17. FILE MANAGER

*Warning: The subsystem within Medley used for managing collections of definitions (of functions, variables, etc.) is known as the "File Manager." This terminology is confusing, because the word "file" is also used in the more conventional sense as meaning a collection of data stored on some physical media. Unfortunately, it is not possible to change this terminology at this time, because many functions and variables (*MAKEFILE, FILEPKGTYPES*, etc.) incorporate the word "file" in their names.*

Most implementations of Lisp treat symbolic files as unstructured text, much as they are treated in most conventional programming environments. Function definitions are edited with a character-oriented text editor, and then the changed definitions (or sometimes the entire file) is read or compiled to install those changes in the running memory image. Interlisp incorporates a different philosophy. A symbolic file is considered as a database of information about a group of data objects---function definitions, variable values, record declarations, etc. The text in a symbolic file is never edited directly. Definitions are edited only after their textual representations on files have been converted to data-structures that reside inside the Lisp address space. The programs for editing definitions inside Medley can therefore make use of the full set of data-manipulation capabilities that the environment already provides, and editing operations can be easily intermixed with the processes of evaluation and compilation.

Medley is thus a "resident" programming environment, and as such it provides facilities for moving definitions back and forth between memory and the external databases on symbolic files, and for doing the bookkeeping involved when definitions on many symbolic files with compiled counterparts are being manipulated. The file manager provides those capabilities. It shoulders the burden of keeping track of where things are and what things have changed so that you don't have to. The file manager also keeps track of which files have been modified and need to be updated and recompiled.

The file manager is integrated into many other system packages. For example, if only the compiled version of a file is loaded and you attempt to edit a function, the file manager will attempt to load the source of that function from the appropriate symbolic file. In many cases, if a datum is needed by some program, the file manager will automatically retrieve it from a file if it is not already in your working environment.

Some of the operations of the file manager are rather complex. For example, the same function may appear in several different files, or the symbolic or compiled files may be in different directories, etc. Therefore, this chapter does not document how the file manager works in each and every situation, but instead makes the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

For a simple illustration of what the file manager does, suppose that the symbolic file FOO contains the functions FOO1 and FOO2, and that the file BAR contains the functions BAR1 and BAR2. These two files could be loaded into the environment with the function LOAD:

```
_ (LOAD 'FOO)
FILE CREATED  4-MAR-83 09:26:55
FOOCOMS
```

```
{DSK}FOO.;1

_ (LOAD 'BAR)
FILE CREATED  4-MAR-83 09:27:24
BARCOMS
{DSK}BAR.;1
```

Now, suppose that we change the definition of FOO2 with the editor, and we define two new functions, NEW1 and NEW2. At that point, the file manager knows that the in-memory definition of FOO2 is no longer consistent with the definition in the file FOO, and that the new functions have been defined but have not yet been associated with a symbolic file and saved on permanent storage. The function FILES? summarizes this state of affairs and enters into an interactive dialog in which we can specify what files the new functions are to belong to.

```
_ (FILES?)
FOO...to be dumped.
    plus the functions: NEW1,NEW2
want to say where the above go ? Yes
(functions)
NEW1  File name: BAR
NEW2  File name: ZAP
  new file ? Yes
NIL
```

The file manager knows that the file FOO has been changed, and needs to be dumped back to permanent storage. This can be done with MAKEFILE.

```
_(MAKEFILE 'FOO)
{DSK}FOO.;2
```

Since we added NEW1 to the old file BAR and established a new file ZAP to contain NEW2, both BAR and ZAP now also need to be dumped. This is confirmed by a second call to FILES?:

```
_(FILES?)
BAR, ZAP...to be dumped.
FOO...to be listed.
FOO...to be compiled
NIL
```

We are also informed that the new version we made of FOO needs to be listed (sent to a printer) and that the functions on the file must be compiled.

Rather than doing several MAKEFILEs to dump the files BAR and ZAP, we can simply call CLEANUP. Without any further user interaction, this will dump any files whose definitions have been modified. CLEANUP will also send any unlisted files to the printer and recompile any files which need to be recompiled. CLEANUP is a useful function to use at the end of a debugging session. It will call FILES? if any new objects have been defined, so you do not lose the opportunity to say explicitly where those belong. In effect, the function CLEANUP executes all the operations necessary to make the your permanent files consistent with the definitions in the current core-image.

```
_ (CLEANUP)
FOO...compiling {DSK}FOO.;2
    .
    .
```

```
   .
BAR...compiling {DSK}BAR.;2
   .
   .
   .
ZAP...compiling {DSK}ZAP.;1
   .
   .
   .
```

In addition to the definitions of functions, symbolic files in Interlisp can contain definitions of a variety of other types, e.g. variable values, property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file manager uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a symbol), a definition of a given type (called the file manager type). Note that the same name may have several definitions of different types. For example, a symbol may have both a function definition and a variable definition. The file manager also keeps track of the files that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

Symbolic files on permanent storage devices are referred to by names that obey the naming conventions of those devices, usually including host, directory, and version fields. When such definition groups are noticed by the file manager, they are assigned simple *root names* and these are used by all file manager operations to refer to those groups of definitions. The root name for a group is computed from its full permanent storage name by applying the function ROOTFILENAME; this strips off the host, directory, version, etc., and returns just the simple name field of the file. For each file, the file manager also has a data structure that describes what definitions it contains. This is known as the commands of the file, or its "filecoms". By convention, the filecoms of a file whose root name is *X* is stored as the value of the symbol *X*COMS. For example, the value of FOOCOMS is the filecoms for the file FOO. This variable can be directly manipulated, but the file manager contains facilities such as FILES? which make constructing and updating filecoms easier, and in some cases automatic. See the Functions for Manipulating File Command Lists section.

The file manager is able to maintain its databases of information because it is notified by various other routines in the system when events take place that may change that database. A file is "noticed" when it is loaded, or when a new file is stored (though there are ways to explicitly notice files without completely loading all their definitions). Once a file is noticed, the file manager takes it into account when modifying filecoms, dumping files, etc. The file manager also needs to know what typed definitions have been changed or what new definitions have been introduced, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file manager operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file manager. Also, *typed-in* assignment of variables or property values is noticed by the file manager. (Note that modifications to variable or property values during the execution of a function body are not noticed.) In some cases the marking procedure can be subtle, e.g. if you edit a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

All file manager operations can be disabled with FILEPKGFLG.

FILEPKGFLG [Variable]

The file manager can be disabled by setting FILEPKGFLG to NIL. This will turn off noticing files and marking changes. FILEPKGFLG is initially T.

The rest of this chapter goes into further detail about the file manager. Functions for loading and storing symbolic files are presented first, followed by functions for adding and removing typed definitions from files, moving typed definitions from one file to another, determining which file a particular definition is stored in, and so on.

Loading Files

The functions below load information from symbolic files into the Interlisp environment. A symbolic file contains a sequence of Interlisp expressions that can be evaluated to establish specified typed definitions. The expressions on symbolic files are read using FILERDTBL as the read table.

The loading functions all have an argument *LDFLG*. *LDFLG* affects the operation of DEFINE, DEFINEQ, RPAQ, RPAQ?, and RPAQQ. While a source file is being loaded, DFNFLG (Chapter 10) is rebound to *LDFLG*. Thus, if *LDFLG* = NIL, and a function is redefined, a message is printed and the old definition saved. If *LDFLG* = T , the old definition is simply overwritten. If *LDFLG* = PROP, the functions are stored as "saved" definitions on the property lists under the property EXPR instead of being installed as the active definitions. If *LDFLG* = ALLPROP, not only function definitions but also variables set by RPAQQ, RPAQ, RPAQ? are stored on property lists (except when the variable has the value NOBIND, in which case they are set to the indicated value regardless of DFNFLG).

Another option is available for loading systems for others to use and who wish to suppress the saving of information used to aid in development and debugging. If *LDFLG* = SYSLOAD, LOAD will:

1. Rebind DFNFLG to T, so old definitions are simply overwritten
2. Rebind LISPXHIST to NIL, thereby making the LOAD not be undoable and eliminating the cost of saving undo information (Chapter 13)
3. Rebind ADDSPELLFLG to NIL, to suppress adding to spelling lists
4. Rebind FILEPKGFLG to NIL, to prevent the file from being "noticed" by the file manager
5. Rebind BUILDMAPFLG to NIL, to prevent a file map from being constructed
6. After the load has completed, set the filecoms variable and any filevars variables to NOBIND
7. Add the file name to SYSFILES rather than FILELST

A filevars variable is any variable appearing in a file manager command of the form (*FILECOM* * *VARIABLE*) (see the FileVars section). Therefore, if the filecoms includes (FNS * FOOFNS),FOOFNS is set to NOBIND. If you want the value of such a variable to be retained, even when the file is loaded with *LDFLG* = SYSLOAD, then you should replace the variable with an equivalent, *non-atomic* expression, such as (FNS * (PROGN FOOFNS)).

All functions that have *LDFLG* as an argument perform spelling correction using LOADOPTIONS as a spelling list when *LDFLG* is not a member of LOADOPTIONS. LOADOPTIONS is initially (NIL T PROP ALLPROP SYSLOAD).

(**LOAD** *FILE LDFLG PRINTFLG*) [Function]

Reads successive expressions from *FILE* (with FILERDTBL as read table) and evaluates each as it is read, until it reads either NIL, or the single atom STOP. Note that LOAD can be used to load both symbolic and compiled files. Returns *FILE* (full name).

If *PRINTFLG* = T, LOAD prints the value of each expression; otherwise it does not.

(**LOAD?** *FILE LDFLG PRINTFLG*) [Function]

Similar to LOAD except that it does not load *FILE* if it has already been loaded, in which case it returns NIL.

LOAD? loads *FILE* except when the *same* version of the file has been loaded (either from the same place, or from a copy of it from a different place). Specifically, LOAD? considers that *FILE* has already been loaded if the full name of *FILE* is on LOADEDFILELST (see the Noticing Files section) or the date stored on the FILEDATES property of the root file name of *FILE* is the same as the FILECREATED expression on *FILE*.

(**LOADFNS** *FNS FILE LDFLG VARS*) [Function]

Permits selective loading of definitions. *FNS* is a list of function names, a single function name, or T, meaning to load all of the functions on the file. *FILE* can be either a compiled or symbolic file. If a compiled definition is loaded, so are all compiler-generated subfunctions. The interpretation of *LDFLG* is the same as for LOAD.

If *FILE* = NIL, LOADFNS will use WHEREIS (see the Storing Files section) to determine where the first function in *FNS* resides, and load from that file. Note that the file must previously have been "noticed". If WHEREIS returns NIL, and the WHEREIS library package has been loaded, LOADFNS will use the WHEREIS data base to find the file containing *FN*.

VARS specifies which non-DEFINEQ expressions are to be loaded (i.e., evaluated). It is interpreted as follows:

T Means to load all non-DEFINEQ expressions.

NIL Means to load none of the non-DEFINEQ expressions.

VARS Means to evaluate all variable assignment expressions (beginning with RPAQ, RPAQQ, or RPAQ?, see the Functions Used Within Source Files section).

Any other symbol Means the same as specifying a list containing that atom.

A list If *VARS* is a list that is not a valid function definition, each element in *VARS* is "matched" against each non-DEFINEQ expression, and if any elements in *VARS* "match" successfully,

the expression is evaluated. "Matching" is defined as follows: If an element of *VARS* is an atom, it matches an expression if it is EQ to either the CAR or the CADR of the expression. If an element of *VARS* is a list, it is treated as an edit pattern (see Chapter 16), and matched with the entire expression (using EDIT4E, described in Chapter 16). For example, if *VARS* was (FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO))), this would cause (RPAQQ FOOCOMS ...), all DECLARE:s, and all DEFLISTs which set up MACROs to be read and evaluated.

A function definition — If *VARS* is a list and a valid function definition ((FNTYP *VARS*) is true), then LOADFNS will invoke that function on every non-DEFINEQ expression being considered, applying it to two arguments, the first and second elements in the expression. If the function returns NIL, the expression will be skipped; if it returns a non-NIL symbol (e.g., T), the expression will be evaluated; and if it returns a list, this list is evaluated instead of the expression. The file pointer is set to the very beginning of the expression before calling the *VARS* function definition, so it may read the entire expression if necessary. If the function returns a symbol, the file pointer is reset and the expression is READ or SKREAD. However, the file pointer is not reset when the function returns a list, so the function must leave it set immediately after the expression that it has presumably read.

LOADFNS returns a list of:

1. The names of the functions that were found
2. A list of those functions not found (if any) headed by the symbol NOT-FOUND:
3. All of the expressions that were evaluated
4. A list of those members of *VARS* for which no corresponding expressions were found (if any), again headed by the symbol NOT-FOUND:

For example:

```
_ (LOADFNS '(FOO FIE FUM) FILE NIL '(BAZ (DEFLIST &)))
(FOO FIE (NOT-FOUND: FUM) (RPAQ BAZ ...) (NOT-FOUND:
(DEFLIST &)))
```

(**LOADVARS** *VARS FILE LDFLG*) [Function]

Same as (LOADFNS NIL *FILE LDFLG VARS*).

(**LOADFROM** *FILE FNS LDFLG*) [Function]

Same as (LOADFNS *FNS FILE LDFLG* T).

Once the file manager has noticed a file, you can edit functions contained in the file without explicitly loading them. Similarly, those functions which have not been modified do not have to be loaded in order to write out an updated version of the file. Files are normally noticed (i.e., their contents become known to the file manager) when either the symbolic or compiled versions of the file are loaded. If the file is *not* going to be loaded completely, the preferred way to notice it is with LOADFROM. You can also load some functions at the same time by giving LOADFROM a second argument, but it is normally used simply to inform the file manager about the existence and contents of a particular file.

