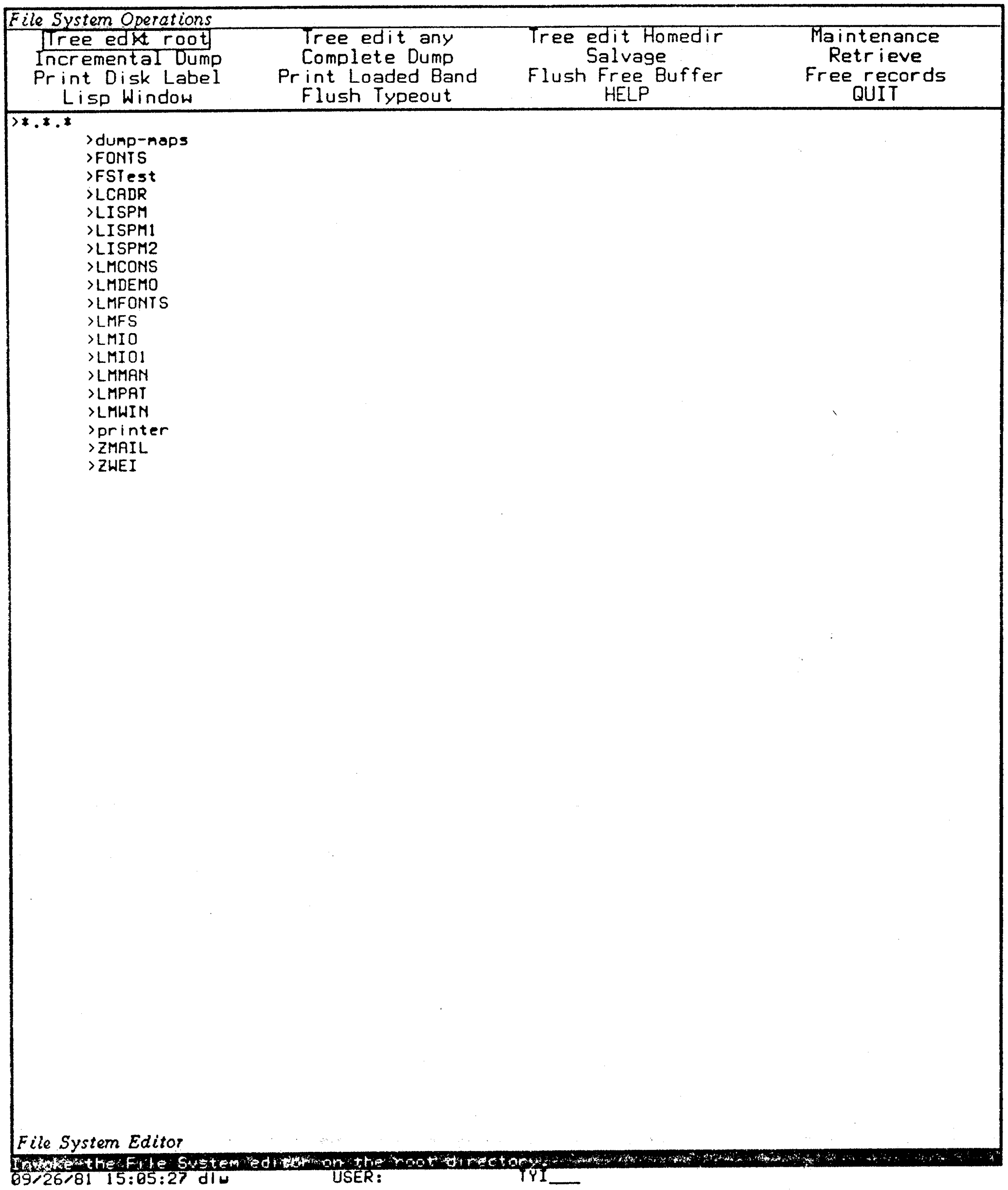
When we open a directory, a line is inserted in the display for each object in the directory. For every directory, there is a line with the pathname of the directory, and nothing else; these directories are all closed. For every file, there is a line with the name, type, and version of the file, and other information about the file. For every link, there is a line with the name, type, and version of the link, followed by => and the pathname of the target of the link, and other information.

