5 Reference Guide to Streams, Files, and I/O

symbolics

5 Reference Guide to Streams, Files, and I/O

symbolics™

Reference Guide to Streams, Files, and I/O

# 999006

August 1986

Symbolics, Inc.
4 New England Tech Center
555 Virginia Road
Concord, MA 01742

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.
Text masters produced on Symbolics 3600-family computers and printed on Symbolics LGP2 Laser Graphics Printers.
Cover Design: Schafer|LaCasse
Printer: CSA Press
Printed in the United States of America.

Printing year and number: 88 87 86 9 8 7 6 5 4 3 2 1

Table of Contents

List of Figures

List of Tables

PART I.

Introduction to the I/O System

Symbolics Common Lisp provides a powerful and flexible system for performing input and output to peripheral devices. To allow device-independent I/O (that is, to allow programs to be written in a general way so that the program's input and output may be connected with any device), the I/O system provides the concept of an "I/O stream". What streams are, the way they work, and the functions to create and manipulate streams, are described in this document. This document also describes the Lisp "I/O" operations **read** and **print**.

PART II.

Streams

1. Introduction to Streams

Many programs accept input characters and produce output characters. Methods for performing input and output vary greatly from one device to another. Programs should be able to use any device available without each program having to know about each device.

The concept of *streams* solves this problem. A stream is a source and/or sink of data. A set of *operations* is available with every stream; operations include such actions as "output a character" and "input a character". The way to perform an operation to a stream is the same for all streams, although what happens inside a stream depends on the kind of stream it is. Thus a program needs to know only how to deal with streams in general.

In Genera, streams are implemented as flavors. You can operate on a stream by using generic functions or by sending it messages, depending on what type of operations the stream supports. Flavors, generic functions, and message-passing are described elsewhere: See the section "Flavors" in *Symbolics Common Lisp: Language Concepts*.

Some streams can do only input, some only output, and some can do both. Some streams support only some operations; however, unsupported operations might work, although slowly, because the **sys:stream-default-handler** can handle them. An operation called **:which-operations** returns a list of the names of all operations that are supported "natively" by a stream. (All streams support **:which-operations**, so it might not be in the list itself.)

2. Types of Streams

2.1 Standard Common Lisp Streams

Several variables whose values are streams are used by many functions in the Lisp system. By convention, variables that are expected to hold a stream capable of input have names ending with **-input**. Similarly, variables expected to hold a stream capable of output have names ending with **-output**. Those expected to hold a bidirectional stream have names ending with **-io**.

The variables ***standard-input***, ***standard-output***, ***error-output***, ***trace-output***, ***query-io***, and ***debug-io*** are initially bound to synonym streams that pass all operations on to the stream that is the value of ***terminal-io***. Thus any operation performed on those streams goes to the terminal.

No user program should ever change the value of ***terminal-io***. For example, a program to divert output to a file should do so by binding the value of ***standard-output***; that way, error messages sent to ***error-output*** can still get to the user by going through ***terminal-io***, which is usually what is desired.

standard-input *Variable*

In the normal Lisp top-level loop, input is read from whatever stream is the value of ***standard-input***. Many input functions, including **read** and **read-char**, take a stream argument that defaults to ***standard-input***.

standard-output *Variable*

In the normal Lisp top-level loop, output is sent to whatever stream is the value of ***standard-output***. Many input functions, including **write** and **write-char**, take a stream argument that defaults to ***standard-output***.

error-output *Variable*

The value of ***error-output*** is a stream to which error messages should be sent. Normally, this is the same as ***standard-output***, but ***standard-output*** might be bound to a file and ***error-output*** left going to the terminal or a separate file of error messages.

terminal-io *Variable*

The value of ***terminal-io*** is ordinarily the stream that connects to the user's console. In an "interactive" program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the terminal. However, in a "background" program that normally does not talk to the user, ***terminal-io*** defaults to a stream that does not expect to be used. If it is used, perhaps by an error notification, it turns into a "background" window and requests the user's attention.

query-io *Variable*

The value of ***query-io*** is a stream to be used when asking questions of the user. The question should be output to this stream, and the answer read from it. When the normal input to a program comes from a file, questions such as "Do you really want to delete all of the files in your directory?" should be sent directly to the user and the answer should come from the user also, not from the data file. For these purposes, ***query-io*** should be used instead of ***standard-input*** and ***standard-output***. ***query-io*** is used by such functions as **yes-or-no-p**.

debug-io *Variable*

The value of ***debug-io*** is a stream to be used for interactive debugging purposes.

trace-output *Variable*

The value of ***trace-output*** is the stream on which the **trace** function prints its output.

2.2 Standard Zetalisp Streams

The variables **zl:standard-input**, **zl:standard-output**, **zl:error-output**, **zl:trace-output**, and **zl:query-io** are initially bound to synonym streams that pass all operations on to the stream that is the value of **zl:terminal-io**. Thus any operation performed on those streams goes to the terminal.

These variables are synonyms for the Common Lisp variables with similar names. For example, **zl:standard-input** is a synonym for ***standard-input***. When writing new programs, you should use the Common Lisp variables instead of the Zetalisp variables.

zl:standard-input *Variable*

In the normal Lisp top-level loop, input is read from **zl:standard-input** (that is, whatever stream is the value of **zl:standard-input**). Many input functions, including **zl:tyi** and **zl:read**, take a stream argument that defaults to **zl:standard-input**. **zl:standard-input** is equivalent to ***standard-input***.

zl:standard-output *Variable*

In the normal Lisp top-level loop, output is sent to **zl:standard-output** (that is, whatever stream is the value of **zl:standard-output**). Many output functions, including **zl:tyo** and **print**, take a stream argument that defaults to **zl:standard-output**. **zl:standard-output** is equivalent to ***standard-output***.

zl:error-output *Variable*

The value of **zl:error-output** is a stream to which error messages should be sent. Normally this is the same as **zl:standard-output**, but **zl:standard-output** might be bound to a file and **zl:error-output** left going to the terminal. **zl:error-output** is equivalent to ***error-output***.

zl:query-io *Variable*

The value of **zl:query-io** is a stream that should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program might be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. **zl:query-io** is used by **fquery** and related functions. **zl:query-io** is equivalent to ***query-io***.

zl:terminal-io *Variable*

The value of **zl:terminal-io** is the stream that connects to the user's console. In an "interactive" program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the terminal. However, in a "background" program that does not normally talk to the user, **zl:terminal-io** defaults to a stream that does not ever expect to be used. If it is used, perhaps by an error notification, it turns into a "background" window and requests the user's attention. **zl:terminal-io** is equivalent to ***terminal-io***.

zl:trace-output *Variable*

The value of **zl:trace-output** is the stream on which the **trace** function prints its output. **zl:trace-output** is equivalent to ***trace-output***.

zl:debug-io *Variable*

If not **nil**, this is the stream that the Debugger should use. The default value is a synonym stream that is synonymous with **zl:terminal-io**. If the value of **dbg:*debug-io-override*** is not **nil**, the Debugger uses the value of that variable as the stream instead of the value of **zl:debug-io**.

The value of **zl:debug-io** can also be a string. This causes the debugger to use the cold-load stream; the string is the reason why the cold-load stream should be used.

No program other than the Debugger should do stream operations on the value of **zl:debug-io**, since the value cannot be a stream. Other programs should use **zl:query-io**, **zl:error-output**, or **zl:trace-output**. **zl:debug-io** is equivalent to ***debug-io***.

dbg:*debug-io-override* *Variable*

This is used during debugging to divert the Debugger to a stream that is known to work. If the value of this variable is **nil** (the default), the Debugger uses the stream that is the value of **zl:debug-io**. But if the value of **dbg:*debug-io-override*** is not **nil**, the Debugger uses the stream that is the value of this variable instead. This variable should always be set (using **setq**), not bound, so all processes and stack groups can see it.

2.3 Coroutine Streams

Functions that produce data as output (output functions) are written in terms of **:tyo** and other output operations. Functions that receive data as input (input functions) are written in terms of **:tyi** and other input operations. Output functions operate on output streams, which handle the **:tyo** message. Input functions operate on input streams, which handle the **:tyi** message. Sometimes it is desirable to view an output function as an input stream, or an input function as an output stream. You can do this with coroutine streams.

Here is a simplified explanation of how coroutine streams work. A coroutine input stream can be built from an output function. Whenever that stream receives a **:tyi** message, it invokes the output function in a separate stack group so that the function can produce the data that the **:tyi** message returns. A coroutine output stream can be built out of an input function; it works in the opposite fashion. Whenever the output stream receives a **:tyo** message, it invokes the input function in a separate stack group so that the function can receive the data transmitted by the **:tyo** message. It is also possible to connect functions that do both input and output, by using bidirectional coroutine streams. Since you can use coroutine streams to connect two functions, they are the logical inverse of **stream-copy-until-eof**, a function used to connect two streams.

To create a coroutine stream, use one of three functions.

- If you want to make an input stream from an output function, use **si:make-coroutine-input-stream**.
- If you want to make an output stream to an input function, use **si:make-coroutine-output-stream**.
- If you want to make a bidirectional stream for a function that does both input and output, use **sys:make-coroutine-bidirectional-stream**.

Following is an example using a coroutine input stream:

```
(setq input-stream
      (si:make-coroutine-input-stream
        #'(lambda (stream) (print-disk-label 0 stream))))
```

```
(send input-stream ':line-in) →
  "1645 free, 260499//262144 used (99%)"
```

Following is an example using a coroutine output stream:

```
(setq output-stream
      (si:make-coroutine-output-stream
        #'(lambda (stream) (setq x (read stream)))))

(send output-stream ':string-out "(a b c)")

(send output-stream ':force-output)

x → (A B C)
```

Coroutine streams are implemented as buffered character streams. Each function that makes a coroutine stream actually creates two streams and one new stack group. One stream is associated with the new stack group and the other stream with the stack group that is current when the stream-making function is called. If you use **si:make-coroutine-input-stream** or **si:make-coroutine-output-stream**, one stream is an input stream and the other is an output stream; they share a common buffer. If you use **sys:make-coroutine-bidirectional-stream**, both streams are bidirectional; the input buffer of each stream is the output buffer of the other.

With **si:make-coroutine-input-stream**, the output function runs in the new stack group. With **si:make-coroutine-output-stream**, the input function runs in the new stack group. With bidirectional streams, the function that does input or output runs in the new stack group.

In the case of **si:make-coroutine-input-stream**, for example, you typically send **:tyi** messages to the input stream that **si:make-coroutine-input-stream** returns. The input stream is associated with the new stack group. When the input stream receives a **:tyi** message, the new stack group is resumed, and the output function runs in that stack group. The output function typically sends **:tyo** messages to the output stream associated with the stack group from which **si:make-coroutine-input-stream** was called. When the output stream receives a **:tyo** message, the associated stack group is resumed. The data transmitted to the output stream become input to **:tyi** via the buffer that the two streams share. **si:make-coroutine-output-stream** and **sys:make-coroutine-bidirectional-stream** work in analogous fashion.

In addition to **:tyi** and **:tyo**, coroutine streams support other standard input and output operations, such as **:line-in** and **:string-out**. Actually, the **:next-input-buffer** method of the input stream and the **:send-output-buffer** method of the output stream resume the new stack group, not the receipt of **:tyi** and **:tyo** messages. Because the streams are buffered, you must send a **:force-output** message to an output stream to cause the new stack group to be resumed.

The instantiable flavors of coroutine streams are **si:coroutine-input-stream**, **si:coroutine-output-stream**, and **si:coroutine-bidirectional-stream**.

Do not confuse coroutine streams with pipes. Coroutine streams are used for intraprocess communication; pipes are used for interprocess communication. 3600-family machines do not currently support pipes.

si:make-coroutine-input-stream *function* &rest *arguments* *Function*

Creates two coroutine streams, an input stream and an output stream, with a shared buffer. **si:make-coroutine-input-stream** returns the input stream. The input stream is associated with a new stack group and the output stream with the stack group that is current when **si:make-coroutine-input-stream** is called. **:tyi** messages to the input stream cause the new stack group to be resumed and *function* to be called from that stack group. The first argument to *function* is the output stream; any additional arguments come from *arguments*. *function* should send **:tyo** messages to the output stream. These messages resume the stack group in which **si:make-coroutine-input-stream** was called. In this way, output from *function* becomes input to the caller of **si:make-coroutine-input-stream** through the shared buffer.

si:make-coroutine-output-stream *function* &rest *arguments* *Function*

Creates two coroutine streams, an output stream and an input stream, with a shared buffer. **si:make-coroutine-output-stream** returns the output stream. The output stream is associated with a new stack group and the input stream with the stack group that is current when **si:make-coroutine-output-stream** is called. **:tyo** messages to the output stream cause the new stack group to be resumed and *function* to be called from that stack group. The first argument to *function* is the input stream; any additional arguments come from *arguments*. *function* should send **:tyi** messages to the input stream. These messages resume the stack group in which **si:make-coroutine-output-stream** was called. In this way, output from the caller of **si:make-coroutine-output-stream** becomes input to *function* through the shared buffer.

sys:make-coroutine-bidirectional-stream *function* &rest *arguments* *Function*

Creates two bidirectional coroutine streams. The input buffer of each stream is the output buffer of the other. One stream is associated with a new stack group and the other with the stack group that is current when **sys:make-coroutine-bidirectional-stream** is called. **sys:make-coroutine-bidirectional-stream** returns the stream associated with the new stack group.

:tyi and **:tyo** messages to the stream associated with the new stack group cause that stack group to be resumed and *function* to be called from that stack group. The first argument to *function* is the stream associated with the stack group from which **sys:make-coroutine-bidirectional-stream** was called. Any additional arguments come from *arguments*. *function* should send **:tyi** or **:tyo** messages to the stream that is its first argument. These messages resume the stack group in which **si:make-coroutine-output-stream** was called. In this way *function* and the caller of **sys:make-coroutine-bidirectional-stream** communicate through the shared buffers; output from one function becomes input to the other.

si:coroutine-input-stream *Flavor*

Coroutine input stream. Defines a **:next-input-buffer** method. Use this to construct an input stream from a function written in terms of output operations.

si:coroutine-output-stream *Flavor*

Coroutine output stream. Defines **:new-output-buffer** and **:send-output-buffer** methods. Use this to construct an output stream to a function written in terms of input operations.

si:coroutine-bidirectional-stream *Flavor*

Bidirectional coroutine stream. Defines **:next-input-buffer**, **:new-output-buffer**, and **:send-output-buffer** methods. Use this to construct a bidirectional stream to a function written in terms of input and output operations.

2.4 Direct Access File Streams

Direct access file streams are supported by LMFS. They are designed to facilitate reading and writing data from many different points in a file. They are typically used to construct files organized into discrete extents or *records*, whose positions within a file are known by programs that access them in nonsequential order. Although this could be done with the **:set-pointer** message to input file streams, the direct access facility provides the following additional functions:

- Direct access to output files.
- Bidirectional file streams, which allow interspersed reading and writing of data to and from varied locations in a file.
- No use of network connections or file buffers during the time between data reading and the next call to position. In contrast, using the **:set-pointer**

message with ordinary ("sequential") input file streams incurs a significant network and data transfer overhead if the program repeatedly positions, reads several bytes, and then computes for a time.

2.4.1 Stream Messages

The following messages are relevant to direct access file streams.

:read-bytes *n-bytes file-position* *Message*

Sent to a direct access input or bidirectional file stream, this requests the transfer of *n-bytes* bytes from position *file-position* of the file. The message itself does not return any data to the caller. It causes the stream to be positioned to that point in the file, and the transfer of *n-bytes* bytes to begin. An EOF is sent following the requested bytes. The bytes can then be read using **:tyi**, **:string-in**, or any of the standard input messages or functions.

The stream enforces the byte limit, and presents an EOF if you attempt to read bytes beyond that limit. You must actually read all the bytes and read past (that is, consume from the stream) the EOF.

It is also possible, before all the bytes have been read, to perform stream operations other than reading bytes. For example, an application might read several records at a time, to optimize transfer and buffering, and decide, after reading the first record, to position somewhere else. Direct access file streams handle this properly. Nevertheless, network and buffering resources allocated to the stream (both on the local machine and server machine) are not freed unless all the requested bytes (of the last **:read-bytes** request) and the EOF following them are read.

If you request more bytes than remain in the file, you receive the remaining bytes followed by EOF.

2.4.2 Direct Access Output File Streams

You create direct access output to output and bidirectional direct access file streams by sending a **:set-pointer** message to the stream, and beginning to write bytes using standard messages, such as **:tyo**, **:string-out**, and so forth. The bytes are written to the file starting at the location requested, at successive file positions. Although you can extend the file in this manner, you cannot do a **:set-pointer** to *beyond* the current end of the file.

Direct access output, therefore, consists of sequences of **:set-pointer** messages and data output. Data are not guaranteed to actually appear in the file until either the stream is closed or a **:finish** message is sent to the stream. See the message **:finish**, page 41.

2.4.3 Direct Access Bidirectional File Streams

Bidirectional direct access file streams combine the features of direct access input and output file streams. Sequences of **:read-bytes** messages and reading data can be interspersed with sequences of **:set-pointer** messages and writing data. The stream is effectively switched between "input" and "output" states by the **:read-bytes** and **:set-pointer** messages. You cannot read data with **:tyi** or similar messages if a **:set-pointer** message has been sent to the stream since the last **:read-bytes** message. Similarly, you cannot write data with **:tyo** or similar messages unless a **:set-pointer** message has been sent to the stream since the last **:read-bytes** or **:tyi** messages, or similar operation.

When the EOF of a byte sequence requested with a **:read-bytes** message has been read for a bidirectional stream, the system frees network and buffering resources.

2.4.4 Effect of Character Set Translation on Direct Access File Streams

The Symbolics generic file access protocol was designed to provide access to ASCII-based file systems for Symbolics computers. Symbolics machines support 8-bit characters and have 256 characters in their character set. This results in difficulties when communicating with ASCII machines that have 7-bit characters.

The file server, on machines not using the Symbolics character set, is required to perform character translations for any character (not binary) opening. Some Symbolics characters expand to more than one ASCII character. Thus, for character files, when we speak of a given position in a file or the length of a file, we must specify whether we are speaking in *Symbolics units* or *server units*.

This causes major problems in file position reckoning. It is useless for the Symbolics machine (or other user side) to carefully monitor file position, counting characters, during output, when character translation is in effect. This is because the operating system interface for "position to point *x* in a file", which the server must use, operates in server units, but the Symbolics machine (or other user end) has counted in Symbolics units. The user end cannot try to second-guess the translation-counting process without losing host independence.

Since direct access file streams are designed for organized file position management, they are particularly susceptible to this problem. As with other file streams, it is only a problem when character files are used.

You can avoid this problem by always using binary files. If you must use character files, consider doing one of the following:

- Know the expansions of the Symbolics machine, that is, characters such as Return that do not expand into single host characters. Note that this sacrifices host independence.

- Do not use these characters. See the section "NFILE Character Set Translation" in *Networks*. This section explains which characters are expanded on the Symbolics computer.

2.5 Hardcopy Streams

The functions in this chapter are provided so that you can write an interface between an applications program and a supported printer. That is, they handle the process of getting the thing to be hardcopied to the printer. They assume that you have one of the printers supported by Symbolics; documentation is not provided to write the support for a different type of printer.

2.5.1 Supported Hardcopy Devices

Symbolics currently supports three types of hardcopy devices: LGP1, LGP2, and DMP1.

The Symbolics LGP1 printer is a laser-beam printer using a Canon print engine and Symbolics proprietary software.

The Symbolics LGP2 laser graphics printer is a table-top laser-beam printer based on the Apple LaserWriter, extended with proprietary Symbolics software.

The Symbolics DMP1 Dot-Matrix Printer is a compact, heavy-duty, impact dot-matrix printer with 24-wire print head.

2.5.2 The Hardcopy Stream Model

The interface between a 3600-family computer and a particular printing device is implemented using a *hardcopy stream*. A hardcopy stream is an output stream. (See the section "Types of Streams", page 7.) It handles the usual output operations, such as **:tyo**, **:line-out**, and **:string-out**. In addition it handles operations such as **:set-cursorpos** and **:allocate-margins**. (See the section "Using Hardcopy Streams", page 19.) It can handle page breaks and formatting information that you specify when the stream is created.

The various hardcopy menus and commands accept a pathname as an argument. **hardcopy:hardcopy-text-file** or **press:hardcopy-press-file** is then called, depending on the type of file, as determined from the file-type extension in the pathname or as specified by the user. **hardcopy:hardcopy-text-file** and **press:hardcopy-press-file** are the *front end* functions that make sure the appropriate file type and format keywords are included with the file. These functions call **hardcopy:hardcopy-file**, which opens the file and then calls the appropriate *formatting* function, **hardcopy:hardcopy-from-stream** or **press:hardcopy-press-stream**, to handle the creation of a hardcopy stream and the actual sending of escape codes to a printer object.

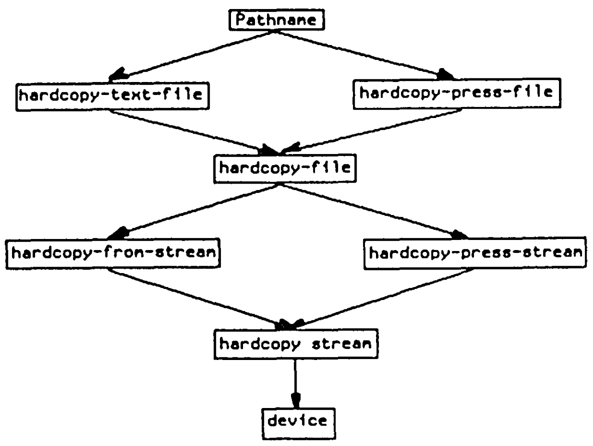


Figure 1. The Hardcopy Stream Model

2.5.3 Making Hardcopy Streams

The functions that create and manipulate hardcopy streams live in the package **zl-user:hardcopy**, with the exception of those functions that handle press format files, which live in the package **zl-user:press**.

The basic function to create a hardcopy stream is **hardcopy:make-hardcopy-stream**:

hardcopy:make-hardcopy-stream *device* &rest *options* *Function*

Returns a hardcopy stream to the given device. *options* can be any of the hardcopy option keywords. See the section "Hardcopy Options", page 26. **hardcopy:make-hardcopy-stream** creates a stream built on **hardcopy:basic-hardcopy-stream** with characteristics determined by the keyword options you specify. This stream accepts the normal output stream messages, such as **:tyo**, **:string-out**, and some specific hardcopy messages.

For example:

```
(with-open-stream
  (stream (hardcopy:make-hardcopy-stream hardcopy:*default-text-printer*))
  (send stream :string-out "this is a test
of the hardcopy system."))
```

hardcopy:get-hardcopy-device *device* &optional (*error-p* **t**) *Function*

Returns a software object named by *device* that can send data to a hardcopy device. Typical hardcopy devices are printers, files, and windows.

```
(with-open-stream
        (stream (hardcopy:make-hardcopy-stream
                  (hardcopy:get-hardcopy-device device)))
        (send stream :string-out "this is a test."))
```

device can be:

- A string treated as the name of a printer. For example: `"Tattler"`.
- A form that evaluates to a printer. For example: **hardcopy:*default-text-printer***
- A list of **'(:window** *window***)** representing a particular window to which to send the output. For example:

  ```
  '(:window (make-window tv:window :expose-p t))
  ```

  See the section "Getting a Window to Use" in *Programming the User Interface, Volume B.* A list of **'(:file** *pathname* &optional *canonical-type***)** that sends bytes to *pathname* in the appropriate format. For example:

  ```
  '(:file "q:>kjones>my-output.text" :lgp2)
  ```

  Creates a file suitable for printing on an LGP2.

- **:debug**, which means print the a description of each message sent to the hardcopy stream to terminal-io. For example:

  ```
  (with-open-stream
          (stream (hardcopy:make-hardcopy-stream
                    (hardcopy:get-hardcopy-device :debug)))
          (send stream :string-out "this is a test."))
  ```

Produces:

```
Set font 0 (#<FONT CPTFONT 107216574>).
Set cursorpos X=0. micas, Y= 26940 micas.
this is a test.
Eject page (eof).
NIL
```

- **:window**, which means create a special window to accept the output.

2.5.4 Using Hardcopy Streams

hardcopy:basic-hardcopy-stream *Flavor*

The basic flavor upon which all hardcopy streams are built. Any hardcopy stream handles the operations defined by the methods of **hardcopy:basic-hardcopy-stream.**

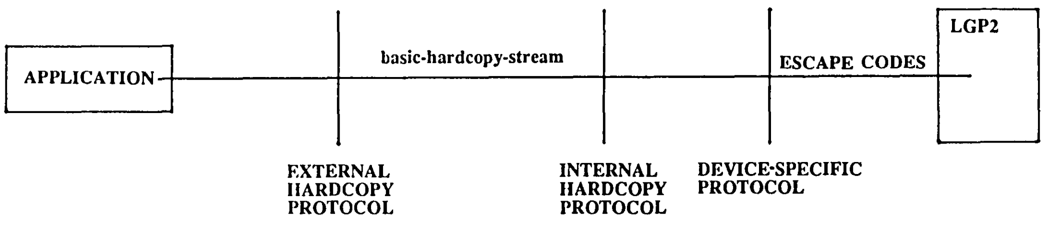


Figure 2. An Instance of a Hardcopy Stream

Hardcopy stream operations measure distances in *pixels* and *micas*. A pixel is a device-dependent measurement. It is one dot on a printer and how large it is depends on the resolution of the device. Device-dependent distances are expressed in pixels.

A mica is a unit of distance equal to 10 microns. Absolute distances are expressed in micas.

The operations handled by hardcopy streams are:

:show-rectangle *width height* of **hardcopy:basic-hardcopy-stream** *Method*

Draws a filled-in rectangle on the page with the lower left corner at the current cursor position of size *width* by *height*. If you are not sure of the current cursor position, use **:set-cursorpos** before **:show-rectangle**. You should not depend on the cursor position after using **:show-rectangle**. If you need the cursor position, do a **:set-cursorpos** after this operation.

width and *height* are always in device-dependent units. Use **:convert-to-device-units** to convert from other units.

:show-line *to-x to-y* of **hardcopy:basic-hardcopy-stream** *Method*

Draws a line on the page from the current position to the position designated by *to-x*, *to-y*. You should not depend on the cursor position after using **:show-line**. If you need the cursor position, do a **:set-cursorpos** after this operation.

The coordinates given to this message are absolute coordinates. If you have coordinates relative to the page margins, for instance arguments to **:set-cursorpos**, use **:un-relative-coordinates** to convert.

:read-cursorpos &optional (*units* **':device**) of **hardcopy:basic-hardcopy-stream** *Method*

Returns the current position of the cursor in *units*, either **:micas** or **:device**. The default is **:device**, meaning the units are device dependent.

:set-cursorpos *x y* &optional (*units* **':device**) of **hardcopy:basic-hardcopy-stream** *Method*

Moves the place where printing occurs on the page to a new position. Unlike the Symbolics console display, the 0,0 point of hardcopy streams is in the lower left corner, the first (*x*) coordinate increasing toward the right of the page, the second (*y*) coordinate increasing toward the top of the page. The coordinates are relative to the margins of the page. If you need absolute coordinates, use **:un-relative-coordinates** to convert.

units specifies the format of *x* and *y*. **:device** means that the interpretation is device dependent. **:micas** means *x* and *y* are in micas.

A value of **nil** for a coordinate means do not set that coordinate. For example,

```
(send stream :set-cursorpos nil 10)
```

sets the cursor position 10 device units above the bottom of the page and leaves its horizontal position unchanged.

:increment-cursorpos *dx dy* &optional *units* of **hardcopy:basic-hardcopy-stream** *Method*

Changes the point on the page where printing occurs with respect to the current position. Unlike the Symbolics console display, the 0,0 point of hardcopy streams is in the lower left corner, the first (x) coordinate increasing toward the right of the page, the second (y) coordinate increasing toward the top of the page.

units specifies the format of x and y. The default is **:device**.

A value of **0** for a coordinate means do not change that coordinate. For example,

```
(send stream :increment-cursorpos 0 10)
```

raises the cursor position 10 device units above where it was and leaves its horizontal position unchanged.

:un-relative-coordinates *x y* &optional (*units* **':device**) of **hardcopy:basic-hardcopy-stream** *Method*

Converts the point *x,y* given to messages like **:set-cursorpos** that take coordinates relative to the page margins to absolute coordinates for use with messages like **:show-line**.

:convert-to-device-units *quantity units direction* of **hardcopy:basic-hardcopy-stream** *Method*

Converts *quantity* in *units* into the corresponding quantity in device dependent units., *units* may be **:micas** or **:pixel**. *direction* is either **:horizontal** or **:vertical**.

:convert-from-device-units *quantity units direction* of **hardcopy:basic-hardcopy-stream** *Method*

Converts *quantity* from device units into *units*. *direction* is either **:horizontal** or **:vertical**.

:home-cursor of **hardcopy:basic-hardcopy-stream** *Method*

Positions the cursor at the upper left hand corner of the page.

:size &optional (*units* **':device**) of **hardcopy:basic-hardcopy-stream** *Method*

Returns the size of the paper in *units*. *units* may be **:micas**, **:pixel**, or **:device**. **:device** is the default and means in device-dependent units.

:inside-size &optional (*units* (**':device**)) of **hardcopy:basic-hardcopy-stream** *Method*

Returns the size of the area on the paper (the *box*) within which printing can occur, in *units*. **:device** means the units are device dependent.

:allocate-margin *size margin* &optional (*units* **':device**) of **hardcopy:basic-hardcopy-stream** *Method*

Adds the amount of space specified by *size* to the margin specified by *margin* in *units*. **:device** means the units are device dependent. Use of **:allocate-margin** increases the margin, making **:inside-size** smaller.

:string-length *string* &optional (*start* **0**) *end* of **hardcopy:basic-hardcopy-stream** *Method*

Returns the length of *string*, which is the horizontal distance the cursor would have to move to print *string*, in device units.

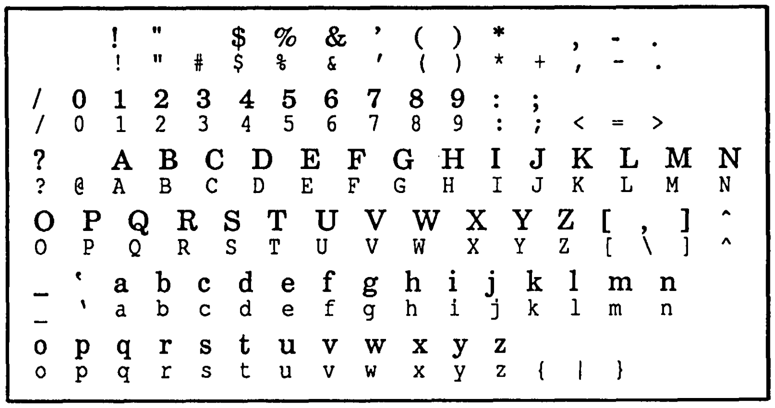
Example of a Hardcopy Stream

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: USER; Base: 10 -*-

;;; Print characters in the character set,
;;; alternating roman and some other font.

(defun font-catalog-page (font &optional
                                (printer hardcopy:*default-text-printer*))
  (setq printer (hardcopy:get-hardcopy-device printer))
  (with-open-stream (stream (hardcopy:make-hardcopy-stream
                              printer))
    (let ((fix-font (send stream :maybe-add-font "FIX9"))
          (catalog-font (send stream :maybe-add-font font)))
      (flet ((send-to-stream-in-font (new-font message &rest args)
               (send stream :set-font new-font)
               (lexpr-send stream message args))
             (draw-line (from-x from-y to-x to-y)
               (send stream :set-cursorpos from-x from-y)
               (multiple-value-bind (x y)
                   ;; Note: :SHOW-LINE takes outside coordinates while
                   ;; :SET-CURSORPOS takes inside coordinates.
                   (send stream :un-relative-coordinates to-x to-y)
                 (send stream :show-line x y))))
```

```
(multiple-value-bind (x-size y-size) (send stream :inside-size)
  (decf x-size) (decf y-size)          ;Leave room for drawing box
  (let* ((line-height-0 (send stream :convert-to-device-units
                              1 :character :vertical))
         (line-height-both (* 2 line-height-0))
         (x 10)
         (y (- y-size (* 1.3 line-height-both)))
         (max-x x)
         (device-units-rounded?
           ;;If the device units are bigger than 0.01 inch, assume they
           ;;are flonums
           (> (send stream :convert-to-device-units 2540. :micas :vertical)
              100.0)))
    (labels
      ((round-device-units (y)
         (if device-units-rounded? (round y) y))
       (draw-box ()
         (decf y line-height-0)
         (draw-line 0     y      0     y-size)
         (draw-line 0   y      max-x y)
         (draw-line max-x y      max-x y-size)
         (draw-line 0   y-size max-x y-size)
         (send stream :set-cursorpos 0 (- y line-height-both))
         (send-to-stream-in-font
           fix-font
           :string-out (format nil "Font ~A catalog" font)))
       (new-page ()
         (send stream :new-page)
         (setq x 10 y (- y-size line-height-both)
               max-x x))
       (new-line ()
         (setq y (round-device-units
                   (- y (* 1.3 line-height-both))))
         (setq max-x (max x max-x))
         (setq x 10)
         (when (< y line-height-both)
           (draw-box)
           (new-page)))
```

```
              (new-character (character)
                (send stream :set-cursorpos x y)
                (send-to-stream-in-font fix-font :tyo character)
                (send stream :set-cursorpos x (+ y line-height-0))
                (send-to-stream-in-font catalog-font :tyo  character)
                (incf x
                  (+ (max (send-to-stream-in-font fix-font
                            :character-width character)
                          (send-to-stream-in-font catalog-font
                            :character-width character))
                    10))
                (when (> x (- x-size 10)) (new-line))))
             (setq y (round-device-units y))
             (loop for char from 32 below 127
                   and character = (code-char char)
                   do
               (new-character character)
                   when (= 15 (mod char 16))
                     do (new-line))
             (draw-box)))))))

(zl-user:font-catalog-page "centuryschoolbook105")
```

Output:



```
     !  "     $  %  &  '  (  )  *     ,  -  .
     !  "  #  $  %  &  '  (  )  *  +  ,  -  .
/  0  1  2  3  4  5  6  7  8  9  :  ;
/  0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >
?     A  B  C  D  E  F  G  H  I  J  K  L  M  N
?  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N
O  P  Q  R  S  T  U  V  W  X  Y  Z  [  ,  ]  ^
O  P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^
_  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n
_  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n
o  p  q  r  s  t  u  v  w  x  y  z
o  p  q  r  s  t  u  v  w  x  y  z  {  |  }
```

```
Font CENTURYSCHOOLBOOK105 catalog
```

The code for this example can be found in sys:examples;hardcopy-stream-example.lisp.

2.5.5 Hardcopy Streams Reference Information

2.5.5.1 Hardcopy Front End and Formatting Functions

hardcopy:hardcopy-text-file *file-name device* &rest *options* *Function*

Called by the various hardcopy commands when the file to be hardcopied is just text (as opposed to press format or other format produced by a text formatting program) or is in an unspecified format.

hardcopy:hardcopy-text-file calls **hardcopy:hardcopy-file** and **hardcopy:hardcopy-from-stream** to do its work.

press:hardcopy-press-file *filename device* &rest *options* *Function*

Called by the various hardcopy commands when the file to be hardcopied is in press format.

hardcopy:hardcopy-file *file-name device* &rest *options* &key *format formatter file-open-options* &allow-other-keys *Function*

Determines the format of the input file, opens it, and passes the input stream to the appropriate formatter function, **hardcopy:hardcopy-from-stream** or **press:hardcopy-press-stream**.

hardcopy:hardcopy-from-stream *stream device* &rest *options* &key (*page-headings* **t**) *starting-page ending-page* &allow-other-keys *Function*

The formatting function for text files. It recognizes character styles in files written by Zmacs. If the file has a `-*- Default Character Style ...` attribute, that style is used as the base for character style merging.

press:hardcopy-press-stream *stream device* &rest *options* &key *starting-page ending-page copies* &allow-other-keys *Function*

The formatting function for press files. It recognizes press format, that is a description of formatted text, and generates the appropriate escape codes for the printing device specified.

2.5.5.2 Hardcopy Options

The functions that do the actual work of hardcopying files, creating hardcopy streams and sending characters to those streams, take a number of keyword options. Each function handles some keywords and passes the remainder along to the function that it calls. As a result, these same keywords are used by **hardcopy:hardcopy-text-file**, **hardcopy:hardcopy-file**, **hardcopy:hardcopy-from-stream** and **hardcopy:make-hardcopy-stream**. Some of the keyword options determine formatter options, some of them are handled directly by the hardcopy stream, and others are passed along to the spooler.

Keyword Options for Formatting

Keyword	*Explanation*
:copies	How many times the formatting function should print the file.
:margins	A list of left margin, top margin, right margin, and bottom margin in *micas*.
:page-headings	Whether to put headings on each page. The default is **t**, which puts headings on each page.
:page-heading	The heading to put on the top of each page. The default is the value of **:banner-file-name**.
:page-heading-date	The date to put in the heading. The default is the value of **:banner-creation-date**.

:output-stream	The destination of bytes for the output device. The formatting function creates an output stream by looking at the options it is given.
:keep-output-stream-open-p	Whether to suppress closing the output stream.
:new-page-hook	A function to call at the start of each page. It receives two arguments, the hardcopy stream and the page number. It can be used to print page headings, for example.
:starting-page	The first physical page to print. The default is the first page of the file. A page is defined by the presence of a `PAGE` character or form feed in the file. Thus plain text files with no page markers in them are single page files. It is important to remember for both **:starting-page** and **:ending-page** that this is physical page and does not use the page number, if any, supplied by a text formatting program.
:ending-page	The last physical page to print. The default is the last page of the file.
:page-number	The number to start with when printing numbers on paper. This is a hardcopy stream option. The default is 1. This is determined by **:starting-page** and **:ending-page**.

Keyword Options for Formatting and Spooling

Keyword	*Explanation*
:body-character-style	The character style to use in printing the main text of the file.
:heading-character-style	The character style to use in printing page headings.
:banner-creation-date	The creation date to print on the banner page and in page headings, in universal time format.
:banner-file-name	The file name to print after `file:` on the banner page and in page headings.

Keyword Options for the Spooler

Keyword	*Explanation*
:spooler-copies	How many times the request should be printed.
:no-banner-page	Whether or not to print a cover (or banner) page. If it is **nil**

	(the default), a banner page is printed. If it is set to **t**, then no banner page is printed.
:notify	Whether or not to notify the user upon completion of printing. If it is **nil** (the default), no notification is given. If it is set to **T**, a message is sent to the originator of the hardcopy request as derived from **:user-name**.
:user-name	The user name of the requestor.
:personal-name	The personal name of the requestor.
:host	The host of the requestor.
:banner-user-name	User name to print on the banner page.
:direct	Whether to print directly on the device without spooling. This is discouraged.
:banner-string	An additional message to put on the banner page.

3. Stream Operations

3.1 Making Your Own Stream

There are some standard functions that make streams for you, such as **make-synonym-stream** and **make-two-way-stream**. There are also forms that allow you to evaluate Lisp forms while performing input or output on a stream, such as **with-open-stream** and **with-input-from-stream**. For more information on **with-open-...** forms: See the section "Accessing Files", page 129.

make-broadcast-stream &rest *streams* *Function*

Returns a stream that works only in the output direction. Any output sent to this stream is sent to all of the streams given. The **:which-operations** is the intersection of the **:which-operations** of all of the streams. The value(s) returned by a stream operation are the values returned by the last stream in *streams*.

make-concatenated-stream &rest *streams* *Function*

Returns a stream that only works in the input direction. Input is taken from the first of the *streams* until it reaches EOF (end-of-file); then that stream is discarded, and input is taken from the next of the *streams*, and so on. If no arguments are given, the result is a stream with no content; any input attempt will result in EOF.

make-echo-stream *input-stream output-stream* *Function*

This function, which is part of the Common Lisp standard, is not currently available in the Symbolics implementation of Common Lisp.

make-string-input-stream *string* &optional (*start* **0**) *end* *Function*

Returns an input stream. The input stream will supply, in order, the characters in the substring of *string* delimited by *start* and *end*. After the last character has been supplied, the stream will then be at end-of-file.

```
(make-string-input-stream "Hello")
  => #<LEXICAL-CLOSURE CLI::STRING-INPUT-STREAM 10223204>

(make-string-input-stream "Hello" 1 3)
  => #<LEXICAL-CLOSURE CLI::STRING-INPUT-STREAM 10224324>
```

make-string-output-stream *Function*

Returns an output stream that will accumulate all string output given it for the benefit of the function **get-output-stream-string**.

```
(setq stream (make-string-output-stream))
  => #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 44310040>

(setq output-string 'hello) => HELLO

(write output-string :stream stream) => HELLO

(get-output-stream-string stream) => "HELLO"
```

get-output-stream-string *stream* *Function*

Returns a string containing all of the characters output to *stream* so far. Works in conjunction with **make-string-output-stream**. *stream* is reset after each call, thus each call to **get-output-stream-string** gets only the characters that have been output to the stream since the last such call (or the creation of *stream*, if no such previous call has been made).

```
(setq s (make-string-output-stream))
  => #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 10602460>

(write-string "Hello" s) => "Hello"

(get-output-stream-string s) => "Hello"

(write-string "Goodbye" s) => "Goodbye"

(get-output-stream-string s) => "Goodbye"
```

make-synonym-stream *stream-symbol* *Function*

Creates and returns a "synonym stream". Any operations on the new stream will be performed on the stream that is then the value of the dynamic variable named by *stream-symbol*. If the value of the variable should change or be bound, then the synonym stream will operate on the new stream.

zl:make-syn-stream *symbol* *Function*

zl:make-syn-stream creates and returns a "synonym stream" (syn for short). *symbol* can be either a symbol or a locative.

If *symbol* is a symbol, the synonym stream is actually an uninterned symbol named #:*symbol*-**syn-stream**. This generated symbol has a property that declares it to be a legitimate stream. This symbol is the value of *symbol*'s **si:syn-stream** property, and its function definition is forwarded to the value cell of *symbol* using a **sys:dtp-external-value-cell-pointer**. Any operations sent to this stream are redirected to the stream that is the value of *symbol*.

If *symbol* is a locative, the synonym stream is an uninterned symbol named

#:syn-stream. This generated symbol has a property that declares it to be a legitimate stream. The function definition of this symbol is forwarded to the cell designated by *symbol*. Any operations sent to this stream are redirected to the stream that is the contents of the cell to which *symbol* points.

Synonym streams should not be passed between processes, since the streams to which they redirect operations are specific to a process.

make-two-way-stream *input-stream output-stream* *Function*

Returns a bidirectional stream that gets its input from *input-stream* and sends its output to *output-stream*.

You can also write your own streams. Here is a sample output stream that accepts characters and conses them onto a list.

```
(defvar the-list nil)
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))
        (otherwise
          (stream-default-handler (function list-output-stream)
                                  op arg1 rest))))
```

The lambda-list for a stream must always have one required parameter (**op**), one optional parameter (**arg1**), and a rest parameter (**rest**). This allows an arbitrary number of arguments to be passed to the default handler. This is an output stream, so it supports the **:tyo** operation. Note that all streams must support **:which-operations**. If the operation is not one that the stream understands (for example, **:string-out**), it calls the **sys:stream-default-handler**. The calling of the default handler is required, since the willingness to accept **:tyo** indicates to the caller that **:string-out** will work.

Here is a typical input stream that generates successive characters of a list.

```
(defvar the-list)       ;Put your input list here
(defvar untyied-char nil)
(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi
     (cond ((not (null untyied-char))
            (prog1 untyied-char (setq untyied-char nil)))
           ((null the-list)
            (and arg1 (error arg1)))
           (t (prog1 (car the-list)
                     (setq the-list (cdr the-list))))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler (function list-input-stream)
                              op arg1 rest))))
```

The important things to note are that **:untyi** must be supported, and that the stream must check for having reached the end of the information and do the right thing with the argument to the **:tyi** operation.

The above stream uses a free variable (**the-list**) to hold the list of characters, and another one (**untyied-char**) to hold the **:untyi**ed character (if any). You might want to have several instances of this type of stream, without their interfering with one another. This is a typical example of the usefulness of closures in defining streams. The following function will take a list and return a stream that generates successive characters of that list.

```
(defun make-a-list-input-stream (list)
   (let-closed ((list list) (untyied-char nil))
      (function list-input-stream)))
```

The above streams are very simple. When designing a more complex stream, it is useful to have some tools to aid in the task. The **defselect** function aids in defining message-receiving functions. The Flavor System provides powerful and elaborate facilities for programming message-receiving objects. See the section "Flavors" in *Symbolics Common Lisp: Language Concepts.*

sys:stream-default-handler *stream op arg1 rest* *Function*

Tries to handle the *op* operation on *stream*, given arguments of *arg1* and the elements of *rest*. The action taken for each of the defined operations is explained with the documentation on that operation. The handler sends the **:any-tyi** message for **:line-in** messages to streams that do not handle **:line-in** themselves.

sys:null-stream *op* &rest *args* *Function*

Can be used as a dummy stream object. As an input stream, it immediately reports end-of-file; as an output stream, it absorbs and discards arbitrary amounts of output. Note: **sys:null-stream** is not a variable; it is defined as a function. Use its definition (or the symbol itself) as a stream, not its value. Examples:

```
(stream-copy-until-eof a 'si:null-stream)
(stream-copy-until-eof a #'si:null-stream)
```

Either of the above two forms reads characters out of the stream that is the value of **a** and throws them away, until **a** reaches the end-of-file.

3.2 General Stream Functions

streamp *x* *Function*

Returns **t** if *x* is a stream, and otherwise it returns **nil**.

input-stream-p *stream* *Function*

Returns *t* if *stream* can handle input operations, and otherwise it returns **nil**.

output-stream-p *stream* *Function*

Returns *t* if *stream* can handle output operations, and otherwise it returns **nil**.

stream-element-type *stream* *Function*

Returns a type specifier which indicates what objects can be read from or written to *stream*. Streams created by **open** will have an element type restricted to a subset of **character** or **integer**, but in principal a stream may transfer any Lisp object.

3.3 General-Purpose Stream Operations

:tyo *char* *Message*

The stream will output the character *char*. For example, if **s** is bound to a stream, then the following form will output a "B" to the stream:

```
(send s :tyo #\B)
```

For binary output streams, the argument is a nonnegative number rather than specifically a character.

:tyi &optional *eof* *Message*

The stream will input one character and return it. For example, if the next character to be read in by the stream is a "C", then the following form returns the value of **#\c** (that is, **103** octal):

```
(send s :tyi)
```

Note that the **:tyi** operation does not "echo" the character in any fashion; it only does the input. The **zl:tyi** function echoes when reading from the terminal.

The optional *eof* argument to the **:tyi** message tells the stream what to do if it reaches the end of the file. If the argument is not provided or is **nil**, the stream returns **nil** at the end of file. Otherwise it signals an error and prints out the argument as the error message. Note that this is not the same as the *eof-option* argument to **read**, **zl:tyi**, and related functions.

The **:tyi** operation on a binary input stream returns a nonnegative number, not necessarily to be interpreted as a character.

An EOF can be forced into the currently selected I/O buffer with the keystrokes FUNCTION END. The next **:tyi** message sent to a window taking input from that I/O buffer will return **nil**.

The EOF indicator is not "sticky," in that the next **:tyi** will take the next character from the I/O buffer. The reason for this is that some programs which read only from the terminal might not be prepared to encounter an EOF, and might loop trying to read input.

This EOF feature makes it possible to fully test programs which use the **:line-in**, **:string-in**, and **:string-line-in** operations by taking input from a window instead of from a file. Typing FUNCTION END causes each of these operations to return. This is especially important when debugging programs which use the **:string-in** operation, since **:string-in** returns only when its buffer is full or an EOF is encountered.

FUNCTION END activates any input buffered in the input editor, since there is no representation for the EOF indicator within text strings.

:untyi *char* *Message*

The stream will remember the character *char*, and the next time a character is input, it will return the saved character. In other words, **:untyi** means "put this character back into the input source". For example:

```
(send s :untyi 120)
(send s :tyi) ==> 120
```

This operation is used by **read**, and any stream that supports **:tyi** must support **:untyi** as well. Note that you are allowed to **:untyi** only one character before doing a **:tyi**, and you can **:untyi** only the last character you read from the stream. Some streams implement **:untyi** by saving the character, while others implement it by backing up the pointer to a buffer. You also cannot **:untyi** after you have peeked ahead with **:tyipeek**.

:which-operations *Message*

The object should return a list of the messages and generic functions it can handle.

The **:which-operations** method supplied by **flavor:vanilla** generates the list once per flavor and remembers it, minimizing consing and compute time. The list is regenerated when a new method is added.

:operation-handled-p *operation* *Message*

operation is a generic function or message name. The object should return **t** if it has a handler for the specified operation, **nil** if it does not.

flavor:vanilla provides a method for **:operation-handled-p**.

Instead of sending this message, you can use the **operation-handled-p** function. See the function **operation-handled-p** in *Symbolics Common Lisp: Language Dictionary.*

:send-if-handles *operation* &rest *arguments* *Message*

operation is a generic function or message name and *arguments* is a list of arguments for the operation.

If a generic function is given, the object should perform the generic function if it has a method for it.

If a message is given, the object should send itself that message with those arguments if it handles the message.

If no method for the generic function or message is available, **nil** is returned.

flavor:vanilla provides a method for **:send-if-handles**.

Instead of sending this message, you can use the **send-if-handles** function. See the function **send-if-handles** in *Symbolics Common Lisp: Language Dictionary.*

:characters *Message*

Returns **t** if the stream is a character stream, **nil** if it is a binary stream.

:direction *Message*

Returns one of the keyword symbols **:input**, **:output**, or **:bidirectional**.

:interactive *Message*

The **:interactive** message to a stream returns **t** if the stream is interactive and **nil** if it is not. Interactive streams, built on **si:interactive-stream**, are streams designed for interaction with human users. They support input editing. Use the **:interactive** message to find out whether a stream supports the **:input-editor** message.

Any stream must either support **:tyo** or support both **:tyi** and **:untyi**. Several more advanced input and output operations work on any stream that can do input or output (respectively). Some streams support these operations themselves; you can tell by looking at the list returned by the **:which-operations** operation. Others are handled by the "stream default handler" even if the stream does not know about the operation itself. However, in order for the default handler to do one of the more advanced output operations, the stream must support **:tyo**, and for the input operations the stream must support **:tyi** (and **:untyi**).

Here is the list of such operations:

:input-wait &optional *whostate function* &rest *arguments* *Message*

This message to an input stream causes the stream to **process-wait** with *whostate* until either of the following conditions is met:

- Applying *function* to *arguments* returns non-**nil**.
- The stream enters a state in which sending it a **:tyi** message would immediately return a value or signal an error.

When either of these conditions is met, **:input-wait** returns. If the stream enters a state in which sending it a **:tyi** message would signal an error, **:input-wait** returns instead of signalling the error. The returned value is not defined.

whostate is what to display in the status line while process-waiting. It can be a string or **nil**. A value of **nil** means to use the normal whostate for this stream, such as "Tyi", "Net In", or "Serial In". For interactive streams, the default whostate is "Tyi".

function can be a function or **nil**. A value of **nil** means that the stream just waits until sending it a **:tyi** message would immediately return a value or signal an error.

This message is intended for programs that need to wait until either input is available from some interactive stream or some other condition, such as the arrival of a notification, occurs. Any stream that can become the value of **zl:terminal-io** must support **:input-wait**.

Following is a simple example of the use of **:input-wait** to wait for input or a notification to an interactive stream. The function just displays notifications and prints representations of characters or blips received as input.

```
(defun my-top-level (stream)
  (error-restart-loop ((error sys:abort) "My top level")
    (send stream :input-wait nil
          #'(lambda (note-cell)
              (not (null (location-contents note-cell))))
          (send stream :notification-cell))
    (let ((note (send stream :receive-notification)))
      (if note
          (sys:display-notification stream note :stream)
          (let ((char (send stream :any-tyi-no-hang)))
            (cond ((null char))
                  ((fixp char)
                   (format stream "~&Character: ~C" char))
                  ((listp char)
                   (format stream "~&Blip: ~S" char))
                  (t (format stream "~&Unknown object: ~S" char)))))))))
```

:listen *Message*

On an interactive device, returns non-**nil** if any input characters are immediately available, or **nil** if no input is immediately available. On a noninteractive device, the operation always returns non-**nil** except at end-of-file, by virtue of the default handler. The main purpose of **:listen** is to test whether the user has pressed a key, perhaps trying to stop a program in progress.

:tyipeek &optional *eof* *Message*

On an input stream, returns the next character that is about to be read, or **nil** if the stream is at end-of-file. The *eof* argument has the same meaning as it does for **:tyi**. **:tyipeek** is defined to have the same effect as a **:tyi** operation, followed by a **:untyi** operation if end-of-file is not reached. Note that this means that you cannot read some character, do a **:tyipeek** to look at the next character, and then **:untyi** the original character.

:fresh-line *Message*

Tells the stream to position itself at the beginning of a new line. If the stream is already at the beginning of a fresh line it does nothing; otherwise it outputs a carriage return. For streams that do not support this, the default handler always outputs a carriage return.

:clear-rest-of-line *Message*

Erases from the current position to the end of the current line.

:string-out *string* &optional *start end* *Message*

The characters of *string* are successively output to the stream. This operation is provided for two reasons: it saves the writing of a frequently used loop, and many streams can perform this operation much more efficiently than the equivalent sequence of **:tyo** operations. If the stream does not support **:string-out** itself, the default handler converts it to **:tyo**s.

If *start* and *end* are not supplied, the entire string is output. Otherwise a substring is output; *start* is the index of the first character to be output (defaulting to **0**), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle **:string-out** must check for them and interpret them appropriately.

:line-out *string* &optional *start end* *Message*

The characters of *string*, followed by a carriage return character, are output to the stream. *start* and *end* optionally specify a substring, as with **:string-out**. If the stream does not support **:line-out** itself, the default handler converts it to **:tyo**s.

:string-in *eof-option string* &optional *(start 0) end* *Message*

Reads characters from an input stream into *string*, using the substring delimited with *start* and *end.*

As is usual with strings, *start* defaults to 0 and *end* defaults to the length of the string. The difference between *end* and *start* constitutes a character count for this operation.

eof-option specifies stopping actions.

Value	*Meaning*
nil	Reading characters into the string stops either when it has transferred the specified character count or when it reaches end-of-file, whichever happens first. For strings with a fill pointer, it sets the fill pointer to point to the location following the last one filled by the read.
not **nil**	If the end-of-file is encountered while trying to transfer a specific number of characters, it signals **sys:end-of-file**, with the value of *eof* as the report string.

:string-in returns two values. The first value is one greater than the last location of *string* into which it stored a character. The second value is **t** if it reached end-of-file and **nil** if it did not. Using **:string-in** at the end of a file returns 0 and **t** and sets the fill pointer of *string* to *start* (if *string* has a fill pointer).

For example, suppose the file my-host:>george>tiny.text contains "Here is some tiny text.".