(**LOADBLOCK** *FN FILE LDFLG*) [Function]

Calls LOADFNS on those functions contained in the block declaration containing *FN* (see Chapter 18). LOADBLOCK is designed primarily for use with symbolic files, to load the EXPRs for a given block. It will not load a function which already has an in-core EXPR definition, and it will not load the block name, unless it is also one of the block functions.

(**LOADCOMP** *FILE LDFLG*) [Function]

Performs all operations on *FILE* associated with compilation, i.e. evaluates all expressions under a DECLARE: EVAL@COMPILE, and "notices" the function and variable names by adding them to the lists NOFIXFNSLST and NOFIXVARSLST (see Chapter 21).

Thus, if building a system composed of many files with compilation information scattered among them, all that is required to compile one file is to LOADCOMP the others.

(**LOADCOMP?** *FILE LDFLG*) [Function]

Similar to LOADCOMP, except it does not load if file has already been loaded (with LOADCOMP), in which case its value is NIL.

LOADCOMP? will load the file even if it has been loaded with LOAD, LOADFNS, etc. The only time it will not load the file is if the file has already been loaded with LOADCOMP.

FILESLOAD provides an easy way for you to load a series of files, setting various options:

(**FILESLOAD** $FILE_1$ ... $FILE_N$) [NLambda NoSpread Function]

Loads the files $FILE_1$ ... $FILE_N$ (all arguments unevaluated). If any of these arguments are lists, they specify certain loading options for all following files (unless changed by another list). Within these lists, the following commands are recognized:

FROM *DIR* Search the specified directories for the file. *DIR* can either be a single directory, or a list of directories to search in order. For example, (FILESLOAD (FROM {ERIS}<LISPCORE>SOURCES>) ...) will search the directory {ERIS}<LISPCORE>SOURCES> for the files. If this is not specified, the default is to search the contents of DIRECTORIES (see Chapter 24).

If FROM is followed by the key word VALUEOF, the following word is evaluated, and the value is used as the list of directories to search. For example, (FILESLOAD

(FROM VALUEOF FOO) ...) will search the directory list that is the value of the variable FOO.

As a special case, if *DIR* is a symbol, and the symbol *DIR*DIRECTORIES is bound, the value of this variable is used as the directory search list. For example, since the variable LISPUSERSDIRECTORIES (see Chapter 24) is commonly used to contain a list of directories containing "library" packages, (FILESLOAD (FROM LISPUSERS) ...) can be used instead of (FILESLOAD (FROM VALUEOF LISPUSERSDIRECTORIES) ...)

If a FILESLOAD is read and evaluated while loading a file, and it doesn't contain a FROM expression, the default is to search the directory containing the FILESLOAD expression before the value of DIRECTORIES. FILESLOAD expressions can be dumped on files using the FILES file manager command.

SOURCE Load the source version of the file rather than the compiled version.

COMPILED Load the compiled version of the file.

If COMPILED is specified, the compiled version will be loaded, if it is found. The source will not be loaded. If neither SOURCE or COMPILED is specified, the compiled version of the file will be loaded if it is found, otherwise the source will be loaded if it is found.

LOAD Load the file by calling LOAD, if it has not already been loaded. This is the default unless LOADCOMP or LOADFROM is specified.

If LOAD is specified, FILESLOAD considers that the file has already been loaded if the root name of the file has a non-NIL FILEDATES property. This is a somewhat different algorithm than LOAD? uses. In particular, FILESLOAD will not load a newer version of a file that has already been loaded.

LOADCOMP Load the file with LOADCOMP? rather than LOAD. Automatically implies SOURCE.

LOADFROM Load the file with LOADFROM rather than LOAD.

NIL, T, PROP ALLPROP

SYSLOAD The loading function is called with its *LDFLG* argument set to the specified token. *LDFLG* affects the operation of the loading functions by resetting DFNFLG (see Chapter

10) to *LDFLG* during the loading. If none of these tokens are specified, the value of the variable LDFLG is used if it is bound, otherwise NIL is used.

NOERROR If NOERROR is specified, no error occurs when a file is not found.

Each list determines how all further files in the lists are loaded, unless changed by another list. The tokens above can be joined together in a single list. For example,

```
(FILESLOAD (LOADCOMP) NET (SYSLOAD FROM VALUEOF
NEWDIRECTORIES) CJSYS)
```

will call LOADCOMP? to load the file NET searching the value of DIRECTORIES, and then call LOADCOMP? to load the file CJSYS with *LDFLG* set to SYSLOAD, searching the directory list that is the value of the variable NEWDIRECTORIES.

FILESLOAD expressions can be dumped on files using the FILES file manager command.

Storing Files

(**MAKEFILE** *FILE OPTIONS REPRINTFNS SOURCEFILE*) [Function]

Makes a new version of the file *FILE*, storing the information specified by *FILE*'s filecoms. Notices *FILE* if not previously noticed. Then, it adds *FILE* to NOTLISTEDFILES and NOTCOMPILEDFILES.

OPTIONS is a symbol or list of symbols which specify options. By specifying certain options, MAKEFILE can automatically compile or list *FILE*. Note that if *FILE* does not contain any function definitions, it is not compiled even when *OPTIONS* specifies C or RC. The options are spelling corrected using the list MAKEFILEOPTIONS. If spelling correction fails, MAKEFILE generates an error. The options are interpreted as follows:

C

RC After making *FILE*, MAKEFILE will compile *FILE* by calling TCOMPL (if C is specified) or RECOMPILE (if RC is specified). If there are any block declarations specified in the filecoms for *FILE*, BCOMPL or BRECOMPILE will be called instead.

If F, ST, STF, or S is the *next* item on *OPTIONS* following C or RC, it is given to the compiler as the answer to the compiler's question LISTING? (see Chapter 18). For example, (MAKEFILE 'FOO '(C F LIST)) will dump FOO, then TCOMPL or BCOMPL it specifying that functions are not to be redefined, and finally list the file.

LIST After making *FILE*, MAKEFILE calls LISTFILES to print a hardcopy listing of *FILE*.

CLISPIFY MAKEFILE calls PRETTYDEF with CLISPIFYPRETTYFLG = T (see Chapter 21). This causes CLISPIFY to be called on each function defined as an EXPR before it is prettyprinted.

Alternatively, if *FILE* has the property FILETYPE with value CLISP or a list containing CLISP, PRETTYDEF is called with CLISPIFYPRETTYFLG reset to CHANGES, which will cause CLISPIFY to be called on all functions marked as having been changed. If *FILE* has property FILETYPE with value CLISP, the compiler will DWIMIFY its functions before compiling them (see Chapter 18).

FAST MAKEFILE calls PRETTYDEF with PRETTYFLG = NIL (see Chapter 26). This causes data objects to be printed rather than prettyprinted, which is much faster.

REMAKE MAKEFILE "remakes" *FILE*: The prettyprinted definitions of functions that have not changed are copied from an earlier version of the symbolic file. Only those functions that have changed are prettyprinted.

NEW MAKEFILE does *not* remake *FILE*. If MAKEFILEREMAKEFLG = T (the initial setting), the default for all calls to MAKEFILE is to remake. The NEW option can be used to override this default.

REPRINTFNS and *SOURCEFILE* are used when remaking a file.

FILE is not added to NOTLISTEDFILES if *FILE* has on its property list the property FILETYPE with value DON'TLIST, or a list containing DON'TLIST. *FILE* is not added to NOTCOMPILEDFILES if *FILE* has on its property list the property FILETYPE with value DON'TCOMPILE, or a list containing DON'TCOMPILE. Also, if *FILE* does not contain any function definitions, it is not added to NOTCOMPILEDFILES, and it is not compiled even when *OPTIONS* specifies C or RC.

If a remake is *not* being performed, MAKEFILE checks the state of *FILE* to make sure that the entire source file was actually LOADed. If *FILE* was loaded as a compiled file, MAKEFILE prints the message CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED. Similarly, if only some of the symbolic definitions were loaded via LOADFNS or LOADFROM, MAKEFILE prints CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED. In both cases, MAKEFILE will then ask you if it should dump anyway; if you decline, MAKEFILE does not call PRETTYDEF, but simply returns (*FILE* NOT DUMPED) as its value.

You can indicate that *FILE* must be block compiled together with other files as a unit by putting a list of those files on the property list of each file under the property FILEGROUP. If

FILE has a FILEGROUP property, the compiler will not be called until all files on this property have been dumped that need to be.

MAKEFILE operates by rebinding PRETTYFLG, PRETTYTRANFLG, and CLISPIFYPRETTYFLG, evaluating each expression on MAKEFILEFORMS (under errorset protection), and then calling PRETTYDEF.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG* = T, so whenever PRETTYPRINT (and hence MAKEFILE) start printing a new function, the name of that function is printed if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

(**MAKEFILES** *OPTIONS FILES*) [Function]

Performs (MAKEFILE *FILE OPTIONS*) for each file on *FILES* that needs to be dumped. If *FILES* = NIL, FILELST is used. For example, (MAKEFILES 'LIST) will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on FILELST, MAKEFILES calls ADDTOFILES? to allow you to specify where these go. MAKEFILES returns a list of all files that are made.

(**CLEANUP** $FILE_1$ $FILE_2$ ... $FILE_N$) [NLambda NoSpread Function]

Dumps, lists, and recompiles (with RECOMPILE or BRECOMPILE) any of the specified files (unevaluated) requiring the corresponding operation. If no files are specified, FILELST is used. CLEANUP returns NIL.

CLEANUP uses the value of the variable CLEANUPOPTIONS as the *OPTIONS* argument to MAKEFILE. CLEANUPOPTIONS is initially (RC), to indicate that the files should be recompiled. If CLEANUPOPTIONS is set to (RC F), no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if $FILE_1$ is a list, it will be interpreted as the list of options regardless of the value of CLEANUPOPTIONS.

(**FILES?**) [Function]

Prints on the terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of any functions and other typed definitions (if any) that are not contained in any file. If there are any, FILES? then calls ADDTOFILES? to allow you to specify where these go.

(**ADDTOFILES?** —) [Function]

Called from MAKEFILES, CLEANUP, and FILES? when there are typed definitions that have been marked as changed which do not belong to any file. ADDTOFILES? lists the names of the changed items, and asks if you want to specify where these items should be put. If you answer N(o), ADDTOFILES? returns NIL without taking any action. If you answer], this is taken to be an answer to each question that would be asked, and all the changed items are marked as dummy items to be ignored. Otherwise, ADDTOFILES? prints the name of each changed item, and accepts one of the following responses:

A file name

A filevar — If you give a file name or a variable whose value is a list (a filevar), the item is added to the corresponding file or list, using ADDTOFILE.

If your response is not the name of a file on FILELST or a variable whose value is a list, you will be asked whether it is a new file. If you say no, then ADDTOFILES? will check whether the item is the name of a list, i.e., whether its value is a list. If not, youwill be asked whether it is a new list.

line-feed — Same as your previous response.

space
carriage return — Take no action.

] — The item is marked as a dummy item by adding it to NILCOMS. This tells the file manager simply to ignore this item.

[— The "definition" of the item in question is prettyprinted to the terminal, and then you are asked again about its disposition.

(— ADDTOFILES? prompts with "LISTNAME: (", you type in the name of a list, i.e. a variable whose value is a list, terminated by a). The item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and you are asked again. For example, you define a new function FOO3. When asked where it goes, you type (FOOFNS). If the command (FNS * FOOFNS) is found, FOO3 will be added to the value of FOOFNS. If instead you type (FOOCOMS), and the command (COMS * FOOCOMS) is found, then FOO3 will be added to a command for dumping functions that is contained in FOOCOMS.

If the named list is not also the name of a file, you can simply type it in without parenthesis as described above.

@ — ADDTOFILES? prompts with "Near: (", you type in the name of an object, and the item is then inserted in a command for dumping objects (of its type) that contains the indicated name. The item is inserted immediately after the indicated name.

(**LISTFILES** *FILE*$_1$ *FILE*$_2$ ... *FILE*$_N$) [NLambda NoSpread Function]

Lists each of the specified files (unevaluated). If no files are given, NOTLISTEDFILES is used. Each file listed is removed from NOTLISTEDFILES if the listing is completed. For each file not found, LISTFILES prints the message *FILENAME* NOT FOUND and proceeds to the next file.

LISTFILES calls the function LISTFILES1 on each file to be listed. Normally, LISTFILES1 is defined to simply call SEND.FILE.TO.PRINTER (see Chapter 29), but you can advise or redefine LISTFILES1 for more specialized applications.

Any lists inside the argument list to LISTFILES are interpreted as property lists that set the various printing options, such as the printer, number of copies, banner page name, etc (see see Chapter 29). Later properties override earlier ones. For example,

```
(LISTFILES FOO (HOST JEDI) FUM (#COPIES 3) FIE)
```

will cause one copy of FOO to be printed on the default printer, and one copy of FUM and three copies of FIE to be printed on the printer JEDI.

(**COMPILEFILES** *FILE$_1$ FILE$_2$ ... FILE$_N$*) [NLambda NoSpread Function]

Executes the RC and C options of MAKEFILE for each of the specified files (unevaluated). If no files are given, NOTCOMPILEDFILES is used. Each file compiled is removed from NOTCOMPILEDFILES. If *FILE$_1$* is a list, it is interpreted as the *OPTIONS* argument to MAKEFILES. This feature can be used to supply an answer to the compiler's LISTING? question, e.g., (COMPILEFILES (STF)) will compile each file on NOTCOMPILEDFILES so that the functions are redefined without the EXPRs definitions being saved.