For example, in figure 3, the > directory and the >LMIO directory are open, and the rest are closed. The objects in > are all directories, as it happens, and the objects in >LMIO are all files. The first object in >LMIO is a file with name CHSAUX, type LISP, and version 158. Its :length-in-blocks is 15. Its :length-in-bytes is 55530, and its byte size is 8 (it is a character file). It was created on 09/11/81 at 18:30:52, and was last referenced on 9/11/81. Its author was MMcM.



```
09/26/81 15:04:35 dlw        USER:           TYI___
```

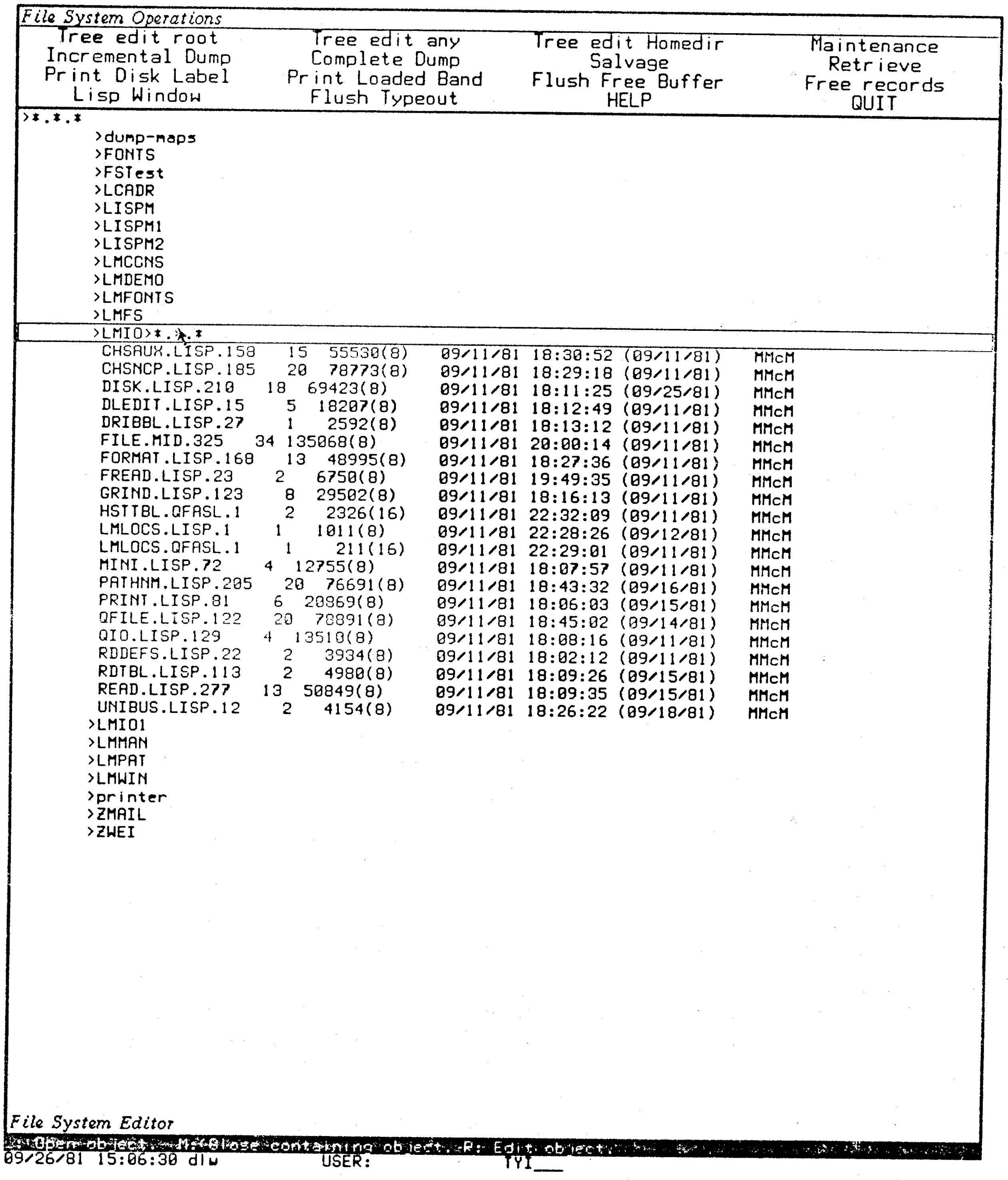
Figure 1: The File System Maintainance program



```
File System Operations
   Tree edit root        Tree edit any       Tree edit Homedir      Maintenance
   Incremental Dump      Complete Dump       Salvage                Retrieve
   Print Disk Label      Print Loaded Band   Flush Free Buffer      Free records
   Lisp Window           Flush Typeout       HELP                   QUIT
>*.*.*
       >dump-maps
       >FONTS
       >FSTest
       >LCADR
       >LISPM
       >LISPM1
       >LISPM2
       >LMCONS
       >LMDEMO
       >LMFONTS
       >LMFS
       >LMIO
       >LMIO1
       >LMMAN
       >LMPAT
       >LMWIN
       >printer
       >ZMAIL
       >ZWEI

File System Editor
Invoke the File System editor on the root directory.
09/26/81 15:05:27 dlw           USER:               TYI___
```

