Streams and File Devices

Edited 30 Nov 84, van Melle
Edited 19 June 86, Jellinek
Edited 22 September 86, Jellinek
Edited 3 October 86, Jellinek

An Interlisp Stream is an object of datatype STREAM that is capable of performing, at the least, sequential input and/or output of bytes. Some streams can do much more. Streams are used for access to open files, for writing to the display, for chatting to remote hosts, and whatever other uses people come up with. This document describes how one goes about defining a new device, the meanings of the record fields of the STREAM and FDEV datatypes, and anything else that seemed relevant at the time.

The implementation of Streams is strongly object-oriented. Every STREAM has a pointer to a device (the datatype FDEV), which contains a vector of functions to be called when certain operations are required of the stream. There can be many streams with the same device. In the object-oriented terms of LOOPS, one can think of the device as a *class*, which provides a set of *methods* that implement class operations, and the streams as *instances*. Devices and streams also have local state, which might be thought of as class and instance variables. Declarations for STREAM and FDEV can be obtained by loading EXPORTS.ALL.

OPENSTREAM, CLOSEF, FORCEOUTPUT, READP, EOFP, GETEOFPTR, GETFILEINFO, SETFILEINFO, DIRECTORY, COPYBYTES, DELFILE, RENAMEFILE, FULLNAME are some of the Lisp functions called by the programmer that ultimately turn into operations at the device level. The descriptions that follow sometimes allude to these functions, and knowledge of how they operate may occasionally give the reader additional clues as to how the device operations work.

Typically, some part of the operation is handled by the ''generic'' file system code, which then calls on the device to handle that part of the operation that is device-specific. For example, the function OPENSTREAM takes the name of the file it is to open and fills in host and directory defaults, and decides which device handles such a file. It then calls on the particular device to actually open the file. After the file is opened, the generic file system code registers the file on (OPENP). As another example, operations involving open streams first coerce non-streams (e.g. filenames) to open streams before calling the device-specific operation. With the advent of multiple streams per file, the last two examples will soon no longer obtain.

Devices

A device is an object of type FDEV (so named for historical reasons: ''File Device''). The standard way to define a new device is to create such an object, by performing (**create** FDEV --), and then pass the newly created FDEV to the function \DEFINEDEVICE. \DEFINEDEVICE is the way a device ''announces'' itself to the generic file system.

(\DEFINEDEVICE *NAME DEV*) [Function]

> Installs device *DEV*, giving it the name *NAME*. *NAME* must be an uppercase litatom. The generic file system code makes use of the name to locate the device that is willing to deal with files whose full name begins {*NAME*}. It is permissible to have more than one name map to the same device; this effectively provides device synonyms. Devices are encouraged, however, to always create file names using the canonical device name, independent of what name was passed in.
>
> *NAME* can be NIL, in which case the generic file system code never consults the device directly. However, its EVENTFN is still run around Lisp exits, and it can be used as the device for a stream

created by nonstandard methods.

If a device never wants to be invoked by name, and has no interesting EVENTFN or HOSTNAMEP methods, then there is no need to ever register it with \DEFINEDEVICE.

(\REMOVEDEVICE *DEV*) [Function]

Removes device *DEV* from the list of known devices, as well as any name that maps to that device.

(\GETDEVICEFROMNAME *NAME NOERROR DONTCREATE*) [Function]

Returns the device associated with *NAME*. *NAME* can be a litatom or string; it is coerced to uppercase, and if it begins with an open brace, is assumed to be a file name, from which the host name is extracted. If no such device is known, attempts to find one by polling the HOSTNAMEP methods of all known devices (see below); if a device is still not found, causes a FILE NOT FOUND error unless *NOERROR* is true. If *DONTCREATE* is true, it never attempts to create a device, just returns an existing device if there is one, NIL otherwise.

The fields of an FDEV are divided up into informational fields and ''methods''.

DEVICENAME — A pointer field, the name of the device, standardly a litatom. Use of this field is largely up to the device, but it is usually selected to be the name that appears inside braces in filenames opened on this device. For devices that do not support the notion of named files, DEVICENAME can be anything that the implementor cares to use for debugging assistance.

RESETABLE — A flag, true if (SETFILEPTR stream 0) can be performed. Currently unused.

RANDOMACCESSP — True if the stream is randomly accessable, i.e., if SETFILEPTR works on this kind of stream.

NODIRECTORIES — True if files opened on this device do not (usually) have a directory as part of their name. The principal use for this is by the CONN command, which will not try to connect to the user's home directory if given a host only, e.g., CONN {DSK}.

BUFFERED — True if streams of this sort are buffered in a manner compatible with the microcoded versions of BIN and BOUT. More specifically, BUFFERED implies that the device implements the GETNEXTBUFFER method. See description of buffered streams.

PAGEMAPPED — True if this stream is implemented by the pagemapped functions. All pagemapped streams are also buffered, so if this flag is true, so should be BUFFERED. See description of pagemapped streams.

FDBINABLE — True if streams on this device obey the rules for microcoded BIN whenever such stream is open for input access.

FDBOUTABLE — True if streams on this device obey the rules for microcoded BOUT whenever such stream is open for output access. Currently unused, as the spec needs revision.

FDEXTENDABLE — Special kind of FDBOUTABLE. Currently unused, as the spec needs revision.

DEVICEINFO — A pointer to arbitrary device-specific information. The standard use for this is to hold local state specific to one of several similar devices that share methods. For example, the Dolphin disk provides a separate FDEV for each partition of the machine; the DEVICEINFO field of each has pointers to the partition's directory and other information specific to files on that

partition only.

OPENFILELST A list of all streams open on this device. Not required; it's provided for convenience only.

The following fields are all pointer fields, and contain functions for implementing various device operations. Not all devices need have all fields filled in; the required ones are listed first and so indicated. Some ''required'' fields have defaults specified in the FDEV (or STREAM) record declaration, so the implementor need not explicitly fill those fields if the default is reasonable. Each field is presented with its arguments, in the style of a function definition; of course, it is the contents of the field, not the field name, that is the function. Using object-oriented terminology, the occupants of these fields are referred to as ''methods''. For example, ''the BIN method'' means ''the function that occupies the BIN field''.

One of the arguments to each method is usually either the device itself, or a stream open on the device, so that the device (and hence its DEVICEINFO) is usually accessible to all these functions. Arguments that are file names or patterns or pieces of file names can be either litatoms or strings, and already have their host and/or directory parts appropriately filled in from the connected directory defaults. The device may assume that the host field of the file name is indeed a name that the device has said it implements (see HOSTNAMEP). ''Full'' file names returned by these functions (or stored in the FULLFILENAME field of a stream) should be litatoms, and at least in the current implementation should be all uppercase.