(**WHEREIS** *NAME TYPE FILES FN*) [Function]

TYPE is a file manager type. WHEREIS sweeps through all the files on the list *FILES* and returns a list of all files containing *NAME* as a *TYPE*. WHEREIS knows about and expands all file manager commands and file manager macros. *TYPE* = NIL defaults to FNS (to retrieve function definitions). If *FILES* is not a list, the value of FILELST is used.

If *FN* is given, it should be a function (with arguments *NAME*, *FILE*, and *TYPE*) which is applied for every file in *FILES* that contains *NAME* as a *TYPE*. In this case, WHEREIS returns NIL.

If the WHEREIS library package has been loaded, WHEREIS is redefined so that *FILES* = T means to use the whereis package data base, so WHEREIS will find *NAME* even if the file has not been loaded or noticed. *FILES* = NIL always means use FILELST.

Remaking a Symbolic File

Most of the time that a symbolic file is written using MAKEFILE, only a few of the functions that it contains have been changed since the last time the file was written. Rather than prettprinting all of the functions, it is often considerably faster to "remake" the file, copying the prettprinted definitions of unchanged functions from an earlier version of the symbolic file, and only prettyprinting those functions that have been changed.

MAKEFILE will remake the symbolic file if the REMAKE option is specified. If the NEW option is given, the file is not remade, and all of the functions are prettprinted. The default action is specified by the value of MAKEFILEREMAKEFLG: if T (its initial value), MAKEFILE will remake files unless the NEW

option is given; if NIL, MAKEFILE will not remake unless the REMAKE option is given.

Note: If the file has never been loaded or dumped, for example if the filecoms were simply set up in memory, then MAKEFILE will never attempt to remake the file, regardless of the setting of MAKEFILEREMAKEFLG, or whether the REMAKE option was specified.

When MAKEFILE is remaking a symbolic file, you can explicitly indicate the functions which are to be prettyprinted and the file to be used for copying the rest of the function definitions from via the *REPRINTFNS* and *SOURCEFILE* arguments to MAKEFILE. Normally, both of these arguments are defaulted to NIL. In this case, *REPRINTFNS* will be set to those functions that have been changed since the last version of the file was written. For *SOURCEFILE*, MAKEFILE obtains the full name of the most recent version of the file (that it knows about) from the FILEDATES property of the file, and checks to make sure that the file still exists and has the same file date as that stored on the FILEDATES property. If it does, MAKEFILE uses that file as *SOURCEFILE*. This procedure permits you to LOAD or LOADFROM a file in a different directory, and still be able to remake the file with MAKEFILE. In the case where the most recent version of the file cannot be found, MAKEFILE will attempt to remake using the *original* version of the file (i.e., the one first loaded), specifying as *REPRINTFNS* the union of all changes that have been made since the file was first loaded, which is obtained from the FILECHANGES property of the file. If both of these fail, MAKEFILE prints the message "CAN'T FIND EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF *FILE*, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, MAKEFILE also checks to see how the file was originally loaded. If the file was originally loaded as a compiled file, MAKEFILE will call LOADVARS to obtain those DECLARE: expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by LOADFNS (but not LOADFROM), then LOADVARS is called to obtain any non-DEFINEQ expressions. Before calling LOADVARS to re-load definitions, MAKEFILE asks you, e.g. "Only the compiled version of FOO was loaded, do you want to LOADVARS the (DECLARE: .. DONTCOPY ..) expressions from {DSK}<MYDIR>FOO.;3?". You can respond Yes to execute the LOADVARS and continue the MAKEFILE, No to proceed with the MAKEFILE without performing the LOADVARS, or Abort to abort the MAKEFILE. You may wish to skip the LOADVARS if you had circumvented the file manager in some way, and loading the old definitions would overwrite new ones.

Remaking a symbolic file is considerably faster if the earlier version has a *file map* indicating where the function definitions are located (see the File Maps section), but it does not depend on this information.

Loading Files in a Distributed Environment

Each Interlisp source and compiled code file contains the full filename of the file, including the host and directory names, in a FILECREATED expression at the beginning of the file. The compiled code file also contains the full file name of the source file it was created from. In earlier versions of Interlisp, the file manager used this information to locate the appropriate source file when "remaking" or recompiling a file.

This turned out to be a bad feature in distributed environments, where users frequently move files

from one place to another, or where files are stored on removable media. For example, suppose you MAKEFILE to a floppy, and then copy the file to a file server. If you loaded and edited the file from a file server, and tried to do MAKEFILE, it would try to locate the source file on the floppy, which is probably no longer loaded.

Currently, the file manager searches for sources file on the connected directory, and on the directory search path (on the variable DIRECTORIES). If it is not found, the host/directory information from the FILECREATED expression be used.

Warning: One situation where the new algorithm does the wrong thing is if you explicitly LOADFROM a file that is not on your directory search path. Future MAKEFILEs and CLEANUPs will search the connected directory and DIRECTORIES to find the source file, rather than using the file that the LOADFROM was done from. Even if the correct file is on the directory search path, you could still create a bad file if there is another version of the file in an earlier directory on the search path. In general, you should either explicitly specify the *SOURCEFILE* argument to MAKEFILE to tell it where to get the old source, or connect to the directory where the correct source file is.

Marking Changes

The file manager needs to know what typed definitions have been changed, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file manager operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file manager by marking changes. Also, *typed-in* assignment of variables or property values is noticed by the file manager. (If a program modifies a variable or property value, this is not noticed.) In some cases the marking procedure can be subtle, e.g. if you edit a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

The various system functions which create or modify objects call MARKASCHANGED to mark the object as changed. For example, when a function is defined via DEFINE or DEFINEQ, or modified via EDITF, or a DWIM correction, the function is marked as being a changed object of type FNS. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type RECORDS, and so on for all of the other file manager types.

You can also call MARKASCHANGED directly to mark objects of a particular file manager type as changed:

(**MARKASCHANGED** *NAME TYPE REASON*) [Function]

Marks *NAME* of type *TYPE* as being changed. MARKASCHANGED returns *NAME*. MARKASCHANGED is undoable.

REASON is a symbol that indicated how *NAME* was changed. MARKASCHANGED recognizes the following values for *REASON*:

DEFINED Used to indicate the creation of *NAME*, e.g. from DEFINEQ

(Chapter 10).

CHANGED Used to indicate a change to *NAME*, e.g. from the editor.

DELETED Used to indicate the deletion of *NAME*, e.g. by DELDEF.

CLISP Used to indicate the modification of *NAME* by CLISP translation.

For backwards compatibility, MARKASCHANGED also accepts a *REASON* of T (=DEFINED) and NIL (=CHANGED). New programs should avoid using these values.

The variable MARKASCHANGEDFNS is a list of functions that MARKASCHANGED calls (with arguments *NAME*, *TYPE*, and *REASON*). Functions can be added to this list to "advise" MARKASCHANGED to do additional work for all types of objects. The WHENCHANGED file manager type property (see the Defining New File Manager Types section) can be used to specify additional actions when MARKASCHANGED gets called on specific types of objects.

(**UNMARKASCHANGED** *NAME TYPE*) [Function]

Unmarks *NAME* of type *TYPE* as being changed. Returns *NAME* if *NAME* was marked as changed and is now unmarked, NIL otherwise. UNMARKASCHANGED is undoable.

(**FILEPKGCHANGES** *TYPE LST*) [NoSpread Function]

If *LST* is not specified (as opposed to being NIL), returns a list of those objects of type *TYPE* that have been marked as changed but not yet associated with their corresponding files (see the File Manager Types section). If *LST* is specified, FILEPKGCHANGES sets the corresponding list. (FILEPKGCHANGES) returns a list of *all* objects marked as changed as a list of elements of the form (*TYPENAME . CHANGEDOBJECTS*).

Some properties (e.g. EXPR, ADVICE, MACRO, I.S.OPR, etc.) are used to implement other file manager types. For example, if you change the value of the property I.S.OPR, you are really changing an object of type I.S.OPR. The effect is the same as though you had redefined the i.s.opr via a direct call to the function I.S.OPR. If a property whose value has been changed or added does not correspond to a specific file manager type, then it is marked as a changed object of type PROPS whose *name* is (*VARIABLENAME PROPNAME*) (except if the property name has a property PROPTYPE with value IGNORE).

Similarly, if you change a variable which implements the file manager type ALISTS (as indicated by the appearance of the property VARTYPE with value ALIST on the variable's property list), only those entries that are actually changed are marked as being changed objects of type ALISTS. The "name" of the object will be (*VARIABLENAME KEY*) where *KEY* is CAR of the entry on the alist that is being marked. If the variable corresponds to a specific file manager type other than ALISTS, e.g., USERMACROS, LISPXMACROS, etc., then an object of that type is marked. In this case, the name of the changed object will be CAR of the corresponding entry on the alist. For example, if you edit LISPXMACROS and change a definition for PL, then the object PL of type LISPXMACROS is marked as being changed.

Noticing Files

Already existing files are "noticed" by LOAD or LOADFROM (or by LOADFNS or LOADVARS when the *VARS* argument is T. New files are noticed when they are constructed by MAKEFILE, or when definitions are first associated with them via FILES? or ADDTOFILES?. Noticing a file updates certain lists and properties so that the file manager functions know to include the file in their operations. For example, CLEANUP will only dump files that have been noticed.

You can explicitly tell the file manager to notice a newly-created file by defining the filecoms for the file, and calling ADDFILE:

(**ADDFILE** *FILE*) [Function]

Tells the file manager that *FILE* should be recognized as a file; it adds *FILE* to FILELST, and also sets up the FILE property of *FILE* to reflect the current set of changes which are "registered against" *FILE*.

The file manager uses information stored on the property list of the root name of noticed files. The following property names are used:

FILE [Property Name]

When a file is noticed, the property FILE, value ((*FILECOMS* . *LOADTYPE*)) is added to the property list of its root name. *FILECOMS* is the variable containing the filecoms of the file. *LOADTYPE* indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with LOADFNS, loaded as a compiled file, etc.

The property FILE is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped. CDR of the FILE property records by type those items that have been changed since the last MAKEFILE. Whenever a file is dumped, these items are moved to the property FILECHANGES, and CDR of the FILE property is reset to NIL.

FILECHANGES [Property Name]

The property FILECHANGES contains a list of all changed items since the file was loaded (there may have been several sequences of editing and rewriting the file). When a file is dumped, the changes in CDR of the FILE property are added to the FILECHANGES property.

FILEDATES [Property Name]

The property FILEDATES contains a list of version numbers and corresponding file dates for this file. These version numbers and dates are used for various integrity checks in connection with *re*making a file.

FILEMAP [Property Name]

The property FILEMAP is used to store the filemap for the file. This is used to directly load individual functions from the middle of a file.

To compute the root name, ROOTFILENAME is applied to the name of the file as indicated in the FILECREATED expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. The file manager detects that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one FILECREATED expressions. In this case, each of the files mentioned in the following FILECREATED expressions are noticed. For example, if you perform (BCOMPL '(FOO FIE)), and subsequently loads FOO.DCOM, both FOO and FIE will be noticed.

When a file is noticed, its root name is added to the list FILELST:

FILELST [Variable]

Contains a list of the root names of the files that have been noticed.

LOADEDFILELST [Variable]

Contains a list of the actual names of the files as loaded by LOAD, LOADFNS, etc. For example, if you perform (LOAD '<NEWLISP>EDITA.COM;3), EDITA will be added to FILELST, but <NEWLISP>EDITA.COM;3 is added to LOADEDFILELST. LOADEDFILELST is not used by the file manager; it is maintained solely for your benefit.

Distributing Change Information

Periodically, the function UPDATEFILES is called to find which file(s) contain the elements that have been changed. UPDATEFILES is called by FILES?, CLEANUP, and MAKEFILES, i.e., any procedure that requires the FILE property to be up to date. This procedure is followed rather than updating the FILE property after each change because scanning FILELST and examining each file manager command can be a time-consuming process; this is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

UPDATEFILES operates by scanning FILELST and interrogating the file manager commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property FILE for the corresponding file. Thus, after UPDATEFILES has completed operating, the files that need to be dumped are simply those files on FILELST for which CDR of their FILE property is non-NIL. For example, if you load the file FOO containing definitions for FOO1, FOO2, and FOO3, edit FOO2, and then call UPDATEFILES, (GETPROP 'FOO 'FILE) will be ((FOOCOMS . T) (FNS FOO2)). If any objects marked as changed have not been transferred to the FILE property for some file, e.g., you define a new function but forget (or declines) to add it to the file manager commands for the corresponding file, then both FILES? and CLEANUP will print warning messages, and then call ADDTOFILES? to permit you to specify on which files these items belong.

You can also invoke UPDATEFILES directly:

(**UPDATEFILES** — —) [Function]

(UPDATEFILES) will update the FILE properties of the noticed files.

File Manager Types

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file manager uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a symbol), a definition of a given type (called the file manager type). Note that the same name may have several definitions of different types. For example, a symbol may have both a function definition and a variable definition. The file manager also keeps track of the file that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

A file manager type is an abstract notion of a class of objects which share the property that every object of the same file manager type is stored, retrieved, edited, copied etc., by the file manager in the same way. Each file manager type is identified by a symbol, which can be given as an argument to the functions that manipulate typed definitions. You may define new file manager types, as described in the Defining New Package Types section.

FILEPKGTYPES [Variable]

The value of FILEPKGTYPES is a list of all file manager types, including any that you may have defined.

The file manager is initialized with the following built-in file manager types:

ADVICE [File Manager Type]

Used to access "advice" modifying a function (see Chapter 15).