Figure 2: The File System Editor



```
>*.*.*
         >dump-maps
         >FONTS
         >FSTest
         >LCADR
         >LISPM
         >LISPM1
         >LISPM2
         >LMCONS
         >LMDEMO
         >LMFONTS
         >LMFS
         >LMIO>*.*.*
          CHSAUX.LISP.158    15  55530(8)    09/11/81 18:30:52 (09/11/81)  MMcM
          CHSNCP.LISP.185    20  78773(8)    09/11/81 18:29:18 (09/11/81)  MMcM
          DISK.LISP.210    18  69423(8)      09/11/81 18:11:25 (09/25/81)  MMcM
          DLEDIT.LISP.15      5  18207(8)    09/11/81 18:12:49 (09/11/81)  MMcM
          DRIBBL.LISP.27      1   2592(8)    09/11/81 18:13:12 (09/11/81)  MMcM
          FILE.MID.325    34 135068(8)       09/11/81 20:00:14 (09/11/81)  MMcM
          FORMAT.LISP.168    13  48995(8)    09/11/81 18:27:36 (09/11/81)  MMcM
          FREAD.LISP.23     2   6750(8)      09/11/81 19:49:35 (09/11/81)  MMcM
          GRIND.LISP.123     8  29502(8)     09/11/81 18:16:13 (09/11/81)  MMcM
          HSTTBL.QFASL.1     2   2326(16)    09/11/81 22:32:09 (09/11/81)  MMcM
          LMLOCS.LISP.1     1   1011(8)      09/11/81 22:28:26 (09/12/81)  MMcM
          LMLOCS.QFASL.1     1     211(16)   09/11/81 22:29:01 (09/11/81)  MMcM
          MINI.LISP.72    4  12755(8)        09/11/81 18:07:57 (09/11/81)  MMcM
          PATHNM.LISP.205    20  76691(8)    09/11/81 18:43:32 (09/16/81)  MMcM
          PRINT.LISP.81    6  20869(8)       09/11/81 18:06:03 (09/15/81)  MMcM
          QFILE.LISP.122    20  78891(8)     09/11/81 18:45:02 (09/14/81)  MMcM
          QIO.LISP.129    4  13510(8)        09/11/81 18:08:16 (09/11/81)  MMcM
          RDDEFS.LISP.22     2   3934(8)     09/11/81 18:02:12 (09/11/81)  MMcM
          RDTBL.LISP.113     2   4980(8)     09/11/81 18:09:26 (09/15/81)  MMcM
          READ.LISP.277    13  50849(8)      09/11/81 18:09:35 (09/15/81)  MMcM
          UNIBUS.LISP.12     2   4154(8)     09/11/81 18:26:22 (09/18/81)  MMcM
         >LMIO1
         >LMMAN
         >LMPAT
         >LMWIN
         >printer
         >ZMAIL
         >ZWEI
```

```
L: Open object, M: Close containing object, R: Edit object.
09/26/81 15:06:30 dlw          USER:          TYI___
```

Figure 3: Opening a Directory

Whenever you click left on a closed directory, it is opened, and its contents appear. By clicking on successive directories inside other directories, you can poke around in the file system and see what is there. The base directory is automatically opened as soon as you start using the File System Editor. When you are finished with a directory, you can *close* the directory by clicking middle on any of the objects inside that directory. So, if you click left on a file, then that file and everything at its level will disappear from the display. It is pretty easy for the display to become larger than the size of the window when you move around in large directories; you can use the usual mouse scrolling commands to move the display up and down in the window. Using these commands, you can get at any part of the file system underneath the base directory, and see everything that is there.