```
(setq string (make-array 100 ':type 'art-string ':fill-pointer 0))
""

(with-open-file (stream "my-host:>george>tiny.text")
  (send stream ':string-in nil string))
23

string => "Here is some tiny text."
```

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

string can be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

The **:string-in** message can be sent to windows. It interacts correctly with the input editor, including correct handling of activation characters.

The interface to this method for windows and the returned value is exactly the same as the equivalent methods for **si:input-stream** and **si:unbuffered-line-input-stream.**

:line-in &optional *leader* *Message*

The stream should input one line from the input source and return it as a string with the carriage return character stripped off. Despite its name, this operation is not much like the **zl:readline** function.

Many streams have a string that is used as a buffer for lines. If this string itself were returned, there would be problems if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. To solve this problem, the string must be copied. On the other hand, some streams do not reuse the string, and it would be wasteful to copy it on every **:line-in** operation. This problem is solved by using the *leader* argument to **:line-in**. If *leader* is **nil** (the default), the stream does not copy the string, and the caller should not rely on the contents of that string after the next operation on the stream. If *leader* is **t**, the stream makes a copy. If *leader* is an integer then the stream makes a copy with an array-leader *leader* elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches the end-of-file while reading in characters, it returns the characters it has read in as a string, and returns a second value of **t**. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was an whole line or only the trailing characters after the last carriage return in the input source.

The **:line-in** message can be sent to windows. It interacts correctly with the input editor, including correct handling of activation characters.

:string-line-in *eof string* &optional (*start* **0**) *end* *Message*

:string-line-in is a combination of **:string-in** and **:line-in**. It allows you to read many lines successively into the same buffer without creating strings. **:string-line-in** reads a line from a file into a string (or other array) supplied by the user. It returns the array index plus one, whether an *eof* was encountered and whether the entire line was read into the buffer.

This message fills up a string as does **:string-in**, but reads only one line, as does **:line-in**. As with **:line-in**, the carriage return character at the end of the line is not stored into your buffer. **:line-in** reads a line from a stream and creates a string with that line in it. **:string-line-in** is given a string; it fills in the string (or other array) that you give it from the stream.

:string-line-in reads a line from a stream and fills the supplied array with that line. As with **:string-in**, if the string (or other array) has a fill pointer, it is set to the number of characters placed into the buffer plus the *start* offset.

:string-line-in returns three values:

- The number of active characters in the string or array. The number is calculated as one plus the array index into the buffer of the last item added to the string by this call.
- Whether the end of the input stream was encountered while trying to read in the string. *eof* is identical to the *eof-option* argument in **:string-in**.
- **nil** if the entire line fit in the buffer supplied, otherwise **t**. If **t** is returned for this value, as much of the line as could fit was stored in the buffer and more of the line is waiting to be read.

If the second and third values are both **nil**, a carriage return was read. If either is **t**, no carriage return was read from the stream.

:clear-input *Message*

The stream clears any buffered input. If the stream does not handle this, the default handler ignores it.

:clear-output *Message*

The stream clears any buffered output. If the stream does not handle this, the default handler ignores it.

:force-output *Message*

Causes any buffered output to be sent to a buffered asynchronous device, such as the Chaosnet. It does not wait for it to complete; use **:finish** for that. If a stream supports **:force-output**, then **:tyo**, **:string-out**, and **:line-out** might have no visible effect until a **:force-output** is done. If the stream does not handle this, the default handler ignores it.

:finish *Message*

Does a **:force-output** to a buffered asynchronous device, such as the Chaosnet, then waits until the currently pending I/O operation has been completed. If the stream does not handle this, the default handler ignores it.

For file output streams, **:finish** finalizes file content. It ensures that all data have actually been written to the file, and sets the byte count. It converts non-direct output openings into append openings. It allows other users to access the data that have been written before the **:finish** message was sent.

:close &optional *mode* *Message*

The stream is "closed", and no further operations should be performed on it; you can, however, **:close** a closed stream. If the stream does not handle **:close**, the default handler ignores it.

The *mode* argument is normally not supplied. If it is **:abort**, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly created file is deleted, as if it were never opened in the first place. Any previously existing file with the same name remains, undisturbed.

:eof *Message*

Indicates the end of data on an output stream. This is different from **:close** because some devices allow multiple data files to be transmitted without closing. **:close** implies **:eof** when the stream is an output stream and the close mode is not **:abort**.

3.4 Special-Purpose Stream Operations

See the section "General-Purpose Stream Operations", page 33. There are several other defined operations that the default handler cannot deal with; if the stream does not support the operation itself, sending that message causes an error. This section describes the most commonly used, least device-dependent stream operations. Windows, files, and Chaosnet connections have their own special stream operations, which are documented separately.

:input-editor *function* &rest *arguments* *Message*

This is supported by interactive streams such as windows. It is described in its own section: See the section "The Input Editor Program Interface", page 265.

Most programs should not send this message directly. See the special form **with-input-editing**, page 270.

:beep &optional *type* *Message*

This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. *type* is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; *type* is currently ignored and should always be **nil**.

:tyi-no-hang &optional *eof* *Message*

Identical to **:tyi** except that if it would be necessary to wait in order to get the character, returns **nil** instead. This lets the caller efficiently check for input being available and get the input if there is any. **:tyi-no-hang** is different from **:listen** because it reads a character and because it is not simulated by the default handler for streams that do not support it.

:untyo-mark *Message*

This is used by the grinder if the output stream supports it. See the special form **grindef**, page 329. It takes no arguments. The stream should return some object that indicates where output has reached in the stream.

:untyo *mark* *Message*

This is used by the grinder in conjunction with **:untyo-mark**. See the special form **grindef**, page 329. It takes one argument, which is something returned by the **:untyo-mark** operation of the stream. The stream should back up output to the point at which the object was returned.

The following operations are only implemented by window streams. There are many other special-purpose stream operations for graphics. See the section "Using the Window System" in *Programming the User Interface, Volume B.*

:read-cursorpos &optional *(units* **':pixel**) *Message*

This operation is supported by windows. It returns two values, the current *x* and *y* coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units *x* and *y* should be; the symbols **:pixel** and

:character are understood. **:pixel** means that the coordinates are measured in display pixels (bits), while **:character** means that the coordinates are measured in characters horizontally and lines vertically.

This operation and **:set-cursorpos** are used by the **zl:format** "~t" request, which is why "~t" does not work on all streams. Any stream that supports this operation must support **:set-cursorpos** as well.

:set-cursorpos *x y* &optional *(units* **':pixel***)* *Message*

This operation is supported by the same streams that support **:read-cursorpos**. It sets the position of the cursor. *x* and *y* are similar to the values of **:read-cursorpos** and *units* is the same as the *units* argument to **:read-cursorpos**.

:clear-window *Message*

Erases the window on which this stream displays. Non-window streams do not support this operation.

The following operations are only implemented by streams to random-access devices, principally files.

:read-pointer *Message*

Returns the current position within the file, in characters (bytes in fixnum mode). For text files on PDP-10 file servers, this is the number of Symbolics characters, not PDP-10 characters. The numbers are different because of character-set translation.

:set-pointer *new-pointer* *Message*

Sets the reading position within the file to *new-pointer* (bytes in fixnum mode). For text files on PDP-10 file servers, this does not do anything reasonable unless *new-pointer* is 0, because of character-set translation. This operation is for input streams only.

The following operations are implemented by buffered input streams. They allow increased efficiency by making the stream's internal buffer available to the user.

:read-input-buffer &optional *eof no-hang-p* *Message*

Returns three values: a buffer array, the index in that array of the next input byte, and the index in that array just past the last available input byte. These values are similar to the *string*, *start*, *end* arguments taken by many functions and stream operations.

If the end of the file has been reached and no input bytes are available, the stream returns **nil** or signals an error, based on the *eof* argument, just like the **:tyi** message. If the argument *no-hang-p* is t and no input is available, the call returns **nil** and **nil**.

After reading as many bytes from the array as you care to, you must send the **:advance-input-buffer** message. The data in the buffer is valid only until the **:advance-input-buffer** message is given. At that point, the stream may reuse the buffer for other storage.

:advance-input-buffer &optional *new-pointer* *Message*

If *new-pointer* is non-**nil**, it is the index in the buffer array of the next byte to be read. If *new-pointer* is **nil**, the entire buffer has been used up.

The following operations are provided for buffered output streams. They allow you to hand the stream's output buffer to a function that can fill it up.

:get-output-buffer *Message*

Returns an array and starting and ending indices.

:advance-output-buffer &optional *index* *Message*

Says that the array returned by the last **:get-output-buffer** operation was filled up through *index*. If *index* is omitted, the array was filled completely.

The following stream operations are obsolete and should no longer be used:

:rewind
:get-input-buffer

3.5 File Stream Operations

The following messages can be sent to file streams, in addition to the normal I/O messages that work on all streams. Note that several of these messages are useful to send to a file stream which has been closed. Some of these messages use pathnames. See the section "Naming of Files", page 51.

:pathname *Message*

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

:truename *Message*

Returns the pathname of the file actually open on this stream. This can be different from what **:pathname** returns because of file links, logical devices, mapping of "newest" version to a particular version number, and so on. For some systems (such as ITS) the truename of an output stream is not meaningful until after the stream has been closed, at least on an ITS file server.

:length *Message*

Returns the length of the file, in bytes or characters. For text files on PDP-10 file servers, this is the number of PDP-10 characters, not Symbolics characters. The numbers are different because of character-set translation. (See the section "The Character Set", page 355.) For an output stream the length is not meaningful until after the stream has been closed, at least on an ITS file server.

:characters *Message*

Returns **t** if the stream is a character stream, **nil** if it is a binary stream.

:creation-date *Message*

Returns the creation date of the file, as a number which is a universal time. See the section "Dates and Times" in *Programming the User Interface, Volume B.* See the function **fs:directory-list**, page 161.

:info *Message*

Returns a cons of the truename and creation date of the file. The creation date is a number that is a universal time. This can be used to tell if the file has been modified between two **opens**. For an output stream the info is not meaningful until after the stream has been closed, at least on an ITS file server.

:delete *Message*

Deletes the file open on this stream. The file does not really go away until the stream is closed. You should not use **:delete**. Instead, use **zl:deletef**.

:rename *new-name* *Message*

Renames the file open on this stream. You should not use **:rename**. Instead, use **zl:renamef**.

:properties *Message*

Returns two values:

- A list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a "disembodied" property list and **zl:get** can be used to access the file's properties.

- A list of what properties of this file are "changeable".

:change-properties *Message*

Changes the file properties of the file open on this stream. **:change-properties** signals an error rather than returning one.

:finish *Message*

Does a **:force-output** to a buffered asynchronous device, such as the Chaosnet, then waits until the currently pending I/O operation has been completed. If the stream does not handle this, the default handler ignores it.

For file output streams, **:finish** finalizes file content. It ensures that all data have actually been written to the file, and sets the byte count. It converts non-direct output openings into append openings. It allows other users to access the data that have been written before the **:finish** message was sent.

File output streams implement the **:finish** and **:force-output** messages.

3.6 Network Stream Operations

:connected-p *Message*

Returns **t** if the stream is fully connected to an active network connection, **nil** otherwise. If the stream is in a transitory state that is not completely connected, **:connected-p** returns **nil**.

:connected-p must be callable in a scheduler context. That is, it cannot call **:process-wait**.

:start-open-auxiliary-stream *active-p* &key *local-id foreign-id stream-options application-id* *Message*

This message is sent to a stream to establish another stream, via another connection, over the same network medium, to the same host. It is used for either end of the connection.

If *active-p* is **t**, it means this side will connect and the remote side should listen; if *active-p* is **nil**, the remote side will connect and this side will listen.

If this side is active, *foreign-id* is the foreign contact identifier to connect to.

If this side is not active, *local-id* is the local identifier to listen on. The content of *foreign-id* and *local-id* depends on the network implementation. If this side is not active, and no *local-id* is supplied, *application-id* must be supplied. *application-id* is a string that the network uses as part of the the contact identifier it will create and return.

Returns *stream* and *contact-identifier.*

stream is a new stream. It is not yet usable. You can do one of two things with it:

- Terminate the establishment of the new connection by sending the message **:close :abort** or **:close-with-reason :abort** to the stream.
- Wait for the connection to be fully established, by sending **:complete-connection** to the stream.

contact-identifier is a string representing the contact name actually being listened to, in the case that this side is not active. This is the string to convey to the other side, so that the other side can supply it as the *foreign-id* argument of **:start-open-auxiliary-stream**, to connect back to this side.

:complete-connection &key *(timeout (* 60. 6.))* *Message*

This message is sent to a new stream created by **:start-open-auxiliary-stream,** in order to wait for the connection to be fully established. **:complete-connection** is used whether or not this side is active.

Timeout is interpreted as the number of sixtieths of a second to wait before timing out.

When **:complete-connection** returns, the stream is fully connected to an active network connection. At this point, **:connected-p** to that stream returns **t**.

:complete-connection signals an error if the connection times out or does not complete for another reason.

:set-input-interrupt-function *function* &rest *args* *Message*

This message assigns a *function* to be applied to any *args* whenever input becomes available on the connection, or the connection goes into an unusable state. The function is called in a non-simple process, and therefore can use **:process-wait.**

3.7 The :read And :print Stream Operations

A stream can specially handle the reading and printing of objects by handling the **:read** and **:print** stream operations. Note that these operations are optional and that most streams do not support them.

If the **read** function is given a stream that has **:read** in its which-operations, then instead of reading in the normal way it sends the **:read** message to the stream with one argument, **read**'s *eof-option* if it had one or a magic internal marker if it did not. Whatever the stream returns is what **read** returns. If the stream wants to implement the **:read** operation by internally calling **read**, it must use a different stream that does not have **:read** in its which-operations.

If a stream has **:print** in its which-operations, it can intercept all object printing operations, including those due to the **print**, **prin1**, and **princ** functions, those due to **zl:format**, and those used internally, for instance in printing the elements of a list. The stream receives the **:print** message with three arguments: the object being printed, the *prindepth* (for comparison against the **zl:prinlevel** variable), and *slashify-p* (**t** for **prin1**, **nil** for **princ**). If the stream returns **nil**, then normal printing takes place as usual. If the stream returns non-**nil**, then **print** does nothing; the stream is assumed to have output an appropriate printed representation for the object. The two following functions are useful in this connection; however, they are in the **system-internals** package and might be changed without much notice.

PART III.

Files

4. Naming of Files

A Symbolics computer generally has access to many file systems. While it can have its own file system on its own disks, a community of Symbolics users often has many shared file systems accessible by any of the Symbolics computers over a network. These shared file systems can be implemented by any computers that are capable of providing file system service. A *file server computer* might be a special-purpose computer that does nothing but service file system requests from computers on a network, or it might be an existing timesharing system.

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

```
/usr2/george/foo.bn
```

In this example, /usr2/george is the *directory name*, foo is the *file name* and bn is the *file type*. However, in TOPS-20, a similar file name is expressed as follows:

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

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists; in fact, new formats could be added in the future, and existing programs should retain their abilities to manipulate files in a system-independent fashion.

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

4.1 Pathnames

All file systems dealt with by the Symbolics computer are mapped into a common model, in which files are named by a conceptual object called a *pathname*. The Symbolics computer system, in fact, represents pathnames by objects of flavor **pathname**, and the flavors built upon it. A pathname always has six conceptual components, described below. These components provide the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. This mapping is described elsewhere for each file system. See the section "The Character Set", page 355.

The following are the conceptual components of a pathname. They will be clarified by examples below.

Host	The computer system, the machine, on which the file resides.
Device	Corresponds to the "device" or "file structure" concept in many host file systems. Often, it designates a group of disks, or removable storage media, or one of several different media of differing storage densities or costs.
Directory	An organizational structure in which files are "contained" on almost all file systems. Files are "stored in", or "reside in" directories. The directories have names; the files' names are only valid within the context of a given directory. Some systems (*hierarchical* file systems) allow directories to be contained in other directories; others do not.
Name	The name of a group of files that can be thought of as conceptually the "same" file. In many systems, this is the "first name" of the file. For instance, source and object files for the same program generally have the same *name*, but differing *type*.
Type	Corresponds to the "filetype" or "extension" concept in many host file systems. This usually indicates the kind of data stored in the file, for example, binary object code, a Lisp source program, a FORTRAN source program, and so forth.
Version	Corresponds to the "version number" concept in many host file systems. Some systems implement this concept, others do not. A version number is a number, part of the conceptual name of the file, that distinguishes succeeding versions of a file from each other. When a user of such a file system writes out a file he or she does not modify the file on the host computer but writes a *new version*, that is, one with a higher version number, automatically. The Symbolics computer system allows a version component of "newest" or "oldest", represented by the keyword symbols **:newest** and **:oldest**, respectively, to designate "the newest (oldest) version of the file, whichever that might be".

As an example, consider a TOPS-20 user named "George", who writes a Lisp program that he thinks of as being named "conch". If George uses the TOPS-20 host named FISH, the source for his program might be in a file on the host FISH with the following name:

```
<GEORGE>CONCH.LISP.17
```

In this case, the host is FISH, the device would be some appropriate default, and the directory would be <GEORGE>. This directory would probably contain a number of files related to the "conch" program. The source code for this program would live in a file with name CONCH, type LISP, and versions 1, 2, 3, and so on. The compiled form of the program would live in a file named CONCH with type BIN.

Now suppose George is a UNIX user, using the UNIX host BIRD. The source for his program would probably be in a file on the host BIRD with the following name:

```
/usr2/george/conch.l
```

In this case, the host is BIRD, and the directory would be /usr2/george. This directory would probably contain a number of files related to the "conch" program. The source code for this program would live in a file with name conch, type l. The compiled form of the program would live in a file named conch, with type bn. There are no version numbers on UNIX.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; a pathname need not correspond to any file that actually exists; and more than one pathname can refer to the same file. For example, the pathname with a version of "newest" will refer to the same file as a pathname with the same components except a certain number as the version. In systems with links, multiple file names, logical devices, and so forth, two pathnames that look quite different can turn out to designate the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

4.1.1 Simple Usage of the Pathname System

The pathname system can be very easy to use if you know a few simple techniques. It often seems that there are many different ways to do anything, and that only one of the is right for any circumstance, but most of these features only exist for special needs. This section shows you how to easily do some of the simple things.

4.1.1.1 Getting a Filename From the User

The simplest and most common application for using a pathname is simply to read or write a file. For example, a program to do some *very* simple processing of a database (it reads the file and ignores it):

```
(defun process-example-database (database-pathname)
  (with-open-file (database-stream database-pathname)
    (format t "~&Ignoring database ~A ..." (send database-stream :truename))
    (stream-copy-until-eof stream #'si:null-stream)
    (format t " ignored.~%")))
```

This simple example is adequate for a program interface, but for a user, it is rather awkward. The user must supply all components of the pathname, plus the quotation marks around the strings. Also, the user has no completion available. In this example, the user does not have to parse the pathname; **open** will do that for him. (Sometimes we will not be so lucky).

The user's job can be made easier by providing a function to read a pathname and pass it to **process-example-database.** To do this, **prompt-and-read** is used. See the function **prompt-and-read** in *Programming the User Interface, Volume B.*

In our first version, we will just ask the user for the pathname.

```
(defun run-example ()
  (let ((pathname (prompt-and-read :pathname "Where is your database? ")))
    (process-example-database pathname)))

Where is your database?  Y:>user>databases>dummy.database
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

prompt-and-read does much of what we are looking for. It provides the following:

- Prompting, including reprompting when the user presses REFRESH
- Parsing
- Completion
- Merging with defaults

In this case, we supplied no default, so the "default default", **fs:*default-pathname-defaults*** is used. But this default is not very helpful to the user, because it is not visible; it could even be confusing if the user expected one default and got another. Good practice dictates telling the user what the default is. **prompt-and-read** makes this easy with the **:visible-default** suboption to **:pathname**, **:pathname-or-nil**, and **:pathname-list.**

```
(defun run-example ()
  (let ((pathname (prompt-and-read
                    '(:pathname :visible-default ,fs:*default-pathname-defaults
                    "Where is your database? ")))
    (process-example-database pathname)))
Where is your database?  (Default Y:>user>foo.lisp) databases>dummy.database
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

Now that the user can see the defaults, he or she can make use of them. Note that in the above example, the user did not have to type the "Y:>user>", because the default was available.

Tailoring Pathname Defaults

fs:*default-pathname-defaults* is a global default, with nothing particularly appropriate to any specific application. Often, when an application is writing or reading a file, it knows more about the file than is implied by **fs:*default-pathname-defaults***. This information can be used to help prompt the user for a suitable filename and help reduce the amount of typing needed to specify a suitable filename.

For example, consider our example of reading a database. (See the section "Getting a Filename From the User", page 53.) In this example, we are just prompting for the filename and ignoring the actual database.

```
(defun run-example ()
  (let ((pathname (prompt-and-read
                    '(:pathname :visible-default ,fs:*default-pathname-default
                    "Where is your database? ")))
    (process-example-database pathname)))
Where is your database?  (Default Y:>user>foo.lisp) databases>dummy.database
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

First, if we are going to seriously use our own special file type, we need to define the type so that it can be used successfully on different systems. See the special form **fs:define-canonical-type**, page 90.

```
(fs:define-canonical-type :database "DATABASE"
  ((:vms :vms4) "DBS")
  (:unix "DB"))
```

Now this type can be used as the default type for our example databases.

```
(defun run-example ()
  (let* ((default (fs:default-pathname fs:*default-pathname-defaults*
                                        nil            ;Host
                                        :database))    ;Type
         (pathname (prompt-and-read '(:pathname :visible-default ,default)
                                    "Where is your database?~%")))
    (process-example-database pathname)))

Where is your database?  (Default Y:>user>foo.database) databases>dummy
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

4.1.1.2 More About Defaults

Most simple programs use **fs:*default-pathname-defaults*** as the source for their defaults. However, as a program makes more use of pathname reading and defaults, there are some things we can do to make things easier for the user.

- Provide a default based on other files in an operation, for example, defaulting an output file pathname from the input file.
- Provide "sticky" defaults, where the new default is based on the last file the user gave.
- Provide a default based on the current context, as in "pathname of the current buffer" in Zmacs.

Defaulting an Output File Pathname From an Input File

Perhaps the most common defaulting situation is that of defaulting an output file pathname from the input file. Usually, the output file differs from the input file only in file type and version, and we would like to have the user provide explicit information only when his or her desires differ from the usual case.

```
(defun my-compile-file (input-file output-file)
  (format t "~&Compiling ~A into ~A.~%"
          input-file output-file)
  (compiler:compile-file input-file output-file))
```

```
(defun comp-it ()
  (let* ((input-default (fs:default-pathname nil nil :lisp :newest))
         (input-file (prompt-and-read
                       '(:pathname :visible-default ,input-default)
                       "Input file: "))
         (output-default (fs:default-pathname input-file nil :bin :newest))
         (output-file (prompt-and-read
                       '(:pathname :visible-default ,output-default)
                       "Output file: ")))
    (my-compile-file input-file output-file)))
```

The above example works well for single files, but it does not handle wildcards. To handle wildcards, we need to introduce the use of **:translate-wild-pathname** and **fs:directory-link-opaque-dirlist**. **:translate-wild-pathname** does the work of interpreting how a given input file is to be mapped to its corresponding output file, and **fs:directory-link-opaque-dirlist** takes care of finding all the input files.

Note that we use **fs:directory-link-opaque-dirlist** rather than **fs:directory-list**. In general, this is necessary whenever the **:translate-wild-pathname** message is used. **:translate-wild-pathname** expects the input pathname to match the input pattern. **fs:directory-list**, in the presence of directory links or VAX/VMS logical devices, can have a different directory or a different device.

If the input pattern has wildcards in its directory component, **fs:directory-link-opaque-dirlist** currently does no better than **fs:directory-list**. This is a difficult problem still under investigation.

```
(defun comp-one-file (input-file-pattern output-file-pattern input-file)
  (let ((output-file (send input-file-pattern :translate-wild-pathname
                           output-file-pattern input-file)))
    (my-compile-file input-file output-file)))
```

```
(defun comp-files ()
  (let* ((input-default (fs:default-pathname nil nil :lisp :newest))
         (input-pattern (prompt-and-read
                          `(:pathname :visible-default ,input-default)
                          "Input file: "))
         (output-default (fs:default-pathname input-file nil :bin :newest))
         (output-pattern (prompt-and-read
                           `(:pathname :visible-default ,output-default)
                           "Output file: ")))
    (if (not (send input-file :wild-p))
        (comp-one-file input-pattern output-pattern input-pattern)
      (loop for (file) in (cdr (fs:directory-link-opaque-dirlist
                                 input-pattern :fast))
            do (comp-one-file input-pattern output-pattern file)))))
```

Note that in the above example, we just call **comp-one-file** directly if the input pathname is not wild. While it is not strictly necessary to do this (**fs:directory-link-opaque-dirlist** works on non-wildcard pathnames), it does eliminate an unneeded operation.

Sticky Pathname Defaults

Often, when a single command or a related set of commands are to be repeated, the next command should operate on a file related to the one the current command is operating on. In this case, it would be most convenient for the default to be the previous pathname. This is called *sticky defaulting.*

For example, consider a simple user-written tool to either show or delete files.

```
(defun show-or-delete ()
  (loop with default = (fs:default-pathname)
        for ch = (prompt-and-read :character "Cmd>")
        do (multiple-value-bind (prompt function)
               (selector char-equal ch
                 (#\S (values "Show File" #'viewf))
                 (#\D (values "Delete File" #'deletef))
                 (#\Q (return nil))
                 (#\Help (format t "~&S = Show File~@
                                    D = Delete File~@
                                    Q = Quit~%")
                  (values nil nil))
                 (otherwise
                  (tv:beep)
                  (format t "~&~:C is an unknown command.~%" ch)))
             (when prompt
               (let ((file (prompt-and-read
                             '(:pathname :visible-default ,default)
                             prompt)))
                 ; The following is done for us by prompt-and-read
                 ;(setq default (fs:merge-pathnames file default))
                 (funcall function file))))))
```

Each time around the loop, when the user specifies a file, it is remembered to serve as the default the next time around. Note the commented out **(setq default (fs:merge-pathnames file default))**. This isn't needed in this example, since **prompt-and-read** does this for us, but if we were reading pathnames via some other mechanism, it is important to keep the default as a fully specified pathname. Otherwise, the second time around the loop, we could end up with defaults like "Q:", which is not of much use if the user is then forced to type all the components of the pathname and may get an error if he or she does not.

If you wish to use a default such as this and not keep it in a local variable, you should use a defaults alist. This serves as a registered place to remember a pathname, so that if the world is moved to another site, it can be reset. Defaults alists can be passed to **fs:default-pathname** to extract a fully-merged default. See the function **fs:set-default-pathname**, page 89. See the function **fs:make-pathname-defaults**, page 89.

Pathname Defaulting From the Current Context

Often, an application program involves the user working on a single context for an extended time. For example, in the editor, the user is working on a single named buffer. In the font editor, the user is working on a single named font.

Often, the object being worked on was read in from a file. This file can serve as a default for further file operations, such as listing the directory, or resaving the object. Consider a picture editor, which lets the user edit multiple pictures, as the Zmacs editor lets the user edit multiple buffers. This picture editor stores its files in .BIN files.

```
(defflavor picture (name
                    (pathname sys:fdefine-file-pathname)
                    (array (make-array '(100. 100.) :type 'art-1b)))
           ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(defvar *pictures* nil
        "List of pictures being edited")

(defvar *current-picture* nil)

(defvar *picture-defaults* (fs:make-pathname-defaults))

(defun add-picture (picture)
  (setq *pictures* (del #'(lambda (p1 p2)
                             (string-equal (send p1 :name) (send p2 :name)))
                         picture
                         *pictures*))
  (push picture *pictures*)
  (setq *current-picture* picture))

(defmethod (picture :fasd-form) ()
  `(make-instance ',(typep self)
                  :name ',name
                  :array ',array))
```

```
(defun picture-default-pathname (&key type (version :newest))
  (let ((bare-default (fs:default-pathname *picture-defaults*
                                           nil type version))
        (path (when *current-picture*
                (send *current-picture* :pathname))))
    (if (not *current-picture*)
        bare-default
      (if path
          (setq path (fs:merge-pathnames path bare-default version))
        ;; A new picture, so no pathname.  Let's make a guess from the name.
        (let ((name (send *current-picture* :name)))
          (setq path
                (condition-case ()
                    (fs:merge-pathnames name bare-default version)
                  ;; If name isn't parsable, just use the bare default.
                  (error bare-default)))))
      path)))

(defun com-create-picture ()
  (let ((name (prompt-and-read :string "Picture name: ")))
    (add-picture (make-instance 'picture :name name))))

(defun com-save-picture ()
  (let* ((default (picture-default-pathname :type :bin))
         (file (prompt-and-read
                 `(:pathname :visible-default ,default)
                 "Save to picture file: ")))
    ;; Remember the pathname given, so the next time we
    ;; get a new picture, we can have a better default.
    (fs:set-default-pathname file *picture-defaults*)
    (sys:dump-forms-to-file
      file
      `((add-picture ',*current-picture*)))))
```

In this example, **picture-default-pathname** computes the default. If the current picture has a file associated with it, that serves as the default. If there is no pathname with the current picture, we attempt to make a pathname using the name. If that fails (or if there is no current picture), we just use the bare default.

Finally, the pathname we read is remembered, so the next time a default is needed for a new picture, we will have a more recent default.

Note that when the picture is loaded, **sys:fdefine-file-pathname** is used to get the file being loaded. This works well when the file being loaded is a .bin file, since **zl:load** binds this variable. However, in other situations, you need to make other arrangements to set the pathname.

4.1.2 Host Determination in Pathnames

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

Since each kind of file system has its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and determining the value of each component. The parsers, therefore, all work differently. How does the parsing operation know which parser to use? It determines for which host the pathname is intended, and uses the appropriate parser. A filename character string can specify a host explicitly, by having the name of the host, followed by a colon, at the beginning of the string, or it can assume a default, if there is no host name followed by a colon at the beginning of the string.

Here is how the pathname system determines for which host a pathname being parsed is intended. The first colon in a pathname being parsed *always* delimits the host name. You can also enter pathname strings that are for a specific host and do not contain any host name. In that case, a *default host* is used. Normally, the identity of the default host is displayed to the user entering a pathname. See the section "Pathname Defaults and Merging", page 73.

However, pathnames can have colons in them that do not designate hosts, such as filenames constructed from clock times, and the like. Some systems use the colon character to delimit devices. This creates a problem in parsing such pathnames. See the function **fs:parse-pathname**, page 83. The standard Symbolics computer user interface does not use such pathnames, but they can be used by other programs, particularly those that deal with files whose format is defined by a foreign operating system.

The rule for parsing file names containing colons is, again, that any string used before a colon is *unconditionally* interpreted as a file computer. If the string cannot be interpreted as a host, an error is signalled.

If you must type a pathname that has an embedded colon *not* meaning a host, you omit the host and place a colon at the beginning of the string. This "null host" tells the parser that it should *not* look further for a colon, but instead assume the host from the defaults. Examples:

- SS:<FOO>BAR refers to a host named "SS". :SS:<FOO>BAR refers to no explicit host; if parsed relative to a TOPS-20 default, "SS" probably refers to a device.
- 09:25:14.data refers to a host named "09". :09:25:14.data refers to no explicit host.
- AI: COMMON; GEE WHIZ refers to a host named "AI".
- AI: ARC: USERS1; FOO BAR refers to a host named "AI". "ARC" is the name of a device in the ITS operating system.
- EE:PS:<COMMON>GEE.WHIZ.5 specifies host EE (TOPS-20).
- PS:<COMMON>GEE.WHIZ.5 specifies a host named PS, which is almost certainly not what is intended! The user probably intended the "PS" device on some TOPS-20 host.
- :PS:<COMMON>GEE.WHIZ.5, assuming that the default host is some TOPS-20, specifies a device named "PS" on that host.

There are a few "pseudohost" names, which are recognized as host names even though they are not actually the names of hosts:

"local"	This pseudohost name always refers to the local file system (LMFS) of the machine that you are using. It does not matter whether or not a local file system actually exists on that machine; an attempt will be made to reference it. "Local" is always equivalent to the name of the local host.
"FEP"	This pseudohost name always refers to a FEP (front-end processor) file system on the machine you are using, specifically, the one on the disk unit from which the system was booted.
"FEP*n*"	This pseudo name always refers to a FEP file system on the machine you are using. The single digit *n* specifies the disk unit number; there is a separate FEP file system on each drive. This can access the boot unit, or any other disk unit, when multiple units are present.
"*host*|FEP*n*"	*host* must be a valid host name. This pseudohost name refers to a FEP file system on a remote 3600-family computer. The syntax "*host*|FEP" is not acceptable: you cannot access the

"boot unit" of a remote machine in this fashion. You must know the disk unit number. The disk unit number of a host having only one disk unit is **0**.

If the string to be parsed does not specify a host explicitly, the parser assumes that some particular host is the one in question, and it uses the parser for that host's file system. The optional arguments passed to the parsing function (**fs:parse-pathname**) tell it which host to assume.

4.1.3 Interning of Pathnames

Pathnames, like symbols, are *interned*. This means that there is only one pathname object with a given set of components. If a character string is parsed into components, and some pathname object with exactly those components already exists, then the parser returns the existing pathname object rather than creating a new one. The main reason for this is that a pathname has a property list. See the section "Property Lists" in *Symbolics Common Lisp: Language Concepts*. The system stores properties on pathnames to remember information about the file or family of files to which that pathname refers. (In fact, some of the *properties* stored on a generic pathname come from the file's *attribute* list when the file is edited or loaded, so they can be retrieved later without having to perform I/O on the file.) So you can parse a character string that represents a filename, and then look at its property list to get various information known about that pathname. The components of a pathname are never modified once the pathname has been created, just as the print name of a symbol is never modified. The only thing that can be modified is the property list.

When using property lists of pathnames, you have to be very careful which pathname you use to hold properties, in order to avoid a subtle problem: many different pathnames can refer to the same file, because of the **:newest** component, file system links, multiple naming in the file system, and so on. If you put a property on one of these pathnames because you want to associate some information with the file itself, somebody else might look at another pathname that refers to the same file, and not find the information there. If you really want to associate information with the file itself rather than some particular pathname, you can get a canonical pathname for the file by using the **:truename** message to a stream opened to that file. See the message **:truename**, page 44. You might also want to store properties on "generic" pathnames. See the section "Generic Pathnames", page 75.

4.1.4 Printing Pathnames

A pathname can be converted back into a string, which is in the file name syntax of its host's file system. Although such a string (the *string for host*) can be produced from a pathname (by sending it the **:string-for-host** message), we discourage this practice. The Genera user interface prefers a string called the *string for printing*, which is the same as the string for host, except that it is preceded by the host name and a colon. This leaves no ambiguity about the host on which the file resides, when seen by a user. It is also capable of being reparsed, unambiguously, back into a pathname. **prin1** of a pathname (**~s** in **format**) prints it like a Lisp object (using the usual "#<" syntax), while **princ** of a pathname (**~a** in **format**) prints the string for printing. The **string** function, applied to a pathname, also returns the string for printing.

Not all the components of a pathname need to be specified. If a component pathname is missing, its value is **nil**. Before a file server can utilize a pathname to manipulate or otherwise access a file, all the pathname's missing components must be filled in from appropriate defaults. Pathnames with missing components are nevertheless often passed around by programs, since almost all pathnames typed by users do not specify all the components explicitly. The host is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to explicitly designate a host. Every pathname has a host attribute, even if the string that was parsed to create it did not specify one explicitly.

All pathname parsers support the cross-system convention that the double-shafted arrow character (↔) can be used to specify a null directory, name, type, or version component explicitly. Thus, for LMFS or TOPS-20, you can type the following:

```
    ↔.↔.5
```

This example specifies a version of 5, but no name or type. This is useful when typing against the default and attempting to change just the version of that default.

The keyword symbol **:unspecific** can also be a component of a pathname. This means that the component is not meaningful on the type of file system concerned. For example, UNIX pathnames do not have a concept of "version", so the version component of every UNIX pathname is **:unspecific**. When a pathname is converted to a string, **nil** and **:unspecific** both cause the component not to appear in the string. The difference occurs in the merging operation, where **nil** is replaced with the default for that component, while **:unspecific** is left alone.

The special symbol **:wild** can also be a component of a pathname. This is only useful when the pathname is being used with a directory listing primitive such as **fs:directory-list** or **fs:all-directories**, where it means that this pathname component matches anything. See the function **fs:directory-list**, page 161. The printed representation of a pathname usually designates **:wild** with an asterisk; however, this is host-dependent.

:wild is one of several possible *wildcard* components, which are given to directory-listing primitives to filter file names. Many systems support other wildcard components, such as the string "foo*". This string, when supplied as a file name to a directory list operation on any of several system types, specifies all files whose name starts with "foo". In other contexts, it might not represent a wildcard at all. The component **:wild** matches all possible values for any component for which it appears. Other wildcard possibilities for directories exist, but they are more complicated, and are explained elsewhere. See the section "Values of Pathname Components", page 66. See the section "Directory Pathnames and Directory Pathnames as Files", page 68.

4.1.5 Values of Pathname Components

The set of permissible values for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually treated as things chosen by the user that need to be preserved and passed around.

Most programs do not use or specify the components of a pathname explicitly, or only in a very limited way. In this way, they can remain operating-system-independent, while letting the pathname system take care of most issues of compatibility. In general, you should avoid where possible using specific values of pathname components in your programs. The descriptions here are illustrative but not complete, and programs should be written to expect component values other than those given here.

It is important to remember that not all pathname flavors accept all the values indicated here. For example, UNIX pathnames accept a type or version of **:unspecific**; few other pathnames do. Some systems do not allow certain characters or limit certain fields to a certain length.

It is generally *not* possible to simply copy components from one flavor of pathname to another. It is often necessary to perform substitutions in order to produce a legal pathname. The **:new-default-pathname** message can be used instead of **:new-pathname** to get this substitution where necessary. The **:new-default-pathname** message attempts to substitute something as close as possible in meaning to the original component; however, the substitution can be arbitrary if necessary. For this reason, it is better to avoid copying components between pathnames of differing flavor, where possible.

The type is always a string (unless it is one of the special symbols **nil**, **:unspecific**, or **:wild**). Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are "lisp", compiled Lisp programs (on, for example, a LMFS host) are "bin", text files are "text", and so on. The set of characters allowed in the type, and the number of characters, are system-dependent. In order to process file types in a system-independent way, the *canonical type* mechanism has been devised. A canonical type is a system-independent keyword symbol representing the conceptual type of a file. For instance, a Lisp source file on VMS has a file type of "LSP", and one on UNIX has a file type of "l". When we ask pathnames of either of these natures for their canonical type, we receive the keyword symbol **:lisp**. See the section "Canonical Types in Pathnames", page 77.

The version is either a number (specifically, a positive fixnum), or one of the symbols **nil**, **:unspecific**, **:wild**, **:newest**, or **:oldest**. **nil**, **:unspecific**, and **:wild** have been explained above. **:newest** refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. **:oldest** refers to the smallest existing version number.

The host component of a pathname is always a host object. See the section "Namespace System Host Objects" in *Networks.*

The device component of a pathname can be one of the symbols **nil** or **:unspecific**, or a string designating some device, for those file systems that support such a notion.

The file name can be **nil** or a string, or **:wild**.

The directory component is highly system-specific. While it can be **nil** for any type of host, values designating actual directories, or partially wild specifications for directories, are more complicated. On nonhierarchical file systems, the directory component is usually a string such as "LMDOC", designating the name of the directory.

On hierarchical file systems, the directory component, when not **nil**, is a list of *directory level components.* For example:

LMFS pathname	*Directory component*
`>sys>io>qfile.lisp.2357`	`("sys" "io")`

"**sys**" and "**io**" are the directory level components. Since the "root directory" of hierarchical file systems has no directory level components, it would be represented as **nil**, but this is impermissible, since **nil** already means that the directory component has not been specified. Thus, **:root** is used as the directory component in that case.

Relative pathnames on hierarchical file systems are represented by directory components having the level component **:relative**, followed by a number of occurrences of the symbol **:up** equal to the number of "upward relativization symbols", followed by the remaining directory level components. For example:

LMFS relative pathname	*Directory component*
`<<x>y>z.lisp`	`(:relative :up :up "x" "y")`

Directory components of pathnames for hierarchical file systems, on some systems, can also have the symbol **:wild** or a partially wild string (such as "foo*") as directory level components, to do level-by-level matching of level components. Also, on some systems, the level component **:wild-inferiors** (which is printed as "**" on LMFS and logical pathnames, and "..." on VMS, currently the only ones supporting it) to designate "any number, including zero of directory levels" to a directory list operation.

Note that some systems (currently VMS) do not allow using zero directory levels to denote their root directory. In this case, **:wild-inferiors** cannot stand alone, but must follow some other directory spec. For example: "[FOO...]" or "[*...]".

4.1.6 Directory Pathnames and Directory Pathnames as Files

In almost all systems having hierarchical directories, and certainly all the ones supported by the Symbolics computer as file server systems, the directories are implemented internally as special files, known about by the operating system. The data in these files is not accessible to the user except through the defined operating system interfaces for dealing with directories.

Typically, listings of the contents of directories on hierarchical directory systems display names of both files and directories contained in the listed directory (as well as of links, on systems that support links).

Directories on hierarchical directory systems and files thus some things have in common. Appearing in directories is one. Another is that directories can usually be renamed, as can files, or, when the appropriate restrictions of the operating system are met (for instance, being empty), deleted. You can ask about the properties of a directory, or change some of them, with **fs:file-properties** and **fs:change-file-properties**, respectively, just as you do with a file.

Using LMFS as an example, consider the directory named "bar", which is contained in the directory named "foo", which itself is contained in the ROOT. A file in this directory named "tables.lisp.6" would have the following pathname:

```
>foo>bar>tables.lisp.6
```

The directory in which it is contained, bar, has the following pathname:

```
>foo>bar.directory.1
```

The file type of a directory, on LMFS, is "directory", and the version number of all directories is 1. The file types of directories, and their versions, if appropriate, vary among operating systems. If you wanted to rename, delete, or deal with the properties of the directory bar, you would have to present the above filename for this directory. A pathname of this type, which names a directory, as though it were a file, is called a *directory pathname as file.*

Directory pathnames as files are appropriate only to systems with hierarchical directories. On other systems, you cannot address directories directly.

The most common use of directories, however, is to reference files in them. The following pathname mentions the directory "bar" in this way:

```
>foo>bar>tables.lisp.6
```

This filename, when parsed into a pathname for the appropriate LMFS host, has a name component of "tables", a type component of "lisp", a version component of 6, and a directory component (in fact **("foo" "bar")**) that designates the directory bar, inferior of foo, inferior of the ROOT. Such a pathname, which designates a given directory via its directory component, is called a *pathname as directory* for that directory. Of course, since the file name, type, and version are irrelevant to the specification of the directory, it is only one of many possible "pathnames as directory" for the directory bar.

The concept of pathname as directory is more general than the concept of directory pathname as file, since directories on nonhierarchical systems be described by their pathnames as directories as well. For instance, the following TENEX pathname, which describes a file in the "LMDOC" directory, is a pathname as directory for the LMDOC directory:

```
<LMDOC>CHFILE.TEXT;7
```

Note, also, that any pathname whose directory component is not **nil** is a pathname as directory for *some* directory.

Therefore, the Symbolics Common Lisp primitives and operations that deal with directories explicitly (for example, **fs:expunge-directory** and **fs:all-directories**) expect pathnames of directories to be represented in the "pathname as directory" form. It is the canonical, system-independent way to represent pathnames of directories in the Symbolics system.

The following two messages convert between directory pathnames as files and pathnames as directories:

:directory-pathname-as-file of **pathname** *Method*

Every pathname whose directory component is not **nil** is a pathname as directory for *some* directory. This method returns the directory pathname as file for that directory.

```
(setq p (fs:parse-pathname "Quabbin:>sys>lmfs>fsstr.lisp.243"))
#<LMFS-PATHNAME "Q:>sys>lmfs>fsstr.lisp.243">
(send p ':directory-pathname-as-file)
#<LMFS-PATHNAME "Q:>sys>lmfs.directory.1">
```

:pathname-as-directory of **pathname** *Method*

This method is intended to be sent to a pathname that is the valid directory pathname as file for some directory. It produces one of many possible pathnames as directory for that directory, namely, the one whose name, type, and version are all **nil**.

```
(setq p1 (fs:parse-pathname "Quabbin:>sys>io.directory.1"))
#<LMFS-PATHNAME "Q:>sys>io.directory.1">
(setq p2 (send p1 ':pathname-as-directory))
#<LMFS-PATHNAME "Q:>sys>io>">
(send p2 ':directory-pathname-as-file)
#<LMFS-PATHNAME "Q:>sys>io.directory.1">
```

If you are used to other systems' file-naming conventions, you may be confused by pathnames that have real directory components, but no name, type, or version. When typed in or printed, they look like the following:

```
>jones>book>examples>
```

Users who are familiar with Multics or UNIX immediately see such pathnames as invalid, even though they are often used on the Symbolics computer to access Multics and UNIX. When parsed for LMFS or Multics, the above filename string produces a pathname whose directory component designates the directory "examples", which is contained in "book", which itself is contained in "jones", an inferior of the ROOT. The name, type, and version components of this pathname are **nil**. This pathname is equivalent to the following:

```
>jones>book>examples>↔.↔.↔
```

Either of these is a canonical pathname as directory for the directory "examples". Typing such pathnames as input is exceedingly common, since the merging process, given such a pathname as its unmerged input, replaces the directory component of the default with a directory component specifying the directory named by the "pathname as directory". See the section "Pathname Defaults and Merging", page 73. For example:

```
Default:        Q:>abel>baker>cakes.list
User Typein:    >Romanoli>weddings>
Merged output:  >Romanoli>weddings>cakes.list
```

Compare this with the following:

```
Default:        Q:>abel>baker>cakes.list
User Typein:    >Romanoli>weddings
Merged output:  >Romanoli>weddings.list

Default:        Q:>abel>baker>cakes.list
User Typein:    >Romanoli>weddings>↔.↔.73
Merged output:  >Romanoli>weddings>cakes.list.73
```

All the Symbolics hierarchical directory parsers recognize a trailing directory delimiter as an instruction to construct a pathname with **nil** name, type, and version, for the directory designated–a "pathname as directory". (The version component, however, remains **:unspecific** for systems not supporting file versions.) This is true even of the parsers for UNIX and Multics, on which systems such syntax is never seen.

This mode of directory naming is usually familiar to users of nonhierarchical systems. The following TENEX pathname results, when parsed, in a pathname as directory for the LMDOC directory (on the appropriate TENEX host), with name, type, and version of **nil**, that can be used in merging operations in a way similar to that shown in the above LMFS example.

```
<LMDOC>
```

As a side-effect of these conventions, the following kinds of pathnames occasionally occur on LMFS or Multics:

```
<lmdoc>
```

As explained above, thi sis a valid way of entering the following relative pathname:

```
<lmdoc>↔.↔.↔
```

4.1.7 Case in Pathnames

The pathname system handles alphabetic case in pathnames and transferring of pathname components between hosts with different preferred alphabetic cases.

The components of a pathname (directory, name, type, and so on) have two possible representations for case, *raw* (also called *native*) and *interchange.* The raw case representation keeps the case in whatever form is normal for that system (for example, lowercase for UNIX, uppercase for TOPS-20). Interchange representation is a format for manipulating pathname components in a host-independent manner. All pathname defaulting and cross-host translation functions use the interchange form of pathname messages.

All the standard messages to pathnames (for example, **:directory**, **:name**) return pathname components in interchange case rather than raw case.

The components are stored internally in raw case, that is, the actual alphabetic case in which the names of the files are stored, or to be stored, in the host's file system. It is possible to access the raw case representation via the set of messages **:raw-directory**, **:raw-name**, and so forth. However, programs seeking to be system-independent should not use these messages, but the standard ones, **:directory**, **:name**, and so forth. Doing so ensures that pathname components transferred between system types stay in the preferred case for each of the systems concerned.

The raw forms of the messages are provided for writing host-specific code or for manipulating several pathname objects known to be on the same host.

Interchange case form	*Raw case form*
:device	**:raw-device**
:directory	**:raw-directory**
:name	**:raw-name**
:type	**:raw-type**

The interchange form of the message specifies the following effect:

Case of component	*Translated case returned*
System default	Uppercase
Mixed case	Mixed case
Opposite to default	Lowercase

Uppercase was chosen as the interchange case because strings like "LISP", representing pathname components, appear in many programs. Either choice (upper or lower) would have been natural for some hosts and not for others.

This facility provides more features for dealing with pathname components independent of the case-sensitivity of file names of different hosts. The following table shows some examples for different host types.

Host	*Message*	*Applied to raw form*	*Returns interchange form*
UNIX	**:name**	"foo-bar.baz"	"FOO-BAR"
	:name	"FOO-BAR.BAZ"	"foo-bar"
	:name	"Foo-Bar.Baz"	"Foo-Bar"
Lisp Machine File System	**:name**	"foo-bar.baz"	"FOO-BAR"
	:name	"FOO-BAR.BAZ"	"FOO-BAR"
	:name	"Foo-Bar.Baz"	"FOO-BAR"
TOPS-20	**:name**	"FOO-BAR.BAZ"	"FOO-BAR"
	:name	"foo-bar.baz"	"foo-bar"
	:name	"Foo-Bar.Baz"	"Foo-Bar"
VMS4	**:name**	"FOO_BAR.BAZ"	"FOO-BAR"

Note that VMS has only one example; that is because VMS supports upper-case only. VMS uses the underscore character "_" where other operating systems use the hyphen "-".

Note that the Lisp Machine File System (LMFS) appears not to follow the interchange case rules. This is because, for LMFS, case is usually maintained but is not significant ("foo", "Foo", and "FOO" are all the same). Thus any mixture of cases in a file name satisfies the "system default" condition and returns all uppercase for the interchange form.

Functions that manipulate pathnames, such as **fs:make-pathname**, **fs:merge-pathnames**, and **fs:merge-pathname-and-set-defaults**, manipulate components in interchange case.

Pathname-constructing functions such as **fs:make-pathname** and pathname messages such as **:new-pathname** and **:new-default-pathname** accept both **:directory** and **:raw-directory**, to allow specification of components in either interchange case or raw case.

4.2 Defaults and Merging

It is unreasonable to require the user to type a complete pathname, containing all components. Instead the program is expected to supply a *default pathname*, from which values of components not specified by the user can be taken.

Every program that prompts the user for a pathname should maintain some default pathname, display it to the user when prompting for a pathname, and merge the parsed input from the user with that default. The function **prompt-and-read** provides easy ways to do all of these things. See the function **prompt-and-read** in *Programming the User Interface, Volume B.* No program should use any pathname obtained from user input without merging it against *some* default. Since it is impossible for a user to type a pathname correctly without knowing against which default it will be merged, the default must be displayed to the user.

A *default default* is available for programs that have no better idea of a default pathname, and a function (**fs:default-pathname**) for customizing default pathnames.

Typically, a program might take the default default, customize it, perhaps by supplying a specific file type (usually via the canonical type mechanism), prompt the user for the name of a file, displaying that default, and merge the user's parsed input against that default.

A more complex program, one that requires an input file and an output file, might proceed as follows: It obtains the pathname of its input file as above, and prepares a default pathname for its output file by customizing the input file pathname, usually by supplying a new type, and presents and uses that as a default for the prompt for the output file pathname.

The *merging* operation is performed by the function **fs:merge-pathnames**. It takes as input an unmerged pathname and a default pathname and returns a *merged pathname*, which has no missing components. Basically, the missing components in the unmerged pathname are filled in from the default pathname. The merging operation also takes a *default version* argument, which specifies the version number of the output pathname, *if there is no version mentioned in the unmerged pathname.* That is, the version number is almost never defaulted from the default pathname. If the default version argument is not supplied, it is assumed to be **:newest**. The version number of the default is used as a default version in the following cases:

- Neither name, type, nor version is specified by the unmerged pathname.
- The unmerged pathname does not have a version, and the value of the default version argument is **:default**.

The full details of the merging rules are as follows.

1. If the unmerged pathname does not supply a device, the device is the default file device for that host.
2. If the unmerged pathname does not specify a host, device, directory, name, or type, that component comes from the defaults.
3. If the unmerged pathname supplies a version, it is used.
4. If it does not supply a version, the default version as explained above is used.

Thus, if the user supplies just a name, the host, device, directory and type will come from the default, with the default version argument (or **:newest** if there was none). If the user supplies nothing, or just a directory, the name, type, and version comes over from the default together. If the host's file name syntax provides a way to input a type or version without a name, the user can let the name default but supply a different type or version than the ones in the default.

The system also defines an object called a *defaults alist*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults alist is basically an object containing a replaceable pathname. **fs:merge-pathnames** accepts a defaults alist as its default pathname argument as well as a pathname. **fs:merge-pathnames-and-set-defaults** is like **fs:merge-pathnames** but *requires* a defaults alist as its default pathname argument. When it has completed its merge, it stores the result back into the defaults alist before returning it. See the function **fs:merge-pathnames-and-set-defaults**, page 84. It is important that you do not attempt to construct a defaults alist, but instead use the primitives provided. See the function **fs:make-pathname-defaults**, page 89. See the function **fs:copy-pathname-defaults**, page 89. See the function **fs:set-default-pathname**, page 89.

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

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

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

fs:load-pathname-defaults *Variable*

The defaults alist for the **zl:load** and **compiler:compile-file** functions. Other functions can share these defaults.

4.3 Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the file attribute list line of a source file in the family. See the section "File Attribute Lists", page 156. All types of files with that name, in that directory, belong together. They are different members of the same family; for example, they might be source code, compiled code, and documentation for a program. All versions of files with that name, in that directory, belong together.

The generic pathname of pathname *p* has the same host, device, directory, and name as *p* does. However, it has a version of **nil**. Furthermore, if the canonical type of *p* is one of the elements of **fs:*known-types***, then it has a type of **nil**; otherwise it has the same type as *p*. The reason that the type of the generic pathname works this way is that in some file systems, such as that of ITS, the type component can actually be part of the file name; ITS files named "DIRECT IONS" and "DIRECT ORY" do not belong together.

The **:generic-pathname** message to a pathname returns its corresponding generic pathname. See the method **(flavor:method :generic-pathname pathname)**, page 98.

fs:*known-types* *Variable*

This is a list of the canonical file types that are "not important"; constructing a generic pathname will strip off the file type if it is in this list. File types not in this list are really part of the name in some sense. The following is the initial list:

```
(:LISP :QBIN :BIN NIL :UNSPECIFIC)
```

Some users might need to add to this list. See the section "Canonical Types in Pathnames", page 77.

4.4 Relative Pathnames

Many operating systems support a notion called *relative pathnames* in order to simplify the typing of filenames by their users. Typically, a user on a system such as Multics or UNIX tells the system what directory on the system is his or her *working directory*. These operating systems assume the working directory as the default directory for filenames whose directory is not specified. For example, when the user types a filename, perhaps as an argument to a command (such as "print foo") the system assumes that the name foo refers to a file named foo in the working directory, as long as the user did not specify another directory (for instance, by saying "print >sources>c>foo").

On hierarchical systems, such as UNIX and Multics, the working directory can often be several levels deep, and have a full name that is therefore cumbersome to type. The concept of working directory is all the more powerful in these cases. Since the hierarchical organization of directories exists to facilitate relating files by placing them in directories in common subtrees, it is common for users working on such systems to want to reference files in "siblings" of their working directory, or "uncles", or even "inferiors" or "inferiors of inferiors", that is, directories near in the directory hierarchy to their working directory.

In order to facilitate the referencing of files in directories "near" the working directory, without having to type full pathnames of directories, these systems support *relative pathnames*, which are interpreted relative to the working directory. Relative pathnames are always syntactically distinguishable from other pathnames. For instance, on Multics, if the working directory is >udd>Proj>Username, the pathname

```
<Othername>stuff>x.pl1
```

refers to the file

```
>udd>Proj>Othername>stuff>x.pl1
```

Although it supports relative pathnames, the Lisp Machine File System does not support a concept of working directory. One rationale for this is the fact that the user might be communicating with many systems at once, and might have several working directories to remember. The merging and defaulting system takes the place of the working directory concept. See the section "Pathname Defaults and Merging", page 73. The default pathname, which is displayed when a user is asked to enter a pathname, determines the default directory for a pathname having no directory explicitly specified. What is more, it specifies the default values of other components as well.

Systems supporting relative pathnames usually have some special syntax to indicate a pathname that is relative to a superior of the working directory, and another to indicate pathnames relative to superiors of the working directory. We call these "upward relativization" and "downward relativization". In this context, a pathname with an explicit directory specified is called an *absolute pathname*, and one without an explicit directory, a relative pathname. However, since specification of no directory at all is a very common case handled by systems that do not otherwise support relative directories, namely, by simply defaulting an entire directory component, this is not considered a relative pathname by the Symbolics system.

The Symbolics system supports relative directories for those hierarchical systems that support it themselves. As might be expected, the "resolution" of relative pathnames entered by the user is performed relative to the default pathname, as opposed to any working directory. Resolution of relative pathnames is performed by **fs:merge-pathnames** as part of its normal operation.

The following examples, using LMFS pathnames, show some examples of relative pathnames and their resolution via merging:

Default:	`>sys>lmfs>new>xst.lisp`	
Unmerged:	`test>xst.lisp`	
Merged:	`>sys>lmfs>new>test>xst.lisp`	
Default:	`>sys>lmfs>new>xst.lisp`	
Unmerged:	`<test>thing.lisp`	`;upward relativization`
Merged:	`>sys>lmfs>test>thing.lisp`	
Default:	`>sys>lmfs>new>xst.lisp`	
Unmerged:	`<<test>`	`;upward relativization`
Merged:	`>sys>test>xst.lisp`	
Default:	`>sys>lmfs>new>xst.lisp`	
Unmerged:	`test>best>`	`;downward relativization`
Merged:	`>sys>new>test>best>xst.lisp`	
Default:	`>sys>lmfs>new>xst.lisp`	
Unmerged:	`<xst.lisp`	
Merged:	`>sys>lmfs>xst.lisp`	
Default:	`>sys>lmfs>new>xst.lisp`	
Unmerged:	`<<abel>baker>foo.lisp`	
Merged:	`>sys>abel>baker>foo.lisp`	

4.5 Canonical Types in Pathnames

A *canonical type* for a pathname is a symbol that indicates the nature of a file's contents. To compare the types of two files, particularly when they could be on different kinds of hosts, you compare their canonical types. (**fs:*default-canonical-types*** and **fs:*canonical-types-alist*** show the canonical types and the default surface types for various hosts.)

Some terminology:

canonical type A host-independent name for a certain type of file, for example,

	Lisp compiled code files or LGP font files. A canonical type is a keyword symbol.
file specification	What you type when you are prompted to supply a string for the system to build a pathname object.
surface type	The appearance of the type component in a file specification. This is a string in native case.
default surface type	Each canonical type has as part of its definition a representation for the type when it has to be used in a string. Default surface type is the string (in interchange case) that would be used in a string being generated by the system and shown to the user. See the special form **fs:define-canonical-type**, page 90.
preferred surface type	Some canonical types have several different possible surface representations. The definition for the type designates one of these as the preferred surface type. It is a string in interchange case. ("Default surface type" implies "preferred surface type" when one has been defined.)

Each canonical type has a default surface representation, which can be different from the surface file type actually appearing in a file specification. **:lisp** is a canonical type for files containing Lisp source code. For example, on UNIX, the default surface representation of the type for **:lisp** files is "L". (Remember, the default surface representation is kept in interchange case.) The surface type in a file specification containing lisp code is different on different systems, "LISP" for Lisp Machine file system, "l" for UNIX. You can find out from a pathname object both the canonical type for the pathname and the surface form of the type for the pathname by using the **:canonical-type** message. See the method **(flavor:method :canonical-type pathname)**, page 93.

The following tables illustrate the terminology.

		UNIX	
Surface type	"l"	"lisp"	"foo"
Raw type	"l"	"lisp"	"foo"
Type	"L"	"LISP"	"FOO"
Canonical type	**:lisp**	**:lisp**	"FOO"
Original type	**nil**	"LISP"	"FOO"

		Lisp machine	
Surface type	"l"	"lisp"	"foo"
Raw type	"l"	"lisp"	"foo"
Type	"L"	"LISP"	"FOO"
Canonical type	"L"	**:lisp**	"FOO"
Original type	"L"	**nil**	"FOO"

To translate the type field of a pathname from one host to another, determine the canonical type, using the surface type on the original host. Then find a surface type on the new host for that canonical type.

Copying operations can preserve the surface type of the file through translations and defaulting rather than by converting it to the surface form for the canonical type. For example:

```
(multiple-value-bind (ctype otype)
    (send p ':canonical-type)
  (send p ':new-pathname
          ':canonical-type ctype
          ':original-type otype
          ':name "temp-p"))
```

4.5.1 Correspondence of Canonical Types and Editor Modes

fs:*file-type-mode-alist* is an alist that associates canonical types (in the car) with editor major modes (in the cdr).