ALISTS [File Manager Type]

Used to access objects stored on an association list that is the value of a symbol (see Chapter 3).

A variable is declared to have an association list as its value by putting on its property list the property VARTYPE with value ALIST. In this case, each dotted pair on the list is an object of type ALISTS. When the value of such a variable is changed, only those entries in the association list that are actually changed or added are marked as changed objects of type ALISTS (with "name" (SYMBOL *KEY*)). Objects of type ALISTS are dumped via the ALISTS or ADDVARS file manager commands.

Note that some association lists are used to "implement" other file manager types. For example, the value of the global variable USERMACROS implements the file manager type USERMACROS and the values of LISPXMACROS and LISPXHISTORYMACROS implement the file manager type LISPXMACROS. This is indicated by putting on the property list of the variable the property VARTYPE with value a list of the form (ALIST *FILEPKGTYPE*). For example, (GETPROP 'LISPXHISTORYMACROS 'VARTYPE) => (ALIST LISPXMACROS).

COURIERPROGRAMS [File Manager Type]

Used to access Courier programs (see Chapter 31).

EXPRESSIONS [File Manager Type]

Used to access lisp expressions that are put on a file by using the REMEMBER programmers assistant command (Chapter 13), or by explicitly putting the P file manager command on the filecoms.

FIELDS [File Manager Type]

Used to access fields of records. The "definition" of an object of type FIELDS is a list of all the record declarations which contain the name. See Chapter 8.

FILEPKGCOMS [File Manager Type]

Used to access file manager commands and types. A single name can be defined both as a file manager type and a file manager command. The "definition" of an object of type FILEPKGCOMS is a list structure of the form ((COM . *COMPROPS*) (TYPE . *TYPEPROPS*)), where *COMPROPS* is a property list specifying how the name is defined as a file manager command by FILEPKGCOM (see the Defining New File Manager Commands section), and *TYPEPROPS* is a property list specifying how the name is defined as a file manager type by FILEPKGTYPE (see the Defining New File Manager Types section).

FILES [File Manager Type]

Used to access files. This file manager type is most useful for renaming files. The "definition" of a file is not a useful structure.

FILEVARS [File Manager Type]

Used to access Filevars (see the FileVars section).

FNS [File Manager Type]

Used to access function definitions.

I.S.OPRS [File Manager Type]

Used to access the definitions of iterative statement operators (see Chapter 9).

LISPXMACROS [File Manager Type]

Used to access programmer's assistant commands defined on the variables LISPXMACROS and LISPXHISTORYMACROS (see Chapter 13).

MACROS [File Manager Type]

Used to access macro definitions (see Chapter 10).

PROPS [File Manager Type]

Used to access objects stored on the property list of a symbol (see Chapter 2). When a

property is changed or added, an object of type PROPS, with "name" (SYMBOL *PROPNAME*) is marked as being changed.

Note that some symbol properties are used to implement other file manager types. For example, the property MACRO implements the file manager type MACROS, the property ADVICE implements ADVICE, etc. This is indicated by putting the property PROPTYPE, with value of the file manager type on the property list of the property name. For example, (GETPROP 'MACRO 'PROPTYPE) => MACROS. When such a property is changed or added, an object of the corresponding file manager type is marked. If (GETPROP *PROPNAME* 'PROPTYPE) => IGNORE, the change is ignored. The FILE, FILEMAP, FILEDATES, etc. properties are all handled this way. (IGNORE cannot be the name of a file manager type implemented as a property).

RECORDS [File Manager Type]

Used to access record declarations (see Chapter 8).

RESOURCES [File Manager Type]

Used to access resources (see Chapter 12).

TEMPLATES [File Manager Type]

Used to access Masterscope templates (see Chapter 19).

USERMACROS [File Manager Type]

Used to access user edit macros (see Chapter 16).

VARS [File Manager Type]

Used to access top-level variable values.

Functions for Manipulating Typed Definitions

The functions described below can be used to manipulate typed definitions, without needing to know how the manipulations are done. For example, (GETDEF 'FOO 'FNS) will return the function definition of FOO, (GETDEF 'FOO 'VARS) will return the variable value of FOO, etc. All of the functions use the following conventions:

1. All functions which make destructive changes are undoable.

2. Any argument that expects a list of symbols will also accept a single symbol, operating as though it were enclosed in a list. For example, if the argument *FILES* should be a list of files, it may also be a single file.

3. *TYPE* is a file manager type. *TYPE* = NIL is equivalent to *TYPE* = FNS. The singular form of a file manager type is also recognized, e.g. *TYPE* = VAR is equivalent to *TYPE* = VARS.

4. *FILES* = NIL is equivalent to *FILES* = FILELST.

5. *SOURCE* is used to indicate the source of a definition, that is, where the definition should be found. *SOURCE* can be one of:

CURRENT Get the definition currently in effect.

SAVED Get the "saved" definition, as stored by SAVEDEF.

FILE Get the definition contained on the (first) file determined by WHEREIS.

WHEREIS is called with *FILES* = T, so that if the WHEREIS library package is loaded, the WHEREIS data base will be used to find the file containing the definition.

? Get the definition currently in effect if there is one, else the saved definition if there is one, otherwise the definition from a file determined by WHEREIS. Like specifying CURRENT, SAVED, and FILE in order, and taking the first definition that is found.

a file name
a list of file names Get the definition from the first of the indicated files that contains one.

NIL In most cases, giving *SOURCE* = NIL (or not specifying it at all) is the same as giving ?, to get either the current, saved, or filed definition. However, with HASDEF, *SOURCE* = NIL is interpreted as equal to *SOURCE* = CURRENT, which only tests if there is a current definition.

The operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file manager type using the function FILEPKGTYPE, described in the Defining New File Manager Types section.

(**GETDEF** *NAME TYPE SOURCE OPTIONS*) [Function]

Returns the definition of *NAME*, of type *TYPE*, from *SOURCE*. For most types, GETDEF returns the expression which would be pretty printed when dumping *NAME* as *TYPE*. For example, for *TYPE* = FNS, an EXPR definition is returned, for *TYPE* = VARS, the value of *NAME* is returned, etc.

OPTIONS is a list which specifies certain options:

NOERROR GETDEF causes an error if an appropriate definition cannot be found, unless *OPTIONS* is or contains NOERROR. In this case, GETDEF returns the value of the NULLDEF file manager type property (see the Defining New File Manager Types section), usually NIL.

a string If *OPTIONS* is or contains a string, that string will be returned if no definition is found (and NOERROR is not among the options). The caller can thus determine whether

a definition was found, even for types for which NIL or NOBIND are acceptable definitions.

NOCOPY GETDEF returns a copy of the definition unless *OPTIONS* is or contains NOCOPY.

EDIT If *OPTIONS* is or contains EDIT, GETDEF returns a copy of the definition unless it is possible to edit the definition "in place." With some file manager types, such as functions, it is meaningful (and efficient) to edit the definition by destructively modifying the list structure, without calling PUTDEF. However, some file manager types (like records) need to be "installed" with PUTDEF after they are edited. The default EDITDEF (see the Defining New File Manager Types section) calls GETDEF with *OPTIONS* of (EDIT NOCOPY), so it doesn't use a copy unless it has to, and only calls PUTDEF if the result of editing is not EQUAL to the old definition.

NODWIM A FNS definition will be dwimified if it is likely to contain CLISP unless *OPTIONS* is or contains NODWIM.

(**PUTDEF** *NAME TYPE DEFINITION REASON*) [Function]

Defines *NAME* of type *TYPE* with *DEFINITION*. For *TYPE* = FNS, does a DEFINE; for *TYPE* = VARS, does a SAVESET, etc.

For *TYPE* = FILES, PUTDEF establishes the command list, notices *NAME*, and then calls MAKEFILE to actually dump the file *NAME*, copying functions if necessary from the "old" file (supplied as part of *DEFINITION*).

PUTDEF calls MARKASCHANGED (see the Mrking Changes section) to mark *NAME* as changed, giving a reason of *REASON*. If *REASON* is NIL, the default is DEFINED.

If *TYPE* = FNS, PUTDEF prints a warning if you try to redefine a function on the list UNSAFE.TO.MODIFY.FNS (see Chapter 10).

(**HASDEF** *NAME TYPE SOURCE SPELLFLG*) [Function]

Returns (OR *NAME* T) if *NAME* is the name of something of type *TYPE*. If not, attempts spelling correction if *SPELLFLG* = T, and returns the spelling-corrected *NAME*. Otherwise returns NIL. HASDEF for type FNS (or NIL) indicates that *NAME* has an editable source definition. If *NAME* is a function that exists on a file for which you have loaded only the compiled version and not the source, HASDEF returns NIL.

(HASDEF NIL *TYPE*) returns T if NIL has a valid definition.

If *SOURCE* = NIL, *HASDEF* interprets this as equal to *SOURCE* = CURRENT, which only tests if there is a current definition.

(**TYPESOF** *NAME POSSIBLETYPES IMPOSSIBLETYPES SOURCE*) [Function]

Returns a list of the types in *POSSIBLETYPES* but not in *IMPOSSIBLETYPES* for which

NAME has a definition. FILEPKGTYPES is used if *POSSIBLETYPES* is NIL.

(**COPYDEF** *OLD NEW TYPE SOURCE OPTIONS*) [Function]

Defines *NEW* to have a copy of the definition of *OLD* by doing PUTDEF on a copy of the definition retrieved by (GETDEF *OLD TYPE SOURCE OPTIONS*). *NEW* is substituted for *OLD* in the copied definition, in a manner that may depend on the *TYPE*.

For example, (COPYDEF 'PDQ 'RST 'FILES) sets up RSTCOMS to be a copy of PDQCOMS, changes things like (VARS * PDQVARS) to be (VARS * RSTVARS) in RSTCOMS, and performs a MAKEFILE on RST such that the appropriate definitions get copied from PDQ.

COPYDEF disables the NOCOPY option of GETDEF, so *NEW* will always have a *copy* of the definition of *OLD*.

COPYDEF substitutes *NEW* for *OLD* throughout the definition of *OLD*. This is usually the right thing to do, but in some cases, e.g., where the old name appears within a quoted expression but was not used in the same context, you must re-edit the definition.

(**DELDEF** *NAME TYPE*) [Function]

Removes the definition of *NAME* as a *TYPE* that is currently in effect.

(**SHOWDEF** *NAME TYPE FILE*) [Function]

Prettyprints the definition of *NAME* as a *TYPE* to *FILE*. This shows you how *NAME* would be written to a file. Used by ADDTOFILES? (see the Storing Files section).

(**EDITDEF** *NAME TYPE SOURCE EDITCOMS*) [Function]

Edits the definition of *NAME* as a *TYPE*. Essentially performs

```
(PUTDEF NAME TYPE
   (EDITE (GETDEF NAME TYPE SOURCE)
          EDITCOMS))
```

(**SAVEDEF** *NAME TYPE DEFINITION*) [Function]

Sets the "saved" definition of *NAME* as a *TYPE* to *DEFINITION*. If *DEFINITION* = NIL, the current definition of *NAME* is saved.

If *TYPE* = FNS (or NIL), the function definition is saved on *NAME*'s property list under the property EXPR, or CODE (depending on the FNTYP of the function definition). If (GETD *NAME*) is non-NIL, but (FNTYP *FN*) = NIL, SAVEDEF saves the definition on the property name LIST. This can happen if a function was somehow defined with an illegal expr definition, such as (LAMMMMDA (X) ...).

If *TYPE* = VARS, the definition is stored as the value of the VALUE property of *NAME*. For other types, the definition is stored in an internal data structure, from where it can be retrieved by GETDEF or UNSAVEDEF.

(**UNSAVEDEF** *NAME TYPE*) [Function]

Restores the "saved" definition of *NAME* as a *TYPE*, making it be the current definition.

Returns *PROP*.

If *TYPE* = FNS (or NIL), UNSAVEDEF unsaves the function definition from the EXPR property if any, else CODE, and returns the property name used. UNSAVEDEF also recognizes *TYPE* = EXPR, CODE, or LIST, meaning to unsave the definition only from the corresponding property only.

If DFNFLG is not T (see Chapter 10), the current definition of *NAME*, if any, is saved using SAVEDEF. Thus one can use UNSAVEDEF to switch back and forth between two definitions.

(**LOADDEF** *NAME TYPE SOURCE*) [Function]

Equivalent to (PUTDEF *NAME TYPE* (GETDEF *NAME TYPE SOURCE*)). LOADDEF is essentially a generalization of LOADFNS, e.g. it enables loading a single record declaration from a file. (LOADDEF *FN*) will give *FN* an EXPR definition, either obtained from its property list or a file, unless it already has one.

(**CHANGECALLERS** *OLD NEW TYPES FILES METHOD*) [Function]

Finds all of the places where *OLD* is used as any of the types in *TYPES* and changes those places to use *NEW*. For example, (CHANGECALLERS 'NLSETQ 'ERSETQ) will change all calls to NLSETQ to be calls to ERSETQ. Also changes occurrences of *OLD* to *NEW* inside the filecoms of any file, inside record declarations, properties, etc.

CHANGECALLERS attempts to determine if *OLD* might be used as more than one type; for example, if it is both a function and a record field. If so, rather than performing the transformation *OLD* -> *NEW* automatically, you are allowed to edit all of the places where *OLD* occurs. For each occurrence of *OLD*, you are asked whether you want to make the replacement. If you respond with anything except Yes or No, the editor is invoked on the expression containing that occurrence.