To do something to an object, you click right on the object. This pops up a menu of commands, each of which specifies something to do with the object. Some command make sense for all three kinds of objects; others are specific to certain kinds of object. Here is a list of all the commands:

`Delete` *(Files, directories, links)*
: Delete this object. This command is only displayed for objects that are not already deleted. [You should not delete directories that have anything in them.]

`Undelete` *(Files, directories, links)*
: Undelete this object. This command is only displayed for objects that are deleted.

`Rename` *(Files, directories, links)*
: Rename this object; you will be prompted for a new name. If the object is not a directory, you may optionally type in a whole pathname specifying a new directory, and the file or link will be moved to the new directory as well as being given the new name.

`View Properties` *(Files, directories, links)*
: Type out one line for each property of the object, giving the name and the value of the property. This information types out on top of the display, and prompts you to type any character when you are ready to proceed. After you type this character, the properties vanish and the window is redisplayed. You can also use the `Flush Typeout` command in the command menu to make the typeout vanish; this is convenient since you don't have to move from the mouse to the keyboard.

`Edit Properties` *(Files, directories, links)*
: Pop up a Choose Variable Values window that lets you change the value of any changeable system property or user property of the object.

`New Property` *(Files, directories, links)*
: Create a new user property for the object. You will first be prompted for the name of the property, and then the value. The name will be upper-cased. If you give an empty string as the value, the property will be removed.

`View` *(Files, links)*
: Print out the file. The file is typed out on top of the display, and you will be prompted to type any character when you are ready to proceed. The `:reference-date` of the file (see section 1.4, page 3) of the file will not be changed. If the object is a link, it must be transparent to `:read` and its target must be a file; the target will be printed.

`Create Inferior Directory` *(Directories)*
: Create a new directory inside this directory. You will be prompted for the name (just type in the name, not the whole pathname).

Create Link *(Directories)*

Create a new link inside this directory. You will first be prompted for the name of the link, and then for the full pathname of the target of the link. See section 1.6, page 7 for details.

Expunge *(Directories)*

Expunge the directory (see section 1.5, page 6). [This generally takes a while.]

Open *(Directories)*

Open the directory. This is the same as clicking left on it. This command is only displayed for closed directories.

Selective Open *(Directories)*

Prompts for a "star name", e.g. a file name containing "*" characters to indicate a wild-card component. The directory will be opened, but only those objects in the directory that match this pathname will be displayed. Unspecified components default to "*". The normal Open command is like a Selective Open of *.*.*, displaying all files. For example, if you do a Selective Open of "*.lisp", only files whose type is "lisp" will be displayed. (In this example, the version was unspecified and defaulted to "*".) The line in the display that corresponds to the directory will show this star name.

Close *(Directories)*

Close the directory. This is the same as clicking middle on one of its inferiors. This command is only displayed for open directories.

Edit Link Transparencies *(Links, directories)*

Lets you change the :link-transparencies of a link, or the :default-link-transparencies of a directory. This command displays a menu showing all of the operations to which a link may or may not be transparent. Each one that the link actually *is* transparent to is highlighted with reverse video. By clicking on the name of any operation, you can turn the highlighting on or off. When you are done changing the transparencies, click on *Do It*, and the transparencies (or default tansparencies, if this is a directory) will be set. You can click on *Abort* to abort the operation.

Decache *(Directories)*

When a directory is opened, the File System Editor examines the directory, sees what is there, and remembers it. If some other process in the Lisp Machine changes the contents of the directory while you are in the middle of editing that directory, the File System Editor will not know that anything has changed, and so what it shows you won't really correspond with the state of the file system. The Decache command tells the File System Editor to forget what it thinks it knows about the contents of the directory, and makes it go back to the file system to see what is really in the directory now.