Fields required of every device:

(HOSTNAMEP *HOSTNAME DEVICE*)

Called by the generic file system code when presented with a host name for which there is as yet no device defined. The function should return non-NIL if it ''recognizes'' *HOSTNAME*. There are two ways in which this method is invoked:

(1) To obtain a device for *HOSTNAME*, for example, so that a file can be opened on it. In this case, *DEVICE* is an already defined device (the one whose HOSTNAMEP method is being called), and the function should return either a new device, or T, meaning it is willing to take responsibility for this host name as well as any previous name under which the device was registered. In either case, the caller will install the returned device, or *DEVICE* if value was T, as the device to which *HOSTNAME* maps.

(2) As a pure predicate. In this case, *DEVICE* is NIL, and the function need only return T or NIL, indicating whether it believes that *HOSTNAME* is the name of a host.

In practice, the HOSTNAMEP method need only take care of the first case, since that also takes care of the second case. The second case is provided so that the device need not be created until there is an actual use for it, should the device wish to avoid unnecessary work. In practice it is rare that anyone tests a host name without subsequently needing to have the device created in full.

There are basically three kinds of devices in the system as distinguished by their HOSTNAMEP methods.

(1) Predefined devices with exactly one name, or strictly internal devices with no notion of name. For example, the CORE device always exists, and has exactly one name; the SPP device (a network byte stream) has no name (it supports no files directly). Such devices have a HOSTNAMEP method of NILL—the only name they ever go by is the one they gave to \DEFINEDEVICE, if any. This is the default.

(2) Devices that don't know ahead of time what their name will be, but for which there might be many incarnations. This is the model for remote file servers. The standard way of handling this case is to define a dummy device that has only a HOSTNAMEP method, and no name. When the HOSTNAMEP method gets called with a name that the device knows it can service, it creates a device by that name. If given a name that is a synonym of another name, it might just return the

existing device of the canonical name (using \GETDEVICEFROMNAME to find the right device). In either case, the HOSTNAMEP method of the new device is usually NILL—the original device is the only one that worries about creating new instances of this class of device.

(3) Like (2), but all the different names are handled by a single device, which takes care internally of the multiplexing among, say, different remote hosts. HOSTNAMEP returns T in this case. This is usually clumsier than (3), so discouraged.

(EVENTFN *DEVICE EVENT*)

Called around Lisp exits, to allow the device to do any necessary cleaning up, clearing of caches, disconnects with remote hosts, revalidation of files, etc. *EVENT* is one of the following litatoms: BEFORELOGOUT, BEFORESYSOUT, BEFOREMAKESYS, BEFORESAVEVM, AFTERLOGOUT, AFTERSYSOUT, AFTERMAKESYS, AFTERSAVEVM, AFTERDOSYSOUT, AFTERDOMAKESYS, AFTERDOSAVEVM. The AFTERxxx events are all run when Lisp is booted from a memory image that resulted from a LOGOUT, SYSOUT, etc. The AFTERDOxxx events run when continuing Lisp in the same incarnation following the SYSOUT, etc. (there is no such event for LOGOUT, of course). The ''after'' events are called in the same order in which the devices were defined; the ''before'' events in the reverse order.

For example, the BEFORELOGOUT event for the Leaf remote file server devices performs a FORCEOUTPUT on all its open files and then breaks the connection with the file server. The AFTERxxx events for the Leaf devices calls \REMOVEDEVICE on itself to flush any connection between the name and the server (since names and addresses can change over exit), and then revalidates all of the device's open files. The AFTERxxx events for the Dorado disk device rebuilds its cache of the disk's directory.

There are a few devices in the system that exist only for their EVENTFN. In most cases, a simpler way to tell the system you want something performed around exit is to add your event function to the list AROUNDEXITFNS instead of going to the expense of defining a device for it. There is yet another list, \SYSTEMCACHEVARS, for handling a more specialized ''around exit'' operation: every time Lisp is booted, each of the variables in the list \SYSTEMCACHEVARS is set to NIL.

The following are required of all named devices, that is, devices that map from some hostname to the device, upon which files might be opened or otherwise manipulated:

(DIRECTORYNAMEP *HOST/DIR DEVICE*)

True if *HOST/DIR* is a valid directory name on *DEVICE*. Function should ideally perform recognition as well, and return the ''true'' name. For example, given ''{PHYLEX:}<LISP>'' as argument, it might return {Phylex:PARC:Xerox}<Lisp>. *HOST/DIR* might include a subdirectory name. The device should attempt to tell the truth about whether the subdirectory exists or not, though this may not be possible for devices with fake subdirectories. Defaults to NILL, i.e., device supports no directories. Used by the command CONN and the function DIRECTORYNAMEP.

(OPENFILE *NAME ACCESS RECOG PARAMETERS DEVICE*)

Used to implement the OPENFILE and OPENSTREAM functions. Opens the file named *NAME* on this device for access *ACCESS*, returning a STREAM. The stream is usually on *DEVICE* (its DEVICE field is *DEVICE*), but is not required to be. The arguments *ACCESS, RECOG, PARAMETERS* are as with the OPENFILE function in the manual. Thus, if *NAME* does not include a version number, recognition is according to *RECOG*, which should be appropriately defaulted per *ACCESS* (INPUT implies OLD, OUTPUT implies NEW, BOTH implies OLD/NEW).

The argument *NAME* can also be a STREAM, which must be a closed stream. OPENFILE should ''reopen'' the stream. The value returned in this case may be a new stream (with the same name as the

old), or the old stream (*NAME*) itself. It is likely that the specification will be changed at some point to require that the old stream itself be returned, suitably reopened.

The argument *PARAMETERS* is a list of pairs (*OPTION VALUE*). The most interesting *OPTION*s are as follows:

`TYPE` For new files, the type of the file (`TEXT` or `BINARY`). If this parameter is not specified, the value of the global variable `DEFAULTFILETYPE` (initially `TEXT`) should be used.

`CREATIONDATE` For new files, the date of its creation. The device should use this if at all possible instead of letting the creation date default to the current date and time.

`LENGTH` The intended length of the file, in bytes. This need not be accurate—it is only a hint that may allow smarter allocation. For example, if the device knows that it does not have room for a file of the specified length, it should immediately cause a `FILE SYSTEM RESOURCES EXCEEDED` error for the intended file.

`DON'T.CHANGE.DATE` For old files being opened for access `BOTH`, don't change the creation date of the file. *ACCESS* = `BOTH` would normally imply that the content of the file is to change, and thus its creation date should be updated. Use of this parameter is a form of ''cheating'' to make it look as though the file had not changed. For example, the code that rewrites filemaps uses this parameter, since rewriting the filemap does not logically change the file's content.

`SEQUENTIAL` If `T`, is a hint that the file will, or need, only be accessed sequentially, which may allow the device to open the file in a more efficient mode.

Any parameters that the device does not understand should be ignored, rather than be cause for an error. All devices are encouraged to support at least `TYPE` and `CREATIONDATE`.