There are two different methods for determining which functions are to be examined. If *METHOD* = EDITCALLERS, EDITCALLERS is used to search *FILES* (see Chapter 16). If *METHOD* = MASTERSCOPE, then the Masterscope database is used instead. *METHOD* = NIL defaults to MASTERSCOPE if the value of the variable DEFAULTRENAMEMETHOD is MASTERSCOPE and a Masterscope database exists, otherwise it defaults to EDITCALLERS.

(**RENAME** *OLD NEW TYPES FILES METHOD*) [Function]

First performs (COPYDEF *OLD NEW TYPE*) for all *TYPE* inside *TYPES*. It then calls CHANGECALLERS to change all occurrences of *OLD* to *NEW*, and then "deletes" *OLD* with DELDEF. For example, if you have a function FOO which you now wish to call FIE, simply perform (RENAME 'FOO 'FIE), and FIE will be given FOO's definition, and all places that FOO are called will be changed to call FIE instead.

METHOD is interpreted the same as the *METHOD* argument to CHANGECALLERS, above.

(**COMPARE** $NAME_1$ $NAME_2$ *TYPE* $SOURCE_1$ $SOURCE_2$) [Function]

Compares the definition of $NAME_1$ with that of $NAME_2$, by calling COMPARELISTS (Chapter 3) on (GETDEF $NAME_1$ *TYPE* $SOURCE_1$) and (GETDEF $NAME_2$ *TYPE* $SOURCE_2$),

which prints their differences on the terminal.

For example, if the current value of the variable A is (A B C (D E F) G), and the value of the variable B on the file <lisp>FOO is (A B C (D F E) G), then:

```
_(COMPARE 'A 'B 'VARS 'CURRENT '<lisp>FOO)
A from CURRENT and B from <lisp>TEST differ:
(E -> F) (F -> E)
T
```

(**COMPAREDEFS** *NAME TYPE SOURCES*) [Function]

Calls COMPARELISTS (Chapter 3) on all pairs of definitions of *NAME* as a *TYPE* obtained from the various *SOURCES* (interpreted as a list of source specifications).

Defining New File Manager Types

All manipulation of typed definitions in the file manager is done using the type-independent functions GETDEF, PUTDEF, etc. Therefore, to define a new file manager type, it is only necessary to specify (via the function FILEPKGTYPE) what these functions should do when dealing with a typed definition of the new type. Each file manager type has the following properties, whose values are functions or lists of functions:

These functions are defined to take a *TYPE* argument so that you may have the same function for more than one type.

GETDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *OPTIONS*, which should return the current definition of *NAME* as a type *TYPE*. Used by GETDEF (see the Functions for Manipulating Typed Definitions section), which passes its *OPTIONS* argument.

If there is no GETDEF property, a file manager command for dumping *NAME* is created (by MAKENEWCOM). This command is then used to write the definition of *NAME* as a type *TYPE* onto the file FILEPKG.SCRATCH (in Medley, this file is created on the {CORE} device). This expression is then read back in and returned as the current definition.

In some situations, the function HASDEF needs to call GETDEF to determine whether a definition exists. In this case, *OPTIONS* will include the symbol HASDEF, and it is permissable for a GETDEF function to return T or NIL, rather than creating a complex structure which will not be used.

NULLDEF [File Manager Type Property]

The value of the NULLDEF property is returned by GETDEF (see the Functions for Manipulating Typed Definitions section) when there is no definition and the NOERROR option is supplied. For example, the NULLDEF of VARS is NOBIND.

FILEGETDEF [File Manager Type Property]

This enables you to provide a way of obtaining definitions from a file that is more efficient than the default procedure used by GETDEF (see the Functions for Manipulating Typed

Definitions section). Value is a function of four arguments, *NAME*, *TYPE*, *FILE*, and *OPTIONS*. The function is applied by GETDEF when it is determined that a typed definition is needed from a particular file. The function must open and search the given file and return any *TYPE* definition for *NAME* that it finds.

CANFILEDEF [File Manager Type Property]

If the value of this property is non-NIL, this indicates that definitions of this file manager type are not loaded when a file is loaded with LOADFROM (see the Loading Files section). The default is NIL. Initially, only FNS has this property set to non-NIL.

PUTDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *DEFINITION*, which should store *DEFINITION* as the definition of *NAME* as a type *TYPE*. Used by PUTDEF (see the Functions for Manipulating Typed Definitions section).

HASDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *SOURCE*, which should return (OR *NAME* T) if *NAME* is the name of something of type *TYPE*. *SOURCE* is as interpreted by HASDEF (see the Functions for Manipulating Typed Definitions section), which uses this property.

EDITDEF [File Manager Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *SOURCE*, and *EDITCOMS*, which should edit the definition of *NAME* as a type *TYPE* from the source *SOURCE*, interpreting the edit commands *EDITCOMS*. If sucessful, should return *NAME* (or a spelling-corrected *NAME*). If it returns NIL, the "default" editor is called. Used by EDITDEF (see the Functions for Manipulating Typed Definitions section).

DELDEF [File Manager Type Property]

Value is a function of two arguments, *NAME*, and *TYPE*, which removes the definition of *NAME* as a *TYPE* that is currently in effect. Used by DELDEF (see the Functions for Manipulating Typed Definitions section).

NEWCOM [File Manager Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *LISTNAME*, and *FILE*. Specifies how to make a new (instance of a) file manager command to dump *NAME*, an object of type *TYPE*. The function should return the new file manager command. Used by ADDTOFILE and SHOWDEF.

If *LISTNAME* is non-NIL, this means that you specified *LISTNAME* as the filevar in interaction with ADDTOFILES? (see the FileVars section).

If no NEWCOM is specified, the default is to call DEFAULTMAKENEWCOM, which will construct and return a command of the form (*TYPE NAME*). You can advise or redefine DEFAULTMAKENEWCOM.

WHENCHANGED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *REASON* when *NAME*, an instance of type *TYPE*, is changed or defined (see MARKASCHANGED, in the Marking Changes section). Used for various applications, e.g. when an object of type I.S.OPRS changes, it is necessary to clear the corresponding translatons from CLISPARRAY.

The WHENCHANGED functions are called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and execute (RETFROM 'MARKASCHANGED).

The *REASON* argument passed to WHENCHANGED functions is either DEFINED or CHANGED.

WHENFILED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is added to *FILE*.

WHENUNFILED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is removed from *FILE*.

DESCRIPTION [File Manager Type Property]

Value is a string which describes instances of this type. For example, for type RECORDS, the value of DESCRIPTION is the string "record declarations".

The function FILEPKGTYPE is used to define new file manager types, or to change the properties of existing types. It is possible to redefine the attributes of system file manager types, such as FNS or PROPS.

(**FILEPKGTYPE** *TYPE* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N) [NoSpread Function]

Nospread function for defining new file manager types, or changing properties of existing file manager types. $PROP_i$ is one of the property names given above; VAL_i is the value to be given to that property. Returns *TYPE*.

(FILEPKGTYPE *TYPE PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *TYPE*) returns a property list of all of the defined properties of *TYPE*, using the property names as keys.

Specifying *TYPE* as the symbol TYPE can be used to define one file manager type as a synonym of another. For example, (FILEPKGTYPE 'R 'TYPE 'RECORDS) defines R as a synonym for the file manager type RECORDS.

File Manager Commands

 The basic mechanism for creating symbolic files is the function MAKEFILE (see the Storing Files

section). For each file, the file manager has a data structure known as the "filecoms", which specifies what typed descriptions are contained in the file. A filecoms is a list of file manager commands, each of which specifies objects of a certain file manager type which should be dumped. For example, the filecoms

```
((FNS FOO)
   (VARS FOO BAR BAZ)
   (RECORDS XYZZY))
```

has a FNS, a VARS, and a RECORDS file manager command. This filecoms specifies that the function definition for FOO, the variable values of FOO, BAR, and BAZ, and the record declaration for XYZZY should be dumped.

By convention, the filecoms of a file *X* is stored as the value of the symbol *X*COMS. For example, (MAKEFILE 'FOO.;27) will use the value of FOOCOMS as the filecoms. This variable can be directly manipulated, but the file manager contains facilities which make constructing and updating filecoms easier, and in some cases automatic (see the Functions for Manipulating File Command Lists section).

A file manager command is an instruction to MAKEFILE to perform an explicit, well-defined operation, usually printing an expression. Usually there is a one-to-one correspondence between file manager types and file manager commands; for each file manager type, there is a file manager command which is used for writing objects of that type to a file, and each file manager command is used to write objects of a particular type. However, in some cases, the same file manager type can be dumped by several different file manager commands. For example, the file manager commands PROP, IFPROP, and PROPS all dump out objects with the file manager type PROPS. This means if you change an object of file manager type PROPS via EDITP, a typed-in call to PUTPROP, or via an explicit call to MARKASCHANGED, this object can be written out with any of the above three commands. Thus, when the file manager attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three file manager commands PROP, IFPROP, and PROPS, to see if the corresponding atom and property are specified. It is also permissible for a single file manager command to dump several different file manager types. For example, you can define a file manager command which dumps both a function definition and its macro. Conversely, some file manager comands do not dump any file manager types at all, such as the E command.

For each file manager command, the file manager must be able to determine what typed definitions the command will cause to be printed so that the file manager can determine on what file (if any) an object of a given type is contained (by searching through the filecoms). Similarly, for each file manager type, the file manager must be able to construct a command that will print out an object of that type. In other words, the file manager must be able to map file manager commands into file manager types, and vice versa. Information can be provided to the file manager about a particular file manager command via the function FILEPKGCOM (see the Defining New File Manager Commands section), and information about a particular file manager type via the function FILEPKGTYPE (see the prior section). In the absence of other information, the default is simply that a file manager command of the form (*X NAME*) prints out the definition of *NAME* as a type *X*, and, conversely, if *NAME* is an object of type *X*, then *NAME* can be written out by a command of the form (*X NAME*).

If a file manager function is given a command or type that is not defined, it attempts spelling correction using FILEPKGCOMSPLST as a spelling list (unless DWIMFLG or NOSPELLFLG = NIL; see

Chapter 20). If successful, the corrected version of the list of file manager commands is written (again) on the output file, since at this point, the uncorrected list of file manager commands would already have been printed on the output file. When the file is loaded, this will result in *FILE*COMS being reset, and may cause a message to be printed, e.g., (FOOCOMS RESET). The value of FOOCOMS would then be the corrected version. If the spelling correction is unsuccessful, the file manager functions generate an error, BAD FILE PACKAGE COMMAND.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, advised functions, edit macros, record declarations, etc. The interpretation of each file manager command is documented in the following sections.

(**USERMACROS** $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Each symbol $SYMBOL_i$ is the name of a user edit macro. Writes expressions to add the edit macro definitions of $SYMBOL_i$ to USERMACROS, and adds the names of the commands to the appropriate spelling lists.

If $SYMBOL_i$ is not a user macro, a warning message "no EDIT MACRO for $SYMBOL_i$" is printed.

Functions and Macros

(**FNS** FN_1 ... FN_N) [File Manager Command]

Writes a DEFINEQ expression with the function definitions of FN_1 ... FN_N.

You should never print a DEFINEQ expression directly onto a file (by using the P file manager command, for example), because MAKEFILE generates the filemap of function definitions from the FNS file manager commands (see the File Maps section).

(**ADVISE** FN_1 ... FN_N) [File Manager Command]

For each function FN_i, writes expressions to reinstate the function to its advised state when the file is loaded. See Chapter 15.

When advice is applied to a function programmatically or by hand, it is additive. That is, if a function already has some advice, further advice is added to the already-existing advice. However, when advice is applied to a function as a result of loading a file with an ADVISE file manager command, the new advice replaces any earlier advice. ADVISE works this way to prevent problems with loading different versions of the same advice. If you really want to apply additive advice, a file manager command such as (P (ADVISE ...)) should be used (see the Miscellaneous File Manager Commands section).

(**ADVICE** FN_1 ... FN_N) [File Manager Command]

For each function FN_i, writes a PUTPROPS expression which will put the advice back on the property list of the function. You can then use READVISE (see Chapter 15) to reactivate the advice.

(**MACROS** *SYMBOL$_1$ ... SYMBOL$_N$*) [File Manager Command]

Each *SYMBOL$_i$* is a symbol with a MACRO definition (and/or a DMACRO, 10MACRO, etc.). Writes out an expression to restore all of the macro properties for each *SYMBOL$_i$*, embedded in a DECLARE: EVAL@COMPILE so the macros will be defined when the file is compiled. See Chapter 10.

Variables

(**VARS** *VAR$_1$ ... VAR$_N$*) [File Manager Command]

For each *VAR$_i$*, writes an expression to set its top level value when the file is loaded. If *VAR$_i$* is atomic, VARS writes out an expression to set *VAR$_i$* to the top-level value it had at the time the file was written. If *VAR$_i$* is non-atomic, it is interpreted as (*VAR FORM*), and VARS write out an expression to set *VAR* to the value of *FORM* (evaluated when the file is loaded).

VARS prints out expressions using RPAQQ and RPAQ, which are like SETQQ and SETQ except that they also perform some special operations with respect to the file manager (see the Functions Used within Source Files section).

VARS cannot be used for putting arbitrary variable values on files. For example, if the value of a variable is an array (or many other data types), a symbol which represents the array is dumped in the file instead of the array itself. The HORRIBLEVARS file manager command provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

(**INITVARS** *VAR$_1$ ... VAR$_N$*) [File Manager Command]

INITVARS is used for initializing variables, setting their values only when they are currently NOBIND. A variable value defined in an INITVARS command will not change an already established value. This means that re-loading files to get some other information will not automatically revert to the initialization values.