Dump *(Files, directories, links)*

Invoke the dumper. See section 1.10.2, page 19 for details.

[Some old versions of the File System Editor have slightly different names for some of these commands.]

1.9 Program Access

The Symbolics File System has many features that the Lisp Machine software does not assume that every file system will have. To access these new features, there are some new options for `with-open-file` (or `open`), and some new messages accepted by pathnames.

1.9.1 Opening Files

In addition to the standard options used by `with-open-file` (see *The Lisp Machine Manual*), the file system also support the following options:

`:binary` A synonym for `:fixnum`.

`:append` This option is an alternative to `:read` and `:write`. It opens an existing file and lets you add characters or bytes to the end; as soon as the file is opened, the current position is set to the end of the file, and then you can write new characters or bytes. If the file does not exist, it is created.

`:dont-chase-links`
This is meaningful only together with the `:probe` option. If this option is specified, the returned information will be about the link itself; otherwise the information will be about the target.

`:incremental-update`
If this option is specified, then every time a new record of the file is created, the internal file system data about the file will be written out as well. This means that if the file is never closed, a partially-written valid version of it will probably exist in the file system anyway. Specifying this option will substantially degrade the performance of writing the file.

`:no-update-reference-time`
Do not update the `:reference-date` of the file. The File System Editor's `View` command does this.

The `:raw` and `:super-image` options are ignored, since there is no character set translation; the file system uses the Lisp Machine character set internally. `:deleted` is supported, but `:temporary` is not.

The stream created by opening a file in `:probe` mode must be closed when you are finished using it. The `probef` function does this automatically.

Internally, files are packed into bytes whose bit size is a power of two, so, for example, using a byte size of 9 has no advantage over a byte size of 16.

[These new options only work when you are using the file system locally; they are not supported by remote file sytsems. This will be fixed.]

1.9.2 Pathname Messages

Pathnames for files in the Symbolics File System understand the following messages (as well as all the standard ones):

`:condense-directory`
This should only be sent to a pathname of a directory. It returns a new pathname, whose directory component is this directory, and whose `name`, `type`, and `version` are all `nil`.

`:create-directory` &optional (*error-p t*) &rest *options*
Create new directory. The directory created will be the `directory` component of the pathname receiving this message. The parent directory must already exist. `t` will be returned if the creation is successful. If the creation is not successful, an error will be signalled unless *error-p* is given as `nil`, in which case an error string is returned. No options are currently defined.

`:create-link` *target* &optional *transparencies*
Creates a link. The pathname itself is the pathname of the link, and must be complete. The target of the link will be *target*, whose version may be `nil` but the rest of which must be specified. The transparencies are initialized from the parent's default link transparencies, and if *transparencies* is supplied, the `:transparencies` option is set from it (see section 1.4, page 3).

`:expunge`
Expunge the directory corresponding to the directory component of the this pathname. Two values are returned; the first is either the number or disk records released, or an error string, if the operation could not be done. The second is a list of per-file errors encountered during the operation.

[Except for `:condense-directory`, these new messages only work when you are using the file system locally; they are not supported by remote file sytsems. This will be fixed.]

1.10 File System Maintenance Program

The File System Maintenance Program includes the File System Editor, described above (section 1.8, page 10), as well as other commands for manipulating backup tapes, running the salvager, and doing other maintenance operations.

The program uses a frame with a command menu pane at the top and a large pane beneath it. Commands are given by clicking on the command menu pane. The large pane can either be a *Lisp Interaction Window* or a *File System Editor* window, depending on whether or not you are in the File System Editor. When it is in the former state, it is a Lisp listener, and you can type Lisp forms at it and have them evaluated and printed. When it is in the latter state, you are in the File System Editor and can click on the pane to get File System Editor commands.

1.10.1 Commands

The following list explains all of the commands in the program. Some commands are described as "typing out" certain information. If the large pane is a *Lisp Interaction Window*, the information is simply printed on that window; if it is a *File System Editor*, then the information is typed out over the file system editor information using a "typeout window". You can flush the typeout and restore the display of the File System Editor by typing any character or by clicking on the `Flush Typeout` menu item.