The additional options `ENDOFSTREAMOP` and `BUFFERS` are handled by the generic file system code; specifying them is equivalent to calling `SETFILEINFO` (q.v.) immediately after the open.

Fine point about *ACCESS* = `OUTPUT`: this operation always produces a new, empty file, independent of whether its name is exactly the name of an existing file. That is, it replaces any old file by the same name. On opening, such a file has an end of file of zero. Of course, since *RECOG* defaults to `NEW` in this case, the name can only clash with an old file name if a version was explicitly specified, or *RECOG* is `OLD` or `OLD/NEW`. To open an old file for output but preserve its contents, i.e., only write over part of the file, one should open for *ACCESS* = `BOTH` (since to preserve the old contents one implicitly reads them).

Exception handling: If the desired file is not found, the `OPENFILE` method should return `NIL` rather than cause a `FILE NOT FOUND` error. This is so that the generic file system code can cause the error using the original file name, not the one packed with host and directory passed in to the `OPENFILE` method. The device should feel free to signal any other errors itself on failing to open the file, e.g., `FILE WON'T OPEN` for a busy file, `PROTECTION ERROR`, or `FILE SYSTEM RESOURCES EXCEEDED`. Ideally, this error should be signaled in a way that is resumable, i.e., so that a user could, in the break, take some action to remedy the condition and then type `OK` to continue. In most cases it suffices that all the internal functions below the `OPENFILE` be named with backslashes, so that the error code will choose to resume by reverting to the `OPENFILE` and trying again.

The device must keep track of the set of files open on it, for use in access-conflict detection, and as an assist to the user in closing "dropped" streams. This is done by the generic file system, employing the `REGISTERFILE` method, below.

(`CLOSEFILE` *STREAM*)

Closes *STREAM*. Performs all necessary ''finalization'' on *STREAM*, including doing a FORCEOUTPUT or equivalent if *STREAM* was open for output. De-registers *STREAM* from its device's list of open streams.

(REOPENFILE *NAME ACCESS RECOG PARAMETERS DEVICE OLDSTREAM*)

This is exactly like OPENFILE, except that it is called after LOGOUT (or other ''after'' events) by the device's EVENTFN on the name of any stream that was left open over exit. The idea is to maintain the illusion that the file really was open over LOGOUT, but check and make sure nothing changed. The generic file system code uses the VALIDATION field to test whether the file changed behind your back.

OLDSTREAM is the stream that was open before exit, and is supplied for the benefit of devices where there is no possibility that the file changed (e.g., {CORE}), so that they can just return *OLDSTREAM* directly. *OLDSTREAM* is also of use for those devices that have to cheat in order to maintain the illusion.

(GETFILENAME *NAME RECOG DEVICE*)

Performs ''recognition'' on *NAME*. That is, it returns the full name of the file that would be opened by OPENFILE in the indicated recognition mode, or NIL if the file is not found. It is not necessary that OPENFILE actually be capable of opening the file (there is no need to check protection, for example). Used by INFILEP, OUTFILEP, FULLNAME.

(DELETEFILE *NAME DEVICE*)

Deletes the file named *NAME*, returning its full name on success, NIL on failure. Recognition mode is implicitly OLDEST. Local devices, after recognizing the file, should make sure that it is not among the device's open files (open files cannot be deleted). This, OPENFILE, and RENAMEFILE are usually the only device methods that need to know anything about what files are open.

(GENERATEFILES *DEVICE PATTERN DESIREDPROPS OPTIONS*)

Enumerates files matching *PATTERN*. Returns a ''file generator object'' of the form (*NEXTFILEFN INFOFN . ArbitraryState*). This is described in more gory detail under **Directory Enumeration**.

(RENAMEFILE *OLD-DEVICE OLD-NAME NEW-DEVICE NEW-NAME*)

(A method of *OLD-DEVICE*.) Renames the file named *OLDNAME* on device *OLD-DEVICE* to have name *NEWNAME on NEW-DEVICE*. Returns the full name of the new file if successful, NIL if not. Recognition mode is implicitly OLD for *OLDNAME*, NEW for *NEWNAME*. The generic file system code always invokes this method to implement the function RENAMEFILE. If *OLD-DEVICE* and *NEW-DEVICE* are not EQ, the RENAMEFILE method can call \GENERIC.RENAMEFILE to do the job; \GENERIC.RENAMEFILE is defined to copy *OLDNAME* to *NEWNAME* and then delete *OLDNAME*. Defaults to \GENERIC.RENAMEFILE. RENAMEFILE needs to check if OLD-NAME is open; only closed files can be renamed.

(OPENP *FILENAME ACCESS DEVICE*)

Returns all of *DEVICE*'s open streams with name *FILENAME* that are open in mode *ACCESS*. *FILENAME* and/or *ACCESS* may be NIL, which matches all names and access modes. If supplied, *FILENAME* should be a complete and exact filename, including a version number. The device field OPENFILELST provides a convenient place to store the open streams. \GENERIC.OPENP is a sample OPENP method that can be used in conjunction with \ADD-OPEN-STREAM and \DELETE-OPEN-STREAM; it consults the device's OPENFILELST.

(REGISTERFILE *DEVICE STREAM*)

Invoked by \OPENFILE, the REGISTERFILE method places the newly-opened stream on *DEVICE*'s list of open streams. Note that this need not be an actual list, but might be contained in the device's directory data structure or other convenient place. The function \ADD-OPEN-STREAM, though not the default, is a simple function that adds STREAM

to DEVICE's OPENFILELST.

(UNREGISTERFILE *DEVICE STREAM*)

Invoked by \CLOSEFILE, the UNREGISTERFILE method removes *STREAM* from *DEVICE*'s list of open streams. The revalidation code also uses this method to silently remove a device's invalid streams. Note that this need not be an actual list, but might be contained in the device's directory data structure.

The function \GENERIC-UNREGISTER-STREAM, though not the default, is a simple function that removes STREAM from DEVICE's OPENFILELST. The revalidation code also uses this method to silently remove a device's invalid streams.

The following methods are invoked for open streams. They are all required:

(BIN *STREAM*)

Returns the next byte of input from *STREAM*, or takes the appropriate action if at end of file. Unless a device has a good reason not to, it should call (\EOF.ACTION *STREAM*) at end of file/stream.

The device BIN method is actually not used directly. Rather, every stream has a STRMBINFN field, which is the function actually applied to do the input. The STRMBINFN field could thus be used to fake a specialization of the device differing only in the BIN method. However, the typical use of STRMBINFN is to temporarily override the device default. In particular, setting a stream's access to INPUT or BOTH automatically sets the stream's STRMBINFN to be the device's BIN method; setting access to NIL or OUTPUT sets the STRMBINFN to be an error. This relieves the device's BIN method of any need to check the stream's access on every call to BIN. Some network streams temporarily set their STRMBINFN to be an input eater when they receive a ''clear output'' command.