```
((:LISP . :LISP) (:SYSTEM . :LISP) (:TEXT . :TEXT) ...)
```

4.6 Wildcard Pathname Mapping

In the Symbolics system, as in some other systems, wildcard pathnames are used not only to specify groups of files, but to specify mappings between pairs of pathnames, for operations such as renaming and copying files.

For example, you might ask to copy *foo*.lisp to *bar*.lisp. All the files to be copied match the wildcard name *foo*.lisp. *bar*.lisp is a specification of how to construct the pathname of the new file. The two wildcard pathnames, as in the above example, are called the *source pattern* and *target pattern*. The original name of any file to be copied is called the *starting instance*. Here is an example:

Source pattern:	`f:>fie>*old*.lisp`
Target pattern:	`vx:/usr2/fum/*older*.l`
Starting instance:	`f:>fie>--oldfoo.lisp`
Target instance:	`vx:/usr2/fum/--olderfoo.l`

A more abstract description of this terminology:

Source pattern	A pathname containing wild components.
Target pattern	A pathname containing wild components.
Source instance	A pathname that matches the source pattern.
Target instance	A pathname specified by applying the common sequences between the source and target patterns to the source instance.

Two Zmacs commands accept pairs of wildcard file specifications:

Copy File (m-X)
Rename File (m-X)

The components of the target instance are determined component-by-component for all components except the host. (The host component is always determined literally from the source and target patterns; it cannot be wild.) The mapping of pathnames is done in the native case of the target host. The source pattern and source instance are coerced to the target host via the **:new-default-pathname** message before the mapping takes place. See the method **(flavor:method :new-default-pathname pathname)**, page 97.

When the type of the target pattern is **:wild**, it uses the canonical type for the target, regardless of the surface form for the type in the source pattern and instance.

NOTE

In the Lisp Machine File System, * as the directory portion of a file specification specifies a relative pathname. You must use >** to indicate a wild directory component that matches any directory at all. See the section "LMFS Pathnames", page 101.

Here are the rules used in constructing a target instance, given the source and target patterns and a particular source instance. This set of rules is applied separately to each component in the pathname. In the mapping rules, a * character as the only contents of a component of a file spec is considered to be the same as the keyword symbol **:wild**. The rule uses the patterns from the example above.

1. If the target pattern does not contain *, copy the target pattern component literally to the target instance.

2. If the target pattern is **:wild**, copy the source component to the target literally with no further analysis. The type component is handled somewhat differently – when source and target hosts are of different system types, it uses the canonical-type mechanism to translate the type. This does not apply when the target pattern is **:wild-inferiors**, in directory specifications.

3. Find the positions of all * characters in the source and target patterns. Take the characters intervening between * characters as a literal value. Literal values for the name component:
   Source: `old`
   Target: `older`

4. Find each literal value from the source pattern in the source instance. Take the characters intervening between literal values as a matching value for the * from the source pattern. The matching value could be any number of characters, including zero. Matching values for the name component:
   -- and `foo`

5. Create the component by assembling the literal and matching values in left to right order, substituting the matching values where * appears in the target pattern. For the name component:
   `--olderfoo`

   When not enough matching values are available (due to too few * in the source pattern) use the null string as the matching value. When the source pattern has too many *, ignore the first extra * and everything following it.

Some examples:

Source pattern	*Source instance*	*Target pattern*	*Target instance*
*report	6802-report	*summary	6802-summary
lmfs-*	lmfs-errors	*	lmfs-errors
l*	l	l*	l
l*	lisp	l*	lisp
OLD-DIR	OLD-DIR	NEW-PLACE	NEW-PLACE
*	doc	*-extract	doc-extract
doc	doc	doc-extract	doc-extract

4.6.1 Wildcard Directory Mapping

The rules for mapping directory components between two wildcard pathnames and a starting instance are parallel to the rules for single names. Directory-level components play roughly the roles of characters in the name-translating algorithm. See the section "Wildcard Pathname Mapping", page 79.

Consider a directory component as a sequence of directory level components. The levels are separated by level delimiters (> in LMFS). Example: In the pathname >foo>bar>*>mumble*>x>**>y>a.b.3, the directory-level components are foo, bar, *, mumble*, x, **, and y. The source and target patterns, as well as the starting instance, are considered as sequences of directory-level components, and are matched and translated level by level.

For this purpose, each directory-level component can be classified as one of three types:

Type	*Directory representation*
constant	String containing no *'s
wild-inferiors	** in LMFS, ... in VMS
must-match	* or string containing at least one * (but not the string representing wild-inferiors)

The matching and mapping of constant and wild-inferiors levels proceeds in a manner identical to the matching and mapping of constant substrings and *s for single names. See the section "Wildcard Pathname Mapping", page 79. Constant directory level components act as constant substrings in that algorithm, and wild-inferiors levels as *s. That is, wild-inferiors level components match and, on the target side, carry, zero to any number of constant directory-level components. Examples:

```
Source pattern:     >sys>**>*.*.newest
Target pattern:     >old-systems>release-5>**>*.*.*
Starting instance:  >sys>lmfs>patch>lmfs-33.patch-dir.66
Target instance:    >old-systems>release-5>lmfs>patch>lmfs-33.patch-dir.66

Source pattern:     >a>b>c>**>d>e>**>x.y.*
Target pattern:     >t>u>**>m>**>w>*.*.*
Starting instance:  >a>b>c>p>q>d>e>f>g>x.y.1
Target instance:    >t>u>p>q>m>f>g>w>x.y.1
```

Must-match components are matched with exactly one directory-level component, which must be present. They are mapped according to the string-mapping rules in the name-translating algorithm. See the section "Wildcard Pathname Mapping", page 79.

Example:

```
Source pattern:     >a>b>c>foo*>d>*>*.*.*
Target pattern:     >x>*bar>y>*man>*.*.*
Starting instance:  >a>b>c>foolish>d>yow>a.lisp.1
Target instance:    >x>lishbar>y>yowman>a.lisp.1
```

You can intersperse constants, must-matches, and wild-inferiors directory-level components, as long as the sequence of wildcard types is the same in both patterns.

Example:

```
Source pattern:      >a>*>c>**>*.lisp.*
Target pattern:      >bsg>sub>new-*>q>**>*.*.*
Starting instance:   >a>bb>c>d>e>p1.lisp.6
Target instance:     >bsg>sub>new-bb>q>d>e>p1.lisp.6
```

4.7 Pathname Functions

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

fs:parse-pathname *thing* &optional *with-respect-to (defaults* **fs:*default-pathname-defaults****)* *Function*

Turns *thing*, which can be a pathname, a string, or a Maclisp-style name list, into a pathname. Most functions that take a pathname argument call **fs:parse-pathname** on it so that they accept anything that can be turned into a pathname. Some, however, do it indirectly, by calling **fs:merge-pathnames**.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *with-respect-to* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host.

If *with-respect-to* is supplied, it should be a host or a string to be parsed as the name of a host. If *thing* is a string, it is then parsed as a true string for that host; host names specified as part of *thing* are not removed. Thus, when *with-respect-to* is not **nil**, *thing* should not contain a host name.

If *with-respect-to* is not supplied or is **nil**, any host name inside *thing* is parsed and used as the host. If *with-respect-to* is **nil** and no host is specified as part of *thing*, the host is taken from *defaults*.

Examples, using a LMFS host named Q:

```
(fs:parse-pathname "a:>b.c" "q") => #<LMFS-PATHNAME "Q:a:>b.c">  ;(wrong)
(fs:parse-pathname "q:>b.c" "q") => #<LMFS-PATHNAME "Q:q:>b.c">  ;(wrong)
(fs:parse-pathname "q:>b.c")     => #<LMFS-PATHNAME "Q:>b.c">
(fs:parse-pathname ">b.c" "q")   => #<LMFS-PATHNAME "Q:>b.c">
```

Note that this causes correct parsing of a TOPS-20 pathname when *thing* contains a device but no host and when *with-respect-to* is not **nil**. (Warning: If *thing* contains a device but no host and if *with-respect-to* is **nil** or not supplied, the device is interpreted as a host.) In the following example, X is a TOPS-20 host and A is a device:

```
(fs:parse-pathname "a:<b>c.d" "x") => #<TOPS20-PATHNAME "X:A:<B>C.D">
(fs:parse-pathname "a:<b>c.d")     => Error: "a" is not a known file
                                             server host.
```

In the same TOPS-20 example, if *with-respect-to* is **nil** and the host is to taken from *defaults*, the pathname string must be preceded by a colon to be parsed correctly:

```
(fs:parse-pathname ":a:<b>c.d" nil "x:") => #<TOPS20-PATHNAME "X:A:<B>C.D">
(fs:parse-pathname "a:<b>c.d" nil "x:")  => Error: "a" is not a known file
                                                   server host.
```

If *thing* is a list, *with-respect-to* is specified, and *thing* contains a host name, an error is signalled if the hosts from *with-respect-to* and *thing* are not the same.

fs:merge-pathnames *pathname* &optional (*defaults* **fs:*default-pathname-defaults***) (*default-version* **':newest**) *Function*

Fills in unspecified components of *pathname* from the defaults, and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, or a Maclisp name list. The returned value is always a pathname. The merging rules are documented elsewhere: See the section "Pathname Defaults and Merging", page 73.

If *pathname* is a string, it is parsed before merging. The default pathname is presented to **fs:parse-pathname** as a default pathname, from which the latter defaults the host if there is no explicit host named in the string.

defaults can be a pathname, a defaults alist, or a string. If it is a string, it is parsed against the default defaults. *defaults* defaults to the value of **fs:*default-pathname-defaults*** if unsupplied.

fs:merge-pathnames-and-set-defaults *pathname* &optional (*defaults* **fs:*default-pathname-defaults***) (*default-version* **':newest**) *Function*

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

The following function is what programs use to complete a partially typed-in pathname.

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

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

string is completed relative to a default pathname constructed from *defaults*, the host (if any) specified by *string*, *type*, and *version*, using the function **fs:default-pathname**. See the function **fs:default-pathname**, page 89. If *string* does not contain a colon, the host comes from *defaults*; otherwise the host name precedes the first colon in *string*.

options are keywords (without following values) that control how the completion will be performed. The following option keywords are allowed. Their meanings are explained more fully below.

:deleted	Look for files that have been deleted but not yet expunged. The default is to ignore such files.
:read or **:in**	The file is going to be read. This is the default. The name **:in** is obsolete and should not be used in new programs.
:write or **:print** or **:out**	The file is going to be written (that is, a new version is going to be created). The names **:print** and **:out** are obsolete and should not be used in new programs.
:old	Look only for files that already exist. This is the default. **:old** is not meaningful when **:write** is specified.
:new-ok	Allow either a file that already exists, or a file that does not yet exist. **:new-ok** is not meaningful when **:write** is specified. The **:new-ok** option is no longer used by any system software, because users found its effects (in the Zmacs command Find File (c-X c-F)) to be too confusing. It remains available, but programmers should consider this experience when deciding whether to use it.

The first value returned is always a string containing a file name; either the original string, or a new, more specific string. The second value returned indicates the status of the completion. It is non-**nil** if it was completely successful. The following values are possible:

:old	The string completed to the name of a file that exists.
:new	The string completed to the name of a file that could be created.

nil The operation failed for one of the following reasons:

- The file is on a file system that does not support completion. The original string is returned unchanged.
- There is no possible completion. The original string is returned unchanged.
- There is more than one possible completion. The string is completed up to the first point of ambiguity.
- A directory name was completed. Completion was not successful because additional components to the right of this directory remain to be specified. The string is completed through the directory name and the delimiter that follows it.

Although completion is a host-dependent operation, the following guidelines are generally followed:

When a pathname component is left completely unspecified by *string*, it is generally taken from the default pathname. However, the name and type are defaulted in a special way described below and the version is not defaulted at all; it remains unspecified.

When a pathname component is specified by *string*, it can be recognized as an abbreviation and completed by replacing it with the expansion of the abbreviation. This usually occurs only in the rightmost specified component of *string*. All files that exist in a certain portion of the file system and match this component are considered. The portion of the file system is determined by the specified, defaulted, or completed components to the left of this component. A file's component *x* matches a specified component *y* if *x* consists of the characters in *y* followed by zero or more additional characters; in other words, *y* is a left substring of *x*. If no matching files are found, completion fails. If all matching files have the same component *x*, it is the completion. If there is more than one possible completion, that is, more than one distinct value of *x*, there is an ambiguity and completion fails unless one of the possible values of *x* is equal to *y*.

If completion of a component succeeds, the system attempts to complete any additional components to the right. If completion of a component fails, additional components to the right are not completed.

A blank component is generally treated the same as a missing component; for example, if the host is a LMFS, completion of the strings "foo" and "foo." deals with the type component in the same way. The strings are not completed identically; completion of "foo" attempts to complete the name component, but completion of "foo." leaves the name component alone since it is not the rightmost.

If *string* does not specify a name, then the name of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, completion fails.

- With the **:write** option, the default name is always used when *string* does not specify a name, regardless of what files exist.

- With the **:new-ok** option, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, the default name is used.

The special treatment of the case where all files in the directory have the same name is not very useful and is not implemented by all file systems.

If *string* does not specify a type, then the type of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, completion fails.

- With the **:write** option, the default type is always used when *string* does not specify a type, regardless of what files exist.

- With the **:new-ok** option, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, the default type is used.

In file systems such as LMFS and UNIX that require a trailing delimiter

(> or ⁄) to distinguish a directory component from a name component, the system heuristically decides whether the rightmost component was meant to be a directory or a name, and inserts the directory delimiter if necessary.

If *string* contains a relative directory specification for a host with a hierarchical file system, it is assumed to be relative to the directory in the default pathname and is expanded into an absolute directory specification.

The host and device components generally are not completed; they must be fully specified if they are specified at all. This might change in the future.

If *string* does not specify a version, the returned string does not specify a version either. This differs from file name completion in TOPS-20; TOPS-20 completes an implied version of "newest" to a specific number. This is possible in TOPS-20 because completing a file name also attaches a "handle" to a file. In Genera, the version number of the newest file might change between the time the file name is completed and the time the actual file operation (open, rename, or delete) is performed.

A pathname component must satisfy the following rules in order to appear in a successful completion:

- The host, device, and directory must actually exist.
- The name must be the name of an existing file in the specified directory, unless **:write** or **:new-ok** is included in *options.*
- The type must be the type of an existing file with the specified name in the specified directory, unless **:write** or **:new-ok** is included in *options.*
- A pathname component always completes successfully if it is **:wild**.

When the rules are not satisfied by a component taken from the default pathname, completion fails and that component remains unspecified in the resulting string. When the rules are not satisfied by a component taken from *string*, completion fails and that part of *string* remains unchanged (other components of *string* can still be expanded).

This function yields a pathname, given its components.

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

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

The following functions are used to manipulate defaults alists directly.

fs:make-pathname-defaults *Function*

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

Defaults alists created with **fs:make-pathname-defaults** are remembered, and reset whenever the site is changed. This prevents remembered defaults from pointing to unknown hosts when world load files are moved between sites.

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

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

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

Obtains a pathname suitable for use as a default pathname and customizes it by modification of its type and version. It also extracts pathnames out of default alists.

The pathname returned by **fs:default-pathname** is always fully specified; that is, all components have non-**nil** values. This is needed when defaulting a pathname with **fs:merge-pathnames** to pass to **open** or other file-system operations, as these operations should always receive fully specified pathnames.

Specifying the optional arguments *host*, *default-type*, and *default-version* as not **nil** forces those fields of the returned pathname to contain those values. If *defaults*, which can be a pathname or a defaults alist, is not specified, the default defaults are used.

If *default-type* is a symbol representing a canonical type, that canonical type is used as the canonical type of the pathname returned. That is, the pathname has a type component that is the correct representation of that canonical type for the host.

Users should never supply the optional argument *sample-p.*

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

Updates a defaults alist. It stores *pathname* into *defaults.* If *defaults* is not specified, the default defaults are used.

The following functions return useful information.

fs:user-homedir &optional *(host* **fs:user-login-machine***)* *Function*

Returns the pathname of the logged-in user's home directory on *host*, which defaults to the host the user logged in to. For a registered user (one who logged in without using the **:host** argument to **login**), the host is the user's **home-host** attribute. Home directory is a somewhat system-dependent concept, but from the point of view of the Symbolics computer it is usually the directory where the user keeps personal files such as init files and mail. This function returns a pathname without any name, type, or version component (those components are all **nil**).

fs:init-file-pathname *program-name* &optional *(canonical-type* **nil***)* *(host* **fs:user-login-machine***)* *Function*

Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the **login** function. *canonical-type* is the canonical type of the init file. It should be **nil** when the returned pathname is being passed to **load** so that **load** can look for a file of the appropriate type.

The following function defines a canonical file type.

fs:define-canonical-type *canonical-type default* &body *specs* *Special Form*

Defines a new canonical type. *canonical-type* is the symbol for the new type; *default* is a string containing the default surface type for any kind of host not mentioned explicitly. The body contains a list of specs that define the surface types that indicate the new canonical type for each host. The following example would define the canonical type **:lisp**.

```
(fs:define-canonical-type :lisp "LISP"
  ((:tops-20 :tenex) "LISP" "LSP")
  (:unix "L" "LISP")
  (:vms "LSP"))
```

For systems with more than one possible default surface form, the form that appears first becomes the preferred form for the type. Always use the interchange case.

Define new canonical types carefully so that they are valid for all host types. For example "com-map" would not be valid on VMS because it is both too long and contains an invalid character. You must define them so that the surface types are unique. That is, the same surface type cannot be defined to mean two different canonical types.

Canonical types that specify binary files must specify the byte size for files of the type. This helps **zl:copyf** and other system tools determine the correct byte size and character mode for files. You specify the byte size by attaching a **:binary-file-byte-size** property to the canonical type symbol. For example, the system defines the byte size of press files as follows.

```
(defprop :press 8. :binary-file-byte-size)
```

The following function is useful when dealing with canonical types. Unlike other functions described here, this function actually accesses and searches a host file system. This description is provided here for completeness. For functions and messages that actually access host file systems: See the section "Streams", page 3.

fs:find-file-with-type *pathname canonical-type* *Function*

Searches the file system to determine the actual surface form for a pathname object. Like **probef**, it returns the truename for *pathname*. When no file can be found to correspond to a pathname, it returns **nil**.

If *pathname* is a string, it is parsed against the default defaults to obtain an actual pathname object before processing.

canonical-type applies only when *pathname* has **nil** as its type component. **fs:find-file-with-type** searches the file system for any matching file with *canonical-type*. For example, on a TOPS-20 host, this would look first for ps:<gcw>toolkit.lisp and then for ps:<gcw>toolkit.lsp:

```
(fs:find-file-with-type (fs:parse-pathname "sc:<gcw>toolkit") ':lisp)
```

If it finds more than one file, it returns the one with the preferred surface type for *canonical-type* (or chooses arbitrarily if none of the files has the preferred surface type).

If *pathname* already had a type supplied explicitly, that overrides *canonical-type*. You can ensure that *canonical-type* applies by first setting the type explicitly:

```
(fs:find-file-with-type (send p ':new-type nil) ':lisp)
```

System programs that supply a default type for input files (for example, **load**) could use this mechanism for finding their input files.

The following functions are useful for poking around.

fs:describe-pathname *pathname* *Function*

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

fs:pathname-plist *pathname* *Function*

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

4.8 Pathname Messages

This section documents some of the messages a user can send to a pathname object. These messages are known as the *passive messages* to pathnames. They deal with inspecting and extracting components, constructing new pathnames based on old pathnames and new components, matching pathnames, and so forth. None of these messages actually interact with any host file system; they deal only with pathname objects within the Symbolics computer.

The other common, useful class of messages to pathnames are those that open, delete, and rename files, list directories, find and change file properties, and so forth. These are the *active messages* to pathnames. You usually do not send these messages directly, but use interface functions, such as **open**, **zl:probef**, **zl:deletef**, **zl:renamef**, **fs:directory-list**, **fs:file-properties**, and **fs:change-file-properties**. Neither these functions and messages, nor additional similar ones, are documented here. See the section "Streams", page 3.

Pathnames handle some additional messages that are intended to be sent only by the pathname system itself, and therefore are not documented here. Only someone who wanted to add a new type of file host to the system would need to understand those internal messages. This section also does not document messages that are peculiar to pathnames of a particular type of host.

:host of **pathname** *Method*

Returns the host component of the pathname. The returned value is always a host object. If the pathname is a logical pathname, the logical host is returned. It is an error to send **:host** to a logical host.

:device of **pathname** *Method*

Returns the device component of the pathname. The returned value can be **nil**, **:unspecific**, or a string. The string is in interchange case.

:directory of **pathname** *Method*

Returns the directory component of the pathname. The returned value can be **nil**, **:wild**, or a list of strings and symbols, each representing a directory level. These symbols can be **:wild** or **:wild-inferiors**. Single names of directories in nonhierarchical file systems are returned as single element lists. The strings are in interchange case.

:name of **pathname** *Method*

Returns the name component of the pathname. The returned value can be **nil**, **:wild**, or a string. The string is in interchange case.

:type of **pathname** *Method*

Returns the type component of the pathname. The returned value is always be **nil**, **:unspecific**, **:wild**, or a string. The string is in interchange case.

:version of **pathname** *Method*

Returns the version component of the pathname. The returned value is always be **nil**, **:wild**, **:unspecific**, **:oldest**, **:newest**, or a number.

:raw-device of **pathname** *Method*

Returns the device component of the pathname. The returned value can be **nil**, **:unspecific**, or a string. The string is in its raw case.

:raw-directory of **pathname** *Method*

Returns the directory component of the pathname. The returned value can be **nil**, **:wild**, or a list of strings and symbols, each representing a directory level. These symbols can be **:wild** or **:wild-inferiors**. Single names of directories in nonhierarchical file systems will be returned as single element lists. The strings are in their raw case.

:raw-name of **pathname** *Method*

Returns the name component of the pathname. The returned value can be **nil**, **:wild**, or a string. The string is in its raw case.

:raw-type of **pathname** *Method*

Returns the type component of the pathname. The returned value is always **nil**, **:unspecific**, **:wild**, or a string. The string is in its raw case.

:canonical-type of **pathname** *Method*

Determines the canonical type of a pathname and a surface representation for the type. It returns two values:

Value	*Meaning*
canonical type	This is either a keyword symbol from the set of known canonical types or a string (when the type component of the pathname is not a known canonical type). The string contains the type component from the pathname, in interchange case.
original type	This is **nil** when the type of the pathname is the same as the preferred surface type for the canonical type. See

the special form **fs:define-canonical-type**, page 90. Otherwise, when the type differs from the preferred or default surface type, it is the original type in interchange case.

For example, for a UNIX pathname, sending the message **:canonical-type** to the following pathnames has these results:

Pathname	*Results*	*from* **:canonical-type** *message*	
foo.l	**:lisp**	**nil**	Preferred surface type
foo.lisp	**:lisp**	"LISP"	Alternate surface type
foo.L	"l"	"l"	Not recognized
foo.LISP	"lisp"	"lisp"	Not recognized

Keep in mind that the **:canonical-type** message returns the type string in the interchange case rather than in the raw case.

:new-device *new-device* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the device component has been changed. The valid set of arguments to the **:new-device** message is the set of possible outputs of **:device**. See the method **(flavor:method :device pathname)**, page 92. A string value is expected to be in interchange case.

:new-directory *new-directory* of **pathname** *Method*

Returns a new pathname which is the same as the pathname it is sent to except that the value of the directory component has been changed. The valid set of arguments to the **:new-directory** message is the set of possible outputs of **:directory**. See the method **(flavor:method :directory pathname)**, page 92. String values are expected to be in interchange case.

:new-name *new-name* of **pathname** *Method*

Returns a new pathname which is the same as the pathname it is sent to except that the value of the name component has been changed. The valid set of arguments to the **:new-name** message is the set of possible outputs of **:name**. See the method **(flavor:method :name pathname)**, page 93. String values are expected to be in interchange case.

:new-type *new-type* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the type component has been changed. The valid set of arguments to the **:new-type** message is the set of possible outputs of **:type**. See the method **(flavor:method :type pathname)**, page 93. String values are expected to be in interchange case.

:new-version *new-version* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the version component has been changed. The valid set of arguments to the **:new-version** message is the set of possible outputs of **:version**. See the method **(flavor:method :version pathname)**, page 93.

:system-type of **pathname** *Method*

Returns the type of host that the pathname is intended for. This value is a keyword from the following set:

:its, :lispm, :multics, :tenex, :tops-20, :unix, :vms, :logical

This is the same set as returned by the **:system-type** message to a host object. It is not likely that you need to use this message directly.

:new-raw-device *dev* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the device component has been changed. The valid set of arguments to the **:new-raw-device** message is the set of possible outputs of **:raw-device**. See the method **(flavor:method :raw-device pathname)**, page 93. A string value is expected to be in its raw case.

:new-raw-directory *new-directory* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the directory component has been changed. The valid set of arguments to the **:new-raw-directory** message is the set of possible outputs of **:raw-directory**. See the method **(flavor:method :raw-directory pathname)**, page 93. String values are expected to be in their raw case.

:new-raw-name *new-name* of **pathname** *Method*

Returns a new pathname which is the same as the pathname it is sent to except that the value of the name component has been changed. The valid set of arguments to the **:new-raw-name** message is the set of possible outputs of **:raw-name**. See the method **(flavor:method :raw-name pathname)**, page 93. String values are expected to be in their raw case.

:new-raw-type *new-type* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the type component has been changed. The valid set of arguments to the **:new-raw-type** message is the set of possible outputs of **:raw-type**. See the method **(flavor:method :raw-type pathname)**, page 93. String values are expected to be in their raw case.

:new-canonical-type *canonical-type* &optional *original-type* of **pathname** *Method*

Returns a new pathname based on the old one but with a new canonical type. *canonical-type* specifies the canonical type for the new pathname. The surface type of the new pathname is based on the default surface type of the canonical type, unless the pathname already had the correct type.

When the pathname object receiving the message already has the correct canonical type, the surface type in the new pathname depends on the presence of *original-type.* When *original-type* is omitted, the new pathname type has the same surface type as the old pathname. When *original-type* is supplied, the surface type for the new pathname is *original-type.* This assumes that *original-type* is a valid representation for *canonical-type*; if that assumption is not met, the *canonical-type* prevails and its default surface type is used.

canonical-type is a symbol for a known type, **:unspecific**, **nil**, or a string. Use a string for *canonical-type* to make pathnames with types that are not known canonical types.

The following examples assume that a pathname object for the file specification "vixen:/usr2/jwalker/mild.new" is the value of **zl-user:m**.

```
(send m ':new-canonical-type ':lisp) =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
(send m ':new-canonical-type ':lisp "LISP") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.lisp">
(send m ':new-canonical-type ':lisp "MSS") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
(send m ':new-canonical-type "BAR" "BAR") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.bar">
(send m ':new-canonical-type ':lisp "lisp") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
(send m ':new-canonical-type ':lisp nil) =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
```

:types-for-canonical-type *canonical-type* of **pathname** *Method*

The internal primitive for finding which surface types correspond to *canonical-type.* Normally you would not use this directly. To determine what form of a pathname exists in a file system: See the function **fs:find-file-with-type**, page 91.

:new-pathname &rest *options* of **pathname** *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are **:host**, **:device**, **:directory**, **:name**, **:type**, **:version**, **:raw-name**, **:raw-device**, **:raw-type**, **:raw-directory**, and **:canonical-type**. The **:type** argument also accepts a symbol as an argument, implying canonical type. See the section "Canonical Types in Pathnames", page 77.

:new-default-pathname &rest *options* of **pathname** *Method*

Returns a new valid pathname based on the one receiving the message, using the pathname components supplied by *options*. The components do not need to be known to be valid on a particular host. The method uses the components "as suggestions" for building the new pathname; it is free to make substitutions as necessary to create a valid pathname. It is heuristic, not algorithmic, so it does not necessarily yield valid semantics. The heuristics used, however, seem to produce pathnames that match what many people expect from cross-host defaulting.

It always produces a pathname with valid syntax but not necessarily valid semantics. For example, when it tries to map between a hierarchical file system and a nonhierarchical file system, it uses the least significant of the hierarchical components as the directory component. Sometimes this is not correct, but in all cases it is syntactically valid. The main applications for **:new-default-pathname** are in producing defaults to offer to the user and in copying components from one kind of pathname to another.

Application notes: **:new-pathname** always does what its arguments specify; it never uses heuristics. Thus **:new-pathname** could signal an error in certain cross-host situations where **:new-default-pathname** would not have any problems. Usually, user programs should use **fs:default-pathname**, which sends **:new-default-pathname** as part of its operation. However, if you are copying a single component from one kind of pathname to another, **:new-default-pathname** is the right tool.

For example, the right way to copy the version from an input pathname to an output pathname is as follows:

```
(defun copy-version (input-pathname output-pathname)
  (send output-pathname :new-default-pathname
        :version (send input-pathname :version)))
```

If the above example used **:new-pathname** or **:new-version**, the input pathname were a UNIX pathname, and the output were a LMFS pathname, this example would signal an error, since **:unspecific** is not a valid version in a LMFS pathname. However, using **:new-default-pathname**, the closest equivalent is substituted, namely **:newest**.

:parse-truename *string* &optional (*from-filesystem* **t**) of **pathname** *Method*

Returns the pathname corresponding to the *string* argument. The *string* is parsed, with the pathname supplying the defaults (notably, the host). The method is useful when, for example, a remote file system produces a string naming a file, and you want the corresponding pathname.

:generic-pathname of **pathname** *Method*

Returns the generic pathname for the family of files of which this pathname is a member. See the section "Generic Pathnames", page 75.

The following messages get a pathname string out of a pathname object:

:string-for-printing of **pathname** *Method*

Returns a string that is the printed representation of the pathname. This is the same as what you get if use **princ** or **string** on the pathname. It is the native host form of the pathname string, preceded by the name of the host and colon. This is the preferred user-visible printed representation of pathnames.

:string-for-wholine of **pathname** *Method*

Returns a string that can be compressed in order to fit in the status line.

:string-for-editor of **pathname** *Method*

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

:string-for-dired of **pathname** *Method*

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

:string-for-host of **pathname** *Method*

Returns a string that is the pathname in the form preferred by the host file system.

:string-for-directory of **pathname** *Method*

Returns a string suitable for describing the directory portion of the pathname, in the format that users of the host system are used to seeing it. The host name is not included.

The following messages manipulate the property list of a pathname:

:get *indicator* of **pathname** *Method*

Manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists" in *Symbolics Common Lisp: Language Concepts.* Be careful using property lists of pathnames. See the section "Pathnames", page 51.

:getl *list-of-indicators* of **pathname** *Method*

This manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists" in *Symbolics Common Lisp: Language Concepts.* Please take care in using property lists of pathnames. See the section "Pathnames", page 51.

:putprop *value indicator* of **pathname** *Method*

This manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists" in *Symbolics Common Lisp: Language Concepts.* Please take care in using property lists of pathnames. See the section "Pathnames", page 51.

:remprop *indicator* of **pathname** *Method*

This manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists" in *Symbolics Common Lisp: Language Concepts.* Please take care in using property lists of pathnames. See the section "Pathnames", page 51.

:plist of **pathname** *Method*

This manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists" in *Symbolics Common Lisp: Language Concepts.*

The following messages can be sent to pathnames having wildcard components or suspected of having wildcard components:

:pathname-match *candidate-pathname* &optional *(match-host t)* of **pathname** *Method*

Determines whether *candidate-pathname* would satisfy the wildcard pattern of the pathname receiving the message. (The pathname receiving the message is assumed to be one that would satisfy **:wild-p.**) It compares corresponding components in the pattern pathname and *candidate-pathname.* It returns **nil** when *candidate-pathname* does not satisfy the pattern; otherwise it returns something other than **nil.**

match-host determines whether it requires the host component of the pattern to match as well. When *match-host* is **nil,** it ignores the host component. By default, it does require that the host component match.

A pattern pathname containing no wild components matches only itself.

If the *candidate-pathname* specifies a physical host, and the message is sent to a logical pathname, the physical host is "back-translated," if possible.

:wild-p of **pathname** *Method*

A predicate that determines whether the pathname is syntactically a wildcard pathname. This means that any component is **:wild**, or, for most systems, contains the character *, or that the directory component has any of the valid forms of directory wildcard in it. See the method **(flavor:method :directory-wild-p pathname)**, page 100.

Value	*Meaning*
nil	No component of the name is syntactically a wildcard.
not **nil**	One or more components of the name are syntactically wild. The actual value in this case is the symbol for the most significant wild component: **:device**, **:directory**, and so on.

:device-wild-p of **pathname** *Method*

If the device component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned.

:directory-wild-p of **pathname** *Method*

If the directory component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. All forms of wildcard at each directory level for hierarchical file systems, as well as **:wild-inferiors**, are recognized as constituting a wildcard directory component. Otherwise, **nil** is returned.

:name-wild-p of **pathname** *Method*

If the name component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. Otherwise, **nil** is returned.

:type-wild-p of **pathname** *Method*

If the type component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. Otherwise, **nil** is returned.

:version-wild-p of **pathname** *Method*

If the version component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. Otherwise, **nil** is returned.

:translate-wild-pathname *target-pattern-pathname starting-pathname* of **pathname** *Method*

Produces a new pathname based on *starting-pathname* and the analogies between the pathname receiving the message and *target-pattern-pathname.*

:translate-wild-pathname examines the correspondences between *target-pattern-pathname* and the pathname receiving the message. It then does whatever is necessary to *starting-pathname* to transform it into the target pathname.

It checks to be sure *starting-pathname* matches the pathname receiving the message and signals **zl:ferror** if they do not match. A standard way for generating *starting-pathname* is to send **:directory-list** to the source pattern pathname to generate a set of starting pathnames.

4.9 Pathnames on Supported Host File Systems

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncrasies.

4.9.1 LMFS Pathnames

LMFS is an acronym for Lisp Machine File System, which is the native file system of the Symbolics computer. It is only one of many possible file systems accessible from the Symbolics computer.

LMFS is a hierarchical file system. It supports file versions. Every file has a name, type, and version. Names are virtually unlimited in length (hundreds of characters), but a performance penalty is imposed for names of over 30 characters. Types are limited to 14 characters. There is no limit to the depth of directories. There are no devices (**:device** to a LMFS pathname always returns **:unspecific**).

A LMFS pathname looks as follows:

```
>dir>ectory>name.type.version
```

The greater-than (">") character separates directory levels. Absolute pathnames always start with greater-than's. Pathnames that specify no directory, relative or otherwise, contain no greater-than's, for example:

```
foo.bar.7
```

The topmost directory of the directory tree (the *ROOT* directory) is indicated by the absence of directory names but the continued presence of a greater-than. For example, the following is a file named foo.bar, version 7, in the ROOT directory:

```
>foo.bar.7
```

No file type abbreviations are needed for LMFS.

File and directory names in LMFS can be stored in upper, lower, or mixed case. Lowercase is the preferred case. Case is ignored on lookup.

Due to problems with interning of pathnames it is sometimes difficult to control the casing of a LMFS pathname, and it is almost always impossible to change it once established. See the section "Interning of Pathnames", page 64.

A version component of **:newest** is represented by the string "newest". A version component of **:oldest** is represented by the string "oldest".

Upward relativization in relative directory specifications is designated by a pathname starting with the character less-than ("<"). All and only all absolute pathnames start with the character greater-than (">"). Downward relativization is indicated by a pathname, which although it contains greater-than's, does not start with one. For example, the following specifies a directory named foo, inferior to the superior directory of the default pathname with which it is merged.

```
    <foo>x.y
```

LMFS directories, when referenced as files, have a file type of "directory" and a version of 1. See the section "Directory Pathnames and Directory Pathnames as Files", page 68.

The following example specifies a directory named bar, inferior to the directory of the default pathname with which it is merged.

```
    bar>x.y
```

LMFS supports recursive directory level matching (**:wild-inferiors**). The representation of **:wild-inferiors** in LMFS is **. Any number of ** components can appear in wildcard pathnames as directory levels, and need not be in trailing positions. (The further it gets from the trailing end of the directory name, however, the more expensive it gets to compute.) Here are some examples of the use of **:

Pathname	*What it means*
>**>*.lisp.newest	All the newest lisp files on the whole file system.
>**>*>secret>*.*.*	All files in subdirectories (but not top-level directories) named "secret".
>lmach>**>*.*.newest	All the newest files in >lmach and all its subdirectories.

A component of **:wild**, in any component except the directory component, is represented by *. *, when accompanied by other characters, such as in foo*bar*, matches zero or more characters, as a wildcard. Although * or names containing * are valid as directory-level component names, a directory component of **:wild** cannot be specified through pathname syntax. This is because "any directory at all" is represented by **(:wild-inferiors)**. A directory [illegible] n as * is a specification for a relative pathname, any subdirector [illegible] irectory of the pathname which is merged. That is represented inte [illegible] s **(:wild)**, not **:wild.**

The name of the ROOT directory, as a file (its "directory pathname as file") is

```
>The Root Directory.directory.1
```

Names of files stored in the Lisp Machine File System can not contain *. This restriction is necessary because * is used consistently to indicate wildcards in pathnames.

You can not access files whose names contain * as a character. A special function allows you to rename any file or directories whose names contain *.

lmfs:rename-local-file-tool *from-path to-path* *Function*

Renames a file in which * appears in one of the pathname components. This function works locally only; you must run it on the machine in whose file system the file is stored. It does not rename a file across the network.

from-path and *to-path* must be pathnames or strings coercible to pathnames. *from-path* is parsed against a default on the local host. *to-path* is parsed against *from-path* as the default. The version number for *to-path* is inherited from the file being renamed. Any version number appearing in *to-path* is ignored.

```
(lmfs:rename-local-file-tool ">AUser>*secret-stuff*" "-secret-stuff-")
(lmfs:rename-local-file-tool ">*special*.directory.1" "-special-")
```

4.9.2 FEP File System Pathnames

The syntax of FEP file system pathnames is identical to that of LMFS pathnames, and the semantics are the same as well. The following differences are to be noted.

- The maximum length of a file name is 32 characters.
- The maximum length of file types is 4 characters.
- The type of directories is "DIR".
- Recursive wildcards (**:wild-inferiors**) are not supported.

The name of the ROOT directory, as a file (its "directory pathname as file") is:

```
>ROOT-DIRECTORY.DIRECTORY.1
```

4.9.3 UNIX Pathnames

Since UNIX file names can only be 14 characters long, the representations of most canonical types are stored in abbreviated form, according to the following table. Other values are represented as they are.

Canonical type	*UNIX abbreviation(s)*
:LISP	"l" "lisp"
:TEXT	"tx" "text" "txt"
:MIDAS	"md"
:QFASL	"qf" "qfasl"
:QBIN	"qb" "qbin"
:BIN	"bn" "bin"
:PRESS	"pr" "press"
:LGP	"lg" "lgp"
:PATCH-SYSTEM-DIRECTORY	"sd"
:PATCH-VERSION-DIRECTORY	"pd"
:BABYL	"bb" "babyl"
:XMAIL	"xm" "xmail"
:MAIL	"ma" "mail"
:RMAIL	"rm"
:ZMAIL-TEMP	"_z" "_zmail"
:GMSGS-TEMP	"_g" "_gmsgs"
:UNFASL	"uf" "unfasl"
:OUTPUT	"ot" "output"
:ULOAD	"ul" "uload"
:MCR	"mc" "mcr"
:SYM	"sm" "sym"
:TBL	"tb" "tbl"
:MICROCODE	"mic"
:ERROR-TABLE	"err"
:FEP-LOAD	"flod"
:SYNC-PROGRAM	"sn" "sync"
:CWARNS	"cw" "cwarns"
:SYSTEM	"sy" "system"
:FONT-WIDTHS	"wd" "widths"
:BFD	"bfd"
:KST	"kt" "kst"
:AST	"at" "ast"

:PLT	"pl" "plt"
:DRW	"drw"
:WD	"wd"
:DIP	"dip"
:SAV	"sav"
:MAP	"map"
:CONSOLIDATED-MAP	"cm"
:TAGS	"tg" "tags"
:PALX-BIN	"pb" "pbin"
:XGP	"xg" "xgp"
:LIL	"ll" "lil"
:SAR	"sar"
:SAB	"sab"
:MSS	"mss" "ms"
:FORTRAN	"f"
:LOGICAL-PATHNAME-TRANSLATIONS	"lt" "logtran"
:LOGICAL-PATHNAME-DIRECTORY-TRANSLATIONS	"ld" "logdir"
:NULL-TYPE	:unspecific ""
:FILES	"fl"
:COLD-LOAD	"load"
:PXL	"px" "pxl"
:IMAGE	"im" "image"
:DUMP	"dm" "dump"

As is true with the canonical type mechanism in general, files having the canonical type spelled in full are also recognized as being of that canonical type.

Logical pathname translation must get around the restrictions in UNIX pathnames. When translating logical pathnames an extra translation step is invoked, in some cases, as for VAX/VMS pathnames.

The preferred case on UNIX is lowercase. Pathname components presented to **:new-directory**, **:new-name**, and so forth, are case-inverted in most instances. See the section "Case in Pathnames", page 71.

4.9.4 UNIX 4.2 Pathnames

UNIX 4.2 uses slightly different representations of some canonical types than do other versions of UNIX. In most cases, the representations are the same as for LMFS, but the UNIX versions are also allowed.

Canonical type	*UNIX 4.2 abbreviation(s)*
:LISP	"lisp" "l"
:TEXT	"text" "tx" "txt"
:MIDAS	"midas" "md"
:QFASL	"qfasl" "qf"
:QBIN	"qbin" "qb"
:BIN	"bin" "bn"
:PRESS	"pr" "press"
:LGP	"lgp" "lg"
:PATCH-SYSTEM-DIRECTORY	"system-dir" "sd"
:PATCH-VERSION-DIRECTORY	"patch-dir" "pd"
:BABYL	"babyl" "bb"
:XMAIL	"xmail" "xm"
:MAIL	"mail" "ma"
:RMAIL	"rmail" "rm"
:ZMAIL-TEMP	"_zmail" "_z"
:GMSGS-TEMP	"_gmsgs" "_g"
:UNFASL	"unfasl" "uf"
:OUTPUT	"output" "ot"
:ULOAD	"uload" "ul"
:MCR	"mcr" "mc"
:SYM	"sym" "sm"
:TBL	"tbl" "tb"
:MICROCODE	"mic"
:ERROR-TABLE	"err"
:FEP-LOAD	"flod"
:SYNC-PROGRAM	"sync" "sn"
:CWARNS	"cwarns" "cw"
:SYSTEM	"system" "sy"
:FONT-WIDTHS	"widths" "wd"
:BFD	"bfd"
:AC	"ac"
:AL	"al"
:KS	"ks"
:KST	"kst" "kt"
:AST	"ast" "at"
:PLT	"pl" "plt"
:DRW	"drw"
:WD	"wd"
:DIP	"dip"
:SAV	"sav"
:MAP	"map"

:CONSOLIDATED-MAP	
	"con-map" "cm"
:TAGS	"tags" "tg"
:PALX-BIN	"palx_bin" "pbin" "pb"
:XGP	"xgp" "xg"
:LIL	"lil" "ll"
:FORTRAN	"f"
:SAR	"sar"
:SAB	"sab"
:MSS	"mss" "ms"
:LOGICAL-PATHNAME-TRANSLATIONS	
	"logtran" "lt"
:LOGICAL-PATHNAME-DIRECTORY-TRANSLATIONS	
	"translations" "logdir" "ld"
:NULL-TYPE	:unspecific ""
:COLD-LOAD	"load"
:FILES	"files" "fl"
:PXL	"pxl" "px"
:IMAGE	"image" "im"
:DUMP	"dump" "dm"

As is true with the canonical type mechanism in general, files having the canonical type spelled in full are also recognized as being of that canonical type.

Logical pathname translation must get around the restrictions in UNIX pathnames. When translating logical pathnames, an extra translation step is invoked as for VAX/VMS pathnames.

The preferred case on UNIX is lowercase. Pathname components presented to **:new-directory**, **:new-name**, and so forth, are case-inverted in most instances. See the section "Case in Pathnames", page 71.

4.9.5 VAX/VMS Pathnames

A VAX/VMS V4 pathname looks as follows:

```
[DIR.ECTORY.COM.PONENTS]NAME.TYP;VERSION
```

The semicolon character is the standard delimiter for the version number. Because of it, a version can be specified even though the name and type are omitted. For compatibility with other Digital Equipment Corporation systems, however, a period is also accepted as a version delimiter when name and type are supplied.

Device is specified by a device name followed a colon preceding the pathname. You must take great caution with pathnames specifying devices so as not to confuse the pathname parser about host identity. See the section "Host Determination in Pathnames", page 62.

Uppercase is the only supported alphabetic case. Pathnames typed in lowercase are converted to uppercase on input.

Filenames cannot contain hyphens, so the underscore character is used in place of a hyphen.

Here is a list of canonical types, their VMS representations, their default byte-size used for a binary transfer, and whether records are stored in fixed- or variable-length format:

Canonical type	*VMS representation*	*Byte-size*	*Format*
:LISP	"LSP"		
:TEXT	"TXT"		
:MIDAS	"MID"		
:QFASL	"QFS"	16	var
:QBIN	"QBN"	16	var
:BIN	"BIN"	16	var
:PRESS	"PRS"	8	fix
:PATCH-SYSTEM-DIRECTORY	"SPD"		
:PATCH-VERSION-DIRECTORY	"VPD"		
:BABYL	"BAB"		
:XMAIL	"XML"		
:MAIL	"MAI"		
:RMAIL	"RML"		
:ZMAIL-TEMP	"ZMT"		
:GMSGS-TEMP	"GMT"		
:UNFASL	"UNF"		
:OUTPUT	"OUT"		
:ULOAD	"ULD"		
:MCR	"MCR"		
:SYM	"SYM"		
:TBL	"TBL"		
:MICROCODE	"MIC"	8	var
:ERROR-TABLE	"ERR"		
:FEP-LOAD	"FLD"		
:SYNC-PROGRAM	"SYN"		
:CWARNS	"CWN"		
:SYSTEM	"SYD"		
:FONT-WIDTHS	"WID"	16	fix
:BFD	"BFD"	16	var

:KST	"KST"	9	
:AC	"AC"	16	
:AL	"AL"	16	
:KS	"KS"	16	
:AST	"AST"		
:PLT	"PLT"	9	
:DRW	"DRW"	12	
:WD	"WD"	12	
:DIP	"DIP"	12	
:SAV	"SAV"	12	
:MAP	"MAP"		
:CONSOLIDATED-MAP	"CON"		
:TAGS	"TAG"		
:PALX-BIN	"PXB"	8	var
:XGP	"XGP"		
:LIL	"LIL"		
:FOR	"FOR"		
:SAR	"SAR"		
:SAB	"SAB"	8	
:MSS	"MSS"		
:LOGICAL-PATHNAME-TRANSLATIONS	"LTR"		
:LOGICAL-PATHNAME-DIRECTORY-TRANSLATIONS	"LDT"		
:NULL-TYPE	""		
:COLD-LOAD	"LOD"	16	var
:FILES	"FLS"		
:PXL	"PXL"	8	
:IMAGE	"IMG"		
:DUMP	"IDM"	16	

Logical pathname translation must get around the restrictions in VMS pathnames, including the prohibition against hyphens. Wherever a hyphen appears in the logical pathname, the underscore character is substituted.

When translating logical pathnames for VAX/VMS, an extra translation step is performed before trying the usual translations. See the file sys:sys;sys.logtran.

The VMS pathname mechanism supports recursive directory matching (**:wild-inferiors**). The representation for a directory level component of **:wild-inferiors** is ".."; however it can appear only at the end of a directory name. Thus, the following matches any file in [A.B], or any subdirectory thereof:

```
[A.B...]*.*.*
```

Upward relativization in pathnames is specified by one or more minuses ("-") as the first directory name. Downward relativization is represented by a null (0-character) first directory name. For example, the following specifies a directory named FOO, inferior to the superior directory of the directory of the default pathname with which it is merged.