`Tree Edit Root`
: Enter the File System Editor, using the root directory as the base directory. See section 1.8, page 10. This puts the large pane into the *File System Editor* state.

`Tree Edit any`
: Enter the File System Editor; you will be prompted for the name of the base directory. See section 1.8, page 10. This puts the large pane into the *File System Editor* state.

`Tree Edit Homedir`
: Enter the File System Editor, using the users's home directory as the base directory. See section 1.8, page 10. This puts the large pane into the *File System Editor* state.

`Free Records`
: Type out information about the number of free records in the file system. Most of this information is only of interest to people who understand about the internal organization of file system data bases; the interesting part is the last line, which tells you how many records are marked as free and how many are marked as used, and the sum of these numbers, which is the total number of records in the file system. See section 1.10.3, page 21 for details.

`Salvage`
: Run the salvager. See section 1.10.4, page 22 for details.

`Retrieve`
: Run the retriever (see section 1.7, page 9). [This command is not yet implemented.]

`Incremental Dump`
: Do an incremental dump of the root directory. See section 1.10.2, page 19 for details.

`Complete Dump`
: Do an complete dump of the root directory. See section 1.10.2, page 19 for details.

`Print Disk Label`
: Type out the disk label; see the description of the `print-disk-label` function in *The Lisp Machine Manual*.

`Print Loaded Band`
: Type out information about the versions of software present in the band from which this machine was cold-booted. See the description of the `print-loaded-band` function in *The Lisp Machine Manual*.

`Flush Free Buffer`
: Write the internal pool of free disk records back to the disk. This happens when you log out. After doing this, you may cold-boot without losing records. See section 1.10.3, page 21 for more details.

`Maintenance` Put up a menu of potentially dangerous operations, intended to be used only by people who are thoroughly familiar with the file system, just below the main command menu. These operations are described below.

`Lisp Window` Put the large pane back into the *Lisp Interaction Pane* state.

`Flush Typeout`
When the large pane is in the *File System Editor* state, and you use one of the above command that "types out" information, then the information will appear on top of the File System Editor window, and you will be told "Type any char to flush:". You can use this command to flush typeout and restore the File System Editor window, too; the command is useful because you can give it without removing your hand from the mouse. You can also use this command to proceed from `**MORE**` pauses.

`HELP` Type out general information about the File System Maintenance program.

`QUIT` Bury the frame, possibly reselecting the window you were using before you started using the File System Maintenance program.

The following items appear in the *Maintenance Operations* menu. They are intended only for personnel who are thoroughly familiar with the operations of the File System; misuse of these operations can destroy or damage the File System.

`Initialize` Create a new file system on the `FILE` paritition of disk `0`. This operation asks you several questions and prints out information to make sure you really want to initialize the `FILE` partition, since this would wipe out any previous file system residing there. This takes about a minute for each four thousand records (a record is four 256-word disk blocks). The `FILE` paritition should be at least 1000 records (4000 disk blocks) long, and the bigger it is, the more room there will be for files. Note that each file uses up at least one record.

`Close All Files`
Call `fs:close-all-files` (see *The Lisp Machine Manual*). This has nothing to do with the Symbolics file system *per se*; it closes any files, locally or remotely, on any file system to which the machine is connected. This is occasionally useful for cleaning up after problems, but if any programs in the machine are validly using files at the time you do it, those programs may get confused.

`Reload` Reload the file system from backup tape (see section 1.10.2, page 19).

`Active Structure Edit`
A maintenance tool for inspecting the active structure of the File System [not yet implemented].

1.10.2 Dumping, Reloading, and Retrieving

(Please read section 1.7, page 9, before reading this section.)

To perform a dump, mount a magnetic tape on a tape drive connected to the machine, and then invoke the `Incremental Dump` or `Complete Dump` commands in the File System Maintenance program, or the `Dump` command in the File System Editor. All three of these commands respond by popping up a Choose Variable Values window that lets you set the parameters of the dump; the only difference between the command is the initial value of some of the values in this window. Here are the parameters:

`Dump Type` There are four possible types of dump: ***incremental***, ***consolidated***, ***complete***, and ***archival***. Only complete and incremental dumps are currently implemented. The two File System Maintenance commands initialize this field appropriately; the `Dump` command initializes it to ***complete***.