Currently, all Interlisp-D streams have bytesize 8, so BIN always returns an 8-bit integer.

Calls to the function BIN are compiled into the BIN opcode, which runs in microcode on some machines if the requirements for it are met. More on this later.

(BOUT *STREAM BYTE*)

Outputs *BYTE* to *STREAM*. As with BIN, this method is not used directly. Rather, every stream has a STRMBOUTFN field, which is the function actually applied to do the output. Setting a stream's access to OUTPUT or BOTH automatically sets the stream's STRMBOUTFN to be the device's BOUT method.

There exists a BOUT opcode, but the design is incomplete.

(PEEKBIN *STREAM NOERRORFLG*)

Returns the next input byte from *STREAM*, but does not advance the stream pointer. Thus a subsequent PEEKBIN or BIN will return the same byte. At end of stream, the device should take eof action as with BIN, unless *NOERRORFLG* is true, in which case it should return NIL.

(READP *STREAM FLG*)

Returns true if input is available from *STREAM*, that is, if a BIN right now would succeed without waiting. Defaults to \GENERIC.READP, which uses EOFP and PEEKBIN.

Roughly speaking, READP is the complement of EOFP for streams that are not arriving in real time. It is interestingly different for network streams, or the keyboard.

FLG is a bit of cruft that not everyone pays attention to, and may be flushed at some point: if *FLG* is NIL, then READP should return NIL if the only input waiting is an end of line character.

(EOFP *STREAM*)

Returns true when *STREAM* is ‘‘at end of file’’, i.e., a BIN would cause an end of file action to occur. Note that for a network stream, it is possible for both EOFP and READP to be false simultaneously, viz. when there is no input waiting (buffered locally), but the remote end of the stream has not indicated that there is no more input.

There are some who call EOFP on streams open only for output. This is a crock; output streams are always at end of file. But to avoid complaints, a device could return T for EOFP on an output stream.

(BLOCKIN *STREAM BUFFER BYTEOFFSET NBYTES*)

Performs bulk input transfer: retrieves the next *NBYTES* bytes from *STREAM* and stores them in successive byte positions in *BUFFER* starting at *BYTEOFFSET*. Defaults to \GENERIC.BINS, which repeatedly calls BIN and \PUTBASEBYTE.

It is almost always the case that a device with a non-trivial BLOCKIN method can be made to be a Buffered device, thereby benefiting from other Buffered operations as well.

(BLOCKOUT *STREAM BUFFER BYTEOFFSET NBYTES*)

Performs bulk output transfer: outputs *NBYTES* bytes to *STREAM*, taking the bytes from *BUFFER* starting at *BYTEOFFSET*. Defaults to \GENERIC.BOUTS, which repeatedly calls \GETBASEBYTE and BOUT.

(FORCEOUTPUT *STREAM WAITFORFINISH*)

Forces to its ultimate destination any output buffered on *STREAM* but not yet sent. *WAITFORFINISH* means that the function should not return until it is confident that the output has reached its destination and been committed. Defaults to NILL, which is reasonable for unbuffered streams.

For example, for a network stream, FORCEOUTPUT sends the current packet being buffered up. For a buffered stream to the disk, FORCEOUTPUT writes out to the disk any ‘‘dirty’’ pages, and makes sure the file is in such a state that if the machine were booted after FORCEOUTPUT returns, that the file could be successfully reopened with no information lost.

(GETFILEINFO *NAME/STREAM ATTRIBUTE DEVICE*)

Returns the value of the specified *ATTRIBUTE* of *NAME/STREAM*, which can be an open Stream or the name of a (closed) file. Returns NIL for attributes it doesn't know about. It is considered good citizenship, though not absolutely required, to know about the following attributes:

LENGTH — Length of the stream/file in bytes. If the device's method returns NIL, but the stream is random access, the generic GETFILEINFO code tries the device's GETEOFPTR method instead.

SIZE — Length in pages, i.e., (FOLDHI length BYTESPERPAGE).

CREATIONDATE — Date when the file's contents were created, as a string. The creationdate does not change when a file is copied or renamed, only when it is changed.

WRITEDATE — Date when the file was written to its current place of storage.

READDATE — Date when the file was last read.

ICREATIONDATE, IWRITEDATE, IREADDATE — The creation, write and read dates as integers, such as from the function IDATE.

TYPE — Type of the contents: TEXT for files that contain only ‘‘text’’ (generally meaning 7-bit ascii), BINARY for all others. NIL means unknown.

AUTHOR Name of the user who created the file (a string).

The following ‘‘generic’’ attributes are generally handled by the generic side of GETFILEINFO if the device’s GETFILEINFO method returns NIL:

EOL The end of line convention of the stream (CR, CRLF or LF).

BUFFERS The number of pagemap buffers for use by the stream (see description of MAXBUFFERS field of pagemapped streams).

ENDOFSTREAMOP Action to take on any attempt to read beyond the end of file. This is a function of one argument, the stream. The function can cause an error, or return a value, which is interpreted as a value to return from BIN. The default ENDOFSTREAMOP causes an END OF FILE error.

ACCESS An atom describing the access mode of the stream (INPUT, OUTPUT, etc). This is so generic that it is handled before the device’s method ever sees it.

BYTESIZE, OPENBYTESIZE The size of bytes transmitted on the stream. Always 8 these days.

(SETFILEINFO *NAME/STREAM ATTRIBUTE VALUE DEVICE*)

Sets the value of the specified *ATTRIBUTE* of *NAME/STREAM* to be *VALUE*. Returns T if successful, NIL if unsuccessful, or for attributes it doesn’t know about.

It is not generally required that SETFILEINFO recognize any attributes at all—NILL is a perfectly good filler for this slot. Most devices recognize no more than TYPE and CREATIONDATE (ICREATIONDATE), and even those are not very important, as most applications set those attributes in the *PARAMETERS* argument to OPENFILE when creating a file.

ATTRIBUTE = LENGTH implies actually truncating (or lengthening) the file; however, the SETFILEINFO need not handle this itself—if it returns NIL, then the generic file system will attempt to use the SETEOFPTR method instead.

The following operations are only required of random access streams. They default to the function \IS.NOT.RANDACCESSP, which causes a ‘‘Stream is not randaccessp’’ error when called.

(GETFILEPTR *STREAM*)

Returns the current file pointer (byte position) in *STREAM*. The file pointer is zero when the stream is opened (except for *ACCESS* = APPEND), and is incremented by one for each byte read.

Although this operation is only absolutely required for random access streams, it is desirable to supply it for other streams where possible. For example, when reading a file sequentially through PupFtp, the stream can count the bytes as they go by and thus give an accurate value for GETFILEPTR. If a stream has no idea at all of position, it can make its GETFILEPTR be the function ZERO and thereby at least avoid breaks from code that calls GETFILEPTR carelessly.