```
[-.FOO]X.Y
```

A pathname version component of **:newest** is specified by a version of 0 in the filename string. There is no VMS implementation of **:oldest**.

The percent sign (%) can be used in VMS wildcards to specify the matching of a single character.

The pathname system does not recognize logical device names. They are specified as device names, and are resolved by VMS, not the pathname system. There may be problems defaulting the directory specification of VAX/VMS pathnames when logical devices are used.

VMS directories, when referenced as files, have a type of "DIR", and a version of 1. See the section "Directory Pathnames and Directory Pathnames as Files", page 68.

4.9.6 TOPS-20 and TENEX Pathnames

A TOPS-20 pathname has the form:

```
HOST:DEVICE:<DIRECTORY>NAME.TYPE.VERSION
```

The default device is PS:.

TOPS-20 pathnames are mapped to uppercase. Special characters (including lowercase letters) are quoted with the circle-X (⊗) character, which has the same character code in the Symbolics character set as control-V in the TOPS-20 character set.

TOPS-20 pathnames represent versions of **:oldest** and **:newest** by the strings "..-2" and "..0", respectively.

The directory component of a TOPS-20 pathname is a list of directory level components. The directory <FOO.BAR> is represented as the list ("FOO" "BAR").

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

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

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

4.9.7 Multics Pathnames

Multics possesses a hierarchical file system. Every file has a name, and may or may not have a type. Multics does not support file versions. The sum of the lengths of name and type and the period required to separate them must not exceed 32 characters. A maximum of 16 directory levels is supported. There are no devices (**:device** to a Multics pathname always returns **:unspecific**). A Multics pathname looks as follows:

```
>dir>ectory>name.type
```

The greater-than (">") character separates directory levels. Absolute pathnames always start with greater-than's. Pathnames that specify no directory, relative or otherwise, contain no greater-than's, for example:

```
foo.bar
```

The topmost directory of the directory tree (the *ROOT* directory) is indicated by the absence of directory names but the continued presence of a greater-than. For example, the following is a file named foo.bar, in the ROOT directory:

```
>foo.bar
```

There are no file type abbreviations needed for Multics.

File and directory names can be stored in upper, lower, or mixed case. Lower case is the preferred case. Case is significant. Foo, FOO, and foo could well be the names of three different files in the same directory.

Upward relativization in relative directory specifications is designated by a pathname starting with the character less-than ("<"). All and only all absolute pathnames start with the character greater-than (">"). Downward relativization is indicated by a pathname, which although it contains greater-than's, does not start with one. For example, the following specifies a directory named foo, inferior to the superior directory of the directory of the default pathname with which it is merged.

```
<foo>x.y
```

Multics directories, when referenced as files, have no specific type; they need not have any type at all. See the section "Directory Pathnames and Directory Pathnames as Files", page 68.

The following example specifies a directory named bar, inferior to the directory of the default pathname with which it is merged.

```
bar>x.y
```

Multics does not support **:wild-inferiors**, that is, recursive directory-level matching. For that matter, Multics does not support *any* form of wildcard in the directory component of a pathname. (Although :pathname-match matches such components, Multics does not support them in directory lists.) A component of

:wild, in any component except the directory component, is represented by *. *, when accompanied by other characters, such as in foo*bar*, matches zero or more characters, as a wildcard.

4.9.8 ITS Pathnames

An ITS pathname looks like "HOST: DEVICE: DIR; FOO 69". The default device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: can be used.

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

An ITS filename is converted into a pathname by making the FN2 the version if it is "<", ">", or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols **:oldest** and **:newest**, which correspond to "<" and ">" respectively. If a version is specified, the type is always **:unspecific**. If a type is specified, the version is **:unspecific** so that it does not override the type.

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

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

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

fs:*its-uninteresting-types* *Variable*

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

It is not possible to have two ITS pathnames with the same meaning that differ in an ignored component. **fs:*its-uninteresting-types*** controls which types are ignored in favor of retaining version numbers. The following table summarizes the interaction of type and version components for ITS pathnames.

Type	*Version*	*Result*
supplied	omitted	type is retained, version is **:unspecific**
omitted	supplied	type is **:unspecific**, version is retained
"interesting"	supplied	type is retained, version is **:unspecific**
"uninteresting"	supplied	type is **:unspecific**, version is retained

:fn1 of **fs:its-pathname** *Method*

This message returns a string that is the FN1 host-dependent component of the pathname.

:fn2 of **fs:its-pathname** *Method*

This message returns a string that is the FN2 host-dependent component of the pathname.

4.9.9 MS-DOS Pathnames

An MS-DOS pathname looks like this:

```
HOST:DEVICE:\DIR\ECTORY\NAME.TYPE
```

The default device is C:. Uppercase is the only supported case. Pathnames typed in lowercase are converted to uppercase on input.

File names and directory components are restricted to eight characters. File types are restricted to three characters. The canonical types for MS-DOS are the same as for VAX/VMS.

Relative pathnames are permitted. Upward-level changes are signalled with "..". For example:

```
PC:A:..\..\DIR\FILE.LSP
```

4.9.10 Logical Pathnames

A *logical pathname* does not correspond to a particular file server; its host is called a *logical host.* Every logical pathname can be translated into a corresponding "physical" pathname; a mapping from logical hosts into physical hosts is used to effect this translation.

Logical pathnames make it easy to move bodies of software to more than one file system. An important example is the body of software that constitutes Genera. Any site can have a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites store the sources on a LMFS file system, while others store them on a VAX/VMS system. However, other software in the system must use pathnames for these files in such a way that the software will work correctly at all sites. This is accomplished with a logical host called SYS; all pathnames for system software files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translation is different at each site. For example, at a site where the sources are stored on a certain VAX/VMS system, pathnames of the SYS host are translated into pathnames for that system.

You usually use logical pathnames when you are defining a system that you wish to be portable to other sites. All logical pathnames in your system should translate to a valid pathname on any kind of host to which the system might be distributed. (Currently, this includes LMFS (Symbolics), VAX/VMS, UNIX, and TOPS-20). The converse is not true; logical pathnames make no attempt to provide a way to represent all pathnames on a particular host. For this reason, no way is provided to distinguish between between "foo" and "foo.", or "foo" and "FOO" on UNIX. Your software will be much more portable if you choose good logical pathnames for your files rather than trying to make the logical pathnames conform to the limitations of whatever filesystem you happen to store your system on. For example, even though logical pathnames have a quoting character, it is good practice to avoid using it.

Here, roughly, is how translation is done: To translate a logical pathname, the system finds the mapping for that pathname's host and looks up that pathname's directory in the mapping. If the directory is found, a new pathname is created whose host is the physical host, and whose device and directory come from the mapping. The other components of the new pathname are left the same.

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

However, a logical pathname host can have the same name as a physical host: See the section "Specifying a New Logical Host Name".

A logical pathname has the form HOST: DIRECTORY; NAME.TYPE.VERSION. On input, spaces can separate the name, type, and version. There is no way to specify a device; parsing a logical pathname always returns a pathname whose device component is **:unspecific**. This is because devices have no meaning in logical pathnames. Logical pathnames can be hierarchical; directory levels are separated by semicolons.

Logical pathnames can be *relative*. That is, a pathname can have a directory component whose meaning is "when merging against a default, append these directories onto the end of any default directories." The syntax for this is HOST: ; DIRECTORY; NAME.TYPE.VERSION, that is, a leading bare ; before the directory component. Thus, the above pathname, merged against a default of HOST: USER; FOO.LISP.NEWEST gives
HOST: USER; DIRECTORY; NAME.TYPE.VERSION.

The equivalence-sign character (≡) can be used for quoting special characters such as spaces and semicolons. (The use of this character is discouraged, however, as such files are unlikely to be transportable). The double-arrow character (↔) can be used as a place-holder for unspecified components. Components are not mapped to uppercase. The **:newest**, **:oldest**, and **:wild** values for versions are specified with the strings NEWEST, OLDEST, and * respectively. On input, **:newest** can be represented by > and **:oldest** by <.

There is no init file naming convention for logical hosts; you cannot log into them. The **:string-for-host**, **:string-for-wholine**, **:string-for-dired**, and **:string-for-editor** messages are all passed on to the translated pathname, but the **:string-for-printing** is handled by the **fs:logical-pathname** flavor itself and shows the logical name.

4.9.10.1 Logical Pathname Wildcard Syntax

Logical pathnames support a wildcard syntax meaning "Match any directory, and any subdirectory, to any level." For example:

```
Show Directory SYS:**;*.BFD.*
```

Here, the Show Directory command lists all font files anywhere in the SYS hierarchy, to any level.

This corresponds to the >**> syntax for LMFS pathnames, and the [name...] syntax for VAX/VMS file specifications. See the section "LMFS Pathnames", page 101. See the section "VAX/VMS Pathnames", page 107.

This makes it easy to specify logical pathname translations on Lisp Machines and VAX/VMS. For example:

```
(fs:set-logical-pathname-host "SYS"
   :translations '(("SYS:**;*.*.*" "ACME-LISPM:>Rel-6>**>*.*.*")))

(fs:set-logical-pathname-host "SYS"
  :translations
  '(("SYS:**;*.*.*" "ACME-VMS:SYMBOLICS:[REL6...]*.*;*"))
  :no-translate nil)
```

It is important to note that wherever a "**;" appears in the logical-host pathname, there must be a corresponding "wild-inferiors" pathname on the physical-host pathname.

UNIX and TOPS-20 do not have a syntax with this meaning. For these hosts, it is necessary to list explicitly each level of directory to be translated. For example:

```
(fs:set-logical-pathname-host "SYS"
  :translations
   '(("SYS:*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*.*.*")
     ("SYS:*;*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*//*.*.*")
     ("SYS:*;*;*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*//*//*.*.*")
     ("SYS:*;*;*;*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*//*//*//*.*.*"))
  :no-translate nil)
```

4.9.10.2 Translation Rules

The logical system host **sys** comes preloaded with heuristics that eliminate characters illegal in VAX/VMS file specifications, such as "-".

The heuristics also deal with limitations in the lengths of file specifications on foreign hosts. For example, some file names can be shortened and contracted without changing their meanings. Thus, sys:io;pathnm-cometh.lisp may translate to acmevax:symbolics[rel6.io]pthnmcmth.lsp on a VAX/VMS physical host.

The system keeps careful track of these changes and does not allow two logical pathnames to translate to the same thing. On the attempt to translate a second logical pathname to a physical pathname already found as the result of a logical-pathname translation, an error is signalled. If the first attempt was due to a typographical error made by the user, and the second was due to the system translating a logical pathname, for example in response to the `m-.` command, the error is signalled. However, when **:no-translate nil** is used in the **fs:set-logical-pathname-host** form, the system translates all its logical pathnames when setting the logical system host; then, incorrect translations cannot be entered by mistake.

There also are special translation rules for microcode files, font files, and others, which retain the special characteristics of these file names.

4.9.10.3 Splitting Logical Hosts Across Physical Hosts

It is possible to have a logical host translate to more than one physical host. All that is needed is an explicit specification of the hosts involved, in the translation list given to **fs:set-logical-pathname-host**. For example:

```
(fs:set-logical-pathname-host "SYS"
  :translations '(("SYS:DOC;**;*.*.*" "ACME-LISPM:>Rel-6>doc>**>*.*.*")
                  ("SYS:**;*.*.*" "ACMEVAX:SYMBOLICS:[REL6...]*.*.*"))
  :no-translate nil)
```

Note that it is not necessary to specify the **:physical-host** argument to

fs:set-logical-pathname-host as long as the host names are specified in the translation list. If the argument is specified, it serves as a default when parsing those pathnames.

4.9.10.4 Logical Pathname Translation

This section explains the format of the "translations" list of logical pathnames and the rules for translating a logical pathname to a physical pathname.

Each element of the list (one translation) specifies two wildcard pathnames, the first on the logical host and the second on the physical. In the Lisp form (in the file sys:site;*host*.translations) that specifies this form, they are given as strings to be parsed against these respective hosts. As they are parsed, they are merged with a pathname of wild name, wild type, and wild version.

Following is an example of a translations list. This is a sample LMFS translation table for the SYS host, slightly more complex than the default:

```
'(("SYS:DOC;**;*.*.*" "ACME-S:>Rel-6>doc>**>*.*.*")
  ("SYS:**;*.*.*" "ACME-Q:>Rel-6>**>*.*.*"))
```

There are two phases to the translation process. In the first phase, a logical/physical pathname pair is found in the translation table. This pair is called the *translation pair.*

In the second phase, this pair is presented to a *translation rule* to be processed. Normally, this rule uses **:translate-wild-pathname** to translate the pathname using the translation pair, but there is a wide variety of translation rules.

The first phase consists of matching the pathname to be translated against each first element of each translation, in succession. (The **:pathname-match** message is used.) The order in the list is thus very important. The first match is then taken to be the translation pair for the second phase.

When the physical host supports a syntax for **:wild-inferiors** (for example, >**> on LMFS), that syntax can be used to have a translation that matches "everything else", as in the example above. If no equivalent syntax is supported, a separate wild-card directory for each level of directory likely to be encountered serves the same purpose, as in the example below.

```
'(("SYS:*;*.*.*" "ACME-UNIX://usr//rel-6//*//*.*")
  ("SYS:*;*;*.*.*" "ACME-UNIX://usr//rel-6//*//*//*.*")
  ("SYS:*;*;*;*.*.*" "ACME-UNIX://usr//rel-6//*//*//*//*.*"))
```

This example handles any SYS pathname with up to three directory levels. In the presence of such a translation, it is impossible to have an undefined translation.

The second phase is potentially more complex. In its simplest form, it reduces to producing the translated pathname by sending the **:translate-wild-pathname-reversible** message to the logical pathname, with the first element of the translation as the source pattern and the second element of the translation as the target pattern. See the section "Wildcard Pathname Mapping", page 79. See the section "Wildcard Directory Mapping", page 81. See the section "Reversible Wildcard Pathname Translation", page 118.

However, before deciding to using **:translate-wild-pathname-reversible**, a search is done to find a more suitable rule for performing the translation. With each logical host, there are three sets of translation rules. In addition, there is a global set of rules, and a default.

Here is the order in which these rules are searched:

Permanent	The permanent translation rules are special purpose rules that cannot be overridden. They provide for such things as the translation of patch file pathnames. This table is searched first.
Site	The site translation rules are provided to override the supplied translation rules at a specific site.
Supplied	The normal, supplied translation rules are normally supplied by the author of the software using the logical host.
Global	This is a set of rules independent of any particular host. This table is not currently used for anything, but it is provided for future extension.
Default	This is the rule used when no other rule is found. This is the **:translate-wild** rule, which uses **:translate-wild-pathname-reversible** to translate according to the translation pair found in phase 1 of the translation process.

Back-translation is performed by searching the second elements of the translations list, and translating in the other direction. **:translate-wild-pathname-reversible** is always used for this, so it is not guaranteed to come up with the same logical pathname as might be expected.

Reversible Wildcard Pathname Translation

A special version of wild pathname translation, called "reversible wild pathname translation," is used. The difference between regular wild pathname translation and reversible translation is in the treatment of a target wildcard pattern consisting solely of *. In regular translation, a target pattern of **:wild** causes the source component to be copied verbatim. This is a useful user-interface feature, but it causes dropping of information and resultant noninvertibility of the transformation. In reversible mapping, this feature is not present. Logical pathname translation and back-translation is done in this mode.

Example:

Type	*Source pattern*	*Source instance*	*Target pattern*	*Result*
Regular	foo*	foolish	*	foolish
Reversible	foo*	foolish	*	lish
Either	*	bar	foo-*	foo-bar

Note that the inverse translation of foo-bar to bar cannot be accomplished under regular translation.

Defining a Translation Rule

Translation rules are defined using the **fs:set-logical-pathname-host** function, using the **:rules** or **:site-rules** argument. (The other rule tables are not normally set by the user). These arguments should be an alist of system type and translation rule specifications.

```
((:vms VMS rule specifications ...)
 (:vms4 VMS4 rule specifications ...)
 (:unix UNIX rule specifications ...))
```

Each rule specification consists of a *pattern*, a *rule type*, and optional arguments, as in the following example.

```
("PICTURE:EDITOR;LINE-DRAWING-COMMANDS.*.*" :vms-new-pathname :name "LINECMNDS")
```

In this example, **"picture:editor;line-drawing-commands.*.*"** is the pattern, **:vms-new-pathname** is the rule type, and **:name** and **"linecmnds"** form a keyword/value pair of arguments to the **:vms-new-pathname** rule type.

Normally, translation rules are defined in the system definition file before a **defsystem** form, so that the rules are loaded before they are needed. If you wish to override the translation rules provided either by Symbolics or another vendor, you can use the **:site-rules** argument to the call to **fs:set-logical-pathname-host**, normally placed in the translation file.

The following sections describe the various rule types that exist and their arguments.

:translate-wild &rest *options* *Translation rule*

The default translation rule's type is **:translate-wild**. This simply sends the source pattern a **:translate-wild-pathname-reversible** with the target pattern as target and the pathname being translated as the source pathname. For example:

contents of sys.translations *file:*

```
(fs:set-logical-pathname-host "SYS"
  :translations '(("SYS:DOC;**;*.*.*" "S:>Rel-6>doc>*.*.*")
                  ("SYS:**;*.*.*" "Q:>Rel-6>**>*.*.*")))
```

pathname to translate:
```
SYS:IO;PATHNM.LISP.23
```

translation pair found in phase 1:
```
("SYS:**;*.*.*" "ACME:>Rel-6>**>*.*.*")
```

result of translation:
```
ACME:>Rel-6>io>pathnm.lisp.23
```

In other words, the default is for the translation to occur according to the wildcard mapping given in the translations.

:new-pathname &key *device directory name type version* *Translation rule*

The **:new-pathname** translation rule type is similar to **:translate-wild**, but replaces the *directory*, *name*, *type*, or *version.* Any components not specified in the argument list will not be replaced, and will be derived via **:translate-wild-pathname-reversible** as for the **:translate-wild** translation rule type.

:vms-heuristicate &optional *substitute* *Translation rule*

This translation rule tries hard to make understandable VMS pathnames out of longer, hyphenated filenames. It works for both :VMS and :VMS4 hosts. It produces usually understandable, hopefully unique, legal names and directories. In operation, it is similar to the **:translate-wild** type, but the components translated by wildcards are subjected to heuristics if needed to fit VMS's pathname syntax.

The *substitute* argument is used to perform character substitutions. For example, for VMS4, it can be used to substitute "_" for "-".

```
("SYS:**;*.*.*" :vms-heuristicate ((#\- #\_)))
```

:vms-heuristicate-name &optional *substitute* *Translation rule*

:vms-heuristicate-name is like **:vms-heuristicate**, but heuristicates only the name.

:vms-heuristicate-directory &optional *substitute* *Translation rule*

:vms-heuristicate-directory is like **:vms-heuristicate**, but heuristicates only the directory name.

:vms-new-pathname &key *device directory name type version* *Translation rule*

The **:vms-new-pathname** translation rule is a cross between **:new-pathname** and **:vms-heuristicate**. Components not explicitly specified in the argument list are supplied by wildcard mapping plus heuristics as for **:vms-heuristicate**.

:vms-font &optional *renamings* *Translation Rule*

The **:vms-font** translation rule *parses* the name component of the logical pathname as a font spec. For example, in **fix.roman.normal**, the *family* is **roman**, the *size* is **normal**, and the *style* is **fix**. (The style is optional). The font family is subjected to the VMS heuristics to fit in a smaller space (to allow room for the size and style). The result is concatenated with the size and style to construct a new name.

If the *renamings* argument is supplied, it is an alist of font names and replacement to be used instead of the one produced by the heuristics. This is useful in cases where the heuristic produces a confusing name, or where there would otherwise be name conflicts. For example, the following translation rule is used with the SYS: host for VMS hosts.

```
("SYS: FONTS; LGP-1; *.BFD.*" :vms-font
 (("DANG-MATH" "DANGM")
  ("GHELVETICA" "GHLVT")
  ("HELVETICA" "HELVT")
  ("TIMESROMAN" "TIMSR")
  ("XGP-VGV" "XGPVV")))
```

This translation rule serves to encode the relevant information that makes each font distinct.

In addition, **:vms-font** performs full VMS heuristics on the directory.

:vms-microcode *Translation rule*

This translation rule encodes the microcode names in such a way as to be sure to retain the information that distinguishes different microcodes.

The name component of the logical pathname is parsed into words. Each word is looked up in the alist **fs:*vms-microcode-translation-alist***. (The alist is shared with the equivalent translation for UNIX). If found, it is replaced with the replacement (a single character, except "MIC" maps to "") found in the second element of the alist bucket. This sequence of characters is then concatenated to produce the new filename.

The directory component is subject to full heuristication.

:tops20-heuristicate-directory &optional *(levels* **fs:*default-tops20-directory-levels****)* *Translation rule*

The **:tops20-heuristicate-directory** translation rule compensates for the fact that TOPS20 directories are limited to a size of **fs:*tops-20-max-field-size***, including the "." characters as directory-level separators. Each level of directory is allocated a share of the available space, and is compressed to fit in that space as needed. In determining how much space to allocate to each level, the rule assumes that no more than *levels* directory levels will be needed. The default is **fs:*default-tops20-directory-levels***, or 3 levels.

:unix-microcode *Translation rule*

This translation rule encodes the microcode names in such a way as to be sure to retain the information that distinguishes different microcodes.

The name component of the logical pathname is parsed into words. Each word is looked up in the alist **fs:*unix-microcode-translation-alist***. (The alist is shared with the equivalent translation for VMS). If found, it is replaced with the replacement (a single character, except "MIC" maps to "") found in the second element of the alist bucket. This sequence of characters is then concatenated to produce the new filename.

:unix-font &optional *renamings* *Translation rule*

The **:unix-font** translation rule *parses* the name component of the logical pathname as a font spec. For example, in **fix.roman.normal**, the *family* is **roman**, the *size* is **normal**, and the *style* is **fix**. (The style is optional). The font family is subjected to the VMS heuristics to fit in a smaller space (to allow room for the size and style). The result is concatenated with the size and style to construct a new name.

If the *renamings* argument is supplied, it is an alist of font names and replacement to be used instead of the one produced by the heuristics. This is useful in cases where the heuristic produces a confusing name, or where there would otherwise be name conflicts. For example, the following translation rule is used with the SYS: host for UNIX hosts.

```
("SYS: FONTS; LGP-1; *.BFD.*" :unix-font
 (("DANG-MATH" "DANGMT")
  ("GHELVETICA" "GHELVT")
  ("HELVETICA" "HELVET")
  ("TIMESROMAN" "TIMESR")
  ("XGP-VGV" "XGPVGV")))
```

This translation rule serves to encode the relevant information that makes each font distinct.

:unix-type-and-version &optional *renamings* *Translation rule*

The **:unix-type-and-version** translation rule is used for situations where you need to retain both the type and version. This is usually needed where differing versions of the file need to coexist.

The name component is matched against the *renamings* alist. If it is found, the second element of the alist bucket is used instead. Then, if the last character of the name (or the replacement) is a digit, a "+" is added to the end. Then, the version number (or "" if **nil** or "*" if **:wild**) is added to the end. This is then used as the name component. The type is handled via the normal mechanisms.

The version is added to the name rather than the end of the type, so that the type field can be recognized by programs that look at the type (or canonical type).

:site-directory &key *device directory name type version* *Translation rule*

The **:site-directory** translation rule substitutes the **:site-directory** attribute from the local site object for the host and directory. The arglist is like for **:new-pathname**. This is used to translate SYS:SITE;.

As a special feature, this rule can be overridden by an explicit entry for SYS: SITE; in the translations. This can be useful when debugging, to get a different site directory without modifying your site namespace object.

fs:patch-file *system-name* &optional *file-type* *Translation rule*

fs:patch-file rules, which will often be seen when doing a **fs:describe-logical-host**, are internal to the patch system. They provide for the translation of patch file logical names to physical files, in a system-dependent manner. These rules are added as a result of defining a system to be patchable.

fs:describe-logical-host *host* *Function*

The **fs:describe-logical-host** function takes a logical host (or the name of a logical host) and provides various information about the host, including:

- Default physical host.
- Translations.
- Translation rules sorted by search order.
- Translation rules sorted by group.

It is often useful for determining what went wrong with a translation file.

fs:make-logical-pathname-host *name* &key *no-search-for-shadowed-physical* *Function*

Defines *name*, which should be a string or symbol, to be the name of a logical pathname host. *name* should not conflict with the name of any existing host, logical or physical.

fs:make-logical-pathname-host also loads the file sys:site;*name*.translations and arranges for that file to be reloaded in the future. **load-patches** checks the translations file for each logical host that is defined in the current world; if any file has been changed it is reloaded. **load-patches** does this if and only if no specific systems are specified in its arguments.

fs:make-logical-pathname-host alters the **logical-pathnames-translation-files** system so that it contains the translations files for all logical hosts defined in the current world. **load-patches** loads updated translations files by calling the Compile System command on this system.

An **fs:make-logical-pathname-host** form often appears in the file sys:site;*system-name*.system. The Compile System command looks for this file when given the name of an unknown system. The **fs:make-logical-pathname-host** form must be the first form in the file, as the second form, a call to **si:set-system-source-file**, depends on the previous definition of the logical host.

Example: Following are the contents of the file sys:site;cube.system:

```
;;; -*- Mode: LISP; Package: USER -*-

(fs:make-logical-pathname-host "cube")
(si:set-system-source-file "cube" "cube: cube; cubpkg")
```

The argument **:no-search-for-shadowed-physical** (default **nil**) means to look only in the existing pathname hosts for a host with the same name as the logical host. This saves time by not asking the namespace server whether the name of the newly defined logical host conflicts with the names of any physical hosts, but it prevents you from seeing the following warnings:

```
Warning: the host ~A must now be referred to as ~A: in pathnames,
                        since ~A is now a logical pathname host.
                        This affects ~[no~:;~:*~D~] extant pathnames.

Warning: the nickname ~A: for the physical host ~A
                          will now refer instead to the
                          logical pathname host ~A.
                          Use ~A: in pathnames.
```

fs:add-logical-pathname-host is an obsolete name for this function.

fs:set-logical-pathname-host *logical-host* &key *physical-host* *translations rules site-rules* (*no-translate* **t**) *no-search-for-shadowed-physical* — *Function*

Creates a logical host named *logical-host* if it does not already exist. It then establishes translations of logical directories on *logical-host* to physical directories on various hosts. (*physical-host* serves as a default.) *translations* is a "translations" list of two-element lists of strings representing associated logical directories (source patterns) and physical directories (target patterns). For the format of the lists and the translation rules: See the section "Logical Pathname Translation", page 117.

Source patterns are logical pathnames that are matched against the pathname being translated. The target patterns are physical pathnames and can be on any host.

If the physical pathname is on a TOPS-20 or VAX/VMS host, you should include the device name. In the case of VAX/VMS, it is important that this device name be either a physical device name or the name of a "concealed device." The simplest way to choose a device name is to connect to the VAX/VMS system in question. If you want to use FOO:[BAR...]*.*.* as the target, where FOO is a VAX/VMS system-wide logical name, connect to VAX/VMS and do the following:

```
$ DIRECTORY FOO:[BAR...]*.*.*

Directory USER$DISK:[BAR...]
...
```

In this example, you should use USER$DISK:[BAR...] instead of FOO:[BAR...] in your translations.

If *no-translate* is **nil**, the translation of every interned logical pathname is checked. Properties are copied from the old physical pathname to the the new one, and logical pathnames that now have no corresponding physical pathnames are uninterned. If *no-translate* is not **nil** or not supplied, this mapping is suppressed, and some physical pathnames might not get the properties of the logical pathname. This is not normally of any consequence, so *no-translate* defaults to **t**.

A call to **fs:set-logical-pathname-host** is usually the only form in the file sys:site;*logical-host*.translations. This file is loaded by **fs:make-logical-pathname-host** (always in the **file-system** package), which also arranges for it to be reloaded in the future. **load-patches** checks this file for all logical hosts in the current world and reloads the file if it has changed. Similarly, changing the site object will cause each translation file to be reloaded from the new site directory.

The argument *no-search-for-shadowed-physical* (default **nil**) means to look only in the existing pathname hosts for a host with the same name as the logical host. This saves time by not asking the namespace server whether the name of the newly defined logical host conflicts with the names of any physical hosts, but it prevents you from seeing the following warnings:

```
Warning: the host ~A must now be referred to as ~A: in pathnames,
                   since ~A is now a logical pathname host.
                   This affects ~[no~:;~:*~D~] extant pathnames.

Warning: the nickname ~A: for the physical host ~A
                    will now refer instead to the
                    logical pathname host ~A.
                    Use ~A: in pathnames.
```

Example:

Following is a typical content of the file sys:site;sys.translations:

```
;;; -*- Mode: LISP; Package: FILE-SYSTEM -*-

(set-logical-pathname-host "sys"
 :translations '(("**;*.*.*" ">Rel-6>**>")))
```

:translated-pathname of **fs:logical-pathname** *Method*

Converts a logical pathname to a physical pathname. It returns the translated pathname of this instance: a pathname whose **:host** component is the physical host that corresponds to this instance's logical host. See the section "Logical Pathnames", page 113.

If this message is sent to a physical pathname, it simply returns itself.

:back-translated-pathname *pathname* of **fs:logical-pathname** *Method*

Converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*. See the section "Logical Pathnames", page 113.

This message might be used in connection with truenames. Given a stream that was obtained by opening a logical pathname,

```
(send stream :pathname)
```

returns the logical pathname that was opened.

```
(send stream :truename)
```

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

```
(send (send stream :pathname)
      :back-translated-pathname
      (send stream :truename))
```

If this message is sent to a physical pathname, it simply returns its argument. Thus the above example works no matter what kind of pathname was opened to create the stream. However, it is important to note two situations in which back translation can fail to do what you expect:

Links	If opening the file involved following a link, the truename will no longer match, and back translation

might not be able to convert it to a physical pathname at all.

File-system restrictions

If the translation involved compressing or modifying a name to adapt to a file-system's rules, the physical pathname may be translated to a logical pathname different from the one originally used.

Back translation is useful only in cases where the logical pathname is wanted for informational, not operational, purposes. For example, if you remember a back translation to reopen the file, you may end up with physical instead of logical pathnames in your program. Physical pathnames are not transportable between sites.

One way to avoid this problem is to avoid back translation. Often, all that is needed is the version number, in which case the following code will serve:

```
(send (send stream :pathname)
      :new-default-pathname
      :version (send (send stream :truename) :version))
```

Note that **:new-default-pathname** is used rather than **:new-pathname**. This is necessary because the logical host and the physical host are of different types. When copying components between host types, you need to allow for certain substitutions. In this case, if the physical host is a UNIX system, the version will be **:unspecific**, and **:new-default-pathname** will convert this to the nearest equivalent for logical pathnames: **:newest**.

4.10 Init File Naming Conventions

Init files are of canonical type **:lisp** for source files and **:bin** for compiled files. For hosts that support long file names, the init file name consists of *program-name* with "-INIT" appended. Thus, the standard file name for a Genera init file is LISPM-INIT; for a Zmail init file, it is ZMAIL-INIT. Hosts that do not support long file names have conventions peculiar to each system.

Following are the names of lispm init source files on some hosts:

Host system	*File name*
LMFS/TOPS-20	LISPM-INIT.LISP
UNIX	lispm-init.l
VMS	LISPMINI.LSP

Multics	lispm-init.lisp
ITS	If user has own directory: LISPM >. If user does not have own directory: *USER* LISPM.

5. File and Directory Access

5.1 Accessing Files

Genera lets you access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Symbolics computer itself, if the machine has its own file system. This section tells you how to get a stream that reads or writes a given file, and what the device-dependent operations on that stream are. Files are named with *pathnames*. Since pathnames are quite complex they have their own chapter. See the section "Naming of Files", page 51.

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. The file-opening options control whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, and so on.

The following option keywords are recognized. Unless otherwise noted, they are supported generically. Additional keywords can be implemented by particular file system hosts.

:byte-size The possible values are **nil** (the default), a number in the range 1 to 16 inclusive, which is the number of bits per byte, and **:default**, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, **nil** selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, **nil** selects the default byte size of 16 bits. The preferred way to specify the byte-size for files is to use the **:element-type** keyword.

:characters This option specifies whether the objects contained in the file are characters or fixnums. The preferred way to specify character files is to use the **:element-type** keyword.

Value	*Meaning*
t	Specifies that the file contains character objects. This is the default.
nil	Specifies that the file is a binary file.
:default	On output, **:default** is always **t**, as character files are created by default. On input,

:default specifies that the file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:deleted	The default is **nil**. If **t** is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.
:direct	The default is **nil**. **t** specifies a direct access stream. See the section "Direct Access File Streams", page 13.
:direction	The **:direction** option allows the following values:

:input	The file is being opened for input. This is the default.
:output	The file is being opened for output.
:block	This is a special case of **:output** that is used for the FEP File System.
:io	The file is being opened for intermixed input and output. Bidirectionality is supported only if the stream is to be a direct stream, that is, **:direct t** is given as well. See the section "Direct Access File Streams", page 13.
:probe	A "probe" opening; no data are to be transferred, and the file is being opened to determine whether the file exists, or to gain access to or change its properties. Returns the truename of the object at the end of a link or chain of links. If the value of **:direction** is **:probe** and the value of **:error** is **nil**, then **open** will return the error object instead of **nil**. If the value of **:if-does-not-exist** is **nil**, the error object will still be returned.
:probe-link	The same as **:probe** except that links are not chased. Returns the truename of the object named, even if it is a link.
:probe-directory	The pathname is being opened to find out about the existence of its *directory* component.

Otherwise, the semantics are the same as **:probe**. If the directory is not found, a file lookup error is signalled.

nil	This is the same as probe. No data are transferred, and the file is being opened only to gain access to or change its properties. If the value of **:direction** is **nil** and the value of **:error** is **nil**, then **open** will return the error object instead of **nil**. If the value of **:if-does-not-exist** is **nil**, the error object will still be returned.

:element-type This argument specifies the type of Lisp object transferred by the stream. Anything that can be recognized as being a finite subtype of **character** or **integer** is acceptable. In particular, the following types are recognized:

string-char	The object being transferred is a string-character. The functions **read-char** and/or **write-char** can be used on the stream. This is the default.
character	The object being transferred is any character, not just a string-character. The functions **read-char** and/or **write-char** can be used on the stream.
(unsigned-byte *n***)**	The object being transferred is an unsigned byte (a non-negative integer) of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
unsigned-byte	The object being transferred is an unsigned byte (a non-negative integer) whose size is determined by the file system. The functions **read-byte** and/or **write-byte** can be used on the stream.
(signed-byte *n***)**	The object being transferred is a signed byte of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
signed-byte	The object being transferred is a signed byte whose size is determined by the file system. The functions **read-byte** and/or **write-byte** can be used on the stream.

bit	The object being transferred is a bit (values 0 and 1). The functions **read-byte** and/or **write-byte** can be used on the stream.
(mod *n*)	The object being transferred is a non-negative integer less than *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
:default	On output, **:default** is always **character**, as character files are created by default. On input, **:default** specifies that the file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:error This option controls what happens when any **fs:file-operation-failure** condition is signalled. **t** is the recommended value for this option. The others have been provided for compatibility with previous systems to aid in converting programs. See the section "File-System Errors" in *Symbolics Common Lisp: Language Concepts.*

The option has three possible values:

Value	*Meaning*
t	Signals the error normally. **t** is both the default and the recommended value.
nil	Returns the error object. If the value of either **:if-exists** or **:if-does-not-exist** is **nil**, the error object is still returned.
:reprompt	Reprompts the user for another file name and tries **open** again. When you use this option, remember that the **:pathname** message sent to the stream finds out what file name was really opened. The alternative to **:reprompt** is to use **:error t** and set up a condition handler for **fs:file-operation-failure** that explains the condition and prompts the user.

:estimated-length The value of the **:estimated-length** option can be **nil** (the default), which means there is no estimated length, or a number of bytes indicating the estimated length of a file to be written. Some file systems use this to optimize disk allocation.

:if-does-not-exist	Specifies the action to be taken if the file does not already exist. The following values are allowed:	
	:error	Signals an error. This is the default if the **:direction** is **:input**, **:probe**, or any of the **:probe**-like modes, or if the **:if-exists** argument is **:overwrite**, **:truncate**, or **:append**.
	:create	Creates an empty file with the specified name, and then proceeds as if it had already existed. This is the default if the **:direction** is **:output** and the **:if-exists** argument is anything but **:overwrite**, **:truncate**, or **:append**.
	nil	Does not create a file or even a stream. Instead, simply returns **nil** to indicate failure. This is overridden when the value of **:direction** is either **nil** or **:probe** and the value of **:error** is **nil**. In this case, the error object is returned instead of **nil**.
:if-exists	Specifies the action to be taken if the **:direction** is **:output** and a file of the specified name already exists. If the direction is **:input** or **:probe** (or any of the **:probe**-like directions), this argument is ignored. The following values are allowed:	
	:error	Signals an error. This is the default when the version component of the filename is not either **:newest** or **:unspecific**.
	:new-version	Creates a new file with the same file name but a larger version number. This is the default when the version component of the filename is either **:newest** or **:unspecific**. File systems without version numbers can choose to implement this by effectively treating it as **:supersede**.
	:rename	Renames the existing file to some other name, and then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.

:rename-and-delete
Renames the existing file to some other name and then deletes it (but does not expunge it, on those systems that distinguish deletion from expunging). Then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.

:overwrite The existing file is used, and output operations on the stream destructively modify the file. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.

:truncate The existing file is used, and output operations on the stream destructively modify the file. The file pointer is initially positioned at the beginning of the file; at that time, the file is truncated to length zero, and disk storage occupied by it is freed.

:append The existing file is used, and output operations on the stream modify the file. The file pointer is initially positioned at the current end of the file.

:supersede Supersedes the existing file. If possible, the file system does not destroy the old file until the new stream is closed, against the possibility that the stream will be closed in "abort" mode. This differs from **:new-version** in that **:supersede** creates a new file with the same name as the old one, rather than a file name with a higher version number.

nil Does not create a file or even a stream. Instead, simply returns **nil** to indicate failure. This is overridden when the value of **:direction** is either **nil** or **:probe** and the value of **:error** is **nil**. In this case, the error object is returned instead of **nil**.

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

:raw	The value can be **nil** (the default) or **t**, which disables all character set translation in ASCII files.
:submit	This is an option to **open** used to get batch jobs. Currently, this is implemented only for VAX/VMS. When the file you are writing is closed, the file is submitted as a batch job by using this option.
:super-image	The value can be **nil** (the default), or **t** which disables the special treatment of Rubout in ASCII files. Normally Rubout is an escape that causes the following character to be interpreted specially, allowing all characters from 0 through 376 to be stored. This applies to PDP-10 file servers only.
:temporary	The default is **nil**. If **t** is specified, the file is marked as temporary, if the file system has that concept.

with-open-file *(stream-variable filename . options...)* &body *body...* *Special Form*

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

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

filename is the name of the file to be opened; it can be a pathname object, a string, or a symbol. It can be anything acceptable to **fs:parse-pathname**. See the section "Naming of Files", page 51. The complete rules for parsing pathnames are explained there.

If an error occurs, such as file not found, the user is asked to supply an alternate pathname, unless this is overridden by *options*. At that point, the user can quit out or enter the Debugger, if the error was not due to a misspelled pathname.

If you are opening the file to read it with **read**, and you want to bind the package and so forth, see the special functions for handling file attributes. See the function **fs:read-attribute-list**, page 158. See the function **fs:file-attribute-bindings**, page 159.

with-open-file-case *(var pathname . options)* &body *clauses* *Special Form*

Opens a file, binding the input stream to *var*, using the pathname and options given in the arguments. In the following example, it executes the first clause when the file is not found. When the file is found without error, it executes the second clause, which is the real reason for trying to open the file in the first place.

```
(with-open-file-case (x "f:>dla>foo.lisp" ':direction ':input)
  (fs:file-not-found (send x ':report error-output))
  (:no-error (stream-copy-until-eof x standard-output)))
```

Any errors other than **file-not-found** (for example, access violations or an unresponsive host) cause an error to be signalled normally.

with-open-file-case-if *cond* (*var pathname . options*) &body *clauses* *Special Form*

Opens a file, binding the input stream to *var*, using *pathname* and *options* given in the arguments. All clauses are evaluated, but the error handling for the body is performed only if the predicate specified by *cond* returns **t**.

Any errors other than **file-not-found** (for example, access violations or an unresponsive host) cause an error to be signalled normally.

with-open-stream (*stream-variable construction-form*) &body *body* *Special Form*

Like **with-open-file** except that you specify a form whose value is the stream, rather than arguments to **open**. This is used with nonfile streams. See the special form **with-open-file**, page 135.

with-open-stream-case (*var construction-form*) &body *clauses* *Special Form*

Opens a stream and binds it to *var*, using *construction-form* to create it. It then executes whichever clause is appropriate, given the condition that resulted from the attempt to create the stream. Refer to the example shown for **with-open-file-case**.

with-open-stream-case-if *cond* (*var construction-form*) &body *clauses* *Special Form*

Opens a stream and binds it to *var*, using *construction-form* to create it. All clauses are evaluated, but the error handling for the body is performed only if the predicate specified by *cond* returns **t**.

with-standard-io-environment &body *body* *Special Form*

All output in *body* is printed with ***package***, ***readtable***, and other variables bound to consistent values. This is useful when you wish to write some data the you will use **read** to retrieve later. This is a custom environment that you create, passing all variables and values that are important before *body.*

with-standard-io-environment inhibits the effect of **#.** while reading. This prevents other forms being read and used as trojan horses. This can be inhibited by rebinding **si:*suppress-read-eval** to **nil**.

with-input-from-string (*stream string* &key *index* (*start* **0**) *end*) &body *body* *Special Form*

body is executed as an explicit **progn** with the variable *stream* bound to a character input stream that supplies successive characters from the value of the form *string*. **with-input-from-string** returns the results from the last form of the body.

The input stream is automatically closed on exit from the **with-input-from-string** form, no matter whether the exit is normal or abnormal. The stream should be regarded as having dynamic extent. The following keywords can be used:

keyword	**value**
:index	The form after the **:index** keyword should be a place acceptable to **setf**. If the form is exited normally, then the place will have stored into it the index into *string* indicating the first character not read, or the length of the string if all characters were used. The place is not updated as reading progresses, but only at the end of the operation.
:start	An argument indicating the beginning of a substring of *string* to be used. **:start** defaults to 0.
:end	An argument indicating the end of a substring of *string* to be used. **:end** defaults to the length of the string.

Examples:

```
(values (with-input-from-string
          (stream "A long boring string" :index i)
          (read stream)) i) => A and 2

(values (with-input-from-string
          (stream "A long boring string" :index i :start 2)
          (read stream)) i) => LONG and 7

(values (with-input-from-string
          (stream "A long boring string" :index i :start 9 :end 12)
          (read stream)) i) => RIN and 12
```

zl:with-input-from-string (*var string* &optional *index limit*) &body *body* *Special Form*

The form:

(with-input-from-string (*var string*)
    body)

evaluates the forms in *body* with the variable *var* bound to a stream that reads characters from the string which is the value of the form *string*. The value of the special form is the value of the last form in its body.

The stream is a function that only works inside the **with-input-from-string** special form, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-input-from-string** special forms and use both streams since the special-variable bindings associated with the streams conflict. It is done this way to avoid any allocation of memory.

After *string* you can optionally specify two additional "arguments". The first is *index*:

(with-input-from-string (*var string index*)
    body)

uses *index* as the starting index into the string, and sets *index* to the index of the first character not read when **with-input-from-string** returns. If the whole string is read, it is set to the length of the string. Since *index* is updated it cannot be a general expression; it must be a variable or a **setf**able reference. The *index* is not updated in the event of an abnormal exit from the body, such as a **throw**. The value of *index* is not updated until **with-input-from-string** returns, so you cannot use its value within the body to see how far the reading has proceeded.

Use of the *index* feature prevents multiple values from being returned out of the body, currently.

(with-input-from-string (*var string index limit*)
    body)

uses the value of the form *limit*, if the value is not **nil**, in place of the length of the string. If you want to specify a *limit* but not an *index*, write **nil** for *index*. Examples:

```
(setq i 0) => 0
(values (zl:with-input-from-string
           (stream "A long boring string" i)
           (read stream)) i) => A and 2

(values (zl:with-input-from-string
           (stream "A long boring string" i)
           (read stream)) i) => LONG and 7
```

```
(values (zl:with-input-from-string
          (stream "A long boring string" i 12)
          (read stream)) i) => BORIN and 12
```

with-output-to-string (*stream* &optional *string* &key *index*) &body *body* *Special Form*

body is executed as an explicit **progn** with the variable *stream* bound to a character output stream that saves characters in *string*. If *string* is not specified, **with-output-to-string** returns the results from the last form of the body as a string.

If *string* is specified, it must be a string with a fill pointer. The output is incrementally appended to the string, as if using **vector-push-extend** if the string is adjustable, and as if using **vector-push** otherwise. In this case, **with-output-to-string** returns the results from the last form of the body.

The output stream is automatically closed on exit from the **with-output-to-string** form, no matter whether the exit is normal or abnormal. The stream should be regarded as having dynamic extent.

The form after the **:index** keyword should be a place acceptable to **setf**. If the form is exited normally, then the place will have stored into it the index into *string* indicating the first character not read, or the length of the string if all characters were used. The place is not updated as reading progresses, but only at the end of the operation.

Examples:

```
(setq string (make-array 2 :element-type 'string-char
                          :fill-pointer t)) => ..
(values (with-output-to-string (stream nil :index i)
          (write-string "a happy day" stream :start 2 :end 7))
         string i) => "happy" and .. and 17

(values (with-output-to-string (stream string :index i)
          (write-string "a happy day" stream :start 2 :end 7))
         string i) => "a happy day" and .. and 22
```

zl:with-output-to-string (*var* &optional (*string* **nil** *string-p*) *index*) &body *body* *Special Form*

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream that saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

appends its output to the string that is the value of the form *string*. (This is like the **string-nconc** function). The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have an array-leader; element 0 of the array-leader is used as the fill-pointer. If *string* is too small to contain all the output, **zl:adjust-array-size** is used to make it bigger.

If characters with font information are output, *string* must be of type **sys:art-fat-string**. See the section "**sys:art-fat-string** Array Type" in *Symbolics Common Lisp: Language Concepts*.

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or **setf**able reference that contains the index of the next character to be stored into. It must be initialized outside the **with-output-to-string** and is updated upon normal exit. The value of *index* is not updated until **with-output-to-string** returns, so you cannot use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if it does have one it is updated.

The stream is a "downward closure" simulated with special variables, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-output-to-string** special forms and use both streams since the special-variable bindings associated with the streams conflict. It is done this way to avoid any allocation of memory. Examples:

```
(setq string (zl:make-array 2 :type 'zl:art-string :fill-pointer 2)) => ··
(setq i 0) => 0
(values (zl:with-output-to-string (stream nil i)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "happy" and ·· and 0


(values (zl:with-output-to-string (stream string i)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "a happy day" and "ha" and 5
```

```
(values (zl:with-output-to-string (stream string i)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "a happy day" and "ha" and 10
(values (zl:with-output-to-string (stream string)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "a happy day" and "hahappy" and 10
```

sys:with-open-file-search (*stream-variable* **Special Form**
(*operation defaults auto-retry*) (*type-list-function pathname* **.** *type-list-args*) **.**
open-options) *body...*

Performs a **with-open-file**, searching for a file with one of the types in a list of file types. **load** uses this special form when not given a specific file type to search first for a binary file and then for a source file.

The body is evaluated with *stream-variable* bound to a stream that reads or writes the file. *open-options* are alternating keywords and values to be passed to **open**.

type-list-function should be a function whose first argument is *pathname* and whose remaining arguments are *type-list-args*. The function should return two values: a list of file types to be searched, in order of preference, and a base pathname to be merged with the types and *defaults* in searching for the file. *defaults* can be a pathname or a defaults alist; if omitted, the defaults come from **fs:*default-pathname-defaults***. The special form uses **fs:merge-pathname-defaults** for merging.

If no file is found with any of the types in the list of types, **fs:multiple-file-not-found** is signalled. *operation* is the name of the operation that failed; usually this is the name of the function that contains the **sys:with-open-file-search** form. If *auto-retry* is not **nil** and the condition is not handled, the user is prompted for a new pathname.

open *pathname* &rest *options* *Function*

Returns a stream that is connected to the specified file. The **open** function only creates streams for *files*; streams for other devices are created by other functions. If an error occurs, such as file not found, the user is asked to supply an alternate pathname, unless this is overridden by *options*.

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

See the special form **unwind-protect** in *Symbolics Common Lisp: Language Dictionary.*
For example:

```
(defun bliss-compile (file)
  (setq file (fs:parse-pathname file))
  (with-open-file (str "comet:usrd$:[mydir]tempfile.com"
                       ':direction ':output
                       ':characters t
                       ':submit t)
    (send str ':line-out (format nil "$ BLISS ~A" (send file ':string-for-host)))
```

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. The file-opening options control whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, and so on.

The following option keywords are recognized. Unless otherwise noted, they are supported generically. Additional keywords can be implemented by particular file system hosts.

:byte-size The possible values are **nil** (the default), a number in the range 1 to 16 inclusive, which is the number of bits per byte, and **:default**, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, **nil** selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, **nil** selects the default byte size of 16 bits. The preferred way to specify the byte-size for files is to use the **:element-type** keyword.

:characters This option specifies whether the objects contained in the file are characters or fixnums. The preferred way to specify character files is to use the **:element-type** keyword.

Value	*Meaning*
t	Specifies that the file contains character objects. This is the default.
nil	Specifies that the file is a binary file.
:default	On output, **:default** is always **t**, as character files are created by default. On input, **:default** specifies that the

file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:deleted	The default is **nil**. If **t** is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.
:direct	The default is **nil**. **t** specifies a direct access stream. See the section "Direct Access File Streams", page 13.
:direction	The **:direction** option allows the following values:

:input	The file is being opened for input. This is the default.
:output	The file is being opened for output.
:block	This is a special case of **:output** that is used for the FEP File System.
:io	The file is being opened for intermixed input and output. Bidirectionality is supported only if the stream is to be a direct stream, that is, **:direct t** is given as well. See the section "Direct Access File Streams", page 13.
:probe	A "probe" opening; no data are to be transferred, and the file is being opened to determine whether the file exists, or to gain access to or change its properties. Returns the truename of the object at the end of a link or chain of links. If the value of **:direction** is **:probe** and the value of **:error** is **nil**, then **open** will return the error object instead of **nil**. If the value of **:if-does-not-exist** is **nil**, the error object will still be returned.
:probe-link	The same as **:probe** except that links are not chased. Returns the truename

	of the object named, even if it is a link.
:probe-directory	The pathname is being opened to find out about the existence of its *directory* component. Otherwise, the semantics are the same as **:probe**. If the directory is not found, a file lookup error is signalled.
nil	This is the same as probe. No data are transferred, and the file is being opened only to gain access to or change its properties. If the value of **:direction** is **nil** and the value of **:error** is **nil**, then **open** will return the error object instead of **nil**. If the value of **:if-does-not-exist** is **nil**, the error object will still be returned.

:element-type

This argument specifies the type of Lisp object transferred by the stream. Anything that can be recognized as being a finite subtype of **character** or **integer** is acceptable. In particular, the following types are recognized:

string-char	The object being transferred is a string-character. The functions **read-char** and/or **write-char** can be used on the stream. This is the default.
character	The object being transferred is any character, not just a string-character. The functions **read-char** and/or **write-char** can be used on the stream.
(unsigned-byte *n***)**	The object being transferred is an unsigned byte (a non-negative integer) of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
unsigned-byte	The object being transferred is an unsigned byte (a non-negative integer) whose size is determined by the file

	system. The functions **read-byte** and/or **write-byte** can be used on the stream.
(signed-byte *n***)**	The object being transferred is a signed byte of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
signed-byte	The object being transferred is a signed byte whose size is determined by the file system. The functions **read-byte** and/or **write-byte** can be used on the stream.
bit	The object being transferred is a bit (values 0 and 1). The functions **read-byte** and/or **write-byte** can be used on the stream.
(mod *n***)**	The object being transferred is a non-negative integer less than *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
:default	On output, **:default** is always **character**, as character files are created by default. On input, **:default** specifies that the file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:error This option controls what happens when any **fs:file-operation-failure** condition is signalled. **t** is the recommended value for this option. The others have been provided for compatibility with previous systems to aid in converting programs. See the section "File-System Errors" in *Symbolics Common Lisp: Language Concepts.*

The option has three possible values:

Value	*Meaning*
t	Signals the error normally. **t** is both

	the default and the recommended value.
nil	Returns the error object. If the value of either **:if-exists** or **:if-does-not-exist** is **nil**, the error object is still returned.
:reprompt	Reprompts the user for another file name and tries **open** again. When you use this option, remember that the **:pathname** message sent to the stream finds out what file name was really opened. The alternative to **:reprompt** is to use **:error t** and set up a condition handler for **fs:file-operation-failure** that explains the condition and prompts the user.

:estimated-length The value of the **:estimated-length** option can be **nil** (the default), which means there is no estimated length, or a number of bytes indicating the estimated length of a file to be written. Some file systems use this to optimize disk allocation.

:if-does-not-exist Specifies the action to be taken if the file does not already exist. The following values are allowed:

:error	Signals an error. This is the default if the **:direction** is **:input**, **:probe**, or any of the **:probe**-like modes, or if the **:if-exists** argument is **:overwrite**, **:truncate**, or **:append**.
:create	Creates an empty file with the specified name, and then proceeds as if it had already existed. This is the default if the **:direction** is **:output** and the **:if-exists** argument is anything but **:overwrite**, **:truncate**, or **:append**.
nil	Does not create a file or even a stream. Instead, simply returns **nil** to indicate failure. This is overridden when the value of **:direction** is either **nil** or **:probe** and the value of **:error** is **nil**. In this case, the error object is returned instead of **nil**.

:if-exists	Specifies the action to be taken if the **:direction** is **:output** and a file of the specified name already exists. If the direction is **:input** or **:probe** (or any of the **:probe**-like directions), this argument is ignored.

The following values are allowed:

:error	Signals an error. This is the default when the version component of the filename is not either **:newest** or **:unspecific**.
:new-version	Creates a new file with the same file name but a larger version number. This is the default when the version component of the filename is either **:newest** or **:unspecific**. File systems without version numbers can choose to implement this by effectively treating it as **:supersede**.
:rename	Renames the existing file to some other name, and then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.
:rename-and-delete	Renames the existing file to some other name and then deletes it (but does not expunge it, on those systems that distinguish deletion from expunging). Then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.
:overwrite	The existing file is used, and output operations on the stream destructively modify the file. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.
:truncate	The existing file is used, and output

	operations on the stream destructively modify the file. The file pointer is initially positioned at the beginning of the file; at that time, the file is truncated to length zero, and disk storage occupied by it is freed.
:append	The existing file is used, and output operations on the stream modify the file. The file pointer is initially positioned at the current end of the file.
:supersede	Supersedes the existing file. If possible, the file system does not destroy the old file until the new stream is closed, against the possibility that the stream will be closed in "abort" mode. This differs from **:new-version** in that **:supersede** creates a new file with the same name as the old one, rather than a file name with a higher version number.
nil	Does not create a file or even a stream. Instead, simply returns **nil** to indicate failure. This is overridden when the value of **:direction** is either **nil** or **:probe** and the value of **:error** is **nil**. In this case, the error object is returned instead of **nil**.

:preserve-dates	The default is **nil**. If **t** is specified, the file's reference and modification dates are not updated.
:raw	The value can be **nil** (the default) or **t**, which disables all character set translation in ASCII files.
:submit	This is an option to **open** used to get batch jobs. Currently, this is implemented only for VAX/VMS. When the file you are writing is closed, the file is submitted as a batch job by using this option.
:super-image	The value can be **nil** (the default), or **t** which disables the special treatment of Rubout in ASCII files. Normally Rubout is an escape that causes the following character

to be interpreted specially, allowing all characters from 0 through 376 to be stored. This applies to PDP-10 file servers only.

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

close *stream* &key *abort* *Function*

stream is closed and no further input or output operations can be performed on it. However, certain inquiry operations can still be performed. It is permissible to close an already closed stream.

If the **:abort** parameter is non-**nil** (the default is **nil**), it indicates an abnormal termination of the use of the stream. An attempt is made to clean up any side effects of having created the stream. For example, if the stream performs output to a file that was newly created when the stream was created, then if possible the file is deleted and any previously existing file is not superseded.

zl:close *stream* &optional *abortp* *Function*

Sends the **:close** message to *stream*.

The *abortp* argument is normally not supplied. If it is **t**, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly created file is deleted, as if it were never opened in the first place. Any previously existing file with the same name remains, undisturbed.

zl:renamef *file new-name* &optional *(error-p* **t***)* *Function*

Renames one file. The Rename File (m-X) command in the editor uses this function.

file can be a pathname, a string, or a stream that is open to a file. The specified file is renamed to *new-name* (a pathname or string). If *error-p* is **t**, when an error occurs it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error object is returned; otherwise the three values described below are returned.

file must refer to a unique file; it cannot contain any wild components. *new-name* can contain wild components, which are eliminated after merging the defaults by means of **:translate-wild-pathname**. **zl:renamef** first attempts to open *file*. When that has happened successfully, it parses *new-name* and merges it (using **fs:merge-pathnames**) against the *link-opaque truename* of *file* and version of **:newest**. This has the following result for version numbers.

Source	*Target*	*Result*
>foo>a.b.newest	>bar>	Retains the version number
>foo>a.b.newest	>bar>x	Makes a new version of >bar>x.b

The defaults for *new-name* come from the link-opaque truename of *file*. For systems without links, this is indistinguishable from the truename. Otherwise, the link-opaque truename depends on whether *file* contains an **:oldest** or **:newest** version. If it does not and if it is fully defaulted, with no wild components, the pathname is its own link-opaque truename. If a pathname *x* contains an **:oldest** or **:newest** version, the link-opaque truename is the pathname of the file or link that corresponds to *x*, with the version number filled in. For example, renaming the LMFS file `>a>p1.lisp` to `>b>` results in `>b>p1.lisp`, with the version of `>a>p1.lisp.newest` inherited. This is so whether `>a>p1.lisp.newest` is a real file, a link, or a rename-through link.

zl:renamef returns three values:

1. The pathname produced by merging and defaulting *new-name*. This is the attempted result of the renaming.

2. The pathname of the object that was actually renamed. This might not be the same as *file*. For example, *file* might have an **:oldest** or **:newest** version, or LMFS rename-through links might be involved. This pathname never has an **:oldest** or **:newest** version.

3. The actual pathname that resulted from the renaming. This might not be the same as *new-name*. For example, *new-name* might have an **:oldest** or **:newest** version, or LMFS create-through links might be involved.

The **:rename** message to streams and pathnames returns the second and third of these values.

Examples:

This example is as simple as possible. Using LMFS, on host `johnny`, with no links involved:

```
(renamef "johnny:>a>foo.lisp" "bar") =>
#<LMFS-PATHNAME "johnny:>a>bar.lisp">
#<LMFS-PATHNAME "johnny:>a>foo.lisp.17">
#<LMFS-PATHNAME "johnny:>a>bar.lisp.1">
```

This example is as complex as possible. Using LMFS, on host `eddie`, with links

```
>abel>moe.lisp.4 => >baker>larry.lisp (rename-through) (latest)
>baker>larry.lisp.4 =>
  >charlie>sam.lisp.19 (not rename- or create-through) (latest)
>david>jerry.lisp.5 => >earl>ted.lisp (create-through) (latest)
```

```
(renamef "eddie:>abel>moe.lisp.4" "eddie:>david>jerry") =>
#<LMFS-PATHNAME "eddie:>david>jerry.lisp">
#<LMFS-PATHNAME "eddie:>baker>larry.lisp.4">
#<LMFS-PATHNAME "eddie:>earl>ted.lisp.1">
```

zl:deletef *file* &optional *(error-p* **t**) *Function*

Deletes the specified file. *file* can be a pathname or a stream that is open to a file. If *error-p* is **t**, then if an error occurs it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error object is returned; otherwise **t** is returned.

undelete-file *pathname* &optional (*error-p* **t**) *Function*

Undeletes the specified file. *file* can be a pathname or a stream that is open to a file. If *error-p* is **t** and an error occurs, it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error object is returned; otherwise **t** is returned. **undelete-file** is like **zl:deletef** except that it undeletes the file instead of deleting it. **undelete-file** is meaningful only for files in file systems that support undeletion, such as TOPS-20 and the Lisp Machine File System.

zl:undeletef *file* &optional *(error-p* **t**) *Function*

Undeletes the specified file. *file* can be a pathname or a stream that is open to a file. If *error-p* is **t** and an error occurs, it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error object is returned; otherwise **t** is returned. **zl:undeletef** is like **zl:deletef** except that it undeletes the file instead of deleting it. **zl:undeletef** is meaningful only for files in file systems that support undeletion, such as TOPS-20 and the Lisp Machine File System.

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

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. If *error-p* is **t** (the default) a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned.

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

Some of the properties of a file can be changed, such as its creation date or its author. The properties that can be changed depend on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list**. See the function **fs:directory-list**, page 161.

fs:change-file-properties changes one or more properties of a file. *pathname* names the file. The *properties* arguments are alternating keywords and values. If the *error-p* argument is **t**, a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned. If no error occurs, **fs:change-file-properties** returns **t**.

zl:viewf *pathname* &optional *(stream* **zl:standard-output***)* *leader* *Function*

Prints the file named by *pathname* onto the *stream*. (The optional third argument is passed as the *leader* argument to **stream-copy-until-eof**.) The name **zl:viewf** is analogous with **zl:deletef**, **zl:renamef**, and so on. Note: **zl:viewf** should not be used for copying files; its output is not the same as the contents of the file (for example, it does a **:fresh-line** operation on the stream before printing the file).

zl:copyf *from-path to-path* &key (*characters* **':default**) (*byte-size* **nil**) (*copy-creation-date* **t**) (*copy-author* **t**) (*report-stream* **nil**) (*create-directories* **':query**) *Function*

Copies one file to another. Copy File (m-X) in the editor uses this function.

from-path and *to-path* are the source and destination pathnames, which can be file specifications. *from-path* must refer to a unique file; it cannot contain any wild components. *to-path* can contain wild components, which are eliminated after merging the defaults by means of **:translate-wild-pathname**. **zl:copyf** first attempts to open *from-path*. When that has happened successfully, it parses *to-path* and merges it (using **fs:merge-pathnames**) against the *link-opaque truename* of *from-path* and version of **:newest**. The output file specified by *to-path* is opened with **:if-exists :supersede**. The processing of *to-path* has the following result for version numbers.

Source	*Target*	*Result*
>foo>a.b.newest	>bar>	Retains the version number
>foo>a.b.newest	>bar>x	Makes a new version of >bar>x.b

The defaults for *to-path* come from the *link-opaque truename* of *from-path*. For systems without links, this is indistinguishable from the truename. Otherwise, the link-opaque truename depends on whether *from-path* contains an **:oldest** or **:newest** version. If it does not and if it is fully defaulted, with no wild components, the pathname is its own link-opaque truename. If a pathname *x* contains an **:oldest** or **:newest** version, the link-opaque truename is the pathname of the file or link that corresponds to *x*, with the version number filled in. For example, copying the LMFS file `>a>p1.lisp` to >b> results in `>b>p1.lisp`, with the version of

`>a>p1.lisp.newest` inherited. This is so whether `>a>p1.lisp.newest` is a real file, a link, or a rename-through link.

By default, **zl:copyf** copies the creation date and author of the file.

Following is a description of the other options:

:characters Possible values:

:default	**zl:copyf** decides whether this is a binary or character transfer according to the canonical type of *from-path*. You do not need to supply this argument for standard file types. For types that are not known canonical types, it opens *from-path* in **:default** mode. In that case, the server for the file system containing *from-path* makes the character-or-binary decision.
t	Specifies that the transfer must be in character mode.
nil	Specifies that the transfer must be binary mode (in this case, you must supply *byte-size* if using a byte size other than 16).

:byte-size Specifies the byte size with which both files are opened for binary transfers. You must supply **:byte-size** when **:characters** is **nil** and the byte size is other than 16. Otherwise, **zl:copyf** determines the byte size from the file type for *from-path*. When *from-path* is a binary file with a known canonical type, it determines the byte size from the **:binary-file-byte-size** property of the type. When the file does not have a known type, it requests the byte size for *from-path* from the file server. When the server for the file system containing *from-path* cannot supply the byte size, it assumes that the byte size is 16.

:report-stream When **:report-stream** is **nil** (the default), the copying takes place with no messages. Otherwise, the value must be a stream for reporting the start and successful completion of the copying. The completion message contains the truename of *to-path*.

:create-directories

Determines whether directories should be created, if needed, for the target of the copy. Permissible values are as follows:

t	Try to create the target directory of the copy and all superiors. Report directory creation to **zl:standard-output**.
nil	Do not try to create directories. If the directory does not exist, handle this condition like any other error.
:query	If the directory does not exist, ask whether or not to create it. This is the default.

zl:probef *pathname* *Function*

Returns **nil** if there is no file named *pathname*, or signals an error if anything else goes wrong (such as **sys:host-not-responding**). Otherwise, **zl:probef** returns a pathname that is the truename of the file, which can be different from *pathname* because of file links, version numbers, and so on.

fs:close-all-files *Function*

Closes all open files. This is useful when a program has run wild opening files and not closing them. It closes all the files in **:abort** mode, which means that files open for output will be deleted. Using this function is dangerous, because you might close files out from under various programs such as Zmacs and Zmail; only use it if you have to and if you feel that you know what you're doing.

fs:*remember-passwords* *Variable*

If not **nil**, causes the first password for each file access path to be remembered. This suppresses prompting for passwords on subsequent attempts by the same user to use that access path. The default value is **nil**.

Note that if you set this variable in an init file, your first login password, typed before the init file is loaded, is not remembered.

Caution: Remembered passwords are accessible. Even after you log out the remembered password for each access path is accessible. If password security is important, you probably should not set this variable to a non-**nil** value.

5.1.1 Loading Files

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

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

These functions are for loading single files. There is a system for keeping track of programs that consist of more than one file: See the section "Maintaining Large Programs" in *Program Development Utilities.*

zl:load *pathname* &optional *pkg nonexistent-ok-flag dont-set-default-p no-msg-p* *Function*

Loads the file named by *pathname* into the Lisp environment. The file can be either a Lisp source file or a binary file. If the *pathname* specifies the type, it is used; otherwise, **zl:load** looks first for a binary file, then for a Lisp file. Normally, the file is read into its "home" package, but *pkg* can be supplied to specify the package. *pkg* can be either a package or the name of a package as a string or a symbol. If *pkg* is not specified, **zl:load** prints a message saying what package the file is being loaded into.

nonexistent-ok controls the action of **zl:load** if none of the files is found. If it is **nil** (the default), you are prompted for a new file unless the corresponding condition (**fs:multiple-file-not-found**) is handled. If it is not **nil**, it is the returned value if the file is not found. Other reasons for not finding the file, such as the host being down or the directory not existing, are signalled as different errors. For example, **zl:load** fails when the host is down even when you specified the *nonexistent-ok* argument.

pathname can be anything acceptable to **fs:parse-pathname**. See the section "Naming of Files", page 51. *pathname* is defaulted from **fs:load-pathname-defaults**, which is the set of defaults used by **zl:load** and similar functions. See the variable **fs:load-pathname-defaults**, page 75. Normally **zl:load** updates the pathname defaults from *pathname*, but if *dont-set-default* is specified this is suppressed.

If an ITS *pathname* contains an FN1 but no FN2, **zl:load** first looks for the file with an FN2 of BIN, then it looks for an FN2 of >. For non-ITS file systems, this generalizes to: if *pathname* specifies a type and/or a version, **zl:load** loads that file. Otherwise it first looks for a binary file, then a Lisp file, in both cases looking for the newest version.

If the value of *no-msg-p* is **t** (it defaults to **nil**), then **zl:load** does not print out the message that it usually prints (that is, the message that tells you that a certain file is being loaded into a certain package).

zl:readfile *pathname* &optional *pkg* *no-msg-p* *Function*

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

5.1.2 File Attribute Lists

Any text file can contain an *attribute list* that specifies several attributes of the file. The functions that load files, the compiler, and the editor look at this attribute list. File attribute lists are especially useful in program source files, that is, a file that is intended to be loaded (or compiled and then loaded).

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

```
; -*- Mode:Lisp; Syntax:Zetalisp; Package:User; Base:10 -*-
```

The semicolon makes this line look like a comment rather than a Lisp expression. This example defines four attributes: mode, syntax, package, and base.

The term *attribute list* applies not only to the -*- line in character files, but also to an analogous data structure in compiled files. For example, in both cases the attribute list tells **zl:load** what package to load the file into.

An attribute name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. An attribute value can be such a name, or a decimal number, or several such items separated by commas. Spaces can be used freely to separate tokens. Upper and lowercase letters are not distinguished. There is no quoting convention for special characters such as colons and semicolons. File attribute lists are different from Lisp property lists; attribute lists correspond to the text inside a file, while file properties are characteristics of the file itself, such as the creation date.

The file attribute list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret.

Symbolics Common Lisp has a parser for file attribute lists that creates some Lisp data structure that corresponds to the file attribute list. When a file attribute list is read in and given to the parser (the **fs:read-attribute-list** function), it is converted into Lisp objects as follows: Attribute names are interpreted as Lisp symbols, and interned on the keyword package. Numbers are interpreted as Lisp fixnums, and are read in decimal. If an attribute value contains any commas, then the commas separate several expressions that are formed into a list.

When a file is edited, loaded, or compiled, its file attribute list is read in and the attributes are stored on the attribute list of the generic pathname for that file, where they can be retrieved with the **:get** and **:plist** messages. See the section "Generic Pathnames", page 75. So, to examine the attributes of a file, you usually use messages to a pathname object that represents the generic pathname of a file. Note that there other attributes there, too. The function **fs:read-attribute-list** reads the file attribute list of a file and sets up the attributes on the generic pathname; editing, loading, or compiling a file calls this function, but you can call it yourself if you want to examine the attributes of an arbitrary file.

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

The following are some of the attribute names allowed and what they mean.

Mode	The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are Lisp and Text.
Package	The name of the package into which the file is to be loaded. See the section "The Need for Packages" in *Symbolics Common Lisp: Language Concepts.*
Base	The number base in which the file is written. This affects both **zl:ibase** and **zl:base**, since it is confusing to have different input and output bases. The most common values are 8 and 10. If a file has no Base attribute, the value of the Syntax attribute affects the default of Base. See the Syntax attribute below.
Syntax	The syntax of the programs contained in the file can be either Zetalisp or Common-Lisp. If a file has no Syntax attribute, the value of the Base attribute affects the default of Syntax. • If there is a Base attribute, but no Syntax attribute, the syntax is assumed to be Zetalisp. • If there is a Syntax: Common-Lisp attribute, and no Base attribute, the base is assumed to be 10. • If there is neither a Base nor a Syntax attribute, Base is assumed to be the default base (10) and the syntax is assumed to be Zetalisp. Furthermore, a warning is issued (upon beginning an editing session on the file) to the effect that there is neither a Syntax nor a Base attribute. You should edit your program accordingly.
Lowercase	If the attribute value is not **nil**, the file is written in lowercase

letters and the editor does not translate to uppercase. (The editor does not translate to uppercase by default unless the user selects "Electric Shift Lock" mode.)

Fonts	The attribute value is a list of font names, separated by commas. The editor uses this for files that are written in more than one font.
Backspace	If the attribute value is not **nil**, the file can contain backspaces that cause characters to overprint on each other. The default is to disallow overprinting and display backspaces the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.
Patch-File	If the attribute value is not **nil**, the file is a "patch file". When it is loaded, the system does not complain about function redefinitions. Furthermore, the remembered source file names for functions defined in this file are changed to this file, but are left as whatever file the function came from originally. In a patch file, the **defvar** special-form turns into **zl:defconst**; thus patch files always reinitialize variables.

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

The function **fs:pathname-attribute-list** is generally the most useful function for obtaining a file's attributes.

fs:pathname-attribute-list *pathname* *Function*

Returns the attribute list for a file designated by *pathname.*

fs:read-attribute-list *pathname stream* *Function*

Parses file attribute lists. *pathname* should be a pathname object (*not* a string or namelist, but an actual pathname); usually it is a generic pathname. See the section "Generic Pathnames", page 75.

stream should be a stream that has been opened and is pointing to the beginning of the file whose file attribute list is to be parsed. This function reads from the stream until it gets the file attribute list, parses it, puts corresponding attributes onto the attribute list of *pathname*, and finally sets the stream back to the beginning of the file by using the **:set-pointer** file stream operation. See the message **:set-pointer**, page 43.

The obsolete name of this function is **fs:file-read-property-list.**

Programs in Symbolics Common Lisp generally react to the presence of attributes on a file's file attribute list by examining the attribute list in the generic pathname's property list. However, file attributes can also cause special variables to be bound whenever Lisp expressions are being read from the file–when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base attributes work. You can also deal with attributes this way, by using the following function.

fs:file-attribute-bindings *pathname* *Function*

Examines the property list of *pathname* and finds all those property names that have file-attribute bindings. Its obsolete name is **fs:file-property-bindings.**

Each such pathname-property name specifies a set of variables to bind and a set of values to which to bind them. This function returns two values: a list of all the variables, and a list of all the corresponding values. Usually you call this function on a generic pathname whose attribute list has been parsed with **fs:read-attribute-list.** Then you use the two returned values as the first two subforms to a **zl:progv** special form. Inside the body of the **zl:progv** the specified bindings will be in effect.

Usually, *pathname* is a generic pathname. It can also be a locative, in which case it is interpreted to be the property list itself.

Of the standard names, the following ones have file-attribute bindings, with the following effects:

- **zl:package** binds the variable **zl:package** to the package. See the variable **zl:package** in *Symbolics Common Lisp: Language Dictionary.*

- **zl:base** binds the variables **zl:base** and **zl:ibase** to the value. See the variable **zl:base** in *Symbolics Common Lisp: Language Dictionary.* See the variable **zl:ibase** in *Symbolics Common Lisp: Language Dictionary.*

- **fs:patch-file** binds **fs:this-is-a-patch-file** to the value.

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

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

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

Finally, the function **sys:dump-forms-to-file** offers, among other things, the option of manipulating the attribute list of a binary file. See the section "Putting Data in Compiled Code Files", page 223.

For example, the following form converts a Lisp file to a binary file, without compiling. The attribute list is obtained from the input stream and cached in the generic pathname. The function **fs:file-attribute-bindings** obtains the list of variables to bind from the generic pathname; these bindings are necessary to ensure that the file is read in the right base, syntax, and package. The **zl:progv** actually accomplishes the binding of the variables.

```
(defun binify-file-internal (input-file output-file)
  (setq input-file (fs:parse-pathname input-file))
  (with-open-file (input input-file :direction :input :characters t)
    (let* ((generic-pathname (send input-file :generic-pathname))
           (attribute-list (fs:read-attribute-list generic-pathname input)))
      (multiple-value-bind (variables-list values-list)
          (fs:file-attribute-bindings generic-pathname)
        (progv variables-list values-list
          (loop with eof-val = (ncons 'eof)
                for form = (read input eof-val)
                while (neq form eof-val)
                collect form into forms
                finally
                  (sys:dump-forms-to-file output-file forms
                                          attribute-list))))))
```

5.2 Accessing Directories

To understand the functions in this section, it is imperative that you read some other documentation. See the section "Naming of Files", page 51.

5.2.1 Functions for Accessing Directories

fs:directory-list *pathname* &rest *options* *Function*

Finds all the files that match *pathname* and returns a freshly consed list with one element for each file. *options* are a list of keywords, with no values, that modify the operation. Each element in the returned list is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a "disembodied" property list and **get** can be used to access the file's properties. The car of one element is **nil**; the properties in this element are properties of the file system as a whole rather than of a specific file.

The matching is done using both host-independent and host-dependent conventions. Any component of *pathname* that is **:wild** matches anything; all files that match the remaining components of *pathname* are listed regardless of their values for the wild component. In addition, there is host-dependent matching. Typically, this uses the asterisk character (*) as a wild-card character. A pathname component that consists of just a * matches any value of that component (the same as **:wild**). *, appearing in a pathname component that contains other characters, matches any character (on ITS) or any string of characters (on TOPS-20, LMFS, UNIX, and Multics) in the starred positions and requires the specified characters otherwise. Other hosts follow similar but not necessarily identical conventions.

The *options* are keywords that modify the operation. These keywords *do not* take values. The following options are currently defined:

:noerror	If a file-system error (for example, no such directory) occurs during the operation, an error is normally signalled and the user is asked to supply a new pathname. However, if **:noerror** is specified and an error occurs, an error object describing the error is returned as the result of **fs:directory-list**. This is identical to the **:noerror** option to **open**.
:deleted	This is for file servers with soft deletion, such as TOPS-20, LMFS, and FEP. It specifies that deleted (but not yet expunged) files are to be included in the directory listing. Normally, they are not included.
:no-extra-info	This results in only enough information for listing the directory as in Dired.
:sorted	This causes the directory to be sorted so that at least multiple versions of a file are consecutive in increasing version number.

The properties that might appear in the list of property lists returned by **fs:directory-list** are host-dependent to some extent. The following properties are defined for most file servers.

:length-in-bytes	The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file and binary bytes for a binary file).
:byte-size	The number of bits in a byte.
:length-in-blocks	The length of the file in terms of the file system's unit of storage allocation.
:block-size	The number of bits in a block.
:creation-date	The date the file was created, as a universal time. This does not necessarily mean the time that the file itself was created, but rather, the time that the data in it were created. This property corresponds to the concept of "modification date" on many systems. See the section "Dates and Times" in *Programming the User Interface, Volume B.*
:modification-date	The most recent time at which this file was modified, expressed in Universal Time. This is the same as the creation date if the file has been opened for appending. Operations such as renaming and property changing update this property, but do not update creation date. The dumper, for instance, is driven off this property. See the section "Dates and Times" in *Programming the User Interface, Volume B.*
:directory	A boolean. If **t**, the object in question is a directory, as opposed to a file or a link. This property can only be returned as **t** in a hierarchical file system.
:auto-expunge-interval	For directories, the time interval between automatic expungings of this directory. If, on a file system that supports this feature (such as TOPS-20 or LMFS), a directory is never automatically expunged, the value of the property will be **nil**. The time interval, when supplied, is expressed as a positive integer, in seconds.
:last-expunge-time	For directories, the date that the directory was last expunged. It is **nil** if the directory has never been expunged.

:reference-date	The most recent date that the file was used, as a universal time.
:author	The name of the person who created the data in the file, as a string.
:account	A string. Highly system-dependent in format.
:deleted	A boolean. **t** for a "deleted" file, in file systems supporting "soft deletion".
:dont-delete	A boolean. If it is **t**, an error results if an attempt is made to delete the file.
:dont-dump	A boolean. Suppresses backup dumping.
:dont-reap	A boolean. A flag used by directory maintenance tools.
:dumped	A boolean. **t** if and only if the file has been dumped to backup tape.
:generation-retention-count	A number that specifies how many versions of a file should be saved.
:link-to	A string. This is the target pathname of a link, as a string.
:offline	A boolean. **t** if the file has been moved to archival storage.
:physical-volume	A string. The volume on which the file is mounted.
:protection	A string. What protections have been set for the file.
:reader	A string. The last person to have read the file.
:settable-properties	A list of the properties that may be changed for the file using **fs:change-file-properties**.
:temporary	A boolean. **t** if the file is temporary.

fs:multiple-file-plists *filenames* &rest *options* *Function*

For each file in *filenames*, **fs:multiple-file-plists** returns a corresponding property list. For example:

```
(fs:multiple-file-plists
  (list "sys: doc; str; str1.sar" "sys: sys2; table.lisp")) =>
```

```
((#P"SYS:SYS2;TABLE.LISP.NEWEST" :TRUENAME
#P"Q:>rel-7>sys>sys2>table.lisp.43" :LENGTH 97047 :AUTHOR "Moon"
:BYTE-SIZE NIL :CREATION-DATE 2729141698)
(#P"SYS:DOC;STR;STR1.SAR.NEWEST" :TRUENAME
#P"Q:>rel-7>sys>doc>str>str1.sar.45" :LENGTH 15625 :AUTHOR "nancy"
:BYTE-SIZE NIL :CREATION-DATE 2728753545))
```

options should either be **:characters** or **:element-type**, and it specifies whether the objects contained in the file are characters or fixnums. The possible values for **:characters** are:

Value	*Meaning*
t	Specifies that the file contains character objects. This is the default.
nil	Specifies that the file is a binary file.
:default	On output, **:default** is always **t**, as character files are created by default. On input, **:default** specifies that the file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:element-type specifies the type of Lisp object transferred by the stream. Anything that can be recognized as being a finite subtype of **character** or **integer** is acceptable. In particular, the following types are recognized:

string-char	The object being transferred is a string-character. The functions **read-char** and/or **write-char** can be used on the stream. This is the default.
character	The object being transferred is any character, not just a string-character. The functions **read-char** and/or **write-char** can be used on the stream.
(unsigned-byte *n***)**	The object being transferred is an unsigned byte (a non-negative integer) of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
unsigned-byte	The object being transferred is an unsigned byte (a non-negative integer) whose size is determined by the file system. The functions **read-byte** and/or **write-byte** can be used on the stream.
(signed-byte *n***)**	The object being transferred is a signed byte of size *n*.

	The functions **read-byte** and/or **write-byte** can be used on the stream.
signed-byte	The object being transferred is a signed byte whose size is determined by the file system. The functions **read-byte** and/or **write-byte** can be used on the stream.
bit	The object being transferred is a bit (values 0 and 1). The functions **read-byte** and/or **write-byte** can be used on the stream.
(mod *n*)	The object being transferred is a non-negative integer less than *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
:default	On output, **:default** is always **character**, as character files are created by default. On input, **:default** specifies that the file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

The properties that might appear in the list of property lists returned by **fs:multiple-file-plists** are host-dependent to some extent. The following properties are defined for most file servers:

:truename	Returns the pathname of the file actually open on this stream. This can be different from what **:pathname** returns because of file links, logical devices, mapping of "newest" version to a particular version number, and so on.
:length	The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file and binary bytes for a binary file).
:author	The name of the person who created the data in the file, as a string.
:byte-size	The number of bits in a byte.
:creation-date	The date the file was created, as a universal time. This does not necessarily mean the time that the file itself was created, but rather, the time that the data in it were created. This property corresponds to the concept of "modification date" on many systems. See the section "Dates and Times" in *Programming the User Interface, Volume B.*

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

Some of the properties of a file can be changed, such as its creation date or its author. The properties that can be changed depend on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list**. See the function **fs:directory-list**, page 161.

fs:change-file-properties changes one or more properties of a file. *pathname* names the file. The *properties* arguments are alternating keywords and values. If the *error-p* argument is **t**, a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned. If no error occurs, **fs:change-file-properties** returns **t**.

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

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. If *error-p* is **t** (the default) a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned.

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

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

string is completed relative to a default pathname constructed from *defaults*, the host (if any) specified by *string*, *type*, and *version*, using the function **fs:default-pathname**. See the function **fs:default-pathname**, page 89. If *string* does not contain a colon, the host comes from *defaults*; otherwise the host name precedes the first colon in *string*.

options are keywords (without following values) that control how the completion will be performed. The following option keywords are allowed. Their meanings are explained more fully below.

:deleted	Look for files that have been deleted but not yet expunged. The default is to ignore such files.
:read or **:in**	The file is going to be read. This is the default. The name **:in** is obsolete and should not be used in new programs.
:write or **:print** or **:out**	The file is going to be written (that is, a new version is going to be created). The names **:print** and **:out** are obsolete and should not be used in new programs.

:old	Look only for files that already exist. This is the default. **:old** is not meaningful when **:write** is specified.
:new-ok	Allow either a file that already exists, or a file that does not yet exist. **:new-ok** is not meaningful when **:write** is specified. The **:new-ok** option is no longer used by any system software, because users found its effects (in the Zmacs command Find File (`c-X c-F`)) to be too confusing. It remains available, but programmers should consider this experience when deciding whether to use it.

The first value returned is always a string containing a file name; either the original string, or a new, more specific string. The second value returned indicates the status of the completion. It is non-**nil** if it was completely successful. The following values are possible:

:old	The string completed to the name of a file that exists.
:new	The string completed to the name of a file that could be created.
nil	The operation failed for one of the following reasons:

- The file is on a file system that does not support completion. The original string is returned unchanged.
- There is no possible completion. The original string is returned unchanged.
- There is more than one possible completion. The string is completed up to the first point of ambiguity.
- A directory name was completed. Completion was not successful because additional components to the right of this directory remain to be specified. The string is completed through the directory name and the delimiter that follows it.

Although completion is a host-dependent operation, the following guidelines are generally followed:

When a pathname component is left completely unspecified by *string*, it is generally taken from the default pathname. However, the name and type are defaulted in a special way described below and the version is not defaulted at all; it remains unspecified.

When a pathname component is specified by *string*, it can be recognized as an abbreviation and completed by replacing it with the expansion of the abbreviation. This usually occurs only in the rightmost specified component of *string*. All files that exist in a certain portion of the file system and match this component are considered. The portion of the file system is determined by the specified, defaulted, or completed components to the left of this component. A file's component *x* matches a specified component *y* if *x* consists of the characters in *y* followed by zero or more additional characters; in other words, *y* is a left substring of *x*. If no matching files are found, completion fails. If all matching files have the same component *x*, it is the completion. If there is more than one possible completion, that is, more than one distinct value of *x*, there is an ambiguity and completion fails unless one of the possible values of *x* is equal to *y*.

If completion of a component succeeds, the system attempts to complete any additional components to the right. If completion of a component fails, additional components to the right are not completed.

A blank component is generally treated the same as a missing component; for example, if the host is a LMFS, completion of the strings "foo" and "foo." deals with the type component in the same way. The strings are not completed identically; completion of "foo" attempts to complete the name component, but completion of "foo." leaves the name component alone since it is not the rightmost.

If *string* does not specify a name, then the name of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, completion fails.

- With the **:write** option, the default name is always used when *string* does not specify a name, regardless of what files exist.

- With the **:new-ok** option, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, the default name is used.

The special treatment of the case where all files in the directory have the same name is not very useful and is not implemented by all file systems.

If *string* does not specify a type, then the type of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, completion fails.

- With the **:write** option, the default type is always used when *string* does not specify a type, regardless of what files exist.

- With the **:new-ok** option, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, the default type is used.

In file systems such as LMFS and UNIX that require a trailing delimiter (> or /) to distinguish a directory component from a name component, the system heuristically decides whether the rightmost component was meant to be a directory or a name, and inserts the directory delimiter if necessary.

If *string* contains a relative directory specification for a host with a hierarchical file system, it is assumed to be relative to the directory in the default pathname and is expanded into an absolute directory specification.

The host and device components generally are not completed; they must be fully specified if they are specified at all. This might change in the future.

If *string* does not specify a version, the returned string does not specify a version either. This differs from file name completion in TOPS-20; TOPS-20 completes an implied version of "newest" to a specific number. This is possible in TOPS-20 because completing a file name also attaches a "handle" to a file. In Genera, the version number of the newest file might change between the time the file name is completed and the time the actual file operation (open, rename, or delete) is performed.

A pathname component must satisfy the following rules in order to appear in a successful completion:

- The host, device, and directory must actually exist.

- The name must be the name of an existing file in the specified directory, unless **:write** or **:new-ok** is included in *options*.

- The type must be the type of an existing file with the specified name in the specified directory, unless **:write** or **:new-ok** is included in *options.*

- A pathname component always completes successfully if it is **:wild**.

When the rules are not satisfied by a component taken from the default pathname, completion fails and that component remains unspecified in the resulting string. When the rules are not satisfied by a component taken from *string*, completion fails and that part of *string* remains unchanged (other components of *string* can still be expanded).

zl:listf *path* &optional *(output-stream standard-output)* *Function*

zl:listf is a function for displaying an abbreviated directory listing. The default for name, type, and version of *path* is **:wild**.

```
(listf "f:>jwalker>mit-220")
```

The format of the listing varies with the operating system.

5.3 Access Control Lists

5.3.1 Introduction to Access Control

The site administrator can configure a Symbolics Lisp Machine file server to provide file protection for its LMFS and FEP file system files. The file protection mechanisms then control access to files on the protected server machine. They *do* protect files from network file users, but they *do not* protect files from users with physical access to the file server's console. They *will* protect against mistakes, but they *will not* protect against malicious users.

There are two steps to protecting files in a LMFS file system using the file protection mechanism:

1. Decide what controls you, as owner, want to impose on access to directories. There are six methods of access control available. Details for each level are given in another section. See the section "Access Control Model: What You Can and Cannot Protect", page 171.

2. Configure the server machine. There are three steps involved in configuring a file server. These steps need only be performed once. See the section "Configuring a File Server", page 173.

5.3.2 Access Control Model: What You Can and Cannot Protect

The file protection mechanism offers a number of ways to protect your files from others.

1. **Protection against network users.** The Symbolics Lisp Machine can protect local files against inter-machine access. At this level of control, users accessing files across the network through the standard protocols cannot run arbitrary Lisp programs or access arbitrary data. The code that implements the network servers represents a *reference monitor*, in the sense that all user access to files passes through it. Therefore, a properly configured Symbolics Lisp Machine file server can provide a reasonable level of assurance that the file protection mechanisms do indeed protect files. The site administrator can check this with the file server log described later. See the section "The File Server Activity", page 174.

   The Symbolics Lisp Machine has no software architecture to support intra-machine file access protection. The flexible programming environment allows any user program to access all data on the machine. No computer data security can assure that it is resistant to penetration unless the architecture isolates the user and her programs from the data structures that define and implement system software and, in particular, its security mechanisms. Therefore, the ACL mechanism makes no attempt to protect files from a local user.

2. **The file server can require login.** In configuring a file server, the site administrator can choose whether to require login (identification and authentication) of all users of the file server. Login is implemented via user names and passwords. If the site administrator does not require login, then anyone can establish a file server connection to the server, and access any files which are marked as accessible to everyone. If the file server does require login, then all users must supply their name and password before they are allowed to access the file server. See the section "Administering Names, Capabilities, and Passwords", page 174.

   Supplying a name and password

   - demonstrates that the user is an authorized user of the facility, and thereby protects all the data on the server from unauthorized access

   - allows the site administrator to create and maintain an audit trail of the user's access via log files

3. **The site administrator can define capabilities.** The file protection model allows the owner of a directory to grant or deny access to data in that

directory to individuals on the basis of the user name they supply at login time.

In addition, the site administrator can define capabilities. A capability is an access identity shared by multiple users that describes some common access privileges. For example, she may wish to specify that all of the people on a particular project (the Vision project, for example) have access to a set of files. Rather than listing all of the people on the project (a list that may change over time) when specifying the access, she can grant access to the Vision capability, and then grant all of the appropriate users the right to use the capability. The site administrator grants a user the right to use a capability by giving her a password for the capability. Each user of a capability has a different password for the capability. Each user has multiple passwords: one for her user name, and then one for each capability that she has the right to use.

Users can utilize a capability by turning it on with the Enable Capabilities command. It prompts for a password, then turns on that particular capability for the requesting user.

4. **The owner can specify which users and capabilities can access the files.** The owner of a LMFS file can control access to files and directories by using the Edit ACL command to edit the Access Control List (ACL) associated with each directory. See the section "Edit ACL Command", page 177. The ACL for a directory controls access to the directory and all files within the directory. An access control list consists of an ordered set of pairs. Each pair consists of an access name (a user name or a capability name) and a list of access modes. The modes are:

Access Modes

Mode	The ability to
:read	read, probe, or perform **fs:file-properties** on files
:write	modify existing files
:append	append to existing files
:properties	delete files, expunge files, or change properties of files
:list	list the file names in the directory
:supersede	create new versions of existing files

:create	create new files (not new versions)
:owner	change the directory's ACL and other properties

The user's access to a file or directory is determined by looking through the list of access names for that file or directory. The system determines a user's right to perform an operation by checking her user name and any enabled capabilities against the ACL. Whichever currently enabled access name first matches a name in the ACL determines the user's access. The reserved user name "*" in an ACL matches all access names.

New directories are initialized with an ACL copied from their parent directory. For an example of an ACL: See the section "Edit ACL Command", page 177.

5. **The file server initialization determines the interpretation of an empty ACL.** In configuring the file server, the site administrator chooses whether an empty ACL grants access to all users or to no users. Granting access to all from an empty ACL is called *permissive access.*

5. **The file server can control access to the FEP file system.** There are no ACLs on individual FEP file system directories. There is a single directory in the LMFS hierarchy that controls access to all files and directories in the FEP file system.

   LOCAL:>File-Server>FEP-File-System-Access>

6. **The site administrator controls access to the physical console.** This protection scheme does not protect files from users with physical access to the file server's console. Therefore, the site administrator must consider some method of controlling physical access to all secure consoles.

5.3.3 Configuring a File Server

Configuration of a file server requires three steps that are performed only once.

1. Create the necessary directories. The function **fs:setup-file-server** creates several directories required for the operation of a file server, with or without security. These are:

Directory	Contents
>File-Server>	password files
>File-Server>Server-Logs>	file server log files

\>File-Server>FEP-File-System-Access>
: the ACL for this directory is the protection for the FEP file system

2. Set the access on system directories. Use the Edit ACL command to set appropriate access for all of the directories listed above. It is *very* important that the administrator set the protection for these directories. If she does not, then this protection scheme can be easily broken.

3. Set up the file server init file. The file server init file should contain an invocation of the function **fs:initialize-secure-server.** There is an example of this in the file sys:examples;file-server-init-file.text.

5.3.4 Administering Names, Capabilities, and Passwords

The database of names, capabilities, and passwords is kept in the file

\>File-Server>passwords.data.

This file is encrypted. To define a user name, or to change a user's password, or to change a user's password for a capability, use the Set Password command in the File Server activity.

To make a user name undefined, or to revoke a user's right to a capability, use the Remove Password command in the File Server activity.

5.3.5 The File Server Activity

The File Server activity includes commands for setting and removing passwords, logging file system activity and errors, and displaying file server status. The activity uses a frame accessible via the CP command Select Activity, using the File Server argument. The frame is divided up into five smaller panes. File Server commands can be entered through typein at the bottom pane, or by clicking on commands available in the File Server Command pane.

There are a few commands available in the File Server Command pane. These are:

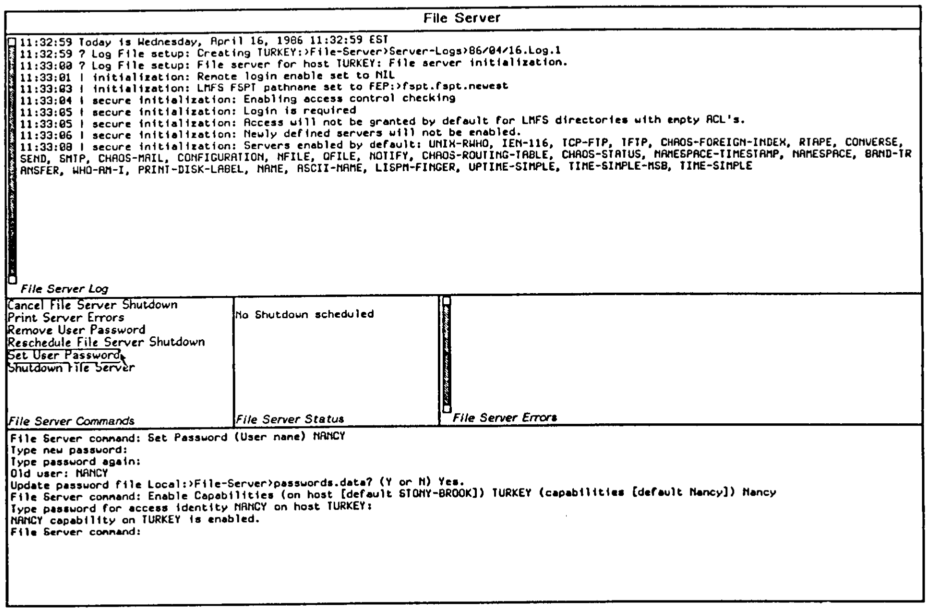
[Cancel File Server Shutdown]
: Cancel an already scheduled File Server shutdown.

[Remove User Password]
: Remove the password for a user, or remove the password for a capability for a user. The command prompts for the user's name.

[Reschedule File Server Shutdown]
: Change the time for a scheduled shutdown. The command prompts for a new number of minutes to shutdown and a string that is used for logging purposes.

[Set User Password]
: Set the password for a user, or for a capability for a user. The command prompts for the user's name.

[Shutdown File Server]
: Schedules a shutdown of the File Server Activity. The command prompts for the number of minutes to shutdown, and for a string. The string is used in the log file, and can be used for logging the reason for the shutdown.



```
11:32:59 Today is Wednesday, April 16, 1986 11:32:59 EST
11:32:59 ? Log File setup: Creating TURKEY:>File-Server>Server-Logs>86/04/16.Log.1
11:33:00 ? Log File setup: File server for host TURKEY: File server initialization.
11:33:01 | initialization: Remote login enable set to NIL
11:33:03 | initialization: LMFS FSPT pathname set to FEP:>fspt.fspt.newest
11:33:04 | secure initialization: Enabling access control checking
11:33:05 | secure initialization: Login is required
11:33:05 | secure initialization: Access will not be granted by default for LMFS directories with empty ACL's.
11:33:06 | secure initialization: Newly defined servers will not be enabled.
11:33:08 | secure initialization: Servers enabled by default: UNIX-RWHO, IEN-116, TCP-FTP, TFTP, CHAOS-FOREIGN-INDEX, RTAPE, CONVERSE,
SEND, SMTP, CHAOS-MAIL, CONFIGURATION, NFILE, QFILE, NOTIFY, CHAOS-ROUTING-TABLE, CHAOS-STATUS, NAMESPACE-TIMESTAMP, NAMESPACE, BAND-TR
ANSFER, WHO-AM-I, PRINT-DISK-LABEL, NAME, ASCII-NAME, LISPM-FINGER, UPTIME-SIMPLE, TIME-SIMPLE-MSB, TIME-SIMPLE
```

```
No Shutdown scheduled
```

```
File Server command: Set Password (User name) NANCY
Type new password:
Type password again:
Old user: NANCY
Update password file Local:>File-Server>passwords.data? (Y or N) Yes.
File Server command: Enable Capabilities (on host [default STONY-BROOK]) TURKEY (capabilities [default Nancy]) Nancy
Type password for access identity NANCY on host TURKEY:
NANCY capability on TURKEY is enabled.
File Server command:
```

Figure 3. File Server Activity Window

The File Server Status pane shows the current status of the File Server Activity. Currently this pane only displays whether a shutdown is scheduled, and if so, how many minutes until the shutdown occurs.

The File Server Errors pane shows any errors that have occurred. This includes all errors specified to the **:log-error-response-flavors** keyword argument to **fs:initialize-secure-server**, such as an unrecognized user trying to log in.

The File Server Log pane shows any File Server accesses that have occurred. This includes all messages specified to the **:logging-keywords** keyword argument to **fs:initialize-secure-server**, such as users logging in and out.

There are a number of new functions and CP commands for managing Access Control Lists. The CP commands Enable Capabilities and Disable Capabilities are user commands which allow access to groups of files. The CP commands Set Password and Remove Password are administrative commands which are used to control access to capabilities. The CP command Edit ACL is a user command which sets the access for directories. The functions **fs:initialize-secure-server** and **fs:setup-file-server** are initialization forms which are used by the file servers to create and maintain the access lists.

Enable Capabilities Command

Enable Capabilities *host* &rest *capabilities*

Turns on specified *capabilities* for *host* after checking access requirements. The host will prompt for your password for *capabilities.*

host	The name of the host on which you want to enable *capabilities.*
capabilities	One or more of the capabilities available for *host.* Each specified capability must be already recognized by the server before access can be enabled.

Disable Capabilities Command

Disable Capabilities *host* &rest *capabilities*

Turns off specified *capabilities* for *host.*

host	The name of the host on which you want to disable *capabilities.*
capabilities	The capabilities you want to disable on *host.*

Set Password Command

Set Password *user-name* &optional *capability*

Sets either the password for *user-name*, or the password for *capability* for *user-name.* This command must be executed from the File Server activity on the console of the host that knows about *capability.* This should be done by the site administrator responsible for assigning passwords.

user-name	The name of the user for whom you want to set a password.

capability	A capability that *user-name* may have access to.

Remove Password Command

Remove Password *user-name* &optional *capability*

Removes either the password for *user-name*, or the password for *capability* for *user-name*. This command must be executed from the File Server activity on the console of the host that knows about *capability*. This should be done by the site administrator responsible for assigning passwords.

user-name	The name of the user for whom you want to remove a password.
capability	A capability that *user-name* may no longer have access to.

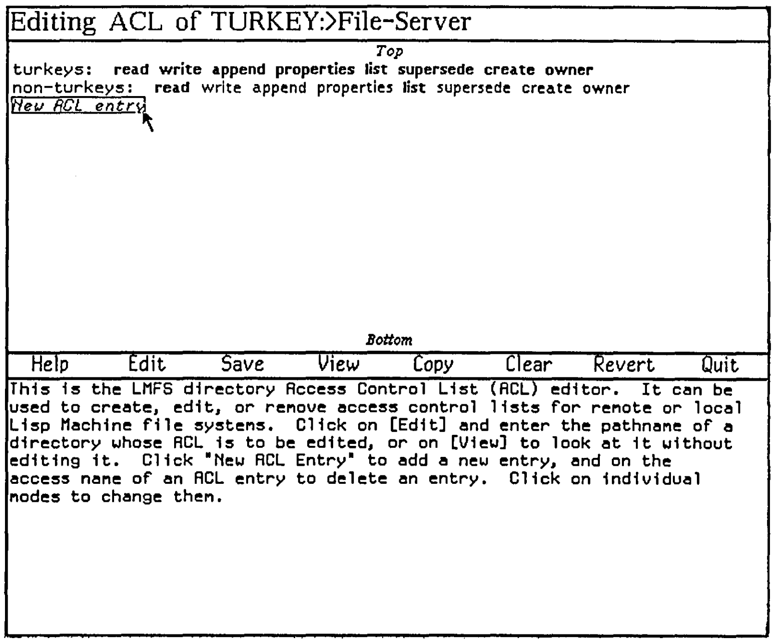
Edit ACL Command

Edit ACL *directory*

Starts up a small window, editing the Access Control List for *directory*. It is used to create, edit or remove access control lists for LMFS directories. This command can be executed from any console, by the owner of *directory*. This should be coordinated with the site administrator, so that passwords can be assigned and any new capabilities can be set up.

The window is divided up into four panes: the top displays the name of the directory, the next pane displays the existing ACLs, next is a command pane, and the bottom pane is minibuffer pane for typein.

[Help]	Type out general information about the Access Control List editor.
[Edit]	Prompts for the name of a directory, and starts an edit of its ACL. If there is no ACL for that directory, the second pane will only contain *New ACL entry*. Click on this command to add a new ACL. To delete an entry, click on the access name of that entry. Each entry consists of an access name, and a list of modes that can be enabled or disabled. Clicking on an mode toggles whether or not it is enabled. Modes that are enabled appear in boldface type, and modes that are disabled appear in roman type.
[Save]	Installs the permissions for the host you are editing by changing the file properties for that directory.
[View]	Prompts for the name of a directory, and displays its ACL. This



```
Editing ACL of TURKEY:>File-Server
                                  Top
turkeys:  read write append properties list supersede create owner
non-turkeys:  read write append properties list supersede create owner
New ACL entry

                                  Bottom
  Help     Edit     Save     View     Copy     Clear     Revert     Quit
This is the LMFS directory Access Control List (ACL) editor.  It can be
used to create, edit, or remove access control lists for remote or local
Lisp Machine file systems.  Click on [Edit] and enter the pathname of a
directory whose ACL is to be edited, or on [View] to look at it without
editing it.  Click "New ACL Entry" to add a new entry, and on the
access name of an ACL entry to delete an entry.  Click on individual
modes to change them.
```

Figure 4. Access Control List Editor

	is done in read-only mode, so you cannot modify the ACL using this command.
[Copy]	Creates a new ACL for a directory by copying the current ACL.
[Clear]	Disables all modes for the current ACL entry.
[Revert]	Reverts the ACL to the modes specified in the saved file for this directory. If the ACL has never been saved, the editor will ask if you want to discard the current ACL information. When you confirm, all entries are discarded.
[Quit]	Return from the Edit ACL command to whatever you were previously doing.

For the example here, the capability named "turkeys" allows all operations on the directory TURKEY:>File-Server. The capability named "non-turkeys" only allows the reading and listing of files in the directory.

fs:initialize-secure-server &key *access-permissive* (*login-required* **t**) *Function*
(*permitted-services* **fs:*trusted-services***)
(*logging-keywords*
fs:*default-secure-server-logging-keywords*)
(*log-error-response-flavors*
fs:*default-secure-server-error-response-log-flavors*)
(*lmfs-fspt-pathname* **lmfs:*fspt-pathname***)

This function starts up the security functions for the file server. The function takes the following keyword arguments. This command should be executed from the file server init file.

:access-permissive
When the value is **t**, an empty ACL gives access to all users. When the value is **nil**, an empty ACL gives access to no users.

:login-required Flags whether or not login is required to access files. When the value is **t** (the default), a login is required.

:permitted-services
A list of network service keyword names, such as **:tcp-ftp**, **:name**, and **:chaos-status**. When running a secure server, services that permit a user to get arbitrary access to the server machine should not be enabled. In particular, the **:eval** and **:login** services should not be enabled. The variable **fs:*trusted-services*** contains the services that Symbolics recommends be enabled.

:logging-keywords
These keywords control what information is put into the file server log. The currently defined keywords are:

:login Log a message for each login and logout.

:server-error-response
Log a message whenever an error condition in the server is translated into an error response to the user, subject to **:log-error-response-flavors**.

:untrusted-transaction
Log all transactions involving users who are not on a trusted subnet.

:server-bug-report	Log a message each time an automatic bug report is sent.
:server-error	Log a message every time the server takes an error that is not just a user error.

:log-error-response-flavors
A list of error conditions that should be logged. Only those errors that satisfy **typep** of a flavor in this list are logged.

:lmfs-fspt-pathname
The pathname of the LMFS File System Partition Table for this file server. It is of the form fep*n*:>fspt.fspt, where *n* is the disk unit where the fspt file resides.

fs:setup-file-server *Function*

This function creates the directories that are required for the operation of a file server, with or without security. **fs:setup-file-server** should be executed when setting up a file server. It creates the following directories required for the operation of a file server:

Directory	Contents
>File-Server>	password files
>File-Server>Server-Logs>	log files
>File-Server>FEP-File-System-Access>	the ACL for this directory is the protection for the FEP file system

6. Lisp Machine File System

6.1 Introduction to LMFS

The Lisp Machine File System (LMFS) provides a file system that runs on a Symbolics computer and stores information on the computer's disks. The information can be accessed locally (from that computer itself) or from other computers.

For information on performing I/O on files: See the section "Streams", page 3.

This discussion does not describe the internal program logic or organizational details of the file system. The methods for performing I/O on files are described elsewhere. See the section "Streams", page 3.

6.2 Concepts

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

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

The name is an arbitrary user-chosen string describing the file. The type is supposed to indicate what type of data the file contains; a type of "lisp" is the system convention for files containing Lisp source programs, "bin" for compiled Lisp programs, and so forth. The version number distinguishes successive generations of a file; to change a file, you normally read the latest version of the file into the Symbolics computer, modify it, and write out a new version with the next highest version number. The general scheme for naming files is covered elsewhere. See the section "Naming of Files", page 51.

Files reside in *directories*. The combination of name, type, and version of any file is unique in the directory in which it is contained. With the exception of a single directory (the *ROOT*), directories also reside in other directories. The directory in which a file or directory resides is called its *parent* directory, and these files and directories are said to be *inferior* to their parent directory. Directories and files thus form a strict tree (hierarchy); the ROOT directory is the root of this tree. Directories have a type of "directory" and a version of 1. Thus, the name of a directory alone identifies it in its containing (*superior*) directory. It is not possible to "fool" LMFS into thinking a file is a directory by giving it a type of "directory" and a version of 1, however.

Links are the third (and last) kind of object that can live in a directory. A link contains the character-string representation of the pathname of something else in the same file system, called the *target* of the link. This pathname specifies a directory, a name and a type, and it can specify a version. A link is conceptually an indirect pointer to something else; when certain operations are done on a link, the operation really gets done to the target instead of the link itself.

It is possible to have "directory links". See the section "LMFS Links", page 189.

The specific syntactic conventions, restrictions, and other information about LMFS pathnames are described elsewhere.. See the section "LMFS Pathnames", page 101.

LMFS also stores and maintains *properties* of objects. For example, for each file it stores the creation date, the author, whether the file has been backed up, and so on. Users can also create their own properties; each file has a property list that lets you store arbitrary associated information with the file. See the section "LMFS Properties", page 182.

The File System Editor is an interactive program that lets you manipulate the file system (the local LMFS system or the system on any host). You can invoke the program by typing `SELECT F`. See the section "File System Editor", page 211.

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

You do not have to take any special action to access the local file system on your Symbolics computer. If you use the host name of the machine, or the special string "local" as a host name, it is accessed automatically, as with any host.

6.3 Properties

Files, directories, and links have various *properties*. There are *system* properties, which are defined and maintained by the file system itself, and *user* properties, which are defined and maintained by programs and people that use the file system. Every property has a *name*, which is a keyword symbol, and a *value*, which is a Lisp object. The names of all of the system properties are listed below. (These properties should not be confused with the *file attribute list*, the **-*-** line in the beginning of the file. See the section "File Attribute Lists", page 156.)

You can examine the values of properties by using either the [View Properties] command in the File System Editor, or View File Properties (m-X) in Zmacs. Users alter the values by using the either the [Edit Properties] command in the File System Editor, or Change File Properties (m-X) in Zmacs. See the section "File System Editor", page 211. Programs access the values of properties by using the **fs:directory-list** and **fs:file-properties** functions and alter the values by using the **fs:change-file-properties** function. See the section "Accessing Directories", page 160.

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

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

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

When the **fs:file-properties** function is called for a LMFS file, it returns a second value: a list of the names of all the properties of the file that are changeable. This function lists all the system properties and all the user properties for the object it is given.

The names of user properties must be symbols on the keyword package, and must not be the same as any of the system property names. The value associated with a user property must be a string. The combined length of the name of the property and its value must not exceed 512 characters. To remove a user property from a file, you set the value of the property to **nil**. **fs:file-properties** returns all the user properties of a file, but **fs:directory-list** does not return any of them.

You can create new user properties with the [New Property] command in the File System Editor; after they are created, you can edit them with [Edit Properties]. Programs create and change user properties by using **fs:change-file-properties**.

User properties involve a subtle trap. If you misspell or otherwise misconstrue the name of a system property, LMFS assumes that you have given the name of a user property, and set it. Thus, LMFS can never admit of, nor diagnose, an unrecognized, or invalid, property name.

Here is a list of all of the standard properties that LMFS maintains. The standard, generic interpretation and representations of the system standard properties among them can be found elsewhere: See the section "Functions for Accessing Directories", page 161. Refer to the table below for the rest.