The format of an INITVARS command is just like VARS. The only difference is that if *VAR$_i$* is atomic, the current value is not dumped; instead NIL is defined as the initialization value. Therefore, (INITVARS FOO (FUM 2)) is the same as (VARS (FOO NIL)(FUM 2)), if FOO and FUM are both NOBIND.

INITVARS writes out an RPAQ? expression on the file instead of RPAQ or RPAQQ.

(**ADDVARS** (*VAR$_1$* . *LST$_1$*) ... (*VAR$_N$* . *LST$_N$*)) [File Manager Command]

For each (*VAR$_i$* . *LST$_i$*), writes an ADDTOVAR (see the Functions Used Within Source Files section) to add each element of *LST$_i$* to the list that is the value of *VAR$_i$* at the time the file is loaded. The new value of *VAR$_i$* will be the union of its old value and *LST$_i$*. If the value of *VAR$_i$* is NOBIND, it is first set to NIL.

For example, (ADDVARS (DIRECTORIES LISP LISPUSERS)) will add LISP and

LISPUSERS to the value of DIRECTORIES.

If LST_i is not specified, VAR_i is initialized to NIL if its current value is NOBIND. In other words, (ADDVARS (*VAR*)) will initialize *VAR* to NIL if *VAR* has not previously been set.

(**APPENDVARS** (VAR_1 . LST_1) ... (VAR_N . LST_N)) [File Manager Command]

The same as ADDVARS, except that the values are added to the end of the lists (using APPENDTOVAR, in the Functions Used Within Source Files section), rather than at the beginning.

(**UGLYVARS** VAR_1 ... VAR_N) [File Manager Command]

Like VARS, except that the value of each VAR_i may contain structures for which READ is not an inverse of PRINT, e.g. arrays, readtables, user data types, etc. Uses HPRINT (see Chapter 25).

(**HORRIBLEVARS** VAR_1 ... VAR_N) [File Manager Command]

Like UGLYVARS, except structures may also contain circular pointers. Uses HPRINT (see Chapter 25). The values of VAR_1 ... VAR_N are printed in the same operation, so that they may contain pointers to common substructures.

UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, UGLYVARS should be used instead of HORRIBLEVARS.

(**ALISTS** (VAR_1 KEY_1 KEY_2 ...) ... (VAR_N KEY_3 KEY_4 ...)) [File Manager Command]

VAR_i is a variable whose value is an association list, such as EDITMACROS, BAKTRACELST, etc. For each VAR_i, ALISTS writes out expressions which will restore the values associated with the specified keys. For example, (ALISTS (BREAKMACROS BT BTV)) will dump the definition for the BT and BTV commands on BREAKMACROS.

Some association lists (USERMACROS, LISPXMACROS, etc.) are used to implement other file manager types, and they have their own file manager commands.

(**SPECVARS** VAR_1 ... VAR_N) [File Manager Command]

(**LOCALVARS** VAR_1 ... VAR_N) [File Manager Command]

(**GLOBALVARS** VAR_1 ... VAR_N) [File Manager Command]

Outputs the corresponding compiler declaration embedded in a DECLARE: DOEVAL@COMPILE DONTCOPY. See Chapter 18.

(**CONSTANTS** VAR_1 ... VAR_N) [File Manager Command]

Like VARS, for each VAR_i writes an expression to set its top level value when the file is loaded. Also writes a CONSTANTS expression to declare these variables as constants (see Chapter 18). Both of these expressions are wrapped in a (DECLARE: EVAL@COMPILE

`...)` expression, so they can be used by the compiler.

Like `VARS`, VAR_i can be non-atomic, in which case it is interpreted as `(`*VAR FORM*`)`, and passed to `CONSTANTS` (along with the variable being initialized to *FORM*).

Symbol Properties

(**PROP** *PROPNAME* $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Writes a `PUTPROPS` expression to restore the value of the *PROPNAME* property of each symbol $SYMBOL_i$ when the file is loaded.

If *PROPNAME* is a list, expressions will be written for each property on that list. If *PROPNAME* is the symbol `ALL`, the values of all user properties (on the property list of each $SYMBOL_i$) are saved. `SYSPROPS` is a list of properties used by system functions. Only properties *not* on that list are dumped when the `ALL` option is used.

If $SYMBOL_i$ does not have the property *PROPNAME* (as opposed to having the property with value `NIL`), a warning message "`NO` *PROPNAME* `PROPERTY FOR` $SYMBOL_i$" is printed. The command `IFPROP` can be used if it is not known whether or not an atom will have the corresponding property.

(**IFPROP** *PROPNAME* $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Same as the `PROP` file manager command, except that it only saves the properties that actually appear on the property list of the corresponding atom. For example, if `FOO1` has property `PROP1` and `PROP2`, `FOO2` has `PROP3`, and `FOO3` has property `PROP1` and `PROP3`, then `(IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3)` will save only those five property values.

(**PROPS** (*SYMBOL*$_1$ *PROPNAME*$_1$)...(*SYMBOL*$_N$ *PROPNAME*$_N$)) [File Manager Command]

Similar to `PROP` command. Writes a `PUTPROPS` expression to restore the value of $PROPNAME_i$ for each $SYMBOL_i$ when the file is loaded.

As with the `PROP` command, if $SYMBOL_i$ does not have the property *PROPNAME* (as opposed to having the property with `NIL` value), a warning message "`NO` $PROPNAME_i$ `PROPERTY FOR` $SYMBOL_i$" is printed.

Miscellaneous File Manager Commands

(**RECORDS** REC_1 ... REC_N) [File Manager Command]

Each REC_i is the name of a record (see Chapter 8). Writes expressions which will redeclare the records when the file is loaded.

(**INITRECORDS** REC_1 ... REC_N) [File Manager Command]

Similar to `RECORDS`, `INITRECORDS` writes expressions on a file that will, when loaded,

perform whatever initialization/allocation is necessary for the indicated records. However, the record declarations themselves are not written out. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(**LISPXMACROS** *SYMBOL$_1$ ... SYMBOL$_N$*) [File Manager Command]

Each *SYMBOL$_i$* is defined on LISPXMACROS or LISPXHISTORYMACROS (see Chapter 13). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to LISPXCOMS

(**I.S.OPRS** *OPR$_1$ ... OPR$_N$*) [File Manager Command]

Each *OPR$_i$* is the name of a user-defined i.s.opr (see Chapter 9). Writes expressions which will redefine the i.s.oprs when the file is loaded.

(**RESOURCES** *RESOURCE$_1$ ... RESOURCE$_N$*) [File Manager Command]

Each *RESOURCES$_i$* is the name of a resource (see Chapter 12). Writes expressions which will redeclare the resource when the file is loaded.

(**INITRESOURCES** *RESOURCE$_1$ ... RESOURCE$_N$*) [File Manager Command]

Parallel to INITRECORDS, INITRESOURCES writes expressions on a file to perform whatever initialization/allocation is necessary for the indicated resources, without writing the resource declaration itself.

(**COURIERPROGRAMS** *NAME$_1$ ... NAME$_N$*) [File Manager Command]

Each *NAME$_i$* is the name of a Courier program (see Chapter 31). Writes expressions which will redeclare the Courier program when the file is loaded.

(**TEMPLATES** *SYMBOL$_1$ ... SYMBOL$_N$*) [File Manager Command]

Each *SYMBOL$_i$* is a symbol which has a Masterscope template (see Chapter 19). Writes expressions which will restore the templates when the file is loaded.

(**FILES** *FILE$_1$ ... FILE$_N$*) [File Manager Command]

Used to specify auxiliary files to be loaded in when the file is loaded. Dumps an expression calling FILESLOAD (see the Loading Files section), with *FILE$_1$ ... FILE$_N$* as the arguments. FILESLOAD interprets *FILE$_1$ ... FILE$_N$* as files to load, possibly interspersed with lists used to specify certain loading options.

(**FILEPKGCOMS** *SYMBOL$_1$ ... SYMBOL$_N$*) [File Manager Command]

Each symbol *SYMBOL$_i$* is either the name of a user-defined file manager command or a user-defined file manager type (or both). Writes expressions which will restore each command/type.

If *SYMBOL$_i$* is not a file manager command or type, a warning message "no FILE PACKAGE COMMAND for *SYMBOL$_i$*" is printed.

(***** . *TEXT*) [File Manager Command]

Used for inserting comments in a file. The file manager command is simply written on the output file; it will be ignored when the file is loaded.

If the first element of *TEXT* is another *, a form-feed is printed on the file before the comment.

(**P** *EXP$_1$ ... EXP$_N$*) [File Manager Command]

Writes each of the expressions *EXP$_1$ ... EXP$_N$* on the output file, where they will be evaluated when the file is loaded.

(**E** *FORM$_1$ ... FORM$_N$*) [File Manager Command]

Each of the forms *FORM$_1$ ... FORM$_N$* is evaluated at *output* time, when MAKEFILE interpretes this file manager command.

(**COMS** *COM$_1$ ... COM$_N$*) [File Manager Command]

Each of the commands *COM$_1$ ... COM$_N$* is interpreted as a file manager command.

(**ORIGINAL** *COM$_1$ ... COM$_N$*) [File Manager Command]

Each of the commands *COM$_i$* will be interpreted as a file manager command without regard to any file manager macros (as defined by the MACRO property of the FILEPKGCOM function, in the Defining New File Manager Commands section). Useful for redefining a built-in file manager command in terms of itself.

Some of the "built-in" file manager commands are defined by file manager macros, so interpreting them (or new user-defined file manager commands) with ORIGINAL will fail. ORIGINAL was never intended to be used outside of a file manager command macro.

DECLARE:

(**DECLARE:** . *FILEPKGCOMS/FLAGS*) [File Manager Command]

Normally expressions written onto a symbolic file are evaluated when loaded; copied to the compiled file when the symbolic file is compiled (see Chapter 18); and not evaluated at compile time. DECLARE: allows you to override these defaults.

FILEPKGCOMS/FLAGS is a list of file manager commands, possibly interspersed with "tags". The output of those file manager commands within *FILEPKGCOMS/FLAGS* is embedded in a DECLARE: expression, along with any tags that are specified. For example, (DECLARE: EVAL@COMPILE DONTCOPY (FNS ...) (PROP ...)) would produce (DECLARE: EVAL@COMPILE DONTCOPY (DEFINEQ ...) (PUTPROPS ...)). DECLARE: is *defined* as an nlambda nospread function, which processes its arguments by evaluating or not evaluating each expression depending on the setting of

internal state variables. The initial setting is to evaluate, but this can be overridden by specifying the DONTEVAL@LOAD tag.

DECLARE: expressions are specially processed by the compiler. For the purposes of compilation, DECLARE: has two principal applications: to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g., to set up macros; and/or to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in CDR of a DECLARE: form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the DECLARE: by means of the tags DONTCOPY, EVAL@COMPILE, etc.

The tags are:

EVAL@LOAD

DOEVAL@LOAD Evaluate the following forms when the file is loaded (unless overridden by DONTEVAL@LOAD).

DONTEVAL@LOAD Do not evaluate the following forms when the file is loaded.

EVAL@LOADWHEN This tag can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading. ... EVAL@LOADWHEN T ... is equivalent to ... EVAL@LOAD ...

COPY

DOCOPY When compiling, copy the following forms into the compiled file.

DONTCOPY When compiling, do not copy the following forms into the compiled file.

Note: If the file manager commands following DONTCOPY include record declarations for datatypes, or records with initialization forms, it is necessary to include a INITRECORDS file manager command (see the prior section) outside of the DONTCOPY form so that the initialization information is copied. For example, if FOO was defined as a datatype,

```
(DECLARE: DONTCOPY (RECORDS FOO))
(INITRECORDS FOO)
```

would copy the data type declaration for FOO, but would not copy the whole record declaration.

COPYWHEN When compiling, if the next form evaluates to non-NIL, copy the following forms into the compiled file.

EVAL@COMPILE

DOEVAL@COMPILE When compiling, evaluate the following forms.

DONTEVAL@COMPILE When compiling, do not evaluate the following forms.

EVAL@COMPILEWHEN When compiling, if the next form evaluates to non-NIL, evaluate the following forms.

FIRST For expressions that are to be copied to the compiled file, the tag FIRST can be used to specify that the following expressions in the DECLARE: are to appear at the front of the compiled file, before anything else except the FILECREATED expressions (see the Symbolic File Format section). For example, (DECLARE: COPY FIRST (P (PRINT *MESS1* T)) NOTFIRST (P (PRINT *MESS2* T))) will cause (PRINT *MESS1* T) to appear first in the compiled file, followed by any functions, then (PRINT MESS2 T).

NOTFIRST Reverses the effect of FIRST.

The value of DECLARETAGSLST is a list of all the tags used in DECLARE: expressions. If a tag not on this list appears in a DECLARE: file manager command, spelling correction is performed using DECLARETAGSLST as a spelling list.

Note that the function LOADCOMP (see the Loading Files section) provides a convenient way of obtaining information from the DECLARE: expressions in a file, without reading in the entire file. This information may be used for compiling other files.

(**BLOCKS** $BLOCK_1$ ... $BLOCK_N$) [File Manager Command]

For each $BLOCK_i$, writes a DECLARE: expression which the block compile functions interpret as a block declaration. See Chapter 18.

Exporting Definitions

When building a large system in Interlisp, it is often the case that there are record definitions, macros and the like that are needed by several different system files when running, analyzing and compiling the source code of the system, but which are not needed for running the compiled code. By using the DECLARE: file manager command with tag DONTCOPY (see the prior section), these definitions can be kept out of the compiled files, and hence out of the system constructed by loading the compiled files files into Interlisp. This saves loading time, space in the resulting system, and whatever other overhead might be incurred by keeping those definitions around, e.g., burden on the record package to consider more possibilities in translating record accesses, or conflicts between system record fields and user record fields.