(GETEOFPTR *STREAM*)

Returns the file pointer of the end of *STREAM*, i.e., the file pointer that GETFILEPTR would return after the last byte of *STREAM* is read. Same as the LENGTH attribute for a stream that represents a file. Of course, non-random access streams may have no idea where the end is, and causing a non-randaccessp error is perfectly acceptable.

(SETFILEPTR *STREAM BYTENUMBER*)

Sets the file pointer of *STREAM* to be *BYTENUMBER*. The special value *BYTENUMBER* = –1 means the end of the stream; other negative values are illegal.

SETFILEPTR beyond the end of the stream is permissible, but it has no immediate effect beyond changing the logical file pointer. Attempting to then BIN causes an EOF error. Attempting to BOUT (for a file open for write) should extend the file, so that its eof is immediately beyond the newly BOUTed byte.

As with GETFILEPTR, there is no requirement that this work on non-random access streams, and it may be completely impossible on some of them. However, for those non-random access streams that perform GETFILEPTR, it is possible to fake SETFILEPTR for values larger than the current file pointer by skipping some number of bytes in the file, e.g., by performing (RPTQ (DIFFERENCE *BYTENUMBER* (GETFILEPTR *STREAM*)) (BIN *STREAM*)). There are some applications for which forward SETFILEPTR is all the random access that is actually required, so it is nice to be able to accommodate such applications.

(BACKFILEPTR *STREAM*)

Backs up the file pointer in *STREAM* by one byte. Functionally the same as (SETFILEPTR *STREAM* (SUB1 (GETFILEPTR *STREAM*)), but may be possible on non-random access streams by maintaining a one-character buffer, which is all the backing up this operation is formally required to perform. I believe the main use for this is in READ, which needs to back up the stream one character when, for example, it reads a break character terminating an atom.

(SETEOFPTR *STREAM LENGTH*)

Changes the length of *STREAM* to be *LENGTH*, i.e., ''sets'' its end of file pointer. This may require lengthening or truncating the file. Used by the function \SETEOFPTR and by SETFILEINFO for attribute LENGTH when the device's SETFILEINFO method doesn't handle it.

The following three fields are place holders for possible future extensions. These fields are not currently used at all:

(LASTC *STREAM*)

Returns the last character read from *STREAM*, i.e., the last byte that was BINed, as a character. LASTC is currently implemented via BACKFILEPTR.

(FREEPAGECOUNT *HOST/DIR DEVICE*)

Intended use is to return the number of free pages on *HOST/DIR*. May be folded into a general GET/SET device/directory info operation.

(MAKEDIRECTORY *HOST/DIR DEVICE*)

Intended use is to create a new directory *HOST/DIR*.

The remaining fields in the FDEV are for buffered and page-mapped streams, and are ignored for non-buffered devices. These fields are described in separate sections.

Streams

The following fields are used by all streams:

DEVICE Pointer to this stream's FDEV.

FULLFILENAME ''Full'' name by which this file is known to the user. Should be an uppercase litatom, fully

qualified so that giving the same name back to the file system should produce the same file (to the extent that the device can support such uniqueness). Is NIL for unnamed streams.

FULLNAME — Access field. Is the same as FULLFILENAME, unless that is NIL, in which case it is the stream itself. This avoids the circularity that would result if the FULLFILENAME field contained the stream datum.

NAMEDP — Access field. Is T if the streams is named, i.e., its FULLFILENAME is non-NIL.

ACCESSBITS — Contains a numeric code describing what access mode the file is open for: there are read, write and append bits. This field is usually accessed indirectly via the ACCESS field. However, there are macros for referring to particular types of access using more efficient bit test operations:

(OPENED *STREAM*)	ACCESS is not NIL.
(READABLE *STREAM*)	Read bit is on: ACCESS is INPUT or BOTH.
(READONLY *STREAM*)	Only the read bit is on: ACCESS is INPUT.
(APPENDABLE *STREAM*)	Append bit is on: ACCESS is OUTPUT, BOTH or APPEND.
(APPENDONLY *STREAM*)	Only the append bit is on: ACCESS is APPEND.
(DIRTYABLE *STREAM*)	Append or write bit is on: ACCESS is OUTPUT, BOTH or APPEND. Yes, this is operationally the same as APPENDABLE, given the four possible values of ACCESS.
(OVERWRITEABLE *STREAM*)	Write bit is on: ACCESS is OUTPUT or BOTH.
(WRITEABLE *STREAM*)	Write bit is on, or append bit is on and file is at EOF. Avoid using this one, it's a little strange.

ACCESS — Access field for referring to the ACCESSBITS field symbolically. Its value is one of the legal values of the *ACCESS* argument to OPENFILE: INPUT, OUTPUT, BOTH, APPEND; or NIL when the stream is closed. Replacing this field has the side effect of setting the BINABLE, BOUTABLE, STRMBINFN and STRMBOUTFN fields appropriately (from the corresponding device fields, or to values consistent with no access).

USERCLOSEABLE — Flag, true if the stream can be closed by CLOSEF. Default is T, but is NIL for such things as dribble files and the terminal.

USERVISIBLE — Flag, true if the stream is to be listed in the result of (OPENP). Default is T, but is NIL for such things as dribble files and the terminal.

BINABLE — True if BIN microcode can be used. Normally set automatically from FDBINABLE when input access is set.

BOUTABLE — True if BOUT microcode can be used. Normally set automatically from FDBOUTABLE when output access is set.

EXTENDABLE — True if BOUT can extend the buffer when COFFSET reaches CBUFSIZE. Obsolete.

STRMBINFN — Function called by BIN. This is normally set indirectly as a side effect of setting the ACCESS field. Setting ACCESS to an input access (INPUT or BOTH) sets the STRMBINFN to be the stream's device's BIN method. Setting to any other access sets the STRMBINFN

to be a ‘‘file not open’’ trap.

STRMBOUTFN — Function called by BOUT. As with STRMBINFN, this is normally set indirectly (from the device's BOUT method) as a side effect of setting the ACCESS field.

OUTCHARFN — Function called to output a single byte. This is like STRMBOUTFN, except for being one level higher: it is intended for text output. Hence, this function should convert (CHARCODE EOL) into the stream's actual end of line sequence, and should adjust CHARPOSITION appropriately before invoking the stream's STRMBOUTFN to actually put the character. Defaults to \FILEOUTCHARFN. The OUTCHARFN for the display additionally worries about such things as ECHOCONTROL.

CHARPOSITION — Current horizontal character position in the stream. Incremented (and reset to zero) by OUTCHARFN. Used by the function POSITION.

LINELENGTH — Maximum line length of the stream, in characters. Used by the function LINELENGTH. Defaults (at creation time) to the value of the global variable FILELINELENGTH.