```
:length-in-bytes
:byte-size
:author
:creation-date
:modification-date
:reference-date
:deleted
:not-backed-up
:dont-delete
:dont-reap
:open-for-writing (LMFS-specific)
:length-in-blocks
:generation-retention-count
:directory
:auto-expunge-interval
:date-last-expunged
:default-generation-retention-count
:default-link-transparencies (LMFS-specific)
:link-to
:link-transparencies (LMFS-specific)
```

The following among them are changeable, that is, users can set their values by means of **fs:change-file-properties**:

```
:generation-retention-count
:modification-date
:reference-date
:creation-date
:author
:deleted
:dont-reap
:dont-delete
:auto-expunge-interval
:default-generation-retention-count
:default-link-transparencies
:link-transparencies
```

The following is a list of all the properties supported by LMFS that are either specific to LMFS or require other special comment.

:byte-size *(Files)*
: For a character file, **8**. For a binary file, the byte size of the file (the number of bits in each byte), a fixnum between **1** and **16**, inclusive. LMFS maintains both the byte size and the binary/character quality of a file natively. It is not permitted to open a binary file with a byte size other than that with which it was written. This property is not currently a changeable one.

:length-in-blocks *(Files, directories, links)*
: A *LMFS record* is 1152 32-bit words. This is the basic allocation unit of the file system. The name of the generic system property is confusing in the case of LMFS, for a LMFS record is composed of multiple disk blocks. This property cannot be meaningful for directories.

:creation-date *(Files, directories, links) (Changeable)*
: LMFS allows setting of creation date by user programs. Creation date, when not set by a user program, is also updated when a file is appended to.

:modification-date *(Files) (Changeable)*
: The most recent time at which this file was modified, expressed in Universal Time. This is the same as the creation date if the file has been opened for appending. Operations such as renaming and property changing update this property, but do not update creation date. The dumper, for instance, is driven off this property.

:author *(Files, directories, links) (Changeable)*
: This property is user-settable in LMFS.

:dont-delete *(Files, directories, links) (Changeable)*
If **t**, does not allow this object to be deleted. The purpose of this attribute is to prevent the accidental deletion of important files. An error results if an attempt is made to delete this file.

:dont-reap *(Files, directories, links) (Changeable)*
This attribute, although maintained internally by LMFS, is not interpreted by LMFS. Dired directory maintenance tools use this property.

:default-generation-retention-count *(Directories) (Changeable)*
The default value for the **:generation-retention-count** property of new objects created in this directory. **:generation-retention-count** is used by LMFS to control the number of versions of each file. See the section "LMFS Deletion, Expunging, and Versions", page 187. The value should be **nil** or a nonnegative fixnum.

:auto-expunge-interval *(Directories) (Changeable)*
LMFS will automatically expunge a directory whenever a file system operation is performed, provided the directory in question has this property and that amount of time has expired since the last time the directory was expunged, whether this previous expunging happened manually, or as a result of **:auto-expunge-interval**. All deleted files will be expunged, and the time of their deletion will not be taken into consideration. See the section "LMFS Deletion, Expunging, and Versions", page 187. The value should be **nil** or a nonnegative fixnum.

:default-link-transparencies *(Directories) (Changeable)*
The initial value for the **:link-transparencies** attribute of links created in this directory. See the section "LMFS Links", page 189. To set this property, use the [Link Transparencies] command in the File System Editor rather than [Edit Properties].

:link-transparencies *(Links) (Changeable)*
The transparencies of this link. See the section "LMFS Links", page 189. To set this property, use either the [Edit Link Transparencies] or [Edit Properties] commands in the File System Editor, or Change File Properties (m-X) in Zmacs.

:complete-dump-date *(Files, directories, links)*
The most recent time at which this object was dumped on a complete dump tape, expressed in Universal Time. See the section "LMFS Backup", page 190. A positive bignum. If this object has never been dumped on a complete dump tape, this property is not present. This property does not appear in directory listings.

:complete-dump-tape *(Files, directories, links)*
The tape reel ID of the complete dump tape on which this object was most

recently dumped. A string. If this object has never been dumped on a complete dump tape, this property is not present. This property does not appear in directory listings.

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

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

6.4 Deletion, Expunging, and Versions

When an object (file, directory, or link) in LMFS is deleted, it does not really cease to exist. Instead, it is marked as "deleted" and continues to reside in the directory. If you change your mind about whether the file should be deleted, you can *undelete* the file, which will bring it back. The deleted objects in a directory actually go away when the directory is *expunged*; this can happen by explicit user command or by means of the auto-expunge feature (see below). When a directory is expunged, the objects in it really disappear, and cannot be brought back (except from backup tapes.) See the section "LMFS Backup", page 190.

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

Users normally delete and undelete objects with the Zmacs commands Delete File (m-X) and Undelete File (m-X), or [Delete] and [Undelete] commands in the File System editor, or D and U in Dired. Directories can be expunged with Dired or the File System Editor, also, and the Expunge Directory (m-X) command in Zmacs. See the section "File System Editor", page 211.

Programs normally delete files using the **zl:deletef** function. See the function **zl:deletef**, page 151. Whether a file is deleted or not also appears as the **:deleted** property of the file, and programs can delete or undelete files by using **fs:change-file-properties** to set this property to **t** or **nil**.

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

The normal way of writing files in the Genera environment is to create a new version of the file each time a file is written. When you edit with Zmacs, for example, every so often the Save File command is issued, and a new version is written out. After a while, you end up with many versions of the same file, which clutters your directory and uses up disk space. Zmacs has some convenient commands that make it easy to identify and automatically delete the old versions.

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

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

When a file version is being created, it is marked with the property **:open-for-writing**. This property is removed when the file is successfully closed. While the file has this property, it is invisible to normal directory operations and to attempts to open or list it. Directory list operations that specify **:deleted** can see the file. Files in this state have the "open for writing" property when you use View Properties in the file system editor, or Show File Properties (m-X) in Zmacs. Files left in this state by crashes have to be removed manually by deleting and expunging. For example, suppose versions 3, 4, and 5 exist, but 5 is open in this state. An attempt to read **:newest** would get version 4; an attempt to write **:newest** would create version 6.

6.5 LMFS Links

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

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

:read	Opening the file for **:input**.
:write	Opening the file for appending, via **:if-exists :append**.
:create	Opening the file for **:output**
:rename	Renaming the file.
:delete	Deleting the file.

You can create new links with the [Create Link] command in the File System Editor, or Create Link (m-X) in Zmacs. See the section "File System Editor", page 211. Programs can use the **:create-link** message to pathnames. See the section "Pathname Messages: Naming of Files", page 92. When a new link is created, its transparencies are set from the **:default-link-transparencies** property of its superior directory. When a new directory is created, its **:default-link-transparencies** property is set to **(:read :write)**.

The value of the **:link-transparencies** property of a link is a list of keywords describing the transparency attributes of which this link is possessed. The value of the **:default-link-transparencies** attribute of a directory is, similarly, a list of all those transparencies to be possessed by newly created links in this directory. When changing the value of either of these properties with **fs:change-file-properties**, the new value of the property is such a list of transparency keywords, chosen from the table above. Transparencies not present in the new value are turned off, and they are not preserved. There is no way to change an individual transparency.

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

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

A directory link is a link whose type is "directory", whose version is 1, and whose target is a real directory or another directory link. The maximum permitted length of such chains of directory links is 10. The system respects a directory link when looking for a directory. By means of directory links, "indirect pointers," or multiple names for directories, can be established. Simply naming a link in this fashion is sufficient; no special action need be taken to declare a link to be a directory link. Transparencies are not interpreted in directory links.

6.6 LMFS Backup

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

In a *complete dump*, all of the files, directories, and links in the file system are written out to tapes. This, obviously, saves all the information needed to reload the file system. However, a complete dump can take a long time and use a lot of tape, especially if the file system is large. In order to make it practical and convenient to dump the file system at short intervals, a second kind of dump can be done, called an *incremental dump.*

In an incremental dump, only those files and links that have been created or modified since the last dump (of either kind) are dumped; things that have stayed the same are not dumped. (All directories are always dumped in an incremental dump.) Now, if the file system is destroyed, you reload it by first reloading from the most recent complete dump and then reloading each of the incremental dump tapes made since that complete dump, in the same order in which they were created. Therefore, you do not need to retain incremental dump tapes that were made *before* the most recent complete dump was done; you can reuse those tapes for future dumps.

Since all tapes containing incremental dumps done since the last complete complete dump must be reloaded in order to restore the file system, doing a complete dump regularly makes recovery time faster. Doing complete dumps also lets you reuse incremental dump tapes, as described above. The more incremental dump tapes you must load at recovery time, the longer it takes to recover, and thus the more chance there is that something will go wrong. Thus, it is advantageous to perform complete dumps periodically.

A *consolidated dump* is like an incremental dump, in that it only dumps files that have been created or changed recently. However, a consolidated dump backs up only those files that have been created or changed since a specified *consolidation date*. A consolidated dump is the appropriate kind to take if some event destroys recent incremental dump tapes, or they are found to be unreadable. If a complete dump extends through several days, it is wise to take an incremental dump between-tape stopping points as appropriate.

See the section "Dumping, Reloading, and Retrieving" in *Site Operations*.

6.7 Multiple Partitions

The Lisp Machine File System (LMFS) allows the use of multiple partitions residing on one or more disk drives. It utilizes one or more files of the FEP file system as the vessels in which it stores its files and directories. These FEP files are called *partitions*. Normally, there is one large partition, usually called LMFS.FILE.1. All the files created by LMFS actually reside inside this FEP file, but the existence of these files is known only to LMFS, whose purpose it is to manage them; they are not known to the FEP file system. Since FEP files are limited to one particular disk drive, if a LMFS file is to utilize the space available on multiple drives, partitions must be created on each drive on which it is desired that LMFS store files. Then, LMFS must be instructed to use these partitions.

The selection of partitions to be used by LMFS is determined by a database called the *file system partition table* (FSPT). It is contained in a FEP file named >fspt.fspt on a boot drive. The FSPT is optional. If it is not present, LMFS uses lmfs.file on the FEP boot drive. The FSPT is a simple character database containing the actual pathnames (in the FEP file system) of the partitions to be used for file system access.

If any machine at your site has more than one disk, it may be difficult to find the disk location of the FSPT. In order to make finding the location of a FSPT easy, insert the Set LMFS FSPT Unit command in your Hello.boot file. This command looks for the file named >fspt.fspt on the unit (disk unit) specified. For example, if you put your FSPT on disk unit 2, put the following in your Hello.boot file:

```
Set LMFS FSPT Unit 2
```

Each partition in the file system knows how many partitions make up the file system. Only the FSPT, which is used only at LMFS startup time, indicates the locations of these partitions. That is, the file system databases in the actual partitions do not contain drive and partition numbers or FEP pathnames. Thus, when LMFS is down, partitions can be moved around using Copy File (m-X); as long as the FSPT is edited to indicate their new locations, LMFS comes up (when required) using the moved partitions. **Note:** Since the Copy File (m-X) command copies files according to byte size, you may need to edit the byte count of the partition for the copy file command to work. To do this, multiply the number of blocks by 1152, since partitions were previously created with a byte size of 0. For example:

```
1152 * number of blocks
```

The FSPT is edited only to move partitions around or to add a partition. When you add partitions to the file system, the file system automatically rewrites the FSPT database to include the locations of new partitions.

6.7.1 Free Records

The basic unit of allocation in the Lisp Machine File System is the *record*. A record is 1152 32-bit words, or four disk blocks. Each file system object is made from an integral number of records. At any time, each record is *in use* (representing an existing file system object) or *free* (not representing anything and free to be used in new objects). When the file system needs to find a new free block to create or grow an object, it does not search through the records looking for a free one, because that would require many disk operations and be very slow. Instead, the file system uses a redundant data structure called the *free record map*, kept in several blocks in a known location in the file system partition. The map has one bit for each record in the file system; this bit marks whether the record is free or in use. The file system can find a free record quickly by examining this map.

If the file system crashes, or something else goes wrong, the contents of the free record map can become inconsistent with the contents of the file system itself. For each record, two different errors are conceivable.

- The record might actually be in use, representing part of an object, but marked as "free" in the map. The system is designed so that this cannot happen, but hardware problems might cause it to happen anyway.

- The record might actually be free, but marked as "in use" in the map.

The first error is much worse than the second; the file system might use the record for a new object even though it is currently representing some existing object, which could destroy the existing object. If the second error occurs, the record simply is not allocated even though it could be. Such a record is said to be *lost*.

The file system is written so that a crash can only cause the second kind of error. While the file system is operating, it maintains a *free buffer* in its data structures in virtual memory. The free buffer is a pool of records that are not actually in use, but are marked as being in use in the free record map on the disk. When it needs to allocate a record, it draws on one of these; when it frees up a record, it adds the record to this buffer. When the buffer gets too big, some records are removed from the buffer and marked as "free" in the map on the disk; when the buffer runs low, more records are marked as "in use" in the map on the disk, and are added to the buffer. So, if the machine is cold booted, or the file system crashes, the records that are in the buffer are lost, but no errors of the first kind are caused. The size of the buffer is maintained at about 30 records, so each crash loses 30 records. To recover, log out of the machine or use the [Flush Free Buffer] command to flush the entire free buffer and mark the records as "free" in the map on the disk. To use the [Flush Free Buffer] command, press SELECT F to enter the File System Editing Program. Click right on [Local LMFS Operations] to invoke the second level of the progam, where you can click on [Flush Free Buffer]. After the buffer has been flushed, you can cold boot the machine without losing any blocks.

Lost records can be found again by the salvager. See the section "Salvager", page 194.

You can check the number of free records in the file system by using the File System Editing Operations program. First, press SELECT F to select the program. Then, click right on [Local LMFS Operations], to invoke the second level of operations. In the second level, if you click left on [Free Records], the program displays a line for each block of the file map, telling you which records are covered by that block, the number of such records, and how many are marked as free. It also tells you how many free records (marked as "in use" in the map) are in the free buffer, and finally displays a grand total of the number of free, used, and total records in the file system.

To find out how many records are actually in use, click middle on [Free Records] to prepare a printable report of record use throughout the file system. This has to pass over every object in the file system, and so it takes some time, especially on large file systems. The discrepancy between the answer of this function and the answer you get when you click left on [Free Records], tells you how many lost records there are; if there are a lot, you might want to run the salvager.

Clicking right on [Free Records] displays how many records are in use in each partition. This information is necessary for commands such as [Grow Partition] that allow you to change the size of partitions, add partitions, or remove partitions.

6.7.2 Salvager

The *salvager* is a program that reads every LMFS record of the file system and finds and fixes certain inconsistencies and errors. It can fix two classes of problems.

- It can see which records are in use and which are free, and update the free record map to reflect the current state of the file system. This is how you recover lost records.

- It can find objects that are stored in a file system partition but are not referenced by any directory. Such objects are called *orphans*; they exist only if some problem has occurred, such as a file system crash during the creation of a file, or an unanticipated failure of some sort. The salvager finds such objects and puts them back into the directory hierarchy (*repatriates* them).

6.7.2.1 Using the Salvager

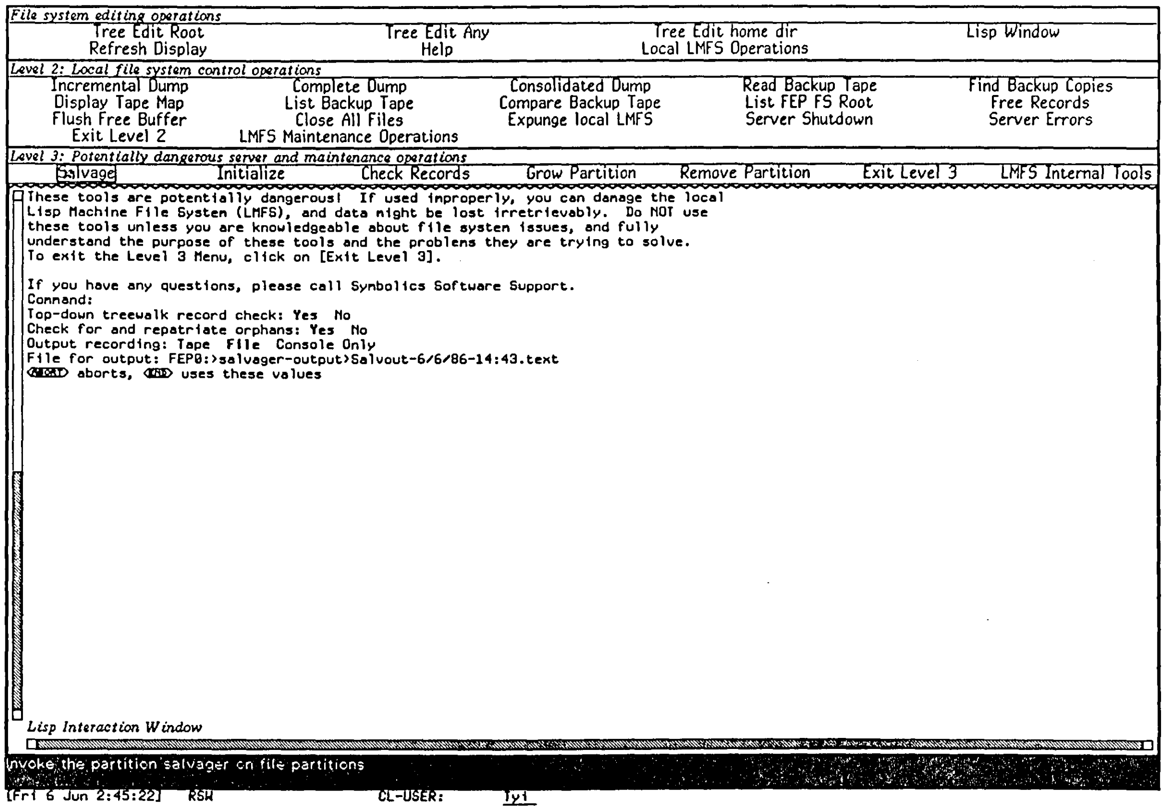
To run the salvager, press `SELECT F` to select the File System Editing Operations program. Click on [Local LMFS Operations] to invoke the second level of the program. Next, click on [LMFS Maintenance Operations] to invoke the third level of the program. Now click on [Salvage] to obtain a menu of options. If you have a local file system of multiple partitions (occupying multiple FEP files), you are presented with a menu of partitions to process. This menu, which is an Accept-Values menu, also includes questions about salvager operations. In addition to listing the partitions to be salvaged, the menu offers you the options as shown in figure 5.

Here are the options:

```
Top-down treewalk record check:  yes no

Check for and repatriate orphans:  yes no

Output recording:  Tape File Console only

File for output:
```

The first items on the menu constitute a list of partitions you can select for processing by the salvager. You can choose some or all of the partitions for processing.

The second menu option, `Top-down treewalk record check`, offers to scan all of the



```
These tools are potentially dangerous!  If used improperly, you can damage the local
Lisp Machine File System (LMFS), and data might be lost irretrievably.  Do NOT use
these tools unless you are knowledgeable about file system issues, and fully
understand the purpose of these tools and the problems they are trying to solve.
To exit the Level 3 Menu, click on [Exit Level 3].

If you have any questions, please call Symbolics Software Support.
Command:
Top-down treewalk record check: [✓ Yes]  No
Check for and repatriate orphans: [✓ Yes]  No
Output recording: Tape  [✓ File]  Console Only
File for output: FEP0:>salvager-output>Salvout-6/6/86-14:43.text
<ABORT> aborts, <END> uses these values
```

```
Invoke the partition salvager on file partitions
[Fri 6 Jun 2:45:22]    RSW                    CL-USER:        Tyi
```

Figure 5. Salvager Options

directories and files in the local LMFS and report any damaged records (hardware or software), disappeared files, or any other problems. This search starts at the root and goes through all of the file system, directory by directory, and is performed after all other salvaging activity.

Note: If you deselect any partition for repatriation, then the next menu item, which offers to check for and repatriate orphans, disappears. This happens because it is impossible to construct an accurate model of the hierarchy if each partition is not scanned.

The third menu option, `Check for and repatriate orphans`, offers to find orphaned objects and put them back into the directory hierarchy. During this scan, the salvager also replaces bad directory records with good ones.

The fourth menu option, `Output recordings`, offers to log the salvager output either to tape, in a file, or only to the console.

- If you choose the `Tape` option for output recording, every message goes onto the tape as soon as it is produced because of a special format that is used. Using an industry-compatible tape ensures that all messages appear on tape. If you use a cartridge tape, this is not fully guaranteed. The following forms may be used to view the tape produced in this way:

  lmfs:print-salvager-output-tape &optional *tape-spec* (*stream* **zl:standard-output**) *Function*

  Prints the contents of the tape created by the salvager. If you do not supply any arguments you are prompted for a tape spec, and output prints to the console.

  lmfs:copy-salvager-output-tape-to-file &optional *tape-spec* *pathname* *Function*

  Prints the contents of the tape created by the salvager to a file. You are prompted for any arguments not given.

- If you choose the `File` option for output recording, you must supply a file name. The default file is a FEP file, on boot unit 0. Every time a message is written to file a **:finish** is done to the file, so that even if the system crashes, the file is intact with all the messages up to the point of failure.

  If you decide to put the output recording in a FEP file, make sure there is enough room, probably about 100 blocks. If you have your output recording sent to another host, choose a host that you are sure will stay on the network during the logging process.

  Warning: If you only have one Symbolics computer or one file server, you can't use the `File` option because you may not put the output recording in a local LMFS file.

  There are currently no tools for automatically processing a file containing a log of salvager output.

- If you choose the `Console only` option for output recording, note that this is not usually the device of choice. You should choose this option when there is no other means of logging available.

- If any problems occur while the log is running, such as a file closing or a disrupted network connection, a menu appears. This menu asks what to do about continuing the salvager's log. If you enter the debugger while the log is being recorded you are offered restart options for discontinuing or re-selecting log options.

6.7.2.2 What the Salvager Does

The salvager always reconstructs the free record maps. Running the salvager takes about two minutes per thousand records of file partition.

When the salvager is repatriating an orphan and it cannot find the directory in which the orphan is supposed to reside, it creates a new directory as an inferior of the directory >repatriations, with a name like lost-1 or lost-2. After a repatriating salvager run, you should examine these directories. When the salvager repatriates an object, it types out a message saying that it did so. One of these messages might cause a **MORE** pause. If you plan to leave your console unattended while the salvager is working, you might want to disable **MORE** pauses before you start it.

Note: The salvager always considers storage occupied by orphans to be "in use" for purposes of the free record map, even if it is not repatriating the orphans. Thus, if many orphans existed, they could use up a great deal of disk space. But normally, orphans do not occur at all. When the salvager repatriates it also "fixes" disk errors and misplaced records or directories by replacing them with fresh, empty ones. By nature of the repatriation process, no files are lost in this way.

6.7.3 Adding a Partition to LMFS

You can add partitions to LMFS by using the File System Editing Operations program. First, press SELECT F to select the menu for that program. Click on [Local LMFS Operations] to invoke the second level of the menu. Then, click on [LMFS Maintenance Operations] to invoke the third level of the menu. In the third level of the menu, clicking right on [Initialize] yields a menu of initialization options, which offers [New File System] and [Auxiliary Partition] as choices. Choosing [New File System] is similar to clicking *left* on [Initialize]; it initializes a partition to be the basis of a file system. Clicking left on [Initialize] prompts for an initial LMFS partition location, offering FEP0:>lmfs.file as a default location.

When you add a new partition or a partition on another disk, the disk should be free of errors and properly initialized and formatted, and the partition should exist.

To add another partition, choose [Auxiliary Partition]. Enter the pathname of the FEP file to be used as the new partition. (The default pathname presented, which is correct for [New File System], is never correct for adding [Auxiliary Partition].) Then choose [Do It]. The system then performs much verification and error checking, roughly as much as when initializing a new partition. It must not be interrupted while performing these actions. When finished, it adds the partition and edits the FSPT automatically.

7. FEP File System

The Symbolics computer disk has a file system called the *FEP file system*. The entire disk is divided up into *FEP files* (that is, files of the FEP file system). FEP files have names syntactically similar to those of files in the Symbolics computer's own local file system. However, the FEP file system and the Lisp Machine File System (LMFS) are completely distinct.

The *FEP file system* manages the disk space available on a disk pack, grouping sets of data into named structures called *FEP files*. All the available space on a disk pack is described by the FEP file system. A single FEP file system cannot extend beyond a single disk pack; each disk pack has its own separate FEP file system.

The FEP file system supports all of the generic file system operations. It also supports multiple file versions, soft deletion and expunging, and hierarchical directories.

Although "FEP" is an acronym for *front-end processor*, the FEP file system is managed by the main Lisp processor. It is called the FEP file system because the FEP can read files stored in the FEP file system. For example, the FEP uses the FEP file system for booting the machine and running diagnostics.

Disk streams access FEP files. A disk stream is an I/O stream that performs input and output operations on the disk. (For information about streams: See the section "Types of Streams", page 7. See the section "Stream Operations", page 29. When disk streams are opened with a **:direction** keyword of **:input** or **:output**, the disk stream reads or writes bytes, respectively, buffering the data internally as required. When the **:direction** is **:block**, the disk stream can both read and write the specified disk blocks. Block mode disk streams address blocks with a block number relative to the beginning of the file, starting at file block number zero. This *file block number* is internally translated into the corresponding disk address. The checkwords of all disk blocks contained in the FEP file system are reserved for use by the FEP file system, so block mode transfers should not use the checkwords stored in the disk array. See the section "3600-Family Disk System Definitions and Constants" in *Internals, Processes, and Storage Management.*

The FEP file system is also used by the system for allocating system overhead files, such as the paging file. See the section "FEP File Types", page 201. This section lists some of these files and what they are used for.

The need to allow the FEP to access FEP files, and also to allow the system to use them imposes some constraints on the design of the FEP file system. The internal data structures of the file system must be simple enough to permit the FEP to read them, and a small amount of concurrent access by both the FEP and Lisp must be tolerated. A FEP file's data blocks should have a high degree of locality on the disk to minimize access times. And the FEP file system must be very reliable, since the FEP needs to use the file system for running diagnostics and for booting the machine.

Note: Because of these constraints, the FEP file system is not intended to be a replacement for LMFS. (See the section "Lisp Machine File System", page 181.) Allocating new blocks for FEP files is slow, so creating many files, especially many small files, might impair the performance of the FEP file system, and ultimately the virtual memory system, if paging files or world load files become highly fragmented.

7.1 Naming of FEP Files

The FEP filename format is similar to the LMFS filename format. See the section "Lisp Machine File System", page 181. There are differences, however. Here are the format details of a FEP filename:

host — The name of the FEP file system host. The format for a FEP host is *host*|FEP*disk-unit*, where the *host* field specifies which machine's FEP file system you are referring to, and *disk-unit* specifies the disk unit number on the machine. The *host* field defaults to the local machine if you omit it and the terminating vertical bar (|). If you omit both the *host* and *disk-unit* fields, the FEP host defaults to the disk unit the world was booted from on the local machine. For example:

`Merrimack|FEP0`	The FEP file system on Merrimack's unit 0.
`FEP2`	The FEP file system on the local machine's unit 2.
`FEP`	The FEP file system the booted world load file resides on.

directory — The name of the directory. The FEP file system supports hierarchical directories in the same format as in LMFS. Each directory name is limited to a maximum of 32 characters; there is no limit on the total length of a hierarchical directory specification.

name — The name of the FEP file, which cannot exceed 32 characters.

type — The type of the FEP file, which cannot exceed 4 characters.

version — The version number of the FEP file, which must be a positive integer or the word "newest".

FEP files can be renamed. For example, if you save a world containing MACSYMA, you might want to rename the world file to >macsyma.load or >macsyma1.load. Be sure to update your boot file if you intend this to be the default world.

7.2 FEP File Types

By convention, the following file types are used by the FEP file system for files used by that system.

boot	The file contains FEP commands that can be read by the FEP's Boot command. *boot* files are text files, and can be manipulated by the editor. See the section "Configuration Files", page 202.
load	The file contains a world load image, or *band*, that is used to boot the system.
mic	The file contains a microcode image, plus the contents of other internal high-speed memories that are initialized when the computer is booted. For example, >tmc5-io4-row-mic.mic.389 contains version 389 of the microcode for version 5 of the TMC.
fspt	The file contains a LMFS partition table. It tells LMFS which FEP files to use for file space. For example, >fspt.fspt.newest is the default partition table used by LMFS.
file	The file contains a LMFS partition which holds the machine's local file system. The entire Symbolics computer local file system normally resides inside one big file of the FEP file system. For example, >lmfs.file.newest is the default LMFS file partition.
page	The file contains disk space that can be used by the virtual memory system. To increase the effective size of virtual memory, you can add additional paging files. See the section "Allocating Extra Paging Space", page 220. For example, >page.page.newest is the default file used by the virtual memory system as storage for swapping pages in and out of main memory.
flod	The file contains a FEP Load file. FEP Load files contain binary code the FEP can load and execute.
fep	The file contains binary information used by the FEP file system. These files should not be written to by user programs. Some examples of these files are:

>root-directory.dir	This is the root directory for the FEP file system.
>free-pages.fep	Describes which blocks on the disk are allocated to existing files.
>bad-blocks.fep	Owns all the blocks that contain a media defect and should not be used.
>sequence-number.fep	Contains the highest sequence number in use. The FEP file system uses sequence numbers internally to uniquely identify files. This is to assist in rebuilding the file system in case of a catastrophic disk failure.
>disk-label.fep	Contains the disk pack's physical disk label. The label is used to identify the pack and describe its characteristics.

dir The file contains a FEP directory. For example, fep0:>root-directory.dir.newest contains the top-level root directory. The directory file for fep0:>dang>examples> would reside in fep0:>dang>examples.dir.1.

7.3 Configuration Files

Configuration files contain FEP commands tailored for a particular Symbolics computer configuration. The commands are executed if you specify the file as argument to a Boot command when cold booting the machine. See the section "FEP Commands" in *Site Operations.*

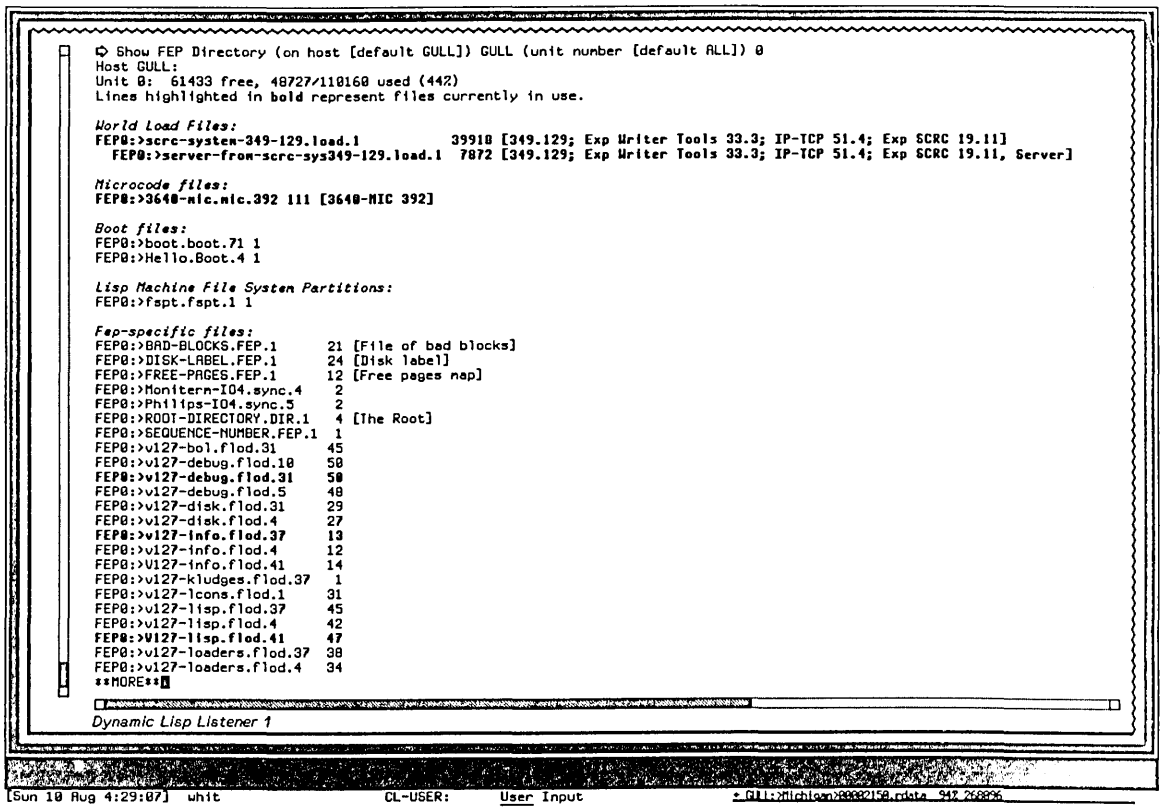
The configuration file >Boot.boot usually contains FEP commands to:

- Clear the internal state of the machine
- Load the microcode
- Load a world
- Set the Chaosnet address
- Start the machine

To change the selection of microcode and world loads that are booted by default, simply use Zmacs to edit the file FEP*n*>Boot.boot, where FEP*n* is the disk unit. Be careful to avoid typographical errors; otherwise, you might have to type in the commands manually in order to boot the machine. Also, be sure that the last command in the file is followed by RETURN.

7.4 FEP File Comment Properties

Comment properties supply additional information about the contents of FEP files. They are listed inside square brackets, where the reference or expunge date appears for other file systems. You can list the contents of the FEP file system by using the Show FEP Directory command. The Zmacs command Dired (m-X) of fep*n*:>*, or the form `(dired "fep:n>*")` (where *n* is the disk unit) invokes the directory editor on the FEP file system. An example of the Show FEP Directory command output is shown in figure 6.



```
Show FEP Directory (on host [default GULL]) GULL (unit number [default ALL]) 0
Host GULL:
Unit 0:  61433 free, 48727/110160 used (44%)
Lines highlighted in bold represent files currently in use.

World Load Files:
FEP0:>scrc-system-349-129.load.1          39918 [349.129; Exp Writer Tools 33.3; IP-TCP 51.4; Exp SCRC 19.11]
  FEP0:>server-from-scrc-sys349-129.load.1  7872 [349.129; Exp Writer Tools 33.3; IP-TCP 51.4; Exp SCRC 19.11, Server]

Microcode files:
FEP0:>3640-mic.mic.392 111 [3640-MIC 392]

Boot files:
FEP0:>boot.boot.71 1
FEP0:>Hello.Boot.4 1

Lisp Machine File System Partitions:
FEP0:>fspt.fspt.1 1

Fep-specific files:
FEP0:>BAD-BLOCKS.FEP.1       21 [File of bad blocks]
FEP0:>DISK-LABEL.FEP.1       24 [Disk label]
FEP0:>FREE-PAGES.FEP.1       12 [Free pages map]
FEP0:>Monitern-I04.sync.4     2
FEP0:>Philips-I04.sync.5      2
FEP0:>ROOT-DIRECTORY.DIR.1    4 [The Root]
FEP0:>SEQUENCE-NUMBER.FEP.1   1
FEP0:>v127-bol.flod.31       45
FEP0:>v127-debug.flod.10     50
FEP0:>v127-debug.flod.31     50
FEP0:>v127-debug.flod.5      48
FEP0:>v127-disk.flod.31      29
FEP0:>v127-disk.flod.4       27
FEP0:>v127-info.flod.37      13
FEP0:>v127-info.flod.4       12
FEP0:>V127-info.flod.41      14
FEP0:>v127-kludges.flod.37    1
FEP0:>v127-lcons.flod.1      31
FEP0:>v127-lisp.flod.37      45
FEP0:>v127-lisp.flod.4       42
FEP0:>V127-lisp.flod.41      47
FEP0:>v127-loaders.flod.37   38
FEP0:>v127-loaders.flod.4    34
**MORE**
```

Dynamic Lisp Listener 1
[Sun 10 Aug 4:29:07] whit
CL-USER:
User Input
GULL:>Michigan>00002150.rdata 942 268896

Figure 6. FEP File Comment Properties

7.5 Accessing FEP Files

FEP files are accessed by open disk streams. A disk stream is opened by the **open** function. (See the section "Accessing Files", page 129. That section contains more details on accessing files.) If a FEP file system residing on a remote host is referred to, a *remote stream* is returned with limited operations, as specified by the remote file protocol.

In addition to the normal **open** options, the following keywords are recognized:

:if-locked This keyword specifies the action to be taken if the specified file is locked. This keyword is not supported by the remote file protocol.

:error	Signal an error. This is the default.
:share	Open the specified file even if it is already locked, incrementing the file's lock count. This mode permits multiple processes to write to the same file simultaneously. (See the section "FEP File Locks", page 208. That section contains more information on file locks.)

:number-of-disk-blocks
The value of this keyword is the number of disk blocks to buffer internally if the **:direction** keyword is **:input** or **:output**. This keyword is ignored for other values of **:direction** or for files on remote hosts. The default **:number-of-disk-blocks** is two.

7.6 Operating on Disk Streams

All disk streams to a local FEP file system handle the following messages:

:grow &optional *n-blocks* &key **:map-area** **:zero-p** *Message*

This message allocates *n-blocks* of free disk blocks and appends them to the FEP file. The value of *n-blocks* defaults to one. If **:zero-p** is true the new blocks are filled with zeros; otherwise, they are not modified. The return value of **:grow** is the file's data map (the format of the data map is described in **:create-data-map**'s description below). The value of **:map-area** is the area to allocate the data map in, which defaults to **default-cons-area**.

:allocate *n-blocks* &key **:map-area** **:zero-p** *Message*

This message ensures that the FEP file is at least *n-blocks* long, allocating additional free blocks as required. Returns the file's data map (the format of the data map is described in **:create-data-map**'s description below). **:map-area** specifies the area to create the data map in, and defaults to **default-cons-area**. The newly allocated blocks are filled with zeros if **:zero-p** is true. **:zero-p** defaults to **nil**.

:file-access-path *Message*

This message returns the disk stream's file access path.

For example, you can find out what unit number a FEP file resides on as follows:

```
(send (send stream :file-access-path) :unit)
```

:map-block-no *block-number grow-p* *Message*

This message translates the relative file *block-number* into a disk address, and returns two values: the first value is the disk address, and the second is the total number of disk blocks, starting with *block-number*, that are in consecutive disk addresses. *grow-p* specifies whether the file should be extended if *block-number* addresses a block that does not exist. When *grow-p* is true, free disk blocks are allocated and appended to the FEP file to extend it to include *block-number*. Otherwise, if *grow-p* is false, **nil** is returned if *block-number* addresses a block that does not exist.

:create-data-map &optional *area* *Message*

This message returns a copy of the FEP file's data map allocated in area *area*, which defaults to **default-cons-area**. A FEP file data map is a one-dimensional **art-q** array. Each entry in the file data map describes a number of contiguous disk blocks, and requires two array elements. The first element is the number of disk blocks described by the entry. The second element is the disk address for the first block described by the entry. The array's fill-pointer contains the number of active elements in the data map times two.

:write-data-map *new-data-map disk-event* *Message*

This message replaces the file's data map with *new-data-map*. *disk-event* is the disk event to associate with the disk writes when the disk copy of the file's data map is updated. This message overwrites the file's contents and should be used with caution.

7.7 Input and Output Disk Streams

Input and output disk streams are buffered streams. In addition to the standard buffered stream messages, local input and output disk streams also support the messages described elsewhere: See the section "Operating on Disk Streams", page 204.

Input disk streams read bytes of data starting at the current byte position in the FEP file, updating the byte position as the data is read. Output disk streams write bytes of data in the same way.

The bytes of data are stored in buffers internal to the stream. The **:number-of-disk-blocks open** keyword controls how many disk blocks the internal buffers can hold. When the current pointer moves beyond a disk block boundary, the buffered disk block is written to the file for an output stream, or the next unbuffered block is read in from the file for an input stream. Output streams also write out all the buffered disk blocks when the stream is sent a **:close** message without an **:abort** option.

7.8 Block Disk Streams

Block disk streams can both read and write disk blocks at specified file block numbers. A file block number is the relative block offset into the file. The first block in the file is at file block number zero, the second is at file block number one, and so on.

Block disk streams do not buffer any blocks internally and are not supported by the remote file protocol.

See the section "Operating on Disk Streams", page 204. In addition to the messages described in that section, block disk streams support the following messages:

:block-length *Message*

The **:block-length** message returns the length of the FEP file in disk blocks.

:block-in *block-number n-blocks disk-arrays* &key **:hang-p :disk-event** *Message*

The **:block-in** message causes the disk to start reading data from the disk into the disk arrays in *disk-arrays*, starting with the file block number *block-number*, and continuing for *n-blocks*. *disk-arrays* can be a disk array or a list of disk arrays. The value of *n-blocks* is the number of disk blocks to read. When *n-blocks* is greater than one, each disk array is completely filled before using the next disk array in *disk-arrays*. The checkwords stored in the disk arrays are reserved for use by the FEP file system. See the section "3600-Family Disk System Definitions and Constants" in *Internals, Processes, and Storage Management.* Unused disk arrays or portions of disk arrays remain unmodified.

When the value of **:hang-p** is true, which it is by default, **:block-in** waits for all the reads to complete before returning. If the value of **:hang-p** is false, **:block-in** returns immediately upon enqueuing the disk reads without waiting for completion. In this case, all *disk-arrays* and the *disk-event* must be wired before sending the **:block-in** message, and must remain wired until the disk reads complete.

If the **:disk-event** keyword is supplied, its value is the disk event to associate with the disk reads. Otherwise the **:block-in** message allocates a disk event for its duration. A **:disk-event** must be supplied when **:hang-p** is false.

:block-out *block-number n-blocks disk-arrays* &key **:hang-p** **:disk-event** *Message*

The **:block-out** message causes the disk to start writing the data in the disk arrays in *disk-arrays* onto the disk, starting with the file block number *block-number*, and continuing for *n-blocks*. The arguments to the **:block-out** message are identical to those of the **:block-in** message.

7.9 FEP File Properties

In addition to having a name and containing data, FEP files also have properties. These properties store information about the file itself, such as when it was last written and whether it can be deleted or not. File properties are read by the **fs:file-properties** function, and modified by the **fs:change-file-properties** function. The **fs:directory-list** function also returns the file properties of several files at once. (See the section "Accessing Directories", page 160.)

The following file properties can be both read and modified:

:creation-date	The universal time the file was last written to. Universal times are integers. (See the section "Dates and Times" in *Programming the User Interface, Volume B.*)
:author	The user-id of the last writer. The user-id must be a string.
:length-in-bytes	The length of the file, expressed as an integer.
:deleted	When **t** the file is marked as being deleted. A deleted file can then be marked as being undeleted by changing this property to **nil**. The disk space used by a deleted file is not actually reclaimed until the file is expunged.

:dont-delete	When **t**, attempting to delete or overwrite the file signals an error. **nil** indicates the file can be deleted or written to.
:comment	A comment to be displayed in brackets in the directory listing. The comment must be a string.

The following file properties are returned by the **:properties** message, but cannot be modified by **:change-properties**:

:byte-size	The number of bits in a byte. The value of this property is always 8.
:length-in-blocks	The block length of the file expressed as an integer.
:directory	If **t**, the file is a directory; otherwise **nil**.

7.10 FEP File Locks

A FEP file is *locked* for the interval from when it is opened for reading or writing until it is closed. If the **:direction** keyword is **:input**, the file is *read-locked*; if the **:direction** keyword is **:output** or **:block**, the file is *write-locked*.

When the **:if-locked** keyword is **:error**, which is its default, a file that is read-locked can still be opened for reading but signals an error if opened for writing; a file that is write-locked cannot be opened for reading or writing. This permits multiple readers to access a file concurrently, while prohibiting writing to the file being read.

When the **:if-locked** keyword is **:share** in an open call for write, it succeeds in opening the file even if it is already read- or write-locked.

An expunge operation on a file that is either read- or write-locked does not expunge the file. If expunging a directory fails to expunge a file, the file must be closed and the directory expunged again.

7.11 Installing Microcode

Use the Copy Microcode command to retrieve any new microcode from the file system of the sys host.

Copy Microcode Command

Copy Microcode {*version or pathname*} *destination keywords*

Installs a version of microcode.

version or pathname	Microcode version number or pathname to copy. *version* is a microcode version number (in decimal). *pathname* rarely needs to be supplied. It defaults to a file on FEP*n*:> (where *n* is unit number of the boot disk) whose name is based on the microcode name and version. (The file resides in the logical directory sys:l-ucode;.) The *version* actually stands for the file *appropriate-hardware*-MIC.MIC.*version* on FEP*n*:>. (See the Section "Genera 7.0 Microcode Types" in *Software Installation Guide*)
destination	FEP file specification. The pathname on your FEP*n*:> directory. The default is created from the microcode version.
keywords	:update boot file

:update boot file
: {FEP-file-spec none}. The default is the current default boot file name.

Initially, the Symbolics personnel who install your system establish these microcode files for you.

7.12 Using a Spare World Load for Paging

You can reuse FEP file space for paging files. You may have a spare world load file, which you can transform into a paging file. For example, once you have successfully installed a new software release, you can rename the old world load to be a paging file. **Note:** Do not use the world load you are currently running for a paging file, as this action overwrites the previous contents of the specified file.

If your old world load is Release-6-1.load, is resident on FEP0:>, and is 36,000 blocks in size, and you want to create a new paging file called FEP0:>page2.page (with a block size of 36,000), follow these steps:

1. You should rename the file FEP0:>release-6-1.load to FEP0:>page2.page using the Rename File command. For example, type:

   ```
   Rename File FEP0:>release-6-1.load  FEP0:>page2.page
   ```

   Now the world load has been renamed to a paging file.

2. Use the Add Paging File command to initialize the paging file from the Lisp environment.

3. Edit your FEP*n*:>Boot.boot file to declare the new paging file. Use the Declare Paging-files command in your boot file to do this. For information about the Declare Paging-file command: See the section "FEP System Commands: General Usage" in *Site Operations*.

You can also create new FEP files and use them for extra paging space: See the section "Allocating Extra Paging Space", page 220.

7.13 Adding a Spare World Load as LMFS File Space

Partitions can be added to LMFS by following these steps:

1. Create the partition you wish to add to LMFS prior to entering the File system editing operations program. In addition, when you add a new partition or a partition on another disk, the disk should be free of errors and properly initialized and formatted.

2. Press `SELECT F` to select the File system editing operations program.

3. Click on [Local LMFS Operations] to invoke the second level of the File System Maintenance Program.

4. Click on [LMFS Maintenance Operations] to invoke the third level menu, which is a menu of file-system maintenance operations.

5. Click right on [Initialize] to invoke a menu of initialization options, which offers [New File System] and [Auxiliary Partition] as choices. Clicking on [New File System] is similar to clicking left on [Initialize]; it initializes a partition to be the basis of a file system.

6. Click on [Auxiliary Partition] to add another partition.

7. Enter the pathname of the FEP file to be used as the new partition. The default presented, which is correct for [New File System], is never correct for adding a partition.

8. Click on [Do It]. The system then performs much verification and error checking, roughly as much as when initializing a new partition. It must not be interrupted while performing these actions.

9. When finished, the File system editing operations program adds the partition and edits the FSPT automatically.

8. FSEdit

8.1 File System Editor

The File System Editor (FSEdit) is an interactive program that lets you examine and modify the contents of a file system. You can create directories and links, view and edit the properties of file system objects, delete objects, and expunge directories. The File System Editor is part of the File System Maintenance program, and it is the only part that most users ever use.

8.1.1 Entering the File System Editor

To get the File System Maintenance program, press SELECT F. At the top of the frame is a menu of commands. Three commands in this menu invoke the File System Editor:

- [Tree Edit Root]
- [Tree Edit Any]
- [Tree Edit home dir]

When click on one of these three commands, the big window in the frame displays a particular tree of a particular file system; that is, it displays a certain directory (the *base* directory) and some of the objects under that directory. If you use:

[Tree Edit Root]	The base directory is the root directory of the local file system; this lets you get at any file in the entire file system.
[Tree Edit Any]	You can specify the base directory by typing in its wildcard pathname; after you click on this command you are prompted for a wildcard pathname.
[Tree Edit home dir]	The base directory is your home directory. [Tree Edit home dir] prompts for a host instead of using only the "logged-in" host (the one designated during login). If you just want to try out the File System Editor, use [Tree Edit home dir].

These commands put you in the File System Editor. You never have to get "out" of it; if you want to do something unrelated to the file system, just select the window you want to use, and if you want to do something else with the File System Maintenance program, you just click on the appropriate command in the command menu.

8.1.2 Using the File System Editor

When you use [Tree Edit Root], at the top of the main window is a line reading >*.*.*. This line represents the root directory, which usually contains only directories. Below the root directory line is a set of indented lines, one representing each object in the root directory.

Move the mouse over any one of these directory lines and notice that the mouse documentation line reflects three actions that you can take:

(L)	Open object: See the section "Opening and Closing a Directory", page 212.
(M)	Close containing object: See the section "Opening and Closing a Directory", page 212.
(R)	Menu of operations: See the section "Using FSEdit Commands", page 213.

8.1.3 Opening and Closing a Directory

Now, suppose you move the mouse over the line that represents a directory, for instance >sys, and click left. That line changes to read >sys>*.*.*, and several lines are inserted just underneath it, one for each object in the >sys directory. You have just *opened* the >sys directory.

When you open a directory, a line is inserted in the display for each object in the directory. For every directory, there is a line with the pathname of the directory and nothing else; these directories are all closed. For every file, there is a line with the name, type, and version of the file, and other information about the file. For every link, there is a line with the name, type, and version of the link, followed by => and the pathname of the target of the link, and other information. See the section "How to Interpret Directory Listings", page 216.

Whenever you click left on a closed directory, FSEdit opens it and displays its contents. By clicking on successive directories inside other directories, you can move around in the file system and see what is there. The base directory is automatically opened as soon as you start using the File System Editor.

When you are finished with a directory, you can *close* the directory by clicking middle on any of the objects inside that directory. So, if you click middle on a file, that file and everything at its level disappears from the display.

Using these commands, you can get at any part of the file system underneath the base directory, and see everything that is there.

It is easy for the display to become longer than the size of the window when you move around in large directories; you can use the usual mouse scrolling commands to move the display up and down in the window. See the section "Scrolling with the Mouse" in *User's Guide to Symbolics Computers.*

8.1.4 Using FSEdit Commands

To do something to an object, click right on the object. This pops up a menu of commands, each of which specifies an action to take regarding the object. Some commands make sense for all three kinds of objects (directories, files and links); others are specific to certain kinds of objects. The menu that appears when you click right on an object offers only the options that you can apply to that type of object on your host type. For example, the menu does not display [Expunge] as an option for files or links, only for directories, and it does not display [Expunge] as option if the directory in question resides on a host that does not support soft deletion.

The following is a list of all these commands with the kind of object(s) to which each command applies:

[Delete] *(Files, directories, links)*
Marks this object for deletion. This command pertains to systems that support soft deletion, for example, Symbolics computers. This command is only displayed for objects that are not already deleted. You should not delete directories that have anything in them.

[Delete (immediate)] *(Files, directories, links)*
Deletes this object. This command pertains to systems that do not support soft deletion, for example, UNIX. This command asks for confirmation and then immediately removes the deleted object from the display. You should not delete directories that have anything in them.

[Wildcard Delete] *(Directories)*
Does wildcard deletion. This command prompts you with a default for deleting everything for the line to which the menu applies. It merges what you enter with * defaults. It lists the files it intends to delete, asks for confirmation, deletes them, reporting any errors, and updates the display.

[Undelete] *(Files, directories, links)*
Undeletes this object. This command pertains to systems that support soft deletion, for example, Symbolics computers, and is displayed only for objects that are deleted (are marked with a D).

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

[View Properties] *(Files, directories, links)*
Types out one line for each property of the object, giving the name and the value of the property. Properties are the qualities of the file that are maintained by the file system on which it resides, such as creation date and time, author, time of last access, and length. For files on a Lisp Machine file system, this means user-

defined properties as well. It prompts for the name of a file and pops up a choose-variable-values window, allowing you to alter various properties of the file. The exact properties that can be altered depend on the file system, but they might include:

- Generation (version) retention count
- Author
- Creation, modification, and reference dates
- Protection flags
- Other file-associated information

This information types out on top of the display, and prompts you to type any character when you are ready to proceed. After you type this character, the properties vanish and the FSEdit window is redisplayed. You can also use [Flush Typeout] in the command menu to make the typeout vanish; this is convenient since you do not have to move from the mouse to the keyboard.

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

[New Property] *(Files, directories, links)*
Creates a new user property for the object. You are first prompted for the name of the property, and then the value. The name is uppercased. To remove a property, give an empty string as the value.

[View] *(Files, links)*
Displays the file. The file is typed out on top of the display, and you are prompted to type any character when you are ready to proceed. The **:reference-date** of the file of the file is not changed. See the section "LMFS Properties", page 182. If the object is a link, it must be transparent to **:read** and its target must be a file; the target is printed.

[Create Inferior Directory] *(Directories)*
Creates a new directory inside this directory. You are prompted for the name (just type in the name, not the whole pathname).

[Create Link] *(Directories)*
Creates a new link inside this directory. You are first prompted for the name of the link, and then for the full pathname of the target of the link. See the section "LMFS Links", page 189.

[Expunge] *(Directories)*
Expunges the directory. See the section "LMFS Deletion, Expunging, and Versions", page 187.

[Open] *(Directories)*
Opens the directory. This is the same as clicking left on the directory name. This command is only displayed for closed directories.

[Selective open] *(Directories)*
Prompts for a wildcard name, for example, a file name containing "*" characters to indicate a wild-card component. The directory is opened and displays only those objects in the directory that match this pattern. Unspecified components default to "*". The normal [Open] command is like a [Selective open] of `*.*.*`, displaying all files. For example, if you do a [Selective open] of "`*.lisp`", only files whose type is "`lisp`" are displayed. (In this example, the version was unspecified and defaulted to "*".) The line in the display that corresponds to the directory shows this wildcard name.

[Close] *(Directories)*
Closes the directory. This is the same as clicking middle on one of the directory's inferiors. This command is only displayed for open directories.

[Link transparencies] *(Links, directories)*
Lets you change the **:link-transparencies** of a link, or the **:default-link-transparencies** of a directory.

Each link has a property called its **:link-transparencies**. The value of this property is a list of keyword symbols. Each symbol specifies an operation to which the link is transparent. If the link is transparent to an operation, that means that if the operation is performed, it will really happen to the target. If the link is not transparent to the operation, then the operation will happen to the link itself. See the section "LMFS Links", page 189.

This command displays a menu showing all of the operations to which a link can or can not be transparent. Each operation to which the link actually is transparent is highlighted with reverse video. By clicking on the name of any operation, you can turn the highlighting on or off. When you are done changing the transparencies, use [Do It], and the transparencies (or default transparencies, if this is a directory) are set. You use [Abort] to abort the operation.

[Decache] *(Directories)*
When a directory is opened, the File System Editor examines the directory, sees what is there, and remembers it. If another user changes the contents of the directory while you are in the middle of editing that directory, the File System Editor does not know that anything has changed, and so what it shows you does not really correspond to the state of the file system. Using [Decache] tells the File System Editor to forget what it thinks it knows about the contents of the directory, and makes it go back to the file system to see what is really in the directory now.

[Hardcopy] *(Files)*
Hardcopies the file. Clicking on this command causes the system hardcopy menu to pop up.

[Edit] *(Files)*
Invokes the Zmacs editor on the file.

[Load] *(Files)*
Loads the file into the Lisp world.

8.2 How to Interpret Directory Listings

The system displays the contents of directories of file systems in three contexts:

- The File System Editor
- The Show Directory (m-X) and Dired (m-X) Zmacs commands
- The Show Directory command

Contents of directories are displayed in a standard format, regardless of the context and regardless of what kind of file system (for example, Symbolics computer, TOPS-20, UNIX) the directory came from. Since this format is designed to express a great deal of information in a single line, it is rather abbreviated. Some of the ways it expresses things might not be clear without an explanation.

The basic format usually looks something like the following:

```
pal.lisp.65    7  25548(8)    03/12/85 12:42:41 (05/13/85)   dlw
```

The following is an explanation of the items in this listing:

item	*explanation*
pal	file name
lisp	file type
65	file version number
7	length of the file in blocks
25548	length of the file in bytes
8	byte-size of the file
03/12/85	date file created
12:42:41	time file created
5/13/85	date file last referred to
dlw	author

Many other things can appear in such a line; some of these things are seen only on certain types of file systems. If the first character in the line is a D, the file has been deleted (this makes sense only on file systems that support undeletion, such as the Lisp Machine and TOPS-20 file systems). After the D, if any, and before the name of the file, is the name of the physical volume that the file is stored on (on ITS, this is the disk-pack number).

On a line that describes a link rather than a file, the length numbers are replaced by an arrow (=>), followed by the name of the target of the link.

On a line that describes a subdirectory rather than a file, the length-in-blocks number is shown (if provided by the file system), but the length-in-bytes is replaced by the string `DIRECTORY`.

Next, before the dates, the line might contain any of several punctuation characters indicating things about the file. Only some of the file systems understand these flags. Following is a list of the various characters and the flags they indicate:

character	*flag*
`!`	not backed up
`@`	do not delete
`$`	do not reap

For lines indicating subdirectories, the reference date can be replaced with a date preceded by `X=`, the date this directory was last expunged. The dates are followed by the file author's name, which is followed by the name of the last user to read the file.

Only certain file systems support certain features. Many file systems do not keep track of the last reader's name and do not have something comparable to a "do not delete" flag. Therefore, any of the above fields might be omitted on certain file systems. However, the same general format is followed for all file systems and so you can interpret the meaning of a line in a directory listing, even for a file system that you are not familiar with.

9. Creating More Room on the Local Disk

There are two file systems available on the Symbolics computer: the Lisp Machine File System (LMFS) and the FEP File System (FEP FS). LMFS is a general-purpose, highly flexible file system, suitable for everyday use. Currently, only the Symbolics processor understands how to operate on LMFS files. The FEP FS is a simple, basic file system that both the Symbolics computer and front-end processors understand how to access.

The FEP FS contains two kinds of files. The first kind, called a *FEP file* is used to store information the FEP uses to do things like boot Lisp and manage virtual memory; this includes world load files, microcode load files, paging files and boot files. The second kind of file is also a FEP file, but it is a very large file and it is called a *file system partition.* One or more partitions are what LMFS uses to store its structure and data. User files are stored by LMFS in partitions.

Sometimes the Save World or Copy World commands might inform you that you have run out of FEP file system space, and offers you the option of editing your FEP directory. For systems with 167-Mbyte or more of storage, you should delete and expunge old, unneeded world loads, and then resume from the Save World "out of room" error or retry the Copy World operation. You should not delete any world loads from a 140-Mbyte system. See the section "Instructions for Managing Disk Space on the 3640 with a 140 Megabyte Disk" in *Site Operations.*

It is wise to keep a large (about 40K), noncritical world load or extra paging file on the Symbolics computer's disk, so it will be available for the FEP Disk Restore command to use in case all world loads become nonfunctional.

Sometimes, writing a file out to a LMFS produces an "out of room" error. This means that the present allocation of that particular LMFS is not large enough to accommodate your request for space. It might help to expunge directories with deleted files in them to remove unneeded versions of files, using the Zmacs command Dired (m-X).

If you still do not have enough space after you have deleted and expunged unnecessary files, consider creating an auxiliary file partition. You should only do so, however, on systems that have at least 280 Mbytes of storage. This is because 140-Mbyte systems have no room at all for an auxiliary file partition, and allocating an auxiliary file partition on a 167-Mbyte system can limit your space for large world loads. Even for 280-Mbyte systems, you are trading off world load space for file space when you create auxiliary partitions.

Be sure to reserve enough FEP file system space for a two world loads (the amount of FEP file space required for this depends on the size of the released worlds): a disk copy of your current world and a spare world load for the FEP Disk Restore command to use.

For details on how to create auxiliary file partitions: See the section "LMFS Multiple Partitions", page 191.

Warning: Once you have created an auxiliary file partition, you should never delete it, because you would lose all the data contained in that partition and make the entire Lisp Machine File System unusable.

If you run out of room while writing a LMFS file and then create a new partition to increase the LMFS space, you cannot resume the file operation that failed. Instead, you must abort that operation by pressing c-ABORT, and then retry the operation.

9.1 Allocating Extra Paging Space

Programs that use large amounts of virtual memory might require you to allocate additional paging space, to perform better or to perform at all. Only systems with at least 280 Mbytes of disk storage have enough room to permit additional paging files without adversely affecting the maintenance of worlds on the machine. In order to add an extra paging file to your virtual memory set, you must first create a FEP file using the Create FEP File command. Then, you can activate the paging file from Lisp by using the Add Paging File command. To create a 20-K block paging file on unit 0 type:

```
Create FEP File fep0:>page1.page 20000
```

After creating the extra paging file, any boot files should be modified to use this new paging file. Use the Declare Paging-files command in the boot file to load any paging files you want to use. A typical boot file before inserting the command to load the paging file might look something like this:

```
Clear Machine
Load Microcode >tmc5-row-mic.mic.384
Load World >Dist-7-0.load
Set Chaos-Address 401
Start
```

After creating the new paging file, edit your boot file to include the Declare Paging-files command. The new Boot.boot file might look something like this:

```
Clear Machine
Load Microcode >tmc5-row-mic.mic.384
Declare Paging-files fep0:page1
Load World >Dist-7-0.load

Set Chaos-Address 401
start
```

For information about the Declare Paging-files command: See the section "FEP System Commands: General Usage" in *Site Operations.*

It is safe to delete extra paging files, but only if they are not in active use. You cannot change a paging file that is being used, without booting. To change the paging area you have set up, first boot without adding the paging file to be deleted. Be sure to cold boot by hand, and when you type the Declare Paging-Files command, *do not* specify the extra paging file that you intend to delete. Once you have booted, you can delete the unwanted paging file by editing the FEP directory. Be sure to remove any references to the file from your boot file as well.

9.2 Adding a Paging File From Lisp

If you want to add a paging file from Lisp, use the new command:

```
Command:  Add Paging File
```

Prior to adding the paging file you may have to create the FEP file by using the command:

```
Create FEP File
```

to create the paging file.

Add Paging File Command

Add Paging File *pathname :prepend*

Adds a pathname as a paging file.

pathname	The pathname of the new paging file. The default pathname is the disk unit from which you most recently booted. For example, if you most recently booted from FEP1:>, the default paging file might look like: `FEP1:>.page`
keywords	:prepend
:prepend	{*yes no*} Yes means to put the paging file at the beginning of the list of swap space to use when new space is needed. This makes the new paging file used almost immediately. No, which is the default, puts the paging file at the end of the list of paging files. Consequently, this new paging file will not be used until the previous swap space is completely used.

10. Putting Data in Compiled Code Files

A compiled code file can contain data rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in a printed representation of that same data structure. Also, certain data structures, such as arrays, do not have a convenient printed representation as text, but can be saved in compiled code files.

In compiled programs, the constants are saved in the compiled code file in this way. The compiler optimizes by making constants that are **zl:equal** become **eq** when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a compiled code file is loaded, objects that were **eq** when the file was written are still **eq**; this does not normally happen with text files.

The following types of objects can be represented in compiled code files:

Symbols
Numbers of all kinds
Lists
Strings
Arrays of all kinds
Instances (for example, hash tables)
Compiled function objects

When an instance is put (dumped) into a compiled code file, it is sent a **:fasd-form** message, which must return a Lisp form that, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply dumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain **eq**; the **:fasd-form** message is sent only the first time a particular instance is encountered during writing of a compiled code file. If the instance does not accept the **:fasd-form** message, it cannot be dumped.

sys:dump-forms-to-file *filename forms* &optional *file-attribute-list* *Function*

sys:dump-forms-to-file writes data to a file in binary form. *forms-list* is a list of Lisp forms, each of which is dumped in sequence. It dumps the forms, not their results. The forms are evaluated when you load the file.

For example, suppose **a** is a variable bound to any Lisp object, such as a list or array. The following example creates a compiled code file that recreates the variable **a** with the same value:

```
(sys:dump-forms-to-file "f:>foo>aval"
       (list `(setq a ',a)))
```

For the purposes of understanding what this function does, you can consider that it is the same as the following:

```
(defun sys:dump-forms-to-file (file forms)
   (with-open-file (s file ':direction ':output)
      (dolist (f forms)
        (print f s))))
```

The actual definition (which is more complicated) writes a binary file in a more easily parsed format so it will load faster. It can also dump arrays, which you cannot write to a Lisp source file.

attribute-list supplies an optional attribute list for the resulting compiled code file. It has basically the same result when loading the binary file as the file attribute list does for **compiler:compile-file**. Its most important application is for controlling the package that the file is loaded into.

```
(sys:dump-forms-to-file "foo" forms-list '(:package "user"))
```

sys:dump-forms-to-file always puts a package attribute into the binary file it writes. If you do not specify the *attribute-list* argument, or if *attribute-list* does not contain a **:package** attribute, the function uses the **cl-user** or **zl-user** package, depending on the context. This is to ensure that package prefixes on symbols are always interpreted when they are loaded as they were intended when the file was dumped.

The *file-attribute-list* argument can be used to store useful information (such as "headers" for special data structures) in the file's attribute list. The information can then be retrieved from the attribute list with **fs:pathname-attribute-list**, without reading the rest of the file.

PART IV.

Input/Output Facilities

11. How the Reader Works

The purpose of the reader is to accept characters, interpret them as the printed representation of a Lisp object, and return a corresponding Lisp object. The reader cannot accept everything that the printer produces; for example, the printed representations of arrays (other than strings), compiled code objects, closures, stack groups, and so on cannot be read in. However, it has many features that are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently used unwieldy constructs.

In general, the reader operates by recognizing *tokens* in the input stream. Tokens can be self-delimiting or can be separated by delimiters such as whitespace. A token is the printed representation of an atomic object such as a symbol or a number, or a special character such as a parenthesis. The reader reads one or more tokens until the complete printed representation of an object has been seen, and then constructs and returns that object.

11.1 What the Reader Recognizes

11.1.1 How the Reader Recognizes Symbols

A string of letters, numbers, and "extended alphabetic" characters is recognized by the reader as a symbol, provided it cannot be interpreted as a number. Alphabetic case is ignored in symbols; lowercase letters are translated to uppercase. When the reader sees the printed representation of a symbol, it *interns* it on a *package*. See the section "Packages" in *Symbolics Common Lisp: Language Concepts*.

Symbols can start with digits; for example, **zl:read** accepts one named "345T". If you want to put strange characters (such as lowercase letters, parentheses, or reader macro characters) inside the name of a symbol, put a slash before each strange character. If you want to have a symbol whose print-name looks like a number, put a slash before some character in the name. You can also enclose the name of a symbol in vertical bars, which quotes all characters inside, except vertical bars and slashes, which must be quoted with slash.

Examples of symbols:

```
foo
bar/(baz/)
34w23
|Frob Sale|
```

When a token could be read as either a symbol or an integer in a base larger than ten, the reader's action is determined by the value of

si:*read-extended-ibase-unsigned-number* and **si:*read-extended-ibase-signed-number***.

11.1.2 How the Reader Recognizes Macro Characters

Certain characters are defined to be macro characters. When the reader sees one of these, it calls a function associated with the character. This function reads whatever syntax it likes and returns the object represented by that syntax. Macro characters are always token delimiters; however, they are not recognized when quoted by slash or vertical bar, nor when inside a string. Macro characters are a syntax-extension mechanism available to the user. Lisp comes with several predefined macro characters:

Quote (')	An abbreviation to make it easier to put constants in programs. *'foo* reads the same as **(quote** *foo***)**.
Semicolon (;)	Used to enter comments. The semicolon and everything up through the next carriage return are ignored. Thus a comment can be put at the end of any line without affecting the reader.
Backquote (`)	Makes it easier to write programs to construct lists and trees by using a template. See the section "Backquote" in *Symbolics Common Lisp: Language Concepts.*
Comma (,)	Part of the syntax of backquote. It is invalid if used other than inside the body of a backquote. See the section "Backquote" in *Symbolics Common Lisp: Language Concepts.*
Sharp sign (#)	Introduces a number of other syntax extensions. See the section "Sharp-sign Reader Macros", page 229. Unlike the preceding characters, sharp sign is not a delimiter. A sharp sign in the middle of a symbol is an ordinary character.

The function **zl:set-syntax-macro-char** can be used to define your own macro characters.

Reader macros that call a read function should call **si:read-recursive**.

si:read-recursive *stream* *Function*

> **si:read-recursive** should be called by reader macros that need to call a function to read. It is important to call this function instead of **zl:read** in macros that are written in Zetalisp but used by the Common Lisp readtable. In particular, this function must be called by macros used in conjunction with the Common Lisp #*n*= and #*n*# syntaxes.
>
> *stream* is the stream from which to read. This function can be called only from inside a **zl:read**.

For example, this is the reader macro called when the reader sees a quote ('):

```
si:(defun xr-quote-macro (list-so-far stream)
     list-so-far                                  ;not used
     (values (list-in-area read-area
                           'quote (read-recursive stream))
             'list))
```

11.2 Sharp-sign Reader Macros

The reader's syntax includes several abbreviations introduced by sharp sign (#). These take the general form of a sharp sign, a second character that identifies the syntax, and following arguments. Certain abbreviations allow a decimal number or certain special "modifier" characters between the sharp sign and the second character.

The function **zl:set-syntax-#-macro-char** can be used to define your own sharp-sign abbreviations.

#\ or #/#\x (or #/*x* in Zetalisp) reads in as the character *x*. For example, #**a**. This is the recommended way to include character constants in your code. Note that the backslash causes this construct to be parsed correctly by the editor.

As in strings, upper- and lowercase letters are distinguished after #\. Any character works after #\, even those that are normally special to **read**, such as parentheses.

#*name* (or #/*name*) reads in as the name for the nonprinting character symbolized by *name*. A large number of character names are recognized. See the section "Special Character Names", page 234. For example, #**return** reads in as an integer, being the character code for the Return character in the Genera character set. In general, the names that are written on the keyboard keys are accepted. The abbreviations #**cr** for #**return** and #**sp** for #**space** are accepted and generally preferred, since these characters are used so frequently. The page separator character is called #**page**, although #**form** and #**clear-screen** are also accepted since the keyboard has one of those legends on the page key. The rules for reading *name* are the same as those for symbols; thus upper- and lowercase letters are not distinguished, and the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

When the system types out the name of a special character, it uses the same table as the #\ reader; therefore, any character name typed out is acceptable as input.

#\ (or #/) can also be used to read in the names of characters that have control and meta bits set. The syntax looks like #**control-meta-b** to get a "B" character with the control and meta bits set. You can use any of the prefix bit names **control, meta, hyper**, and **super**. They can be in any order, and upper- and lowercase letters are not distinguished. The last hyphen can be followed by a single character, or by any of the special character names normally recognized by #\. If it is a single character, it is treated the same way the reader normally treats characters in symbols; if you want to use a lowercase character or a special character such as a parenthesis, you must precede it with a slash character. Examples: #**hyper-super-a**, #**meta-hyper-roman-i**, #**ctrl-meta-/(**.

The character can also be modified with control and meta bits by inserting one or more special characters between the # and the \. This syntax is obsolete since it is not mnemonic and it generally unclear. However, it is used in some old programs, so here is how it is defined. #α\x generates Control-*x*. #β\x generates Meta-*x*. #π\x generates Super-*x*. #λ\x generates Hyper-*x*. These can be combined, for instance #πβ\& generates Super-Meta-ampersand. Also, #ε\x is an abbreviation for #αβ\x. When control bits are specified, and *x* is a lowercase alphabetic character, the character code for the uppercase version of the character is produced.

#^ #^*x* is exactly like #α/*x* if the input is being read by Zetalisp; it generates Control-*x*. In Maclisp *x* is converted to uppercase and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by **zl:tyi** if the user holds down the control key and types *x*. (In Maclisp #α/*x* sets the bit set by the CONTROL key when the TTY is open in **zl:fixnum** mode.)

NOTE: #^ Reader Macro is supported in Zetalisp only.

#' #'*foo* is an abbreviation for **(function** *foo***)**. *foo* is the printed representation of any object. This abbreviation can be remembered by analogy with the ' macro character, since the **function** and **quote** special forms are somewhat analogous.

#, #,*foo* evaluates *foo* (the printed representation of a Lisp form) at read time, unless the compiler is doing the reading, in which case it is arranged that *foo* be evaluated when the QFASL file is loaded. This is a way, for example, to include in your code complex list-structure constants that cannot be written with **quote**. Note that the reader does not put **quote** around the result of the evaluation. You must do this yourself, typically by using the ' macro-character. An example of a case where you do not want **quote** around it is when this object is an element of a constant list.

#. #.*foo* evaluates *foo* (the printed representation of a Lisp form) at read time, regardless of who is doing the reading.

#: #:*name* reads *name* as an uninterned symbol. It always creates a new symbol. Like all package prefixes, #: can be followed by any expression. Example: **#:(a b c)**.

#b #b*rational* reads *rational* (an integer or a ratio) in binary (radix 2). Examples:

```
#B1101 <=> 13.
#B1100\100 <=> 3
```

#o #o *number* reads *number* in octal regardless of the setting of **zl:ibase**. Actually, any expression can be prefixed by #o; it is read with **zl:ibase** bound to 8.

#x #x *number* reads *number* in radix 16. (hexadecimal) regardless of the setting of *ibase*. As with #o, any expression can be prefixed by #x. The *number* can contain embedded hexadecimal "digits" A through F as well as the 0 through 9. See the section "Reading Integers in Bases Greater Than 10" in *Symbolics Common Lisp: Language Concepts.*

#r #*radix***R** *number* reads *number* in radix *radix* regardless of the setting of **zl:ibase**. As with #o, any expression can be prefixed by #*radix***R**; it is read with **zl:ibase** bound to *radix*. *radix* must consist of only digits, and it is read in decimal. *number* can consist of both numeric and alphabetic digits, depending upon *radix*.

For example, **#3R102** is another way of writing **11.** and **#11R32** is another way of writing **35.**

#Q #Q *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace).

Note: #Q is supported only for Zetalisp.

#M #M *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace).

Note: #M is supported only for Zetalisp.

#N #N *foo* reads as *foo* if the input is being read into NIL or compiled to run in NIL, otherwise it reads as nothing (whitespace). Also, during the reading of *foo*, the reader temporarily defines various NIL-compatible sharpsign reader macros (such as #! and #") in order to parse the form correctly, even though it is not going to be evaluated.

Note: #N is supported only for Zetalisp.

#+ This abbreviation provides a read-time conditionalization facility similar to, but more general than, that provided by #m, #n, and #q. It is used as *#+feature form.* If *feature* is a symbol, then this is read as *form* if

(status feature *feature***)** is **t**. If **(status feature** *feature***)** is **nil**, then this is read as whitespace. Alternately, *feature* can be a boolean expression composed of **and**, **or**, and **not** operators and symbols representing items which can appear on the **(status features)** list. **(or lispm amber)** represents evaluation of the predicate **(or (status feature lispm) (status feature amber))** in the read-time environment.

For example, #+**lispm** *form* makes *form* exist if being read by Symbolics Common Lisp, and is thus equivalent to #**q** *form*. Similarly, #+**maclisp** *form* is equivalent to #**m** *form*. #+**(or lispm nil)** *form* makes *form* exist on either Symbolics Common Lisp or in NIL. Note that items can be added to the **(status features)** list by means of **(sstatus feature** *feature***)**, thus allowing the user to selectively interpret or compile pieces of code by parameterizing this list. See the section "**zl:status** And **zl:sstatus**" in *User's Guide to Symbolics Computers*.

#- *#-feature form* is equivalent to #+(**not** *feature*) *form*.

#| #| begins a comment for the Lisp reader. The reader ignores everything until the next |#, which closes the comment. Note that if the |# is inside a comment that begins with a semicolon, it is *not* ignored; it closes the comment that began with the preceding #|. #| and |# can be on different lines, and #|...|# pairs can be nested.

Using #|...|# always works for the Lisp reader. The editor, however, does not understand the reader's interpretation of #|...|#. Instead, the editor retains its knowledge of Lisp expressions. Symbols can be named with vertical bars, so the editor (not the reader) behaves as if #|...|# is the name of a symbol surrounded by pound signs, instead of a comment.

Note: Use #||...||# instead of #|...|# to comment out Lisp code.

The reader views #||...||# as a comment: the comment prologue is #|, the comment body is |...|, and the comment epilogue is |#. The editor, however, interprets #||...||# as a pound sign (#), a symbol with a zero-length print name (||), Lisp code (...), another symbol with a zero length print name (||), and a stray pound sign (#). Therefore, inside a #||...||#, the editor commands that operate on Lisp code, such as balancing parentheses and indenting code, work correctly.

#< This is not valid reader syntax. It is used in the printed representation of objects that cannot be read back in. Attempting to read a #< causes an error.

#◊ #◊ turns infix expression syntax into regular Lisp code. It is intended for people who like to use traditional arithmetic expressions in Lisp code. It is not intended to be extensible or to be a full programming language. We do not intend to extend it into one.

```
(defun my-add (a b)
  #◊a+b◊)
```

The quoting character is backslash. It is necessary for including special symbols (such as -) in variable names.

zl-user:! reads one Lisp expression, which can use this reader-macro inside itself.

#◊ supports the following syntax:

Delimiters
: Begin the reader macro with #◊, complete it with **zl-user:◊**.

```
#◊a+b-c◊
```

Escape characters
: Special characters in symbol names must be preceded with backslash (\). You can escape to normal Lisp in an infix expression; precede the Lisp form with exclamation point (**zl-user:!**).

Symbols
: Start symbols with a letter. They can contain digits and underscore characters. Any other characters need to be quoted with **zl:\\**.

Operators
: It accepts the following classes of operators. Arithmetic operator precedence is like that in FORTRAN and PL/I.

Operator	*Infix*	*Lisp Equivalent*
Assignment	x : y	(setf x y)
Functions	f(x,y)	(f x y) -- also works for defstruct accessors, and so on.
Array ref	a[i,j]	(aref a i j)
Unary ops	+ - not	*same*
Binary ops	+ - * / ^ = ≠ < ≤ > ≥ and or	*same*
Conditional	if p then c	(if p c)
	if p then c else a	(if p c a)
Grouping:	(a, b, c)	(progn a b c) -- even works for (1+2)/3

The following example shows matrix multiplication using an infix expression.

```
(defun matrix-multiply (a b)
  (let ((n (array-dimension-n 2 a)))
    (unless (= n (array-dimension-n 1 b))
      (ferror "Matrices ~S and ~S do not have compatible dimensions") a b)
    (let ((d1 (array-dimension-n 1 a))
          (d2 (array-dimension-n 2 b)))
      (let ((c #◊ make\-array(list(d1, d2), !:type, art\-float)◊ ))
        (dotimes (i d1)
          (dotimes (j d2)
            #◊ c[i,j] : !(loop for k below n sum #◊ a[i,k]*b[k,j] ◊)◊))
        c))))
```

The line containing the infix expression could also have been written like this:

```
(let ((sum 0))
  (dotimes (k n) #◊ sum:sum+a[i,k]*b[k,j] ◊)
  #◊ c[i,j]:sum ◊)
```

11.3 Special Character Names

The following are the recognized special character names, in alphabetical order except with synonyms together and linked with equal signs. These names can be used after a #\ to get the character code for that character. Most of these characters type out as this name enclosed in a lozenge.

The special characters are:

```
Null                Tab                 Abort               Hand-Up
Suspend=Break       Line=Linefeed       Resume              Hand-Left
Clear-Input=Clear   Refresh=Clear-Screen                    Hand-Right
Call                Page=Formfeed       Status              Select=System
Function=Terminal   Return=CR=Newline   End                 Network
Macro=Backnext      Quote               Square=Roman-I      Escape=Altmode
Help                Hold-Output         Circle=Roman-II     Complete
Rubout              Stop-Output         Triangle=Roman-III  Symbol-Help=Top-Help
Back-Space=Overstrike                   Roman-IV
```

The following are special characters sometimes used to represent single and double mouse clicks. The buttons can be called either **l**, **m**, **r** or **1**, **2**, **3** depending on stylistic preference.

Mouse-L-1=Mouse-1-1	Mouse-L-2=Mouse-1-2
Mouse-M-1=Mouse-2-1	Mouse-M-2=Mouse-2-2
Mouse-R-1=Mouse-3-1	Mouse-R-2=Mouse-3-2

11.4 The Readtable

A data structure called the *cl:*readtable** (or readtable) is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard Lisp meanings to all the characters, but you can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the symbol ***readtable***.

readtablep *object* *Function*

Returns **t** if *object* is a readtable, otherwise it returns **nil**.

readtable *Variable*

The value of ***readtable*** is the current readtable. The initial value of this is a readtable set up for standard Common Lisp syntax. You can bind this variable to temporarily change which readtable is being used.

zl:readtable *Variable*

The value of **zl:readtable** is the current readtable. This starts out as a copy of **si:initial-readtable**. You can bind this variable to temporarily change the readtable being used.

si:initial-readtable *Variable*

The value of **si:initial-readtable** is the initial standard readtable. You should never change the contents of either this readtable or **si:initial-readtable**; only examine it, by using it as the *from-readtable* argument to **zl:copy-readtable** or **zl:set-syntax-from-char**. Change **zl:readtable** instead.

You can program the reader by changing the readtable in any of three ways.

- You can create a completely new readtable, using the readtable compiler (sys:io;rtc) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.

- The syntax of a character can be set to one of several predefined possibilities.

- A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function that is called when the character is read.

11.4.1 Functions That Create New Readtables

copy-readtable &optional (*from-readtable* ***readtable***) *to-readtable* *Function*

A copy is made of *from-readtable*, which defaults to the current readtable (the value of the global variable ***readtable***). If *from-readtable* is **nil**, then a copy of a standard Common Lisp readtable is made. For example,

```
(setq *readtable* (copy-readtable nil))
```

will restore the input syntax to standard Common Lisp syntax, even if the original readtable has been clobbered.

If *to-readtable* is unsupplied or **nil**, a fresh copy is made. Otherwise, *to-readtable* must be a readtable, which is destructively copied into.

zl:copy-readtable &optional *from-readtable to-readtable* *Function*

from-readtable, which defaults to the current readtable, is copied. If *to-readtable* is unsupplied or **nil**, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copy. Use **zl:copy-readtable** to get a private readtable before using the other readtable functions to change the syntax of characters in it. The value of **zl:readtable** at the start of a session is the initial standard readtable, which usually should not be modified.

11.4.2 Functions That Change Character Syntax

set-syntax-from-char *to-char from-char* &optional (*to-readtable* ***readtable***) *from-readtable* *Function*

This makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. The *to-readtable* defaults to the current readtable (the value of the global variable ***readtable***), and *from-readtable* defaults to **nil**, meaning to use the syntaxes from the standard Lisp readtable.

The attributes *whitespace*, *constituent*, *macro* and *escape* are copied. If a *macro character* is copied, the macro definition is also copied. The attributes *alphabetic* and *alphadigit*, as well as marker characteristics such as plus sign, dot and float exponent marker, are not copied, since they are "hard-wired" into the extended-token parser. For example, if the definition of **s** is copied to *****, then ***** will become a constituent that is alphabetic but cannot be used as an exponent indicator for short-format floating-point number syntax.

You can copy a macro definition from a character such as " to another character and expect it to work properly, since the standard definition for " looks for another character that is the same as the character that invoked it. You probably don't want to copy the definition of (to {, since it lets you write lists in the form {a b c), not {a b c}, because the definition always looks for a closing parenthesis, not a closing brace.

zl:set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable* *Function*

Makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable. to-readtable* defaults to the current readtable, and *from-readtable* defaults to the initial standard readtable.

set-character-translation *from-char to-char* &optional *readtable* *Function*

Changes *readtable* so that *from-char* is translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lowercase letters to uppercase. Character translations are turned off by slash, string quotes, and vertical bars. *readtable* defaults to the current readtable.

zl:set-syntax-from-description *char description* &optional *readtable* *Function*

Sets the syntax of *char* in *readtable* to be that described by the symbol *description.* The following descriptions are defined in the standard readtable:

si:alphabetic	An ordinary character such as "a".
zl:break	A token separator such as "(". (Obviously left parenthesis has other properties besides being a break.)
si:whitespace	A token separator that can be ignored, such as "@".
si:single	A self-delimiting single-character symbol. The initial readtable does not contain any of these.
si:slash	The character quoter. In the initial readtable this is "/".
si:verticalbar	The symbol print-name quoter. In the initial readtable this is "|".
si:doublequote	The string quoter. In the initial readtable this is '"'.
macro	A macro character. Do not use this; use **zl:set-syntax-macro-char**.
si:circlecross	The octal escape for special characters. In the initial readtable this is "⊗". (**si:circlecross** exists only the the standard Zetalisp readtable, not the Symbolics Common Lisp readtable.)
si:bitscale	A character that causes the integer to its left to be doubled the number of times indicated by the integer to its right. In the initial readtable this is "_". See the section "What the Reader Recognizes", page 227.

si:digitscale	A character that causes the integer to its left to be multiplied by **zl:ibase** the number of times indicated by the integer to its right. In the initial readtable this is "^". See the section "What the Reader Recognizes", page 227.
si:non-terminating-macro	A macro character that is not a token separator. This is a macro character if seen alone but is just a symbol constituent inside a symbol. You can use it as a character of a symbol other than the first without slashing it. (# would be one of these if it were not built into the reader.)

readtable defaults to the current readtable.

11.4.3 Functions That Change Characters Into Macro Characters

make-dispatch-macro-character *char* &optional *non-terminating-p* (*a-readtable* ***readtable***) *Function*

Causes *char* to be a dispatching macro character in *readtable.* If *non-terminating-p* is non-**nil** (it defaults to **nil**), then it will be a non-terminating macro character, which means that it may be embedded within extended tokens. **make-dispatch-macro-character** returns **t**.

Initially, every character in the dispatch table has a character-macro function that signals an error. Use **set-dispatch-macro-character** to define entries in the dispatch table.

set-dispatch-macro-character *disp-char sub-char function* &optional (*a-readtable* ***readtable***) *Function*

Causes *function* to be called when the *disp-char* followed by *sub-char* is read. *function* is called with three arguments, a stream, *sub-char,* and the non-negative integer whose decimal representation appears between *disp-char* and *sub-char,* or **nil** if no decimal integer appeared there. **set-dispatch-macro-character** returns **t**.

An error is signalled if *sub-char* is one of the ten decimal digits, since they are reserved for specifying an infix integer argument. Moreover, if *sub-char* is a lowercase character, its uppercase equivalent is used instead. This is how the rule is enforced that the case of a dispatch sub-character doesn't matter.

An error is also signalled if the specified *disp-char* is not a dispatch character in the specified readtable. It is necessary to use **make-dispatch-macro-character** to set up the dispatch character before specifying its sub-characters.

As an example, the definition of the sharp-sign single-quote dispatch macro character is:

```
(defun sharp-single-quote-reader (stream sub-char arg)
  (declare (ignore char arg))
  (list-in-area 'sys:read-area 'function
    (read stream t nil t)))

(set-dispatch-macro-character #\# #\'  #'sharp-single-quote-reader)
```

sharp-single-quote-reader reads an object following the single-quote and returns a list of the symbol **function** and that object. The *char* and *arg* arguments are ignored for this function. Note that the *recursive-p* argument to **read** is **t**, which means that this call to **read** is imbedded, not top-level.

get-dispatch-macro-character *disp-char sub-char* &optional (*a-readtable* ***readtable***) *Function*

Returns the macro-character function for *sub-char* under *disp-char*, or **nil** if there is no function associated with sub-char. If *sub-char* is one of the ten decimal digits, **get-dispatch-macro-character** always returns **nil**. If *sub-char* is a lowercase character, its uppercase equivalent is always used instead.

An error is signalled if the specified *disp-char* is not a dispatch character in the specified readtable.

```
(get-dispatch-macro-character #\# #\') =>
#<LEXICAL-CLOSURE (:INTERNAL GET-DISPATCH-MACRO-CHARACTER 0)
     36057616>

(get-dispatch-macro-character #\# #\1) => NIL
```

Note that because **get-dispatch-macro-character** returns a lexical closure, subsequent calls will not necessarily return the same object. This may be changed in a future release.

set-macro-character *char function* &optional *non-terminating-p* (*a-readtable* ***readtable***) *Function*

Causes *char* to be a macro character that causes *function* to be called when it is seen by the reader. If *non-terminating-p* is not **nil** (it defaults to **nil**), then it will be a non-terminating macro character, which means that it may be embedded within extended tokens. **set-macro-character** returns **t**.

function is called with two arguments, *stream* and *char*. *stream* is the input stream, and *char* is the macro character itself. In the simplest case, *function* returns a Lisp object. This object is taken to be that whose printed representation was the macro character and any following characters read by the *function*. As an example, the definition of the single-quote macro character is:

```
(defun single-quote-reader (stream char)
  (declare (ignore char))
  (list-in-area 'sys:read-area 'quote (read stream t nil t)))

(set-macro-character #\' #'single-quote-reader)
```

single-quote-reader reads an object following the single-quote and returns a list of the symbol **quote** and that object. The *char* argument is ignored for this function. Note that the *recursive-p* argument to **read** is **t**, which means that this call to **read** is imbedded, not top-level.

function should not have any side effects other than on *stream*. Because of backtracking and restarting of the **read** operation, front ends to the reader, such as editors and rubout handlers, can cause *function* to be called repeatedly during the reading of a single expression in which the macro character only appears once.

get-macro-character *char* &optional (*a-readtable* ***readtable***) *Function*

Returns two values: the function associated with *char*, and the value of the *non-terminating-p* flag. It returns just the symbol **nil** if *char* does not have macro-character syntax. For example:

```
(get-macro-character #\') =>
#<LEXICAL-CLOSURE (INTERNAL GET-MACRO-CHARACTER 0) 16433170>
NIL

(get-macro-character #\-) => NIL
```

Note that because **get-macro-character** returns a lexical closure, subsequent calls will not necessarily return the same object. This may be changed in a future release.

zl:set-syntax-macro-char *char function* &optional *readtable non-terminating-p* *Function*

Changes *readtable* so that *char* is a macro character. When *char* is read, *function* is called. *readtable* defaults to the current readtable.

function is called with two arguments: *list-so-far* and the input stream. When a list is being read, *list-so-far* is that list (**nil** if this is the first element). At the "top level" of **zl:read**, *list-so-far* is the symbol **:toplevel**. After a dotted-pair dot, *list-so-far* is the symbol **:after-dot**. *function* can read any number of characters from the input stream and process them however it likes.

function should return three values, called *thing*, *type*, and *splice-p*. *thing* is the object read. If *splice-p* is **nil**, *thing* is the result. If *splice-p* is non-**nil**, then when reading a list *thing* replaces the list being read – often it is *list-so-far* with something else **nconc**'ed onto the end. At top level and after a dot if *splice-p* is non-**nil** the *thing* is ignored and the macro character does not contribute anything to the result of **zl:read**. *type* is a historical artifact and is not really used; **nil** is a safe value. Most macro character functions return just one value and let the other two default to **nil**.

function should not have any side effects other than on the stream and *list-so-far*. Because of the way the input editor works, *function* can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called **:macro** syntax).

If *non-terminating-p* is **nil** (the default), **zl:set-syntax-macro-char** makes a normal macro character. If it is **t**, **zl:set-syntax-macro-char** makes a nonterminating macro character. A nonterminating macro character is a character that acts as a reader macro if seen between tokens, but if seen inside a token it acts as an ordinary letter; it does not terminate the token.

zl:set-syntax-#-macro-char *char function* &optional *readtable* *Function*

Causes *function* to be called when #*char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters. When *function* is called, the special variable **si:xr-sharp-argument** contains **nil** or a number that is the number or special bits between the # and *char*.

11.4.4 Readtable Functions for Maclisp Compatibility

zl:setsyntax *character arg2 arg3* *Function*

This exists only for Maclisp compatibility. The other readtable functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be an integer, a symbol, or a string, that is, anything acceptable to the **character** function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro	The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, can **zl:tyi** or **zl:read** from **zl:standard-input** (that is, can call **zl:tyi** or

	zl:read without specifying a stream), and returns an object that is taken as the result of the read.
:splicing	Like **:macro**, but the object returned by the macro function is a list that is **nconce**d into the list being read. If the character is read not inside a list (at top level or after a dotted-pair dot), then it can return **()**, which means it is ignored, or (*obj*), which means that *obj* is read.
:single	The character becomes a self-delimiting single-character symbol. If *arg3* is an integer, the character is translated to that character.
nil	The syntax of the character is not changed, but if *arg3* is an integer, the character is translated to that character.
a symbol	The syntax of the character is changed to be the same as that of the character *arg2* in the standard initial readtable. *arg2* is converted to a character by taking the first character of its print name. Also if *arg3* is an integer, the character is translated to that character.

zl:setsyntax-sharp-macro *character type function* &optional *readtable* *Function*

This exists only for Maclisp compatibility. **zl:set-syntax-#-macro-char** is preferred. If *function* is **nil**, #*character* is turned off, otherwise it becomes a macro that calls *function*. *type* can be **:macro**, **:peek-macro**, **:splicing**, or **:peek-splicing**. The splicing part controls whether *function* returns a single object or a list of objects. Specifying peek causes *character* to remain in the input stream when *function* is called; this is useful if *character* is something like a left parenthesis. *function* gets one argument, which is **nil** or the number between the # and the *character*.

12. Input Functions

Most of these functions take an optional argument to specify the stream from which to take characters, called *input-stream* in Common Lisp and *stream* in Zetalisp. *input-stream* is the stream from which the input is to be read; if unsupplied it defaults to the value of ***standard-input***. The special pseudostreams **nil** and **t** are also accepted. **nil** means the value of ***standard-input*** (that is, the default) and **t** means the value of ***terminal-io*** (that is, the interactive terminal). See the section "Introduction to Streams", page 5. Streams are documented in detail in that section.

These functions also take optional end-of-file arguments. In Common Lisp, the options are *eof-error-p* and *eof-value*. *eof-error-p* controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is **t** (the default), an error will be signalled at the end of a file. If it is **nil**, then no error is signalled, and instead the function returns *eof-value*.

In Zetalisp, if no *eof-option* argument is supplied, an error is signalled. If there is an *eof-option*, it is the value to be returned. Note that an *eof-option* of **nil** means to return **nil** if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

Functions such as **read** that read an "object" rather than a single character always signal an error, regardless of *eof-error-p* or *eof-option*, if the file ends in the middle of an object. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** complains. If a file ends in a symbol or a number immediately followed by end-of-file, **read** reads the symbol or number successfully and when called again, sees the end-of-file and obeys *eof-error-p*. If a file contains ignorable text at the end, such as blank lines and comments, **read** does not consider it to end in the middle of an object and obeys *eof-error-p*.

Note that all of these functions except **zl:readline-no-echo** echo their input if used on an interactive stream (one that supports the **:input-editor** operation. The functions that input more than one character at a time (**zl:read**, **zl:readline**) allow the input to be edited using rubout. **zl:tyipeek** echoes all of the characters that were skipped over if **zl:tyi** would have echoed them; the character not removed from the stream is not echoed either.

12.1 Input Functions That Work on Streams

The following functions work on input or bidirectional streams:

read &optional *input-stream* (*eof-error-p* **t**) *eof-value recursive-p* *Function*

Reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object.

The optional arguments *input-stream*, *eof-error-p*, *eof-value* and *recursive-p* affect how **read** reads and interprets the incoming information. *input-stream* is the stream from which to obtain input. If unsupplied or **nil**, it defaults to the value of the special variable ***standard-input***. If **t**, it becomes the value of the special variable ***terminal-io***.

eof-error-p controls what happens if input is from a file (or any other input source that has a definite end) and the end of file is reached. If *eof-error-p* is **t** (the default), an error is signalled at the end of file (EOF). If it is **nil**, then no error is signalled, and instead **read** returns *eof-value*.

Because **read** reads the representation of an object rather than a single character, it always signals an error, regardless of *eof-error-p*, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** will complain. If a file ends in a symbol or a number, immediately followed by EOF, **read** will read the symbol or number successfully and when called again will see the EOF and only then act according to *eof-error-p*. If a file contains ignorable text at the end, such as blank lines and comments, **read** will not consider it to end in the middle of an object. Thus an *eof-error-p* argument controls what happens when the file ends *between* objects.

If *recursive-p* is specified and non-**nil**, this argument specifies that this call is not a top-level call to **read**, but an imbedded call. This typically happens from the function for a macro character. For more information on how *recursive-p* affects input functions: See the section "Input Functions", page 243.

The corresponding output function is **write**.

zl:read &optional (*stream* **zl:standard-input**) *eof-option* *Function*

Reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. For details: See the section "Input Functions", page 243.

(This function can take its arguments in the other order, for Maclisp compatibility only.)

sys:read-character &optional *stream* &key (*fresh-line* **t**) (*any-tyi* **nil**) (*eof* **nil**) (*notification* **t**) (*prompt* **nil**) (*help* **nil**) (*refresh* **t**) (*suspend* **t**) (*abort* **t**) (*status* **nil**) *Function*

Reads and returns a single character from *stream.* This function displays notifications and help messages and reprompts at appropriate times. It is used by **fquery** and the **:character** option for **prompt-and-read.**

stream must be interactive. It defaults to **zl:query-io.**

Following are the permissible keywords:

:fresh-line	If not **nil**, the function sends the stream a **:fresh-line** message before displaying the prompt. If **nil**, it does not send a **:fresh-line** message. The default is **t.**
:any-tyi	If not **nil**, the function returns blips. If **nil**, blips are treated as the **:tyi** message to an interactive stream treats them. The default is **nil.**
:eof	If not **nil** and the function encounters end-of-file, it returns **nil.** If **nil** and the function encounters end-of-file, it beeps and waits for more input. The default is **nil.**
:notification	If not **nil** and a notification is received, the function displays the notification and reprompts. If **nil** and a notification is received, the notification is ignored. The default is **t.**
:prompt	If **nil**, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor", page 278. The default is **nil.**
:help	If not **nil**, the value should be a help option. See the section "Displaying Help Messages in the Input Editor", page 279. Then, when the user presses HELP, the function displays the help option and reprompts. If **nil** and the user presses HELP, the function just returns **#\help.** The default is **nil.**
:refresh	If not **nil** and the user presses REFRESH, the function sends the stream a **:clear-window** message and reprompts. If **nil** and the user presses REFRESH, the function just returns **#\refresh.** The default is **t.**
:suspend	If not **nil** and the user types one of the **sys:kbd-standard-suspend-characters**, a **zl:break** loop is

	entered. If **nil** and the user types a suspend character, the function just returns the character. The default is **t**.
:abort	If not **nil** and the user types one of the **sys:kbd-standard-abort-characters**, **sys:abort** is signalled. If **nil** and the user types an abort character, the function just returns the character. The default is **t**.
:status	This option takes effect only if the stream is a window. If the value is **:selected** and the window is no longer selected, the function returns **:status**. If the value is **:exposed** and the window is no longer exposed or selected, the function returns **:status**. If the value is **nil**, the function continues to wait for input when the window is deexposed or deselected. The default is **nil**.

zl:tyi &optional *stream eof-option* *Function*

Inputs one character from *stream* and returns it. The character is echoed if *stream* is interactive, except that Rubout is not echoed. The Control, Meta, and so on shifts echo as prefix c-, m-, and so on.

The **:tyi** stream operation is preferred over the **zl:tyi** function for some purposes. Note that it does not echo. See the message **:tyi**, page 34.

(This function can take its arguments in the other order, for Maclisp compatibility only)

sys:read-for-top-level &optional (*stream* **zl:standard-input**) *eof-option* *Function*

Differs from **zl:read** only in that it ignores close parentheses seen at top level, and it returns the symbol **si:eof** if the stream reaches end-of-file if you have not supplied an *eof-option* (instead of signalling an error as **zl:read** would). This version of **zl:read** is used in the system's "read-eval-print" loops.

zl:read-expression &optional *stream* &key (*completion-alist* **nil**) (*completion-delimiters* **nil**) *Function*

Like **sys:read-for-top-level** except that if it encounters a top-level end-of-file, it just beeps and waits for more input. This function is used by the **:expression** option for **prompt-and-read**.

stream defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

If *completion-alist* is not **nil**, this function also sets up `COMPLETE` and `c-?` as input editor commands. When the user presses `COMPLETE`, the input editor tries to complete the current symbol over the set of possibilities defined by *completion-alist*. When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. *completion-alist* can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered.
alist	The car of each alist element is a string representing one possible completion.
array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

The default for *completion-alist* is **nil**.

completion-delimiters is **nil** or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If *completion-delimiters* is **nil**, the entire text of the current symbol is a single "chunk". The default is **nil**.

zl:read-form &optional *stream* &key (*edit-trivial-errors-p* **zl:*read-form-edit-trivial-errors-p***) (*completion-alist* **zl:*read-form-completion-alist***) (*completion-delimiters* **zl:*read-form-completion-delimiters***) *Function*

Like **zl:read-expression** except that it assumes that the returned value will be given immediately to **eval**. This function is used by the Lisp command loop and by the **:eval-form** and **:eval-form-or-end** options for **prompt-and-read**.

stream defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

If *edit-trivial-errors-p* is not **nil**, the function checks for two kinds of errors. If a symbol is read, it checks whether the symbol is bound. If a list whose first element is a symbol is read, it checks whether the symbol has a function definition. If it finds an unbound symbol or undefined function, it offers to use a lookalike symbol in another package or calls **zl:parse-ferror** to let the user correct the input. *edit-trivial-errors-p* defaults to the value of **zl:*read-form-edit-trivial-errors-p***. The default value is **t**.

If *completion-alist* is not **nil**, this function also sets up COMPLETE and c-? as input editor commands. When the user presses COMPLETE, the input editor tries to complete the current symbol over the set of possibilities defined by *completion-alist.* When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. *completion-alist* can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered.
alist	The car of each alist element is a string representing one possible completion.
array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

The default for *completion-alist* is the value of **zl:*read-form-completion-alist***. The default value is **:zmacs.**

completion-delimiters is **nil** or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If *completion-delimiters* is **nil**, the entire text of the current symbol is a single "chunk". The default is the value of **zl:*read-form-completion-delimiters***. The default value is **(#/- #/: #\space)**.

read-or-end &optional (*stream* **zl:standard-input**) *reader* *Function*

Like **zl:read-expression** except that if it is reading from an interactive stream and the user presses END as the first character or the first character after only whitespace characters, it returns two values, **nil** and :end. If it encounters any nonwhitespace characters, it calls the *reader* function with an argument of *stream* to read the input. *reader* defaults to **zl:read-expression**. *stream* defaults to **zl:standard-input**.

The **:expression-or-end** and **:eval-form-or-end** options for **prompt-and-read** invoke **read-or-end**.

This function is intended to read only from interactive streams.

zl:read-or-character &optional *delimiters stream reader* *Function*

Like **zl:read-expression**, except that if it is reading from an interactive stream and the user types one of the *delimiters* as the first character or the first character after only whitespace characters, it returns four values: **nil**, **:character**, the character code of the delimiter, and any numeric argument to the delimiter. If it encounters any nonwhitespace characters, it calls the *reader* function with an argument of *stream* to read the input.

delimiters is a character, a list of characters, or **nil**. The default is **nil**. *reader* defaults to **zl:read-expression**. *stream* defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

zl:read-and-eval &optional *stream* (*catch-errors* **t**) *Function*

Calls **zl:read-expression** to read a form, without completion. It then evaluates the form and returns the result. If *catch-errors* is not **nil**, it calls **zl:parse-ferror** if an error occurs during the evaluation (but not the reading) so that the input editor catches the error.

stream defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

read-line &optional *input-stream* (*eof-error-p* **t**) *eof-value recursive-p* *Function*

Reads in a line of text terminated by a newline. It returns the line as a character string, *without* the newline character. This function is usually used to get a line of input from the user. A second returned value is a flag that is considered false if the line was terminated normally, or true if end-of-file (EOF) terminated the non-empty line. If EOF is encountered immediately (that is, appears to terminate an empty line), then the end-of-file processing is controlled in the usual way by the *eof-error-p*, *eof-value*, and *recursive-p* arguments.

The corresponding output function is **write-line**.

zl:readline &optional (*stream* **zl:standard-input**) *eof-option* *Function*

Reads in a line of text. If called from inside the input editor or if reading from a stream that does not support the input editor, the line is terminated by a Newline character. If the stream supports the input editor and **zl:readline** is called from outside the input editor, the line is terminated by RETURN, LINE, or END.

This function is usually used to get a line of input from the user. If *stream* supports the input editor, **zl:readline** calls **zl:read-delimited-string**, and *input-editor-options* is passed as the list of options to the input editor.

zl:readline returns two values:

- The line as a character string, without the Newline character, or if already at end-of-file, **nil**.
- An *eof* flag, if *eof-option* was **nil**. This is **t** if the line was terminated because end-of-file was encountered, or **nil** if it was terminated because of a RETURN, LINE, or END character.

See the function **zl:read-delimited-string**, page 255.

read-line-trim &optional *input-stream* (*eof-errorp* **t**) *eof-value* *recursive-p* *Function*

Trims leading and trailing whitespace from string input. "Whitespace" means spaces, tabs, or newlines. It takes the same arguments as the normal **read-line** and returns the same values.

Examples:

```
(read-line-trim)  exciting option   RETURN =>
"exciting option"
NIL
NIL
NIL

(readline-trim)RETURN =>
""
NIL
NIL
NIL
```

zl:readline-trim &optional (*stream* **zl:standard-input**) *eof-option* *Function*

Trims leading and trailing whitespace from string input. "Whitespace" means spaces, tabs, or newlines. It takes the same arguments as the normal **zl:readline** and returns the same values.

Examples:

```
(readline-trim)   exciting option   RETURN =>
"exciting option"
NIL
#/Return
NIL
```

```
(readline-trim)RETURN =>
""
NIL
#/Return
NIL
```

The **:string-trim** option for **prompt-and-read** and the **:string-trim** **tv:choose-variable-values** keyword use **zl:readline-trim.**

zl:readline-or-nil &optional (*stream* **zl:standard-input**) *eof-option* *Function*

Like **zl:readline-trim,** except that it returns a first value of **nil** instead of the empty string if the input string is empty.

The **:string-or-nil** option for **prompt-and-read** and the **:string-or-nil** **tv:choose-variable-values** keyword use **zl:readline-or-nil.**

See the function **zl:readline-trim,** page 250.

read-line-no-echo &optional *stream* &rest *keywords* &key *Function*
(*terminators* **'(#\return #\line #\end)**)
(*full-rubout* **nil**) (*notification* **t**) (*prompt* **nil**)
(*help* **nil**)

Reads a line of input from *stream* without echoing the input, and returns the input as a string, without the terminating character. This function is used to read passwords and encryption keys. It does not use the input editor but does allow input to be edited using RUBOUT.

stream must be interactive. It defaults to **zl:query-io.**

Following are the permissible keywords:

:terminators	A list of characters that terminate the input. If the user types **#\return**, **#\line**, or **#\end** as a terminator, the function echoes a NEWLINE. If the user types any other character as a terminator, the function echoes that character. The default is **(#\return #\line #\end).**
:full-rubout	If not **nil** and the user rubs out all characters on the line, the function returns **nil.** If **nil** and the user rubs out all characters on the line, the function waits for more input. The default is **nil.**
:notification	If not **nil** and a notification is received, the function displays the notification and reprompts. If **nil** and a notification is received, the notification is ignored. The default is **t.**

:prompt	If **nil**, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor", page 278. The default is **nil**.
:help	If not **nil**, the value should be a help option. See the section "Displaying Help Messages in the Input Editor", page 279. Then, when the user presses HELP, the function displays the help option and reprompts. If **nil** and the user presses HELP, the function just returns **#\help**. The default is **nil**.

zl:readline-no-echo &optional *stream* &key (*terminators* **'(#\return #\line #\end)**) (*full-rubout* **nil**) (*notification* **t**) (*prompt* **nil**) (*help* **nil**) *Function*

Reads a line of input from *stream* without echoing the input, and returns the input as a string, without the terminating character. This function is used to read passwords and encryption keys. It does not use the input editor but does allow input to be edited using RUBOUT.

stream must be interactive. It defaults to **zl:query-io**.

Following are the permissible keywords:

:terminators	A list of characters that terminate the input. If the user types **#\return**, **#\line**, or **#\end** as a terminator, the function echoes a NEWLINE. If the user types any other character as a terminator, the function echoes that character. The default is **(#\return #\line #\end)**.
:full-rubout	If not **nil** and the user rubs out all characters on the line, the function returns **nil**. If **nil** and the user rubs out all characters on the line, the function waits for more input. The default is **nil**.
:notification	If not **nil** and a notification is received, the function displays the notification and reprompts. If **nil** and a notification is received, the notification is ignored. The default is **t**.
:prompt	If **nil**, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor", page 278. The default is **nil**.
:help	If not **nil**, the value should be a help option. See the section "Displaying Help Messages in the Input Editor",

page 279. Then, when the user presses HELP, the function displays the help option and reprompts. If **nil** and the user presses HELP, the function just returns **#\help**. The default is **nil**.

read-delimited-list *char* &optional *stream recursive-p* *Function*

Reads objects from *stream* until the next character after an object's representation (ignoring whitespace characters and comments) is *char*. **read-delimited-list** returns a list of the objects read.

To be more precise, **read-delimited-list** looks ahead at each step for the next non-whitespace character and peeks at it as if with **peek-char**. If it is *char*, then the character is consumed, and the list of objects is returned. If it is a constituent or escape character, then **read** is used to read an object, which is added to the end of the list. If it is a macro character, the associated macro function is called, and if that function returns a value, the returned value is added to the list. Then, the peek-ahead process is repeated.

This function is particularly useful for defining new macro characters. Usually it is desirable for the terminating character *char* to be a terminating macro character, so that it may be used to delimit tokens. However, **read-delimited-list** makes no attempt to alter the syntax specified for *char* by the current readtable. You must make any necessary changes to the readtable syntax explicitly. The following example illustrates this.

Suppose you wanted #{**a b c ... z**} to read as a list of all pairs of the elements **a, b, c, ... z**. For example:

```
#{p q z a} reads as ((p q) (p z) (p a) (q z) (q a) (z a))
```

This can be done by specifying a macro-character definition for #{ that does two things: reads in all of the items up to the }, and constructs the pairs. **read-delimited-list** performs the first task.