`Tape Reel ID`
Every reel of tape produced by the dumper must have a Tape Reel ID, which is a string of up to eight characters. You must explicitly supply some value for this option. The reel ID is how this reel of tape is identified to the backup system; it will appear in the dump maps (see below) and in any messages about the tape. The `:complete-dump-tape` or `:incremental-dump-tape` property of any file dumped will be set to this value, as well. The Tape Reel ID should by physically written onto the tape so that the tape can be identified by sight.

`Starting directory`
The pathname of the directory at which the dump should start. A dump always starts at some particular directory, and dumps that directory and all of its inferiors. The two File System Maintenance commands initialize this value to > (the root directory), which is its usual value; this means to dump the whole file system. The `Dump` command sets this to the directory on which you clicked. The value of this field can also be the pathname of a file or link, to dump just that one file or link.

`Tape drive number`
The drive number of the tape drive on which the reel of tape is mounted. The default value is zero.

`Dump deleted files`
This can be either *Yes* or *No*, and says whether files marked as deleted but not yet expunged should be dumped on the backup tape. The default value is *No*; deleted files normally are not dumped.

`Comment` A string of arbitrary contents, written on each reel of the dump and in the dump map. This might say why the dump was taken or any other special information about this dump.

Note that you ***must*** supply a Tape Reel ID. When you are done, click on the `Done` box; if you decide not to do a dump after all, click on the `Abort` box. [In some early releases, the `Done` box is labelled `Exit`, and you have to type `Control-Meta-Abort` to abort.] If there is something wrong about the set of parameters you have specified, the program will type out a message and present you with the Choose Variable Values window again. Otherwise, it will type out a message saying that the dump has started successfully, and it will proceed.

The dumper creates a file called the ***dump map***. The dump map is a character file giving a complete description of what has been dumped, directory by directory and file by file, including the time of dumping, the tape on which the file was dumped, the tape reel ID of the previous tape on which the file appears (if any), and so on. The dump map is created in the `>dump-maps` directory. Its name is constructed from the type of dump and the date and time at which it was started; the type is `map` and the version is 1. A typical dump map might have the pathname

```
>dump-maps>complete-9/15/81-9:02.map.1
```

The dumper also creates a file called the ***tape directory***, for each separate reel of each dump. This is a binary file saying what is on the tape, with more or less the same information as the dump map. This file can be searched and examined by various program tools [which are not yet implemented]. The tape directory is also created in the `>dump-maps` directory. Its name is the tape reel ID of the tape, its type is `directory`, and its version is 1. A typical directory map might have the pathname

```
    >dump-maps>INC00001.directory.1
```

To perform a reload, use the `Reload` operation in the *Maintenance Operations* menu (which appears in response fo the `Maintenance` command). [The existing reloader is primitive. It always reloads exactly one reel of tape from drive zero. If the file system is destroyed, you should reinitialize it, and reload from dump tapes. Each dump tape can stand alone, so you can just give them to the reloader one by one. The existing reloader has some problems. It never overwrites an existing file; if it finds a file on the tape that already exists in the file system, it ignores the copy on the tape. This means that if, in between a complete dump and a later incremental dump, a file is deleted and a new file is created with the same name, the reloaded file system will have the old version instead of the new version. Also, if a file was deleted between the complete dump and a later incremental dump, the file will re-appear at the end of the reload. These problems will be fixed in future versions. The reloader types out information about what it is doing, as it goes.]

[The retriever is not yet implemented.]

1.10.3 Free Records

The basic unit of allocation in the File System is the *record*. A record is 1024 32-bit words, or four disk blocks. Each file system object is made from an integeral number of records. At any time, each record is *in use* (representing an existing file system object) or *free* (not representing anything and free to be used in new objects). When the file system needs to find a new free block to create or grow an object, it does not search through the records looking for a free one, because that would require many disk operations and be very slow. Instead, the file system keeps a redundant data structure called the *free record map*, kept in several blocks in a known location in the file system partition. The map has one bit for each record in the file system; this bit marks whether the record is in use or is free. The file system can find a free record quickly by examining this map.