EOLCONVENTION — The stream's end of line convention: the manner in which ‘‘end of line’’ is encoded on this stream. That is, output of an end of line (function TERPRI) produces the stream's end of line sequence, and on input, the stream's end of line sequence is converted to (CHARCODE EOL) by READC. This is not necessarily the same as the way that end of line is encoded in the actual file written by, say, a file server. For example, Lisp might open a stream to a Tenex file server with EOLCONVENTION of CR, while the server might choose to take each of the CRs in the stream and actually store a CR, LF sequence in the physical file.

The convention is encoded as a two-bit field; the constants CR.EOLC, LF.EOLC, CRLF.EOLC can be used to refer to the currently known values symbolically. Default in Interlisp-D is CR.EOLC.

ENDOFSTREAMOP — Function of one argument (the stream) called when an attempt to read beyond the end of file occurs. If this function returns something, it should be interpreted as a value to return from BIN (the value T is currently prohibited). Defaults to \EOSERROR, which causes an END OF FILE error.

VALIDATION — Pointer field, some compact encoding of the state of the file such that if the file's content changes, the VALIDATION changes. The file's ICREATIONDATE attribute usually works well enough. The only use for this field is to check whether the file changed over LOGOUT, etc.—if the VALIDATION of the stream returned from REOPENFILE is EQUAL to the VALIDATION of the stream open before LOGOUT, the stream is assumed to be unchnaged. This will probably be the sole concern of the device when we go to multiple streams per file.

BYTESIZE — Byte size of the file, i.e., what BIN and BOUT traffic in. Defaults to 8. This field is not used by many; there are probably a lot of things that won't work if the byte size is not 8.

OTHERPROPS — List in property list format used by the function STREAMPROP. Analogous to WINDOWPROP, etc.

IMAGEOPS — Image operations vector (object of type IMAGEOPS) for use of device-independent graphics operations, such as DSPXPOSITION, DSPFONT. Defaults to \NOIMAGEOPS, a vector suitably defined for non-display devices. See the implementors' manual chapter **Device-Independent Graphics**.

IMAGEDATA Device-dependent data for use by IMAGEOPS.

REVALIDATEFLG Flag. The standard use of this flag is to solve a problem with correctly maintaining the creation date. The problem is that the definition of ''creation date'' is that the creation date changes whenever the contents of the file change. If followed literally, this would mean, for example, that ever time you wrote out a page of a {DSK} file, you would also have to rewrite its leader page with a new creation date. However, it suffices in practice to only change the creation date when it would matter, i.e., when there would be any possibility of some agent other than the currently running Lisp to see the change. Usually, this means the only time to worry about is when the Lisp vmem is saved and a file that was open before the save is written to again afterwards.

Thus, the use of this flag (for those devices that care) is as follows: the device's BEFORExxx events set this flag true for any streams open on the device. Then, whenever the device is about to do something that would change the file's content, e.g., write out a new page, it first tests REVALIDATEFLG. If the flag is true, it updates the file's creation date and clears the flag.

NONDEFAULTDATEFLG Flag. Standard use is in conjunction with REVALIDATEFLG, to mark a file that was opened in a way that the user constrained the creation date of the file (e.g., the *PARAMETERS* argument to OPENFILE included an explicit creation date, or the option DON'T.CHANGE.DATE).

F1, F2, F3, F4, F5 Pointer fields for private use by the stream, to maintain stream-specific state of concern only to the device. Stream clients that wish to hang information on a stream without regard to what kind of stream it is should use the function STREAMPROP.

FW6, FW7, FW8, FW9 16-bit word fields for private use by the stream.

DIRTYBITS Obsolete.

EXTRASTREAMOP ?

Buffered Streams

Buffered streams are ones that constrain themselves to obey a set of conventions that make it easy for an agent (e.g., microcode) to perform input or output on the stream without knowing about the details of the stream's physical i/o. The stream maintains a ''current buffer'' and two indices into that buffer, the offset of the next byte, and the offset of the end of the buffer. As long as the former index is less than the latter, the stream guarantees that the bytes in the buffer between those indices are the true contents of the file/stream starting at the current file pointer. Advancing the first index effectively advances the file pointer. When it reaches the second index, a stream-specific operation is called to ''refill'' the buffer.

The following fields are used by buffered streams:

COFFSET Byte offset in the buffer CBUFPTR of the next BIN or BOUT.

CBUFSIZE ''Size'' of the current buffer, i.e., byte offset that is one beyond the last byte.

CBUFMAXSIZE For output, the maximum size the buffer can be written to. If COFFSET reaches CBUFSIZE, but CBUFSIZE is less than CBUFMAXSIZE, then the buffer can be extended.

CBUFPTR Pointer to current buffer. Must be valid if COFFSET is less than CBUFSIZE and BINABLE

or BOUTABLE is true. It is not necessary that this ''buffer'' be anything other than some chunk of memory, a portion of which contains interesting data. Thus, the bytes from offset COFFSET to CBUFSIZE must be valid, but COFFSET need not start at zero, nor need CBUFSIZE or CBUFMAXSIZE coincide with the end of the underlying structure.

CBUFDIRTY Flag, true if current buffer has been written to.

In general, the device has sole responsibility for setting CBUFSIZE, CBUFMAXSIZE, and CBUFPTR; generic code does not touch those. The fields COFFSET and CBUFDIRTY can be changed by generic stream clients as well as by device-specific code. For example, code that simulates a BIN increments COFFSET; code that writes directly to the stream's buffer sets CBUFDIRTY true.

The following methods are defined for devices implementing buffered streams:

(GETNEXTBUFFER *STREAM WHATFOR NOERRORFLG*) [Device method]

Called when *STREAM* needs to have its buffer fixed, i.e., the state of *STREAM* is such that BIN (*WHATFOR* = READ) or BOUT (*WHATFOR* = WRITE) cannot proceed. This method should do whatever is necessary to allow the operation to proceed. This typically includes disposing of the current buffer somehow (if GETNEXTBUFFER was invoked because the buffer was exhausted), and fetching a new buffer consistent with *STREAM*'s current position.

In the case of *WHATFOR* = READ, GETNEXTBUFFER returns T on success, i.e., if *STREAM* is not at end of file. When *STREAM* is at end of file, GETNEXTBUFFER should take standard end of stream action, returning whatever \EOF.ACTION returns (if anything). However, if *NOERRORFLG* is true, GETNEXTBUFFER should just return NIL immediately.

(RELEASEBUFFER *STREAM BUFFER*) [Device method]

Performs any device-specific operation required when *BUFFER*, which is the current value of *STREAM*'s CBUFPTR field, is ''released'' (when the CBUFPTR field is replaced). This is used so that different pagemap-like devices can share certain code. For example, in the case of pagemapped streams, RELEASEBUFFER marks the buffer dirty in the case that the stream's CBUFDIRTY field has been set.

This method is not currently used.