However, if the implementor wants to debug or compile code in the resulting system, the definitions are needed. And even if the definitions *had* been copied to the compiled files, a similar problem arises if one wants to work on system code in a regular Interlisp environment where none of the system files had been loaded. One could mandate that any definition needed by more than one file in the system should reside on a distinguished file of definitions, to be loaded into any environment where the

system files are worked on. Unfortunately, this would keep the definitions away from where they logically belong. The EXPORT mechanism is designed to solve this problem.

To use the mechanism, the implementor identifies any definitions needed by files other than the one in which the definitions reside, and wraps the corresponding file manager commands in the EXPORT file manager command. Thereafter, GATHEREXPORTS can be used to make a single file containing all the exports.

(**EXPORT** *COM1 ... COMN*) [File Manager Command]

This command is used for "exporting" definitions. Like COM, each of the commands *COM1 ... COMN* is interpreted as a file manager command. The commands are also flagged in the file as being "exported" commands, for use with GATHEREXPORTS.

(**GATHEREXPORTS** *FROMFILES TOFILE FLG*) [Function]

FROMFILES is a list of files containing EXPORT commands. GATHEREXPORTS extracts all the exported commands from those files and produces a loadable file *TOFILE* containing them. If *FLG* = EVAL, the expressions are evaluated as they are gathered; i.e., the exports are effectively loaded into the current environment as well as being written to *TOFILE*.

(**IMPORTFILE** *FILE RETURNFLG*) [Function]

If *RETURNFLG* is NIL, this loads any exported definitions from *FILE* into the current environment. If *RETURNFLG* is T, this returns a list of the exported definitions (evaluable expressions) without actually evaluating them.

(**CHECKIMPORTS** *FILES NOASKFLG*) [Function]

Checks each of the files in *FILES* to see if any exists in a version newer than the one from which the exports in memory were taken (GATHEREXPORTS and IMPORTFILE note the creation dates of the files involved), or if any file in the list has not had its exports loaded at all. If there are any such files, you are asked for permission to IMPORTFILE each such file. If *NOASKFLG* is non-NIL, IMPORTFILE is performed without asking.

For example, suppose file FOO contains records R1, R2, and R3, macros BAR and BAZ, and constants CON1 and CON2. If the definitions of R1, R2, BAR, and BAZ are needed by files other than FOO, then the file commands for FOO might contain the command

```
(DECLARE: EVAL@COMPILE DONTCOPY
   (EXPORT (RECORDS R1 R2)
           (MACROS BAR BAZ))
   (RECORDS R3)
   (CONSTANTS BAZ))
```

None of the commands inside this DECLARE: would appear on FOO's compiled file, but (GATHEREXPORTS '(FOO) 'MYEXPORTS) would copy the record definitions for R1 and R2 and the macro definitions for BAR and BAZ to the file MYEXPORTS.

FileVars

In each of the file manager commands described above, if the symbol * follows the command type, the form following the *, i.e., CADDR of the command, is evaluated and its value used in executing the command, e.g., (FNS * (APPEND FNS1 FNS2)). When this form is a symbol, e.g. (FNS * FOOFNS), we say that the variable is a "filevar". Note that (COMS * *FORM*) provides a way of *computing* what should be done by MAKEFILE.

Example:

```
_ (SETQ FOOFNS '(FOO1 FOO2 FOO3))
      (FOO1 FOO2 FOO3)

_ (SETQ FOOCOMS
      '((FNS * FOOFNS)
       (VARS FIE)
       (PROP MACRO FOO1 FOO2)
       (P (MOVD 'FOO1 'FIE1))]

_ (MAKEFILE 'FOO)
```

would create a file FOO containing:

```
(FILECREATED "time and date the file was made" . "other
information")
       (PRETTYCOMPRINT FOOCOMS)
       (RPAQQ FOOCOMS ((FNS * FOOFNS) ...)
       (RPAQQ FOOFNS (FOO1 FOO3 FOO3))
       (DEFINEQ "definitions of FOO1, FOO2, and FOO3")
       (RPAQQ FIE "value of FIE")
       (PUTPROPS FOO1 MACRO PROPVALUE)
       (PUTPROPS FOO2 MACRO PROPVALUE)
       (MOVD (QUOTE FOO1) (QUOTE FIE1))
       STOP
```

For the PROP and IFPROP commands (see the Litatom Properties section), the * follows the property name instead of the command, e.g., (PROP MACRO * FOOMACROS). Also, in the form (* * comment ...), the word comment is not treated as a filevar.

Defining New File Manager Commands

A file manager command is defined by specifying the values of certain properties. You can specify the various attributes of a file manager command for a new command, or respecify them for an existing command. The following properties are used:

MACRO [File Manager Command Property]

Defines how to dump the file manager command. Used by MAKEFILE. Value is a pair (*ARGS* . *COMS*). The "arguments" to the file manager command are substituted for *ARGS* throughout *COMS*, and the result treated as a list of file manager commands. For example, following (FILEPKGCOM 'FOO 'MACRO '((X Y) . *COMS*)), the file manager command (FOO A B) will cause A to be substituted for X and B for Y throughout *COMS*,

and then *COMS* treated as a list of commands.

The substitution is carried out by SUBPAIR (see Chapter 3), so that the "argument list" for the macro can also be atomic. For example, if (X . *COMS*) was used instead of ((X Y) . *COMS*), then the command (FOO A B) would cause (A B) to be substituted for X throughout *COMS*.

Filevars are evaluated *before* substitution. For example, if the symbol * follows *NAME* in the command, CADDR of the command is evaluated substituting in *COMS*.

ADD [File Manager Command Property]

Specifies how (if possible) to add an instance of an object of a particular type to a given file manager command. Used by ADDTOFILE. Value is *FN*, a function of three arguments, *COM*, a file manager command CAR of which is EQ to *COMMANDNAME*, *NAME*, a typed object, and *TYPE*, its type. *FN* should return T if it (undoably) adds *NAME* to *COM*, NIL if not. If no ADD property is specified, then the default is (1) if (CAR *COM*) = *TYPE* and (CADR *COM*) = *, and (CADDR *COM*) is a filevar (i.e. a literal atom), add *NAME* to the value of the filevar, or (2) if (CAR *COM*) = *TYPE* and (CADR *COM*) is not *, add *NAME* to (CDR *COM*).

Actually, the function is given a fourth argument, *NEAR*, which if non-NIL, means the function should try to add the item after *NEAR*. See discussion of ADDTOFILES?, in the Storing Files section.

DELETE [File Manager Command Property]

Specifies how (if possible) to delete an instance of an object of a particular type from a given file manager command. Used by DELFROMFILES. Value is *FN*, a function of three arguments, *COM*, *NAME*, and *TYPE*, same as for ADD. *FN* should return T if it (undoably) deletes *NAME* from *COM*, NIL if not. If no DELETE property is specified, then the default is either (CAR *COM*) = *TYPE* and (CADR *COM*) = *, and (CADDR *COM*) is a filevar (i.e. a literal atom), and *NAME* is contained in the value of the filevar, then remove *NAME* from the filevar, or if (CAR *COM*) = *TYPE* and (CADR *COM*) is not *, and *NAME* is contained in (CDR *COM*), then remove *NAME* from (CDR *COM*).

If *FN* returns the value of ALL, it means that the command is now "empty", and can be deleted entirely from the command list.

CONTENTS [File Manager Command Property]
CONTAIN [File Manager Command Property]

Determines whether an instance of an object of a given type is contained in a given file manager command. Used by WHEREIS and INFILECOMS?. Value is *FN*, a function of three arguments, *COM*, a file manager command CAR of which is EQ to *COMMANDNAME*, *NAME*, and *TYPE*. The interpretation of *NAME* is as follows: if *NAME* is NIL, *FN* should return a list of elements of type *TYPE* contained in *COM*. If *NAME* is T, *FN* should return T if there are *any* elements of type *TYPE* in *COM*. If *NAME* is an atom other than T or NIL, return T if *NAME* of type *TYPE* is contained in *COM*. Finally, if *NAME* is a list, return a list of those elements of type *TYPE* contained in *COM* that are also contained in *NAME*.

It is sufficient for the CONTENTS function to simply return the list of items of type *TYPE* in

command *COM*, i.e. it can in fact ignore the *NAME* argument. The *NAME* argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type *TYPE* is contained in the command.

If a CONTENTS property is specified and the corresponding function application returns NIL and (CAR *COM*) = *TYPE*, then the operation indicated by *NAME* is performed on the value of (CADDR *COM*), if (CADR *COM*) = *, otherwise on (CDR *COM*). In other words, by specifying a CONTENTS property that returns NIL, e.g. the function NILL, you specify that a file manager command of name FOO produces objects of file manager type FOO and only objects of type FOO.

If the CONTENTS property is not provided, the command is simply expanded according to its MACRO definition, and each command on the resulting command list is then interrogated.

If *COMMANDNAME* is a file manager command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of CONSes and garbage collections. By informing the file manager as to what this command actually does and does not produce via the CONTENTS property, this expansion is avoided. For example, suppose you have a file manager command called GRAMMARS which dumps various property lists but no functions. The file manager could ignore this command when seeking information about FNS.

The function FILEPKGCOM is used to define new file manager commands, or to change the properties of existing commands. It is possible to redefine the attributes of system file manager commands, such as FNS or PROPS, and to cause unpredictable results.

(**FILEPKGCOM** *COMMANDNAME* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N) [NoSpread Function]

Nospread function for defining new file manager commands, or changing properties of existing file manager commands. $PROP_i$ is one of of the property names described above; VAL_i is the value to be given that property of the file manager command *COMMANDNAME*. Returns *COMMANDNAME*.

(FILEPKGCOM *COMMANDNAME PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGCOM *COMMANDNAME*) returns a property list of all of the defined properties of *COMMANDNAME*, using the property names as keys.

Specifying *TYPE* as the symbol COM can be used to define one file manager command as a synonym of another. For example, (FILEPKGCOM 'INITVARIABLES 'COM 'INITVARS) defines INITVARIABLES as a synonym for the file manager command INITVARS.

Functions for Manipulating File Command Lists

The following functions may be used to manipulate filecoms. The argument *COMS* does *not* have to

correspond to the filecoms for some file. For example, *COMS* can be the list of commands generated as a result of expanding a user-defined file manager command.

The following functions will accept a file manager command as a valid value for their *TYPE* argument, even if it does not have a corresponding file manager type. User-defined file manager commands are expanded as necessary.

(**INFILECOMS**? *NAME TYPE COMS*) [Function]

COMS is a list of file manager commands, or a variable whose value is a list of file manager commands. *TYPE* is a file manager type. INFILECOMS? returns T if *NAME* of type *TYPE* is "contained" in *COMS*.

If *NAME* = NIL, INFILECOMS? returns a list of all elements of type *TYPE*.

If *NAME* = T, INFILECOMS? returns T if there are *any* elements of type *TYPE* in *COMS*.

(**ADDTOFILE** *NAME TYPE FILE NEAR LISTNAME*) [Function]

Adds *NAME* of type *TYPE* to the file manager commands for *FILE*. If *NEAR* is given and it is the name of an item of type *TYPE* already on *FILE*, then *NAME* is added to the command that dumps *NEAR*. If *LISTNAME* is given and is the name of a list of items of *TYPE* items on *FILE*, then *NAME* is added to that list. Uses ADDTOCOMS and MAKENEWCOM. Returns *FILE*. ADDTOFILE is undoable.

(**DELFROMFILES** *NAME TYPE FILES*) [Function]

Deletes all instances of *NAME* of type *TYPE* from the filecoms for each of the files on *FILES*. If *FILES* is a non-NIL symbol, (LIST *FILES*) is used. *FILES* = NIL defaults to FILELST. Returns a list of files from which *NAME* was actually removed. Uses DELFROMCOMS. DELFROMFILES is undoable.

Deleting a function will also remove the function from any BLOCKS declarations in the filecoms.

(**ADDTOCOMS** *COMS NAME TYPE NEAR LISTNAME*) [Function]

Adds *NAME* as a *TYPE* to *COMS*, a list of file manager commands or a variable whose value is a list of file manager commands. Returns NIL if ADDTOCOMS was unable to find a command appropriate for adding *NAME* to *COMS*. *NEAR* and *LISTNAME* are described in the discussion of ADDTOFILE. ADDTOCOMS is undoable.

The exact algorithm for adding commands depends the particular command itself. See discussion of the ADD property, in the description of FILEPKGCOM.

ADDTOCOMS will not attempt to add an item to any command which is inside of a DECLARE: unless you specified a specific name via the LISTNAME or NEAR option of ADDTOFILES?.

(**DELFROMCOMS** *COMS NAME TYPE*) [Function]

Deletes *NAME* as a *TYPE* from *COMS*. Returns NIL if DELFROMCOMS was unable to modify

COMS to delete *NAME*. DELFROMCOMS is undoable.

(**MAKENEWCOM** *NAME TYPE*) [Function]

Returns a file manager command for dumping *NAME* of type *TYPE*. Uses the procedure described in the discussion of NEWCOM, in the Defining New File Manager Types section.

(**MOVETOFILE** *TOFILE NAME TYPE FROMFILE*) [Function]

Moves the definition of *NAME* as a *TYPE* from *FROMFILE* to *TOFILE* by modifying the file commands in the appropriate way (with DELFROMFILES and ADDTOFILE).

Note that if *FROMFILE* is specified, the definition will be retrieved from that file, even if there is another definition currently in your environment.

(**FILECOMSLST** *FILE TYPE*) [Function]

Returns a list of all objects of type *TYPE* in *FILE*.