If the file system crashes, or something else goes wrong, the contents of the free record map can become inconsistent with the contents of the file system itself. For each record, there are two different errors that might happen. The first error is that the record might be in use, representing some of an object, but be marked as "free" in the map; the second error is that the record might be free, but be marked as "in use" in the map. The first error is much worse than the second. If the first error happens, then the file system might use the record for a new object even though it is currently representing some existing object, which could destroy the existing object. The second error is less severe: the record simply isn't allocated even though it could be. Such a record is said to be *lost*.

The file system is written so that a crash can only create the second kind of error and not the first. While the file system is operating, it maintains a *free buffer* in its data structures in virtual memory. The free buffer is a pool of records that are not actually in use, but that are marked as being in use in the free map on the disk. When it needs to allocate a record, it draws on one of these; when it frees up a record, it adds the record to this buffer. When the buffer gets too big, some records are removed from the buffer and marked as "free" in the map on the disk; when the buffer runs low, some more records are marked as "in use" in the map on the disk, and are added to the buffer. So, if the machine is cold-booted, or the file system crashes, the records that are in the buffer will be lost, but no errors of the first kind will be created. The size of the buffer is maintained at about 30 records, so each crash will lose 30 records. Logging out of the machine, or using the `Flush Free Buffer` command, will flush the entire free buffer and mark the records as "free" in the map on the disk. After the buffer has been flushed, you can cold-boot the machine without losing any blocks.

Lost records can be found again by the salvager; see section 1.10.4, page 22.

The `Free Records` command first types out a line for each block of the file map, telling you which records are covered by that block, the number of such records, and how many are marked as free. `Free Records` also tells you how many free records (marked as "in use" in the map) are in the free buffer, and finally types out a grand total of the number of free, used, and total records in the file system.

To find out how many records are actually in use, use

```
    (lmfs:compute-records-used-globally)
```

This has to pass over every object in the file system, and so it takes some time, especially on large file systems. It types out messages reporting its progress as it goes. The discrepancy between the answer of this function and the answer of `Free Records` is large tells you how many lost blocks there are; if there are a lot, you may want to run the salvager. [While running this, you must make sure that nothing else tries to use the file system; in particular, make sure that the remote file server isn't running. This will be fixed in the future.]

1.10.4 Salvager

The *salvager* is a program that reads every record of the file system and finds and fixes inconsistencies and errors. There are two classes of problems that the salvager can fix. First of all, it can see which records are in use and which are free, and set the free record map to correctly reflect the current state of the file system. This is how lost records are recovered. Secondly, it can find objects that are stored in the file system partition but are not part of any directory. Such objects are called *orphans*; they only exist if some problem has occured, such as a file system crash during the creation of a file, or due to an unanticipated directory failure of some sort. The salvager can find such objects and put them back into the directory hierarchy (*repatriate* them).

To run the salvager, give the `Salvage` command. The salvager always reconstructs the free record maps; you will be queried as to whether it should repatriate orphans. The salvager takes about two minutes per thousand records of file partition. While it is running, all other file system activity is locked out; any attempt to use the file system will wait until the salvage is finished [this will be fixed in the future].

When the salvager is repatriating an orphan and it cannot find the directory in which the orphan is supposed to reside, it creates a new directory as an inferior of the directory `>repatriations`, with a name like `lost-1` or `lost-2`. After a repatriating salvager run, you should examine this directory. When the salvager repatriates an object, it types out a message saying that it did so. (If you walk away from the machine while the salvager is running, one of these messages might cause a `**MORE**` pause; you may want to disable `**MORE**` pauses in the salvager.)

Note: the salvager always considers storage occupied by orphans to be "in use" for purposes of the free record map, even if it is not repatriating the orphan. If there are a lot of orphans around, they can use up lots of disk space, but normally they don't occur.

[Note: it is currently possible to deal with a broken directory (one whose format is damaged so that any attempt to use it causes an error) by simply deleting the directory and then running the salvager to repatriate all of its inferiors. This operation may require a different sequence of commands in the future.]