The functions \BUFFERED.BIN, \BUFFERED.PEEKBIN, \BUFFERED.BOUT, \BUFFERED.BINS and \BUFFERED.BOUTS are supplied for use by buffered streams; they are standardly used to implement the BIN, PEEKBIN, BOUT, BLOCKIN and BLOCKOUT device methods. In addition, the function COPYBYTES, when presented with a source stream that is buffered, utilizes the GETNEXTBUFFER method to efficiently copy bytes to the destination a buffer-full at a time.

Pagemapped Streams

Pagemapped streams are a particular kind of random access Buffered stream that buffers its data in units of pages. The device provides methods that read or write data in units of pages, while system-supplied Pagemapped functions handle the responsibilities of a Buffered stream, as well as managing the file pointer for random access. In general, a stream can have several pages of a file buffered at a time, allowing the code to make some effort to make efficient use of multi-paged transfers where applicable.

To create a pagemapped device, create an FDEV, fill in the necessary private fields, then call the following function:

(\MAKE.PMAP.DEVICE *DEVICE*) [Function]

Fills in fields in the device appropriate for pagemapped devices, and returns the updated device. The fields it fills are the flag fields FDBINABLE, FDBOUTABLE, RESETABLE, RANDOMACCESSP, PAGEMAPPED, BUFFERED (all true), and the methods BIN, BOUT, PEEKBIN, BLOCKIN, BLOCKOUT, READP, EOFP, GETFILEPTR, BACKFILEPTR, SETFILEPTR, GETEOFPTR, SETEOFPTR, GETNEXTBUFFER and FORCEOUTPUT.

A Pagemapped device is required to supply the following methods (in addition to those required of all devices and not filled in by \MAKE.PMAP.DEVICE):

(READPAGES *STREAM FIRSTPAGE# BUFFERS*) [Device method]

Causes pages of *STREAM* to be read into *BUFFERS*. The first page read is *FIRSTPAGE#* (zero for the first page of the file). *BUFFERS* is either a single page-sized buffer (a VMEMPAGEP), in which case exactly one page is read, or it is a list of such buffers. READPAGES returns the total number of bytes read. If the last page read is not a full page, READPAGES should zero out the rest of its buffer. READPAGES can assume that the buffers are page-aligned, although they need not be consecutive.

(WRITEPAGES *STREAM FIRSTPAGE# BUFFERS*) [Device method]

Writes data from *BUFFERS* out to *STREAM*. The first page written is *FIRSTPAGE#*. *BUFFERS* is as with READPAGES.

Neither READPAGES nor WRITEPAGES affects *STREAM*'s file pointer or end of file; those are managed by higher-level pagemapped routines. WRITEPAGES might, however, want to look at *STREAM*'s EPAGE and EOFFSET fields if it needs to take any special action around the end of the file. It is possible, for no particularly good reason, for READPAGES to get called for a page beyond the end of file; in fact, this standardly happens when writing a new file. The READPAGES method in this case should just clear the buffer and return zero.

(TRUNCATEFILE *STREAM PAGE# OFFSET*) [Device method]

Truncates *STREAM* so that its end of file is *PAGE#*, *OFFSET*, which should be defaulted to *STREAM*'s EPAGE and EOFFSET. Can be used to either shorten or lengthen a file; if lengthening, the file should be padded with nulls. Used by \PAGED.SETEOFPTR and \PAGED.FORCEOUTPUT. As of this writing there are still bugs in this code in certain funny cases, such as when you SETFILEPTR beyond eof and then BOUT.

The following fields of a stream are meaningful for a pagemapped device. The generic pagemapped codes maintain them as operations on the file are performed, but they should all be initialized appropriately by the device's OPENFILE method:

CPAGE — For pagemapped streams, the current page position in the stream. Together with COFFSET, this constitutes the stream's file pointer. The device's OPENFILE method should set CPAGE and COFFSET to zero, except for files opened with access APPEND, in which case they should be set to the end of file.

EPAGE, EOFFSET — For pagemapped files, the page and byte offset of the end of file. Note that this is the *logical* end of the file; it need have nothing to do with the physical end of file, except that when a file is closed, the device should see to it that its logical and physical EOFs are the same (normally seen to by the TRUNCATEFILE inside of \CLEARMAP, below). In fact, as a typical file is being written, EPAGE tends to stay several pages ahead of the physical end of file by virtue of the fact that pages are being buffered before being written out.

BUFFS — For pagemapped streams, a pointer to the stream's BUFFER chain. Initially NIL (no buffers allocated). The device usually has no direct interest in this field.

MAXBUFFERS — For pagemapped streams, the maximum number of buffers desired in the stream's BUFFS

chain. If the code needs another buffer and there are already MAXBUFFERS buffers, it will try to recycle the least recently referenced buffer. Defaults to \STREAM.DEFAULT.MAXBUFFERS. The user can change this field for an open stream by calling SETFILEINFO with attribute BUFFERS.

MULTIBUFFERHINT Flag. For pagemapped streams, is a hint to the pagemap code that the device prefers to transfer data more than one buffer at a time. If this flag is true, the pagemap code tries to write out (WRITEPAGES) more than one buffer at a time when the opportunity arises. A similar improvement is planned, but not implemented, for reading multiple buffers at a time.

The following functions are of use for pagemapped devices:

(\PAGED.FORCEOUTPUT *STREAM WAITFORFINISH*) [Function]

This function implements the FORCEOUTPUT method for pagemapped streams: it causes any dirty pages to be written out (using WRITEPAGES), then calls the TRUNCATEFILE method to set the end of file.

This function is normally installed as the FORCEOUTPUT method by the function \MAKE.PMAP.DEVICE. However, the device can override this default (by supplying its own function in that field), in which case it might want to call the function \PAGED.FORCEOUTPUT explicitly as part of its more comprehensive FORCEOUTPUT method.

There is an unpleasantness in the implementation of pagemapped devices that stems from the fact that originally all devices (the few that existed in the distant past) were made to support the PMAP package, a means whereby a programmer could get direct access to the buffers of a file, much as one can with the PMAP JSYS in Tenex. As a result, the buffers used by pagemapped streams are set up in a special manner so that the garbage collector can tell when the user no longer has access to a PMAP buffer. The PMAP package is being phased out.

This is all exceedingly crufty, and is of little concern to the device implementer, except for the fact that it requires that the buffers be explicitly released when a stream is closed; the buffers are not automatically collected when the stream is dropped.

(FORGETPAGES *STREAM FROMPAGE TOPAGE*) [Function]

''Forgets'' pages *FROMPAGE* thru *TOPAGE* of *STREAM*; i.e., removes those pages from the set of pages being currently buffered, and frees the buffers they were occupying. If *FROMPAGE* = *TOPAGE* = NIL, forgets all pages, and releases all of *STREAM*'s buffers.

(\CLEARMAP *STREAM*) [Function]

Performs a FORCEOUTPUT (if *STREAM* is open for output) followed by a FORGETPAGES. This is the standard action that should be taken by a pagemapped stream's CLOSEFILE method.