(**FILEFNSLST** *FILE*) [Function]

Same as (FILECOMSLST *FILE* 'FNS).

(**FILECOMS** *FILE TYPE*) [Function]

Returns (PACK* *FILE* (OR *TYPE* 'COMS)). Note that (FILECOMS 'FOO) returns the symbol FOOCOMS, not the value of FOOCOMS.

(**SMASHFILECOMS** *FILE*) [Function]

Maps down (FILECOMSLST *FILE* 'FILEVARS) and sets to NOBIND all filevars (see the FileVars section), i.e., any variable used in a command of the form (*COMMAND* * *VARIABLE*). Also sets (FILECOMS *FILE*) to NOBIND. Returns *FILE*.

Symbolic File Format

The file manager manipulates symbolic files in a particular format. This format is defined so that the information in the file is easily readable when the file is listed, as well as being easily manipulated by the file manager functions. In general, there is no reason for you to manually change the contents of a symbolic file. However, to allow you to extend the file manager, this section describes some of the functions used to write symbolic files, and other matters related to their format.

(**PRETTYDEF** *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*) [Function]

Writes a symbolic file in PRETTYPRINT format for loading, using FILERDTBL as its read table. PRETTYDEF returns the name of the symbolic file that was created.

PRETTYDEF operates under a RESETLST (see Chapter 14), so if an error occurs, or a Control-D is typed, all files that PRETTYDEF has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be

undone. The RESETLST also means that any RESETSAVEs executed in the file manager commands will also be protected.

PRTTYFNS is an optional list of function names. It is equivalent to including (FNS * *PRTTYFNS*) in the file manager commands in *PRTTYCOMS*. *PRTTYFNS* is an anachronism from when PRETTYDEF did not use a list of file manager commands, and should be specified as NIL.

PRTTYFILE is the name of the file on which the output is to be written. *PRTTYFILE* has to be a symbnol. If *PRTTYFILE* = NIL, the primary output file is used. *PRTTYFILE* is opened if not already open, and it becomes the primary output file. *PRTTYFILE* is closed at end of PRETTYDEF, and the primary output file is restored.

PRTTYCOMS is a list of file manager commands interpreted as described in the File Manager Commands section. If *PRTTYCOMS* is atomic, its top level value is used and an RPAQQ is written which will set that atom to the list of commands when the file is subsequently loaded. A PRETTYCOMPRINT expression (see below) will also be written which informs you of the named atom or list of commands when the file is subsequently loaded. In addition, if any of the functions in the file are nlambda functions, PRETTYDEF will automatically print a DECLARE: expression suitable for informing the compiler about these functions, in case you recompile the file without having first loaded the nlambda functions (see Chapter 18).

REPRINTFNS and *SOURCEFILE* are for use in conjunction with remaking a file (see the Remaking a Symbolic File section). *REPRINTFNS* can be a list of functions to be prettyprinted, or EXPRS, meaning prettyprint all functions with EXPR definitions, or ALL meaning prettyprint all functions either defined as EXPRs, or with EXPR properties. Note that doing a remake with *REPRINTFNS* = NIL makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to variables or property lists. *SOURCEFILE* is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by *REPRINTFNS*. *SOURCEFILE* = T means to use most recent version (i.e., highest number) of *PRTTYFILE*, the second argument to PRETTYDEF. If *SOURCEFILE* cannot be found, PRETTYDEF prints the message "*FILE* NOT FOUND, SO IT WILL BE WRITTEN ANEW", and proceeds as it does when *REPRINTFNS* and *SOURCEFILE* are both NIL.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG* = T, so whenever PRETTYPRINT starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note that normally if PRETTYPRINT is given a symbol which is not defined as a function but is known to be on one of the files noticed by the file manager, PRETTYPRINT will load in the definition (using LOADFNS) and print it. This is not done when PRETTYPRINT is called from PRETTYDEF.

In Medley the SYSPRETTYFLG is ignored in the Interlisp exec.

(**PRINTFNS** *X*) [Function]

X is a list of functions. PRINTFNS prettyprints a DEFINEQ epression that defines the

functions to the primary output stream using the primary read table. Used by PRETTYDEF to implement the FNS file manager command.

(**PRINTDATE** *FILE CHANGES*) [Function]

Prints the FILECREATED expression at beginning of PRETTYDEF files. *CHANGES* used by the file manager.

(**FILECREATED** *X*) [NLambda NoSpread Function]

Prints a message (using LISPXPRINT) followed by the time and date the file was made, which is (CAR *X*). The message is the value of PRETTYHEADER, initially "FILE CREATED". If PRETTYHEADER = NIL, nothing is printed. (CDR *X*) contains information about the file, e.g., full name, address of file map, list of changed items, etc. FILECREATED also stores the time and date the file was made on the property list of the file under the property FILEDATES and performs other initialization for the file manager.

(**PRETTYCOMPRINT** *X*) [NLambda Function]

Prints *X* (unevaluated) using LISPXPRINT, unless PRETTYHEADER = NIL.

PRETTYHEADER [Variable]

Value is the message printed by FILECREATED. PRETTYHEADER is initially "FILE CREATED". If PRETTYHEADER = NIL, neither FILECREATED nor PRETTYCOMPRINT will print anything. Thus, setting PRETTYHEADER to NIL will result in "silent loads". PRETTYHEADER is reset to NIL during greeting (see Chapter 12).

(**FILECHANGES** *FILE TYPE*) [Function]

Returns a list of the changed objects of file manager type *TYPE* from the FILECREATED expression of *FILE*. If *TYPE* = NIL, returns an alist of all of the changes, with the file manager types as the CARs of the elements..

(**FILEDATE** *FILE*) [Function]

Returns the file date contained in the FILECREATED expression of *FILE*.

(**LISPSOURCEFILEP** *FILE*) [Function]

Returns a non-NIL value if *FILE* is in file manager format and has a file map, NIL otherwise.

Copyright Notices

The system has a facility for automatically printing a copyright notice near the front of files, right after the FILECREATED expression, specifying the years it was edited and the copyright owner. The format of the copyright notice is:

```
(* Copyright (c) 1981 by Foo Bars Corporation)
```

Once a file has a copyright notice then every version will have a new copyright notice inserted into the

file without your intervention. (The copyright information necessary to keep the copyright up to date is stored at the end of the file.).

Any year the file has been edited is considered a "copyright year" and therefore kept with the copyright information. For example, if a file has been edited in 1981, 1982, and 1984, then the copyright notice would look like:

```
(* Copyright (c) 1981,1982,1984 by Foo Bars Corporation)
```

When a file is made, if it has no copyright information, the system will ask you to specify the copyright owner (if COPYRIGHTFLG = T). You may specify one of the names from COPYRIGHTOWNERS, or give one of the following responses:

- Type a left-square-bracket. The system will then prompt for an arbitrary string which will be used as the owner-string
- Type a right-square-bracket, which specifies that you really do not want a copyright notice.
- Type "NONE" which specifies that this file should never have a copyright notice.

For example, if COPYRIGHTOWNERS has the value

```
((BBN "Bolt Beranek and Newman Inc.")
 (XEROX "Xerox Corporation"))
```

then for a new file FOO the following interaction will take place:

```
Do you want to Copyright FOO? Yes
Copyright owner:   (user typed ?)
one of:
BBN - Bolt Beranek and Newman Inc.
XEROX - Xerox Corporation
NONE - no copyright ever for this file
[ - new copyright owner -- type one line of text
] - no copyright notice for this file now

Copyright owner: BBN
```

Then "Foo Bars Corporation" in the above copyright notice example would have been "Bolt Beranek and Newman Inc."

The following variables control the operation of the copyright facility:

COPYRIGHTFLG [Variable]

The value of COPYRIGHTFLG determines whether copyright information is maintained in files. Its value is interpreted as follows:

NIL The system will preserve old copyright information, but will not ask you about copyrighting new files. This is the default value of COPYRIGHTFLG.

T When a file is made, if it has no copyright information, the system will ask you to specify the copyright owner.

NEVER The system will neither prompt for new copyright information nor preserve old copyright information.

DEFAULT The value of DEFAULTCOPYRIGHTOWNER (below) is used for putting copyright information in files that don't have any other copyright. The prompt "Copyright owner for file xx:" will still be printed, but the default will be filled in immediately.

COPYRIGHTOWNERS [Variable]

COPYRIGHTOWNERS is a list of entries of the form (*KEY OWNERSTRING*), where *KEY* is used as a response to ASKUSER and *OWNERSTRING* is a string which is the full identification of the owner.

DEFAULTCOPYRIGHTOWNER [Variable]

If you do not respond in DWIMWAIT seconds to the copyright query, the value of DEFAULTCOPYRIGHTOWNER is used.

Functions Used Within Source Files

The following functions are normally only used within symbolic files, to set variable values, property values, etc. Most of these have special behavior depending on file manager variables.

(**RPAQ** *VAR VALUE*) [NLambda Function]

An nlambda function like SETQ that sets the top level binding of *VAR* (unevaluated) to *VALUE*.

(**RPAQQ** *VAR VALUE*) [NLambda Function]

An nlambda function like SETQQ that sets the top level binding of *VAR* (unevaluated) to *VALUE* (unevaluated).

(**RPAQ?** *VAR VALUE*) [NLambda Function]

Similar to RPAQ, except that it does nothing if *VAR* already has a top level value other than NOBIND. Returns *VALUE* if *VAR* is reset, otherwise NIL.

RPAQ, RPAQQ, and RPAQ? generate errors if *X* is not a symbol. All are affected by the value of DFNFLG (see Chapter 10). If DFNFLG = ALLPROP (and the value of *VAR* is other than NOBIND), instead of setting *X*, the corresponding value is stored on the property list of *VAR* under the property VALUE. All are undoable.

(**ADDTOVAR** *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Each X_i that is not a member of the value of *VAR* is added to it, i.e. after ADDTOVAR

completes, the value of *VAR* will be (UNION (LIST X_1 X_2 ... X_N) *VAR*). ADDTOVAR is used by PRETTYDEF for implementing the ADDVARS command. It performs some file manager related operations, i.e. "notices" that *VAR* has been changed. Returns the atom *VAR* (not the value of *VAR*).

(**APPENDTOVAR** *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Similar to ADDTOVAR, except that the values are added to the end tof the list, rather than at the beginning.

(**PUTPROPS** *ATM* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N) [NLambda NoSpread Function]

Nlambda nospread version of PUTPROP (none of the arguments are evaluated). For $i = 1...N$, puts property $PROP_i$, value $VALi$, on the property list of *ATM*. Performs some file manager related operations, i.e., "notices" that the corresponding properties have been changed.

(**SAVEPUT** *ATM PROP VAL*) [Function]

Same as PUTPROP, but marks the corresponding property value as having been changed (used by the file manager).

File Maps

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end byte address (see GETFILEPTR, in Chapter 25) for each DEFINEQ expression in the file, the begin and end address for each function definition within the DEFINEQ, and the begin and end address for each compiled function.

MAKEFILE, PRETTYDEF, LOADFNS, RECOMPILE, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables LOADFNS to load selected function definitions simply by setting the file pointer to the corresponding address using SETFILEPTR, and then performing a single READ. Similarly, the file map is heavily used by the "remake" option of MAKEFILE (see the Remaking a Symbolic File section): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is written by MAKEFILE, a file map for the new file is built. Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that PRETTYPRINT *know* that it is printing a DEFINEQ expression. For this reason, you should never print a DEFINEQ expression onto a file yourself, but should instead always use the FNS file manager command (see the Functions and Macros section).

The file map is stored on the property list of the root name of the file, under the property FILEMAP. In addition, MAKEFILE writes the file map on the file itself. For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the FILECREATED expression that appears at the beginning of the file so that the file map can be

rapidly accessed without having to scan the entire file. In most cases, LOAD and LOADFNS do not have to build the file map at all, since a file map will usually appear in the corresponding file, unless the file was written with BUILDMAPFLG = NIL, or was written outside of Interlisp.

Currently, file maps for *compiled* files are not written onto the files themselves. However, LOAD and LOADFNS will build maps for a compiled file when it is loaded, and store it on the property FILEMAP. Similary, LOADFNS will obtain and use the file map for a compiled file, when available.

The use and creation of file maps is controlled by the following variables:

BUILDMAPFLG [Variable]

Whenever a file is read by LOAD or LOADFNS, or written by MAKEFILE, a file map is automatically built unless BUILDMAPFLG = NIL. (BUILDMAPFLG is initially T.)

While building the map will not help the first reference to a file, it will help in future references. For example, if you perform (LOADFROM 'FOO) where FOO does not contain a file map, the LOADFROM will be (slightly) slower than if FOO did contain a file map, but subsequent calls to LOADFNS for this version of FOO will be able to use the map that was built as the result of the LOADFROM, since it will be stored on FOO's FILEMAP property.

USEMAPFLG [Variable]

If USEMAPFLG = T (the initial setting), the functions that use file maps will first check the FILEMAP property to see if a file map for this file was previously obtained or built. If not, the first expression on the file is checked to see if it is a FILECREATED expression that also contains the address of a file map. If the file map is not on the FILEMAP property or in the file, a file map will be built (unless BUILDMAPFLG = NIL).

If USEMAPFLG = NIL, the FILEMAP property and the file will not be checked for the file map. This allows you to recover in those cases where the file and its map for some reason do not agree. For example, if you use a text editor to change a symbolic file that contains a map (not recommended), inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error FILEMAP DOES NOT AGREE WITH CONTENTS OF *FILE*. In such cases, you can set USEMAPFLG to NIL, causing the map contained in the file to be ignored, and then reexecute the operation.