```
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (mapcon #'(lambda (x)
              (mapcar #'(lambda (y) (list (car x) y)) (cdr x)))
          (read-delimited-list #\} stream t)))

(set-dispatch-macro-character #\# #{ #'|#{-reader|)

(set-macro-character #\} (get-macro-character #\) nil)
```

It is necessary to give a macro definition to the character } as well, to prevent it from being a constituent, as discussed above. Without the definition, the } in the input expression would be considered a constituent character; part of the symbol named **a}**. You could correct for this by putting a space before the }, but it is cleaner to simply use the call to **set-macro-character**.

Giving } the same definition as the standard definition of the character) has the twin benefit of making it terminate tokens for use with **read-delimited-list**, and also making it illegal for use in any other context. This means that attempting to read a stray } will signal an error.

read-delimited-string *delimiters* &optional *stream eof-error-p* *eof-value* &rest *make-array-args* *Function*

delimiters is either a character or a list of characters. Characters are read from *stream* until one of the delimiter characters is encountered. The characters read up to the delimiter are returned as a string. This function can be invoked from inside or outside the input editor. If invoked from outside the input editor, the delimiter characters are set up as activation characters. *make-array-args* are arguments to be passed to **make-array** when constructing the string to return.

eof-error-p controls what happens if input is from a file (or any other input source that has a definite end) and the end of file is reached. If *eof-error-p* is **t** (the default), an error is signalled at the end of file (EOF). If it is **nil**, then no error is signalled, and instead **read** returns *eof-value*.

read-delimited-string returns four values:

- The string
- An *eof-value*, if the *eof-error-p* parameter was **nil**
- The character that delimited the string
- Any numeric argument given the delimiter character

This function is used by **readline** and the **:delimited-string** option for **prompt-and-read**.

Examples:

The following reads characters until `END` is typed and returns a string at least 200 characters long with a leader-length of 3:

```
(read-delimited-string #\end *standard-input* nil nil
                        200. :leader-length 3)
```

The following is the same as **(readline)**, except that it does not echo a `NEWLINE` after the string is activated:

```
(read-delimited-string '(#\return #\line #\end))
```

A simple word parser:

```
(read-delimited-string '(#\space #/, #/. #/?))
```

zl:read-delimited-string &optional (*delimiters* **#\end**) (*stream* **standard-input**) (*eof* **nil**) (*input-editor-options* **nil**) &rest (*make-array-args* **'(100. :type sys:art-string)**) *Function*

delimiter is either a character or a list of characters. Characters are read from *stream* until one of the delimiter characters is encountered. The characters read up to the delimiter are returned as a string. This function can be invoked from inside or outside the input editor. If invoked from outside the input editor, the delimiter characters are set up as activation characters. The *eof* argument is treated the same way as the *eof* argument to the **:tyi** message to non-interactive streams. *input-editor-options* are passed on as the first argument to the **:input-editor** message, after having an **:activation** entry prepended. *make-array-args* are arguments to be passed to **make-array** when constructing the string to return.

zl:read-delimited-string returns four values:

- The string
- An *eof* flag, if the *eof* parameter was **nil**
- The character that delimited the string
- Any numeric argument given the delimiter character

This function is used by **readline**, **zl:qsend**, and the **:delimited-string** option for **prompt-and-read**.

Examples:

The following reads characters until END is typed and returns a string at least 200. characters long with a leader-length of 3:

```
(read-delimited-string #\end standard-input nil nil 200. :leader-length 3)
```

The following is the same as **(readline)**, except that it does not echo a Newline after the string is activated:

```
(read-delimited-string '(#\return #\line #\end))
```

A simple word parser:

```
(read-delimited-string '(#\space #/, #/. #/?))
```

For a more complex example of a sentence parser that uses **zl:read-delimited-string**: See the section "Examples of Use of the Input Editor", page 279.

read-preserving-whitespace &optional *input-stream* (*eof-error-p* **t**) *eof-value recursive-p* *Function*

Certain printed representations given to **read**, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the close parenthesis marks the end of the list.) Normally, **read** will throw away the delimiting character if it is a whitespace character, but will preserve the character of the next expression.

read-preserving-whitespace is provided for some specialized situations where it is desirable to determine precisely what character terminated the extended token. For example, consider this macro-character definition:

```
(defun slash-reader (stream char)
  (declare (ignore char))
    (do ((path (list (read-preserving-whitespace stream))
               (cons (progn (read-char stream nil nil t)
                            (read-preserving-whitespace stream))
                     path)))
        ((not (char= (peek-char nil stream nil nil t) #\/))
         (cons 'path (nreverse path)))))

(set-macro-character #\/ #'slash-reader)
```

Consider calling **read** now on this expression:

```
(zyedh /usr/games/zork /usr/games/boggle)
```

The / macro reads objects separated by more / characters, thus **/usr/games/zork** is intended to read as **(path usr games zork)**. The entire example expression should therefore be read as:

```
(zyedh (path usr games zork) (path usr games boggle))
```

However, if **read** had been used instead of **read-preserving-whitespace**, then after reading the symbol **zork**, the following space would have been discarded, and the next call to **peek-char** would see the following /. Since the / had already been read, the loop would continue, producing the expression:

```
(zyedh (path usr games zork usr games boggle))
```

Note that **read-preserving-whitespace** behaves *exactly* like **read** when the *recursive-p* argument is non-**nil**. The distinction is established only by calls with *recursive-p* equal to **nil** or omitted.

Note also that this is actually a rather dangerous definition to make, because expressions such as **(/ x 3)** will no longer read properly. The ability to reprogram the reader syntax is very powerful, and must be used with caution. This redefinition of / is shown here purely for the sake of example.

read-char &optional *input-stream* (*eof-error-p* **t**) *eof-value recursive-p* *Function*

Reads one character from *input-stream*, and returns it as a character object.

The corresponding output function is **write-char**.

zl:readch &optional *stream eof-option* *Function*

Provided only for Maclisp compatibility, since in Zetalisp characters are always represented as integers. **zl:readch** is just like **zl:tyi**, except that instead of returning an integer character, it returns a symbol whose print name is the character read in. The symbol is interned in the current package. This is just like a Maclisp "character object". (This function can take its arguments in the other order, for Maclisp compatibility only.)

read-char-no-hang &optional *input-stream* (*eof-error-p* **t**) *eof-value* *recursive-p* *Function*

This function performs the same operation as **read-char**, but if it would be necessary to wait in order to get a character (as from a keyboard), **nil** is immediately returned without waiting. This allows you to check for input availability and get the input, if it is available, in the same operation. This is different from the **listen** operation in two ways. First, **read-char-no-hang** potentially reads a character, whereas **listen** never inputs a character. Second, **listen** does not distinguish between end-of-file (EOF) and no input being available, whereas **read-char-no-hang** does make that distinction. **read-char-no-hang** returns *eof-value* at EOF (or signalling an error of no *eof-error-p* is true), and always returns **nil** if no input is available.

unread-char *character* &optional *input-stream* *Function*

Puts *character* onto the front of *input-stream*. *character* must be the same character that was most recently read from *input-stream*. *input-stream* backs up over this character, so that when a character is next read from *input-stream* it will be the specified character. Successive calls to **read-char** will pick up the previous contents of *input-stream*, as it was before the call to **unread-char**. **unread-char** returns **nil**.

You can apply **unread-char** only to the character most recently read from *input-stream*. Moreover, you can not invoke **unread-char** twice consecutively without an intervening **read-char** operation. The result is that you can back up only by one character, and you can not insert any characters into the input stream that were not already there.

read-byte *binary-input-stream* &optional (*eof-error-p* **t**) *eof-value* *Function*

Reads one byte from *binary-input-stream* and returns it in the form of an integer.

The corresponding output function is **write-byte**.

peek-char &optional *peek-type input-stream* (*eof-error-p* **t**) *eof-value recursive-p* *Function*

The result of **peek-char** depends on *peek-type*, which defaults to nil. The affects of *peek-type* are as follows:

Value	Affect
nil	Returns the next character to be read from *input-stream*, without actually removing it from the stream. The next time input is done from *input-stream*, the character will still be there. It is as if you had called **read-char**, then **unread-char** in succession.
t	Skips over whitespace characters (but not comments), and then performs the peeking operation on the next character. This is useful for finding the beginning of the next printed representation of a Lisp object. The last character examined (the one that starts an object) is not removed from the input stream.
character object	Skips over input characters until a character that is **char=** to that object is found. That character is left in the input stream.

zl:tyipeek &optional *peek-type stream eof-option* *Function*

Provided mainly for Maclisp compatibility; the **:tyipeek** stream operation is usually clearer.

What **zl:tyipeek** does depends on the *peek-type*, which defaults to **nil**. With a *peek-type* of **nil**, **zl:tyipeek** returns the next character to be read from *stream*, without actually removing it from the input stream. The next time input is done from *stream* the character is still there; in general, (= **(zl:tyipeek) (zl:tyi)**) is **t**. See the message **:tyipeek**, page 37.

If *peek-type* is an integer less than 1000 octal, then **zl:tyipeek** reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is **t**, then **zl:tyipeek** skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of **zl:tyipeek** supported by Maclisp, in which *peek-type* is an integer not less than 1000 octal, is not supported, since the readtable formats of the Maclisp reader and the Symbolics Common Lisp reader are quite different.

Characters passed over by **zl:tyipeek** are echoed if *stream* is interactive.

clear-input &optional *input-stream* *Function*

Clears any buffered input associated with *input-stream.* It is primarily useful for removing type-ahead from keyboards when some kind of asynchronous error has occurred. If this operation doesn't make sense for the stream involved, then **clear-input** does nothing. **clear-input** returns **nil.**

listen &optional *input-stream* *Function*

The predicate **listen** returns **t** if there is a character immediately available from *input-stream,* and otherwise it returns **nil.** This is particularly useful when the stream obtains characters from an interactive device such as a keyboard. A call to **read-char** would simply wait until a character was available, but **listen** can sense whether or not to attempt input. On a non-interactive stream, the general rule is that **listen** returns **t** except when it's at EOF.

12.2 Non-stream Input Functions

The following functions are related functions that do not operate on streams:

read-from-string *string* &optional (*eof-errorp* **t**) *eof-value* &key (*start* **0**) *end preserve-whitespace* *Function*

The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect. If *string* has a fill-pointer it controls how much can be read.

The arguments **:start** and **:end** delimit a substring of *string* beginning at the character indexed by **:start** and up to, but not including, the character indexed by **:end.** This is the same as for other string functions.

The flag **:preserve-whitespace,** if provided and non-**nil,** indicates that the operation should preserve whitespace as for **read-preserving-whitespace.** It defaults to **nil.**

As with other reading functions, the arguments *eof-error-p* and *eof-value* control the action of the reader if the end of the string is reached before the operation is completed. Reaching the end of the string is treated as any other EOF event.

read-from-string returns two values: The first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this is the length of the string. For example:

```
(read-from-string "(a b c)") => (A B C) and 7
```

zl:read-from-string *string* &optional (*eof-option* **'si:no-eof-option**) (*start* **0**) *end* (*preserve-whitespace* **zl:read-preserve-delimiters**) *Function*

The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect. If *string* has a fill-pointer it controls how much can be read.

eof-option is what to return if the end of the string is reached, as with other reading functions. *start* is the index in the string of the first character to be read. *end*, if given, is used instead of (**zl:array-active-length** *string*) as the integer that is one greater than the index of the last character to be read.

The flag **:preserve-whitespace**, if provided and non-**nil**, indicates that the operation should preserve whitespace as for **read-preserving-whitespace**. It defaults to **nil**.

zl:read-from-string returns two values: The first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this is the length of the string.

Example:

```
(read-from-string "(a b c)") => (A B C) and 7
```

parse-integer *string* &key (*start* **0**) (*end* **nil**) (*radix* **10**) (*junk-allowed* **nil**) (*sign-allowed* **t**) *Function*

This function examines the substring of *string* delimited by **:start** and **:end** (which default to the beginning and end of the string). It skips over whitespace and then attempts to parse an integer. The **:radix** argument defaults to 10, and must be an integer between 2 and 36.

If **:junk-allowed** is **nil** (the default), then the entire substring is scanned. The returned value is the value of the number parsed as an integer. An error is signalled if the substring does not consist entirely of the representation of an integer, possibly surrounded on either side by whitespace characters.

If **:junk-allowed** is non-**nil**, then the first value returned is the value of the number parsed as an integer, or **nil** if no syntactically correct integer was seen.

In either case, the second value returned is the index into the string of the delimiter that terminated the parse, or it is the index beyond the substring if the parse terminated at the end of the substring (as will be the case of **:junk-allowed** is **nil**).

Note that **parse-integer** does not recognize the syntactic radix-specifier prefixes #**o**, #**b**, #**x**, and #*n***r**, nor does it recognize a trailing decimal point. It permits only an optional sign (+ or -) followed by a non-empty sequence of digits in the specified radix. For example:

```
(parse-integer " -1234567890 " :start 3) => 234567890 and 13
```

zl:readlist *char-list* *Function*

Provided mainly for Maclisp compatibility. *char-list* is a list of characters. The characters can be represented by anything that the function **character** accepts: integers, strings, or symbols. The characters are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect.

If there are more characters in *char-list* beyond those needed to define an object, the extra characters are ignored. If there are not enough characters, an "eof in middle of object" error is signalled.

See the special form **with-input-from-string**, page 137.

12.3 Read Control Variables

There are a number of reader variables that affect the performance of read functions.

read-suppress *Variable*

When the value of ***read-suppress*** is **nil**, the Lisp reader operates normally. When it is non-**nil**, then most of the interesting operations of the reader are suppressed; input characters are parsed, but much of what is read is not interpreted.

The primary purpose of ***read-suppress*** is to support the operation of the read-time conditional constructs #+ and #-. See the section "Sharp-sign Reader Macros", page 229. It is important for these constructs to be able to skip over the printed representation of a Lisp expression despite the possibility that the syntax of the skipped expression may not be legal for the current implementation. This is especially useful because a primary application of #+ and #- is to allow the same program to be share among several Lisp implementations despite small incompatibilities of syntax.

A non-**nil** value of ***read-suppress*** has the following specific effects on the Lisp reader:

- All extended tokens are completely uninterpreted, they are discarded and treated as if they were **nil**. It does not matter whether a token looks like a valid number, or whether the package markers are correct. One consequence of this is that the error concerning improper dotted-list syntax will not be signalled.

- Any standard # macro-character construction that requires, permits, or disallows an infix numerical argument, such as #*n***r**, will not enforce any constraint on the presence, absence, or value of such an argument.

- The #\ construction always produces the value **nil**. It will not signal an error even it an unknown character name is seen.

- Each of the #**b**, #**o**, #**x**, and #**r** constructions always scans over a following token and produces the value **nil**. It will not signal an error even if the token does not have the syntax of a rational number.

- The #* construction always scans over a following token and produces the value **nil**. It will not signal an error even if the token does not consist solely of the characters 0 and 1.

- Each of the #**.** and #**,** constructions reads the following form in suppressed mode but does not evaluate it. The form is discarded and **nil** is produced.

- Each of the #**a**, #**s**, and #**:** constructions reads the following form in suppressed mode but does not interpret it in any way. It need not be a list in the case of #**s**, or a symbol in the case of #**:**. The form is discarded and **nil** is produced.

- The #= construction is totally ignored. It does not read a following form. It produces no object, but is treated as whitespace.

- The ## construction always produces **nil**.

Note that, no matter what the value of ***read-suppress*** is, parentheses continue to delimit (and construct) lists, the #(construction continues to delimit vectors; and comments, strings, and the quote and backquote constructions continue to be interpreted properly. furthermore, such illegal constructions as '), #<, #), and #<**space**> continue to signal errors.

In some cases, it may be appropriate for a user-written macro-character definition to check the value of ***read-suppress*** and avoid certain computations or side effects if its value is not **nil**.

zl:read-preserve-delimiters *Variable*

Certain printed representations given to **zl:read**, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close parenthesis serves to mark the end of the list.) Normally **zl:read** throws away the delimiting character if it is "whitespace", but preserves it (with a **:untyi** stream operation) if the character is syntactically meaningful, since it might be the start of the next expression.

If **zl:read-preserve-delimiters** is bound to **t** around a call to **zl:read**, no delimiting characters are thrown away, even if they are whitespace. This might be useful for certain reader macros or special syntaxes.

zl:*read-form-edit-trivial-errors-p* *Variable*

If not **nil**, **zl:read-form** checks for two kinds of errors. If a symbol is read, it checks whether the symbol is bound. If a list whose first element is a symbol is read, it checks whether the symbol has a function definition. If it finds an unbound symbol or undefined function, it offers to use a lookalike symbol in another package or calls **zl:parse-ferror** to let the user correct the input. The default is **t**.

zl:*read-form-completion-alist* *Variable*

If not **nil**, **zl:read-form** sets up COMPLETE and c-? as input editor commands. When the user presses COMPLETE, the input editor tries to complete the current symbol over the set of possibilities defined by *completion-alist*. When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. **zl:*read-form-completion-alist*** can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered.
alist	The car of each alist element is a string representing one possible completion.
array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

The default value is **:zmacs**.

zl:*read-form-completion-delimiters* *Variable*

The value is **nil** or a list of characters that delimit "chunks" for completion in **zl:read-form**. As in Zwei, completion works by matching initial substrings of "chunks" of text. If **zl:*read-form-completion-delimiters*** is **nil**, the entire text of the current symbol is a single "chunk". The default value is **(#/- #/: #\space)**.

13. The Input Editor Program Interface

13.1 How the Input Editor Works

The input editor is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to let you edit minor mistakes in typein. At the same time, it is not supposed to get in the way; Lisp is to see the input as soon as you have typed a syntactically complete form. The definition of "syntactically complete form" depends on the function that is reading from the stream; for **zl:read**, it is a Lisp expression. This section describes the general protocol used for communication between the input editor and reading functions such as **zl:read** and **zl:readline**.

By *reading function* we mean a function that reads a number of characters from a stream and translates them into an object. For example, **zl:read** reads a Lisp expression and returns an object. **zl:readline** reads a line of characters and returns a string as its first value. Reading functions do not include the more primitive **:tyi** and **:any-tyi** stream operations, which take and return one character or blip from the stream.

The tricky thing about the input editor is the need for it to figure out when you are all done. The idea of an input editor is that as you type in characters, the input editor saves them up in an *input buffer* so that if you change your mind, you can edit them and replace them with different characters. However, at some point the input editor has to decide that the time has come to stop putting characters into the input buffer and let the reading function start processing the characters. This is called "activating".

The right time to activate depends on the function calling the input editor, and determining it may be very complicated. If the function is **zl:read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently defined reader macros do, and so on. The input editor should not have to know how to parse the characters in the input buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The input editor interface is organized so that the calling function can do all the parsing, while the input editor does all the handling of editing commands, and the two are kept completely separate.

Following is a summary of how the input editor works. The input editor used to be called the rubout handler, and some operations and variables still have "rubout-handler" in their names.

When a reading function is called to read from a stream that supports the **:input-editor** operation, that function "enters" the input editor. It then goes ahead **:tyi**'ing characters from the stream. Because control is inside the input editor, the stream echoes these characters so the user can see the input. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the input editor it is also handing it the responsibility for echoing). The input editor is also saving all these characters in the input buffer, for reasons disclosed in the following paragraph. When the reading function decides it has enough input, it returns and control "leaves" the input editor. That was the easy case.

If you press RUBOUT or a keystroke that represents another editing command, the input editor processes the command and lets you insert characters before the last one in the line. The input editor modifies the input buffer and the screen accordingly. Then, when you type the next nonediting character at the end of the line, a **throw** is done, out of all recursive levels of **zl:read**, reader macros, and so forth, back to the point where the input editor was entered. Now the **zl:read** is tried over again, rereading all the characters you had typed and not rubbed out, but not echoing them this time. When the saved characters have been exhausted, additional input is read from you in the usual fashion.

The input editor has options that can cause the **throw** to occur at other times as well. With the **:activation** option, when you type an activation character a **throw** occurs, a rescan is done if necessary, and a final blip is returned to the reading function. With the **:preemptable** and **:command** options, a blip or special character in the input stream causes control to be returned from the input editor immediately, without a rescan. These options let you process mouse clicks or special keystroke commands as soon as they are read.

The effect of all this is a complete separation of the functions of input editing and parsing, while at the same time mingling the execution of these two functions in such a way that input is always "activated" at just the right time. It does mean that the parsing function (in the usual case, **zl:read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have nontrivial side-effects, since it may be called multiple times.

If an error occurs while inside the input editor, the error message is printed and then additional characters are read. When you press RUBOUT, it rubs out the error message as well as the last character. You can then proceed to type the corrected expression; the input is reparsed from the beginning in the usual fashion.

13.2 Invoking the Input Editor

The variable **sys:rubout-handler** indicates the current state of input editing. This variable is not **nil** if the current process is already inside the input editor.

sys:rubout-handler *Variable*

Indicates the status of input editing within a process.

This variable is used internally by the **:input-editor** method and the input editor. It should not be necessary for user programs to examine its value since the **with-input-editing** special form is provided for this purpose.

The possible values for this variable are:

Value	*Meaning*
nil	The process is outside the input editor.
:read	The process is inside the **:input-editor** method.
:tyi	The process is inside the editing portion of the **:tyi** method.

The input editor is invoked on a stream when the stream receives an **:input-editor** message. The **:input-editor** and **:tyi** methods of **si:interactive-stream** contain the code of the input editor. The **:input-editor** method initializes the input editor, establishes its **catch**, and then calls back to the reading function with **sys:rubout-handler** bound to **:read**. When the reading function sends the **:tyi** or **:any-tyi** message, input is taken from the input buffer. If no input is available, the editing or **:tyi** portion of the input editor is invoked, and **sys:rubout-handler** is bound to **:tyi**.

The first argument to the **:input-editor** message is the function that the input editor should call to do the reading, and the rest of the arguments are passed to that function. If the reading function returns normally, the values returned by the **:input-editor** message are just those returned by the reading function. If the input editor returns by throwing out of the reading function, the return values depend on which option caused the input editor to throw: See the option **:full-rubout**, page 272. See the option **:preemptable**, page 276. See the option **:command**, page 276.

The input editor can take a series of options. These are specified dynamically by the special forms **with-input-editing-options** and **with-input-editing-options-if**. For a description of the options: See the section "Input Editor Options", page 272.

with-input-editing-options *options* &body *body* *Special Form*

Specifies input editing options and executes *body* with those options in effect. The scope of the option specifications is dynamic.

options is a list of input editor option specifications. Each element is a list whose car is an option-name specification and whose cdr is a list of forms to be evaluated to yield "arguments" for the option. The option-name specification is a keyword symbol or a list whose car is a keyword symbol. The symbol is the name of the option.

If the option-name specification is a list and if the symbol **:override** is an element of the cdr of the list, this option specification overrides any higher-level specifications for this option. Otherwise, the specification for each option that is dynamically outermost (that is, the specification from the highest-level caller) is in effect during the execution of *body*.

with-input-editing-options returns whatever values *body* returns.

In the following example, the user is prompted for a Lisp expression. Two input editor options are specified. The first says that the caller is also willing to receive mouse or menu blips. The second specifies a prompt.

```
(with-input-editing-options ((:preemptable :blip)
                             (:prompt "Form: "))
  (read))
```

In the following example, the user is prompted for a line of text. The text may be activated by any of the characters RETURN, END, or TRIANGLE. This might be useful if activating with TRIANGLE meant something different from activating with RETURN. This example also demonstrates the use of **:override** to make this **:activation** specification override any higher-level **:activation** specifications.

```
(with-input-editing-options
    (((:activation :override) 'memq '(#\return #\end #\triangle)))
  (prompt-and-read :string "Name: "))
```

For a list of input editor options: See the section "Input Editor Options", page 272. See the special form **with-input-editing-options-if**, page 268.

with-input-editing-options-if *cond options* &body *body* *Special Form*

Executes *body*, possibly with specified input editing options in effect. The scope of the option specifications is dynamic.

cond is a form to be evaluated at run-time. If *cond* returns **non-nil**, the specified input editor options are in effect during the execution of *body*.

options is a list of input editor option specifications. Each element is a list whose car is an option-name specification and whose cdr is a list of forms to be evaluated to yield "arguments" for the option. The option-name specification is a keyword symbol or a list whose car is a keyword symbol. The symbol is the name of the option.

If the option-name specification is a list and if the symbol **:override** is an element of the cdr of the list, this option specification overrides any higher-level specifications for this option. Otherwise, the specification for each option that is dynamically outermost (that is, the specification from the highest-level caller) is in effect during the execution of *body*.

with-input-editing-options-if returns whatever values *body* returns.

For a list of input editor options: See the section "Input Editor Options", page 272. See the special form **with-input-editing-options**, page 267.

This example illustrates the use of the **:command**, **:preemptable**, and **:prompt** input editor options. It is a simple command loop that reads different kinds of commands – typed Lisp expressions, single-keystroke commands, and mouse clicks. The Lisp expressions are read using the **read-or-end** function. You can provide four kinds of input:

Input	*Action*
`END`	Exit the command loop
Lisp form	Print form on next line
Mouse click	Display type of click and mouse coordinates
Single-key command	Display keystroke

The predicate for detecting a single-keystroke command simply checks for the Super bit. In a more complex program, it might look up the character in a command table.

```
(defun command-char-p (c) (char-bit c :super))
```

```
(defun command-loop ()
  (loop
    do (multiple-value-bind (value flag)
           (with-input-editing-options
               ((:command 'command-char-p)
                (:preemptable :blip)
                (:prompt "Command loop input: "))
             (read-or-end))
         (selectq flag
           (:end
            (format t "Done")
            (return t))
           (:blip
            (selectq (car value)
              (:mouse-button
               (destructuring-bind (click nil x y) (cdr value)
                 (format t "~C click at ~D, ~D" click x y)))
              (otherwise (format t "Random blip -- ~S" value))))
           (:command
            (format t "Execute ~:C command" (second value)))
           (otherwise
            (format t "~&Value is ~S" value))))))
```

To write a reading function that invokes the input editor, you should use the **with-input-editing** special form instead of sending the **:input-editor** message directly. Such functions as **zl:read** and **zl:readline** use this special form to provide input editing.

with-input-editing (&optional *stream keyword*) &body *body* *Special Form*

Provides a convenient way of invoking the input editor for use by a reading function. It establishes a context in which input editing should be provided. Use **with-input-editing** instead of sending an **:input-editor** message directly.

Both "arguments" are optional. *stream* is the stream from which characters are read; if *stream* is not provided or is **nil**, ***standard-input*** is used.

keyword determines the activation characters for the input editor:

Value	*Activation characters*
nil	None (unless specified at a higher level). This is the default.

:end-activation	#\end
:line-activation	**#\end**, **#\return**, and **#\line**
:line	**#\end**, **#\return**, and **#\line**. In addition, a Newline is echoed after the reading function returns.

To supply other input editor options: See the special form **with-input-editing-options**, page 267. See the special form **with-input-editing-options-if**, page 268.

with-input-editing defines an internal lexical closure with *body* as its body. When the **with-input-editing** form is evaluated from outside the input editor, the stream is sent an **:input-editor** message if it handles it. The argument to the **:input-editor** message is the lexical closure, except that if the **:line** keyword is supplied, **with-input-editing** also arranges to echo a Newline after the lexical closure returns. If the **with-input-editing** form is evaluated from inside the input editor or if the stream does not handle the **:input-editor** message, the lexical closure is called instead.

with-input-editing returns whatever values *body* returns.

The following example defines a simple sentence parser.

```
(defun read-sentence (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream)
      (loop named sentence
            with sentence = nil
            for word = (make-array 20. :type art-string :fill-pointer 0)
            do (loop for char = (send stream :tyi)
                     do
                 (cond ((memq char '(#\space #\return #/. #/? #/,))
                        (if (not (equal word ""))
                            (push word sentence))
                        (selectq char
                          ((#\space #\return #/,)
                           (return))
                          (#\.
                           (push :period sentence)
                           (return-from sentence (nreverse sentence)))
                          (#\?
                           (push :question-mark sentence)
                           (return-from sentence (nreverse sentence)))))
                       (t (array-push-extend word char))))))))
```

13.3 Input Editor Options

The input editor can take a series of options, specified by the special forms **with-input-editing-options** and **with-input-editing-options-if**. Following are descriptions of the options.

:full-rubout *token* *Option*

If the user rubs out all the characters that were typed, control is returned from the input editor immediately. Two values are returned: **nil** and *token*. If the user does not rub out all the characters, the input editor propagates multiple values back from the function that it calls, as usual. In the absence of this option, the input editor simply waits for more characters to be typed and ignores any additional rubouts.

:pass-through &rest *characters* *Option*

The characters in *characters* are not to be treated as special by the input editor. This option is used to pass format effectors (such as HELP or CLEAR INPUT) through to the reading function instead of interpreting them as input editor commands. **:pass-through** is allowed only for characters with no modifier bits set, that is, for character codes 0 through 377 (octal). For characters that have modifier bits set and must be visible to the reading function, use **:do-not-echo** or **:activation**.

:prompt &rest *prompt-option* *Option*

When it is time for the user to be prompted, the input editor displays *prompt-option*. *prompt-option* can have one element, which can be **nil**, a string, a function, or a symbol other than **nil**; or it can have more than one element: See the section "Displaying Prompts in the Input Editor", page 278.

The difference between **:prompt** and **:reprompt** is that the latter does not display the prompt when the input editor is first entered, but only when the input is redisplayed (for example, after a screen clear). If both options are specified, **:reprompt** overrides **:prompt** except when the input editor is first entered.

:reprompt &rest *prompt-option* *Option*

When it is time for the user to be reprompted, the input editor displays *prompt-option*. *prompt-option* can have one element, which can be **nil**, a string, a function, or a symbol other than **nil**; or it can have more than one element: See the section "Displaying Prompts in the Input Editor", page 278.

Unlike **:prompt**, **:reprompt** displays the prompt only when input is redisplayed (for example, after a screen clear), not when the input editor is first entered. If both **:prompt** and **:reprompt** are specified, **:reprompt** overrides **:prompt** except when the input editor is first entered.

:complete-help &rest *help-option* *Option*

When the user presses HELP, the input editor types out a message determined by *help-option.* None of the standard input editor help is displayed. If a **:brief-help** option has been specified, it overrides **:complete-help**. **:complete-help** overrides **:merged-help** and **:partial-help**.

help-option can have one element, which can be a string, a function, or a symbol; or it can have more than one element. For an explanation: See the section "Displaying Help Messages in the Input Editor", page 279.

This option is intended for programs that supply their own input editor help messages.

:partial-help &rest *help-option* *Option*

When the user presses HELP, the input editor first types out a message determined by *help-option.* It then types out a message describing how to invoke input editor commands and other information about the stream. If a **:brief-help, :complete-help,** or **:merged-help** option has been specified, it overrides **:partial-help.**

help-option can have one element, which can be a string, a function, or a symbol; or it can have more than one element. For an explanation: See the section "Displaying Help Messages in the Input Editor", page 279.

This option is intended for use when inexperienced users might be typing to the input editor. Often *help-option* gives some information about the program to which the user is typing and what the user can do to exit from it.

:merged-help *function* &rest *arguments* *Option*

When the user presses HELP, the input editor types out a message determined by the arguments. *function* is a function that takes at least two arguments. The input editor calls the function to print the help message. The first argument is the stream. The second argument is a continuation (a list) to print a standard message describing how to invoke input editor commands and other information about the stream. When the function wants to print this message, it should apply the car of the continuation to the cdr. If any *arguments* are supplied, they are the remaining arguments to the function.

If a **:brief-help** or **:complete-help** option has been specified, it overrides **:merged-help**. **:merged-help** overrides **:partial-help**.

This option is intended for programs that want to decide when and where to display their own help messages and the standard help message.

:brief-help &rest *help-option* *Option*

When the user presses HELP, the input editor displays a message determined by *help-option* on the same line as the typein. The message is displayed in the default typeout font, and none of the usual conventions about input editor typeout apply. **:brief-help** overrides **:complete-help**, **:merged-help**, and **:partial-help**.

help-option can have one element, which can be a string, a function, or a symbol; or it can have more than one element. For an explanation: See the section "Displaying Help Messages in the Input Editor", page 279.

This option is intended for programs like **fquery** that need to supply only a brief help message, usually about expected typein.

:initial-input *string* &optional *begin end cursor-position* *Option*

When the input editor is entered, *string* is inserted into the input buffer as if the user had typed it. The user can edit the string before activating. *begin* and *end* are indices into *string* and mark the portion of the string to be copied into the input buffer. *begin* defaults to **0**; *end* defaults to **(zl:array-active-length** *string***)**. *cursor-position* is an index into the string where the cursor should initially be placed. The default is to place the cursor at the end of the portion of the string copied into the input buffer. *string* can be **nil**, which is the same as not specifying the option.

In the following example, the user is prompted for a line of text. The input buffer initially contains the name of the user, and the cursor is placed at the beginning of the input buffer.

```
(with-input-editing-options
    ((:initial-input fs:user-personal-name nil nil 0))
  (prompt-and-read :string "Full name: "))
```

Placing a string in the input buffer is one style of input defaulting. Another style leaves the input buffer empty but allows a default to be yanked with c-m-Y. See the option **:input-history-default**, page 274.

:input-history-default *string* *Option*

Specifies *string* as the default to be yanked by c-m-Y. *string* is temporarily placed at the head of the input history. If the user types c-m-Y m-Y, the true first element of the input history is yanked. c-m-0 c-m-Y shows *string* at the head of the input history, and the entries in the input history are shifted down by one.

In the following example, the user is prompted for a line of text. The input buffer is initially empty, but the c-m-Y command yanks a default, which is the name of the user.

```
(with-input-editing-options
    ((:input-history-default fs:user-personal-name))
  (prompt-and-read :string "Full name: "))
```

This option is used by the **:pathname** option for **prompt-and-read.**

:blip-handler *function* *Option*

Specifies a function to handle blips received while inside the input editor. *function* must be a function of two arguments. The first argument is the blip; the second argument is the stream that received the blip. The handler is invoked when the input editor receives a blip. If the handler returns non-**nil**, no further action is taken. If it returns **nil** and a **:preemptable** option is in effect, the actions specified by that option are taken. Otherwise, the default blip handler is invoked.

In the following example, the user is prompted for a line of text. While entering this text, the user may also click the left or middle mouse buttons. If the left mouse button is clicked, the coordinates of the mouse with respect to the window are inserted into the input buffer. If the middle button is clicked, the name of the window is inserted.

```
(defun example-blip-handler (blip ignore)
  (destructuring-bind (type click window x y) blip
    (and (eq type :mouse-button)
         (selectq click
           (#\mouse-l-1
            (si:ie-insert-string (format nil " ~D ~D" x y))
            t)
           (#\mouse-m-1
            (si:ie-insert-string (format nil " ~A" window))
            t)))))

(with-input-editing-options ((:blip-handler 'example-blip-handler))
  (prompt-and-read :string "Blip handler test: "))
```

si:ie-insert-string is an internal function for inserting a string into the input buffer. Since the language for writing input editor commands has not been formalized, this example might not work in a later release.

:do-not-echo &rest *characters* *Option*

The characters in *characters* are interpreted as activation characters and are not echoed. The comparison is done with **char=**, not **char-equal**, so that the control and meta bits are not masked off. The characters are not inserted into the input buffer and are not interpreted as input editor commands. When one of these characters is typed, the final **:tyi** value returned is the character, not a blip.

This option exists only for compatibility with earlier releases. New programs should use the **:activation** option.

:activation *function* &rest *arguments* *Option*

For each character typed, the input editor invokes *function* with the character as the first argument and *arguments* as the remaining arguments. If the function returns **nil**, the input editor processes the character as it normally would. Otherwise, the cursor is moved to the end of the input buffer, a rescan of the input is forced (if one is pending), and the blip (**:activation** *character numeric-arg*) is returned by the final sending of the **:any-tyi** message to the stream. Activation characters are not inserted into the input buffer, nor are they echoed by the input editor. It is the responsibility of the reading function to do any echoing. For instance, **zl:readline**, not the input editor, types a Newline at the end of the input buffer when RETURN, END, or LINE is pressed.

:preemptable *token* *Option*

A blip in the input stream causes control to be returned from the input editor immediately. Two values are returned: the blip and *token*, which is usually a keyword symbol. Any unscanned input typed before the blip remains in the input buffer, available to the next read operation from the stream.

:no-input-save *Option*

The input editor does not save the scanned contents of the input buffer on the input history when returning from the reading function. This is intended for use by functions such as **fquery** that use the input editor to ask simple questions whose responses are not worth saving. **zl:yes-or-no-p** uses **:no-input-save** by default.

:command *function* &rest *arguments* *Option*

This option is used to implement nonediting single-keystroke commands. For each character typed, the input editor invokes *function* with the character as the first argument and *arguments* as the remaining arguments. If the function returns **nil**, the input editor processes the character as it normally would. Otherwise, control is returned from the input editor immediately. Two values are returned: a blip of the form (**:command** *character numeric-arg*) and the keyword **:command**. Any unscanned input typed before the command character remains in the input buffer, available to the next read operation from the stream.

:editor-command &rest *command-alist* *Option*

This option lets you specify your own input editor editing commands. Each element of *command-alist* is a cons whose car is a character and whose cdr is a symbol or a list. If the cdr is a symbol, it is a function to be called with no arguments when the user types the associated character. If the cdr is a list, the car of the list is a function to be applied to the cdr of the list when the user types the associated character. The function can examine the internal special variables that describe the state of the input editor.

If **:editor-command** specifies a command that is invoked by the same character as one of the standard input editor editing commands, the command specified by **:editor-command** overrides the standard command.

:input-wait &optional *whostate function* &rest *arguments* *Option*

When the input editor waits for input, it sends the stream an **:input-wait** message with the arguments to the **:input-wait** option as arguments. In addition, unless the **:suppress-notifications** option has been specified, **:input-wait** returns when a notification is received. See the message **:input-wait**, page 36.

:input-wait-handler *function* &rest *arguments* *Option*

When the input editor is waiting for input it sends the stream an **:input-wait** message. After **:input-wait** returns, the input editor applies *function* to *arguments*. The input editor does not process the input or display the notification until *function* returns.

:suppress-notifications *flag* *Option*

If a notification is received while in the input editor, and *flag* is supplied as **nil**, the input editor itself handles the notification, regardless of any other way you have specified that notifications should be handled. If *flag* is **t**, notifications are handled in the input editor the same way they would be handled if you were not in the input editor. That is, the input editor does not handle the notification itself.

:notification-handler *function* &rest *arguments* *Option*

If a notification is received while in the input editor, *function* is called to handle it. *function* should take at least one argument, the notification (as returned by the **:receive-notification** message to the stream). *arguments* are the remaining arguments to *function*. *function* can do anything it wants with the notification. To display the notification, *function* would usually call **sys:display-notification**.

If this option is not specified, notifications appear one after the other using **:insert**-style typeout.

Following are two simple examples of notification handlers. The first handler assumes that you want each notification to overwrite the previous one. The second handler assumes that you want them to appear one after another. ***window*** should be bound to a window and ***stream*** to a stream where you want the notifications to appear.

```
(defun my-notification-handler-1 (notification)
  (send *window* :clear-window)
  (sys:display-notification *window* notification :window))

(defun my-notification-handler-2 (notification)
  (sys:display-notification *stream* notification :stream))
```

13.4 Displaying Prompts in the Input Editor

The input editor options **:prompt** and **:reprompt** and the functions **zl:readline-no-echo** and **sys:read-character** take *prompt* arguments that let you specify an input editor prompt. *prompt* can be **nil**, a string, a function, a symbol other than **nil**, or a list (for the input editor options, the list is an **&rest** argument):

nil	No prompt is displayed.
string	A **zl:format** control string to be passed to **zl:format** with one argument, the stream on which the prompt is displayed.
function or symbol other than **nil**	A function to display the prompt. The function should take two arguments: the first is the stream on which the prompt is displayed, and the second is a keyword that indicates the origin of the function call.
list	If the first element is **nil**, no prompt is displayed. If the first element is a string, it is a **zl:format** control string to be passed to **zl:format** with the remaining elements of the list as arguments. If the first element is a function or a symbol other than **nil**, it is a function to display the prompt. The first argument to the function is the stream on which the prompt is displayed. The second argument is a keyword that indicates the origin of the function call. The remaining arguments are the remaining elements of the list.

When a function is called to display the prompt, the second argument to the function is a keyword that indicates the origin of the function call:

Keyword	*Function called from*
:prompt	**:input-editor** method of **si:interactive-stream**, when the input editor is entered

:restore	**:restore-input-buffer** method of **si:interactive-stream**
:finish-typeout	**:finish-typeout** method of **si:interactive-stream**
:refresh	Body of the input editor, when the user presses REFRESH
:erase-typeout	Body of the input editor, when the user presses PAGE

13.5 Displaying Help Messages in the Input Editor

The input editor options **:brief-help**, **:partial-help**, and **:complete-help** and the functions **zl:readline-no-echo** and **sys:read-character** take *help* arguments that let you specify input editor help messages. *help* can be a string, a function, a symbol, or a list (for the input editor options, the list is an **&rest** argument):

string	A **zl:format** control string to be passed to **zl:format** with one argument, the stream on which the help message is displayed.
function or symbol	A function to display the help message. The function should take one argument, the stream on which the help message is displayed.
list	If the first element is a string, it is a **zl:format** control string to be passed to **zl:format** with the remaining elements of the list as arguments. If the first element is a function or a symbol, it is a function to display the help message. The first argument to the function is the stream on which the help message is displayed, and the remaining arguments are the remaining elements of the list.

13.6 Examples of Use of the Input Editor

This series of examples shows several different ways of using the input editor, gradually increasing in complexity. The examples are also available in the file sys: examples; interaction.lisp.

We refer to functions whose names begin with "read-" as "reading functions" or "readers", since they read individual characters and construct a Lisp object as a returned value. Examples of readers the Lisp system provides are **read, readline,** and **read-delimited-string. read** returns Lisp objects of many types. **readline** and **read-delimited-string** return strings.

read-two-lines-1 reads two lines of input from the console. You type each line in its own editing context. After you enter the first line by pressing RETURN, LINE, or END, you can no longer rub out or otherwise edit any of the characters in the first line. You can type and edit only the second line at that point.

```
(defun read-two-lines-1 () (list (readline) (readline)))
```

read-two-lines-2 lets you edit both lines in a single context by using the **with-input-editing** special form. Even after entering the first line you can edit it. For example, the m-< input editor command moves the cursor to the first character of the first line. **read-two-lines-2** also adds a stream parameter so that you can read from different streams without having to bind ***standard-input***. You can also use this function for reading from noninteractive streams, such as file streams.

```
(defun read-two-lines-2 (&optional (stream *standard-input*))
  (with-input-editing (stream) (list (readline stream) (readline stream))))
```

read-two-lines-3 demonstrates the use of the **:prompt** input editor option and the **:end-activation** option for **with-input-editing**. When you invoke this function on an interactive stream you receive a prompt. This prompt is redisplayed if typeout to the stream occurs. This might happen if you press HELP or the window receives a notification.

The **:end-activation** option defines **#\end** as an activation character. This lets you activate previous input to **read-two-lines-3**, after yanking and editing it, by pressing END. The **:prompt** and **:end-activation** options have no effect on the behavior of the function for noninteractive streams.

```
(defun read-two-lines-3 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type two lines: "))
    (with-input-editing (stream :end-activation)
      (list (readline stream) (readline stream)))))
```

read-n-lines is like **read-two-lines** except that you specify the number of lines to be read using the **n-lines** argument. It also uses a prompt function instead of a string to generate the prompt.

```
(defun read-n-lines-prompt (stream ignore n-lines)
  (format stream "Type ~R line~:P:~%" n-lines))

(defun read-n-lines (n-lines &optional (stream *standard-input*))
  (with-input-editing-options ((:prompt 'read-n-lines-prompt n-lines))
    (with-input-editing (stream :end-activation)
      (loop repeat n-lines collect (readline stream)))))
```

Next is an example of a simple sentence parser. It builds a list of strings and symbols that represent the words and punctuation marks of the sentence. A sentence may be any number of lines long. It is delimited by a period or a question mark. Words are delimited by a space, newline, or punctuation mark.

This is also an example of a reading function written entirely in terms of **:tyi** as the primitive input operation.

```
(defun read-sentence-1 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream)
      (loop named sentence
            with sentence = nil
            for word = (make-array 20. :type art-string :fill-pointer 0)
            do (loop for char = (send stream :tyi)
                     do
                 (cond ((memq char '(#\space #\return #/. #/? #/,))
                        (if (not (equal word ""))
                            (push word sentence))
                        (selectq char
                          ((#\space #\return #/,)
                           (return))
                          (#\.
                           (push :period sentence)
                           (return-from sentence (nreverse sentence)))
                          (#\?
                           (push :question-mark sentence)
                           (return-from sentence (nreverse sentence)))))
                       (t (array-push-extend word char))))))))
```

Following is a different sentence parser that calls **read-delimited-string** to accumulate characters into a string. It uses the **:end-activation** option for **with-input-editing** so that previous input to **read-sentence-2** can be yanked, edited, and activated using the END key. When it detects incorrect uses of punctuation, it calls **zl:parse-ferror** to signal an error caught by the input editor.

```
(defun read-sentence-2 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream :end-activation)
      (loop with sentence = nil
            do (multiple-value-bind (word nil delimiter)
                   (read-delimited-string
                     '(#\space #\return #/. #/? #/, #/: #/;) stream)
                 (if (not (equal word ""))
                     (push word sentence))
                 (cond ((memq delimiter '(#\space #\return)))
                       ((null sentence)
                        (if (eq delimiter #\end)
                            (return nil)
                            (sys:parse-ferror
                              "The punctuation mark /"~C/" occurred at the ~
                             beginning of the sentence."
                              delimiter)))
                       ((symbolp (car sentence))
                        (sys:parse-ferror
                          "The punctuation mark /"~C/" was typed after a ~@∧."
                          delimiter (car sentence)))
                       (t (selectq delimiter
                            (#/,
                             (push ':comma sentence))
                            (#/:
                             (push ':colon sentence))
                            (#/;
                             (push ':semicolon sentence))
                            (#/.
                             (push ':period sentence)
                             (return (nreverse sentence)))
                            (#/?
                             (push ':question-mark sentence)
                             (return (nreverse sentence))))))))))
```

Sometimes an error in parsing is detected not by the function that invokes the input editor, but by some function that it calls. In the next example, **read-time** invokes **time:parse-universal-time** to do its parsing. If we did not use the **condition-case** form in **read-time**, we would enter the Debugger when **time:parse-universal-time** encountered incorrect input. The **condition-case** form encapsulates the original error in one of flavor **zl:parse-ferror** so that the input editor catches it. Alternately, we could define **time:parse-error** to be a subflavor of **sys:parse-error.**

```
(defun read-time (&optional (stream *standard-input*))
  (with-input-editing (stream :line)
    (let ((string (readline-or-nil stream)))
      (when string
        (condition-case (error)
            (time:parse-universal-time string)
          (time:parse-error
            (sys:parse-ferror "~A" error)))))))
```

13.7 Input Editor Messages to Interactive Streams

:input-editor *read-function* &rest *read-args* of **si:interactive-stream** *Method*

Apply *read-function* to *read-args* after invoking the input editor. For more information: See the section "The Input Editor Program Interface", page 265.

Normally a program does not send this message itself; it uses the special form **with-input-editing**. See the special form **with-input-editing**, page 270.

:start-typeout *type* &optional *spacing* of **si:interactive-stream** *Method*

Informs the input editor that typeout to the window will follow. The word "typeout" is used in the name of this message because this is very similar to typeout in the editor, even though typeout windows are not actually used. *type* can be one of the following keywords:

Keyword	*Action*
:insert	Typeout is inserted before the current input, as is done with notifications or input editor documentation.
:overwrite	Like **:insert**, but the next time **:insert** or **:overwrite** typeout is performed, this typeout is overwritten.
:append	Typeout appears after the current input, which remains visible before the typeout. This is the style used by **zl:break**.
:temporary	Typeout appears after the current input and is erased after the user types a character.
:clear-window	The window is cleared, and typeout appears at the top.

spacing can be one of the following keywords:

Keyword	*Action*
:none	No spacing before typeout.
:fresh-line	Typeout begins at the beginning of a line.
:blank-line	A blank line precedes typeout.

If *spacing* is not specified, a default that depends on *type* is computed.

si:*typeout-default* *Variable*

Controls the style of typeout performed by the input editor. Permissible values are the keywords acceptable as the *type* argument to the **:start-typeout** method of **si:interactive-stream.** These are **:insert, :overwrite, :append, :temporary,** and **:clear-window.** The default value is **:overwrite.**

:finish-typeout &optional *spacing erase?* of **si:interactive-stream** *Method*

Completes typeout to the window and causes the input buffer to be refreshed. In the case of **:temporary** typeout, the *erase?* parameter is used to indicate whether or not the typeout overwrote part of the current input by wrapping around the screen. It is the responsibility of the program doing the typeout to keep track of how much is output.

spacing can be one of the following keywords:

Keyword	*Action*
:none	No spacing before typeout.
:fresh-line	Typeout begins at the beginning of a line.
:blank-line	A blank line precedes typeout.

If *spacing* is not specified, a default that depends on the *type* argument to the **:start-typeout** method is computed.

:rescanning-p of **si:interactive-stream** *Method*

This message can be sent by a read function that uses the input editor to determine whether the next character returned by **:tyi** will come from the input buffer or from the keyboard. If **t** is returned, the input is being rescanned and the next character will come from the input buffer. If **nil** is returned, the next character will come from the keyboard.

:force-rescan of **si:interactive-stream** *Method*

This message can be sent by a read function that uses the input editor to force a rescan of the current input. Before this message is sent, usually some global state has changed and the contents of the input buffer are interpreted differently.

:replace-input *n-chars string* &optional (*begin* **0**) *end* (*rescan-mode* **:ignore**) of **si:interactive-stream** *Method*

This message can be sent by a read function that uses the input editor to provide completion of the current input.

n-chars specifies the number of characters to be removed from the end of the input buffer and erased from the screen. It can be an integer, a string, or **nil**:

integer	Remove *n-chars* characters from immediately before the scan pointer
string	Remove as many characters as the string contains
nil	Remove characters from the beginning of the input buffer to the scan pointer

The substring of *string* determined by *begin* and *end* is then displayed on the screen. *end* defaults to **(string-length** *string***)**. The scan pointer is left after the string, and a rescan does not take place. If a rescan takes place at some later time, the characters in *string* are seen as input.

rescan-mode specifies what action to take if the **:replace-input** message is sent when the scan pointer is not at the end of the input buffer:

:ignore	Don't perform the **:replace-input** operation. This is the default.
:enable	Perform the operation.
:error	Signal an error.

:read-bp of **si:interactive-stream** *Method*

Returns the value of the scan pointer. This is for the benefit of read functions that might want to return a pointer into the input buffer when signalling an error of type **sys:parse-error**.

:noise-string-out *string* &optional (*rescan-mode* **:ignore**) of **si:interactive-stream** *Method*

This message can be sent by a read function to display a string that is not to be treated as input. For example, the string might prompt the user for a particular kind of input. *string* is displayed on the screen without changing the scan pointer, and a rescan does not take place. If a rescan takes place at some later time, the characters in *string* are ignored.

rescan-mode specifies what action to take if the **:noise-string-out** message is sent when the scan pointer is not at the end of the input buffer:

:ignore	Don't perform the **:noise-string-out** operation. This is the default.
:enable	Perform the operation.
:error	Signal an error.

14. Printed Representation

14.1 How the Printer Works

People cannot deal directly with Lisp objects, because the objects live inside the machine. In order to let us get at and talk about Lisp objects, Lisp provides a representation of objects in the form of printed text; this is called the *printed representation.*

Functions such as **print**, **prin1**, and **princ** take a Lisp object and send the characters of its printed representation to a stream. These functions (and the internal functions they call) are known as the *printer.* The **zl:read** function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds a corresponding object and returns it; **zl:read** and its subfunctions are known as the *reader.* See the section "Introduction to Streams", page 5.

The printed representation of an object depends on its type. For descriptions of how different Lisp objects are printed:

See the section "Printed Representation of Symbols", page 288.
See the section "Printed Representation of Common Lisp Character Objects", page 288.
See the section "Printed Representation of Strings", page 288.
See the section "Printed Representation of Instances", page 288.
See the section "Printed Representation of Arrays That Are Named Structures", page 289.
See the section "Printed Representation of Arrays That Are Not Named Structures" page 289.
See the section "Printed Representation of Miscellaneous Data Types", page 289.
See the section "Controlling the Printed Representation of an Object", page 291.

14.2 Effects of Slashification on Printing

Printing is done either with or without *slashification.* The unslashified version is nicer looking, but **zl:read** cannot handle it properly. The slashified version, however, is carefully set up so that **zl:read** is able to read it in.

The primary effects of slashification are:

- Special characters used with other than their normal meanings (for example, a parenthesis appearing in the name of a symbol) are preceded by slashes or cause the name of the symbol to be enclosed in vertical bars.
- Symbols that are not from the current package are printed out with their package prefixes. (A package prefix looks like a symbol followed by a colon).

14.3 What the Printer Produces

14.3.1 Printed Representation of Symbols

If slashification is off, the printed representation of a symbol is simply the successive characters of the print-name of the symbol. If slashification is on, two changes must be made.

1. The symbol might require a package prefix for **zl:read** to work correctly, assuming that the package into which **zl:read** reads the symbol is the one in which it is being printed. (See the section "System Packages" in *Symbolics Common Lisp: Language Concepts.*)
2. If the printed representation would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), the printed representation must have one of the following kinds of quoting for those characters.
   - Slashes ("/") before each special character
   - Vertical bars ("|") around the whole name

The decision whether quoting is required is made using the readtable, so it is always accurate provided that **zl:readtable** has the same value when the output is read back in as when it was printed. See the variable **zl:readtable**, page 235.

Uninterned symbols are printed preceded by #:. You can turn this off by evaluating **(zl:setf (si:pttbl-uninterned-prefix zl:readtable) "")**.

14.3.2 Printed Representation of Common Lisp Character Objects

For Common Lisp, character objects always print as #\char.

14.3.3 Printed Representation of Strings

If slashification is off, the printed