Directory Enumeration

This section describes how directory enumeration works—what you need to know in order to implement the GENERATEFILES device method, and what you need to know as a programmer trying to enumerate a directory via anything more elaborate than the function DIRECTORY.

The general idea is that the directory enumeration code is given a pattern, and it returns a generator that, each time it is poked, returns another file name matching the pattern. In addition, the generator provides a handle for getting file attributes of each enumerated file. This second handle is important for efficiency: although one could just take the

file name given by the enumerator and pass it to GETFILEINFO, the device, in the course of enumeration, usually has its fingers on the file closely enough that it need not perform the second directory lookup that a GETFILEINFO out of the blue would require. The caller of the directory enumeration code specifies ahead of time which, if any, attributes will be required (a necessity for most file server implementations).

Information for device implementors. A *file generator* is an object represented as a list described by the record FILEGENOBJ, exported from FILEIO:

```
(RECORD FILEGENOBJ (NEXTFILEFN FILEINFOFN . GENFILESTATE))
```

NEXTFILEFN and FILEINFOFN are functions of the device's choosing that when called will return the next file, and attributes for that file. GENFILESTATE is arbitrary state maintained by the generator. With that as background, here are the pieces of directory enumeration:

(GENERATEFILES *DEVICE PATTERN DESIREDPROPS OPTIONS*) [Device method]

Returns a generator that enumerates files matching *PATTERN*, which is a string that has host and directories suitably filled in from defaults, and may contain the pattern character ''*'' to match an arbitrary number of characters. *DESIREDPROPS* is a list of file attributes that may be requested during the enumeration; they must be valid *ATTRIBUTE* arguments to GETFILEINFO. *OPTIONS* is a list of options to the enumeration, chosen from among the following:

SORT — The files should be enumerated in sorted order. If this option is not specified, the device is free to enumerate files in any convenient order.

There is some question as to whether files should be enumerated lowest version first (as IFS's do) or highest version first (as Twenex does). I prefer the latter, but given servers that do the former, we currently make no requirement about version order.

RESETLST — Informs the enumerator that the enumeration context is surrounded by a RESETLST, so that it may perform RESETSAVEs to clean up after itself if the enumeration is aborted. Cleaning up can be a very messy business without this information about the scope of the enumeration, so all callers of \GENERATEFILES are strongly encouraged to provide it.

GENERATEFILES should return a file generator with a suitable NEXTFILEFN and FILEINFOFN.

Fine point about missing fields in the pattern: null fields in *PATTERN* match only files for which the corresponding field is null. A null version is interpreted as highest. Thus,

DIR *.* = DIR *.* = DIR *.*;* enumerates everything.

DIR *. = DIR *.;* enumerates all versions of files with null extension.

DIR *.; enumerates highest version of files with null extension.

DIR *.*; enumerates highest version of everything.

It is difficult for some devices to enumerate only highest version of files; there are several devices in the system that treat a null version the same as version *. However, every device should try its best. With some work, any device that can enumerate all versions can enumerate just highest version if it enumerates in sorted order and uses perhaps a little lookahead to assure that any name it returns is the one of highest version.

(NEXTFILEFN *GENFILESTATE NAMEONLY*) [File Generator Component]

Generates the next file, returning its name as a string, or NIL if the generator is exhausted.

GENFILESTATE is the state component of the file generator returned from GENERATEFILES. *NAMEONLY* means that the caller is only interested in the file's Name.Ext fields, not the full file name (and no more than one version of the file need be enumerated); however, it is always permissible to return the full file name. The *NAMEONLY* option is used by SPELLFILE.

(FILEINFOFN *GENFILESTATE ATTRIBUTE*) [File Generator Component]

Returns the value of the *ATTRIBUTE* property of the file most recently generated by the NEXTFILEFN, i.e., effectively (GETFILEINFO latest-name *ATTRIBUTE*), but hopefully much faster. *ATTRIBUTE* must have been a member of the *DESIREDPROPS* argument to GENERATEFILES.

Not all device implementors are enthused about implementing a pattern matcher for file names. The following functions are provided to help out:

(DIRECTORY.MATCH.SETUP *PATTERN*) [Function]

Accepts as *PATTERN* a file name string such as passed to GENERATEFILES. Returns an object suitable as a filter to DIRECTORY.MATCH.

(DIRECTORY.MATCH *FILTER TESTNAME*) [Function]

Matches *TESTNAME*, a file name, against *FILTER*, the object returned from DIRECTORY.MATCH.SETUP. Returns true if *TESTNAME* matches the pattern, false if not. The match is case-insensitive.

(\NULLFILEGENERATOR) [Function]

Returns a file generator that produces no files.

(\GENERATENOFILES *DEVICE PATTERN DESIREDPROPS OPTIONS*) [Function]

Returns a ''stupid'' file generator for devices that don't know how to enumerate in general. If *PATTERN* contains no wildcards, but names a file that is INFILEP, then the generator produces exactly that file. If *PATTERN* contains a wildcard in the version field, it uses GETFILENAME to laboriously generate all the versions of the file. In all other cases, \GENERATENOFILES returns a null file generator.

Information for clients of device enumeration. The following functions make up the ''public'' interface to directory enumeration:

(\GENERATEFILES *PATTERN DESIREDPROPS OPTIONS*) [Function]

Returns a file generator object for enumerating the files matching *PATTERN*. *PATTERN* is expanded by adding the default host and/or directory if appropriate. See description of the GENERATEFILES method for description of *DESIREDPROPS* and *OPTIONS*.

(\GENERATENEXTFILE *GENERATOR NAMEONLY*) [Function]

Returns the next file, as a string. *GENERATOR* is the object returned from \GENERATEFILES; *NAMEONLY* indicates caller does not require that the full name be returned, but that the name and extension are sufficient.

(\GENERATEFILEINFO *GENERATOR ATTRIBUTE*) [Function]

Returns the value of the *ATTRIBUTE* property of the file most recently generated by \GENERATENEXTFILE, i.e., effectively (GETFILEINFO latest-name *ATTRIBUTE*). *ATTRIBUTE* must have been a member of the *DESIREDPROPS* argument to \GENERATEFILES.

(DIRECTORY.FILL.PATTERN *PATTERN DEFAULTEXT DEFAULTVERS*) [Function]

This function is used to fill in defaults in *PATTERN* before passing it to \GENERATEFILES. If *PATTERN* does not include an extension or version, but those fields are not explicitly omitted (e.g., ''FOO'', but not ''FOO.''; ''FOO.BAR'', but not ''FOO.BAR;''), they are filled in with *DEFAULTEXT* and *DEFAULTVERS*, which themselves default to ''*''. This function is used by the DIR command, and should probably be used by any code that takes a user-supplied pattern and enumerates files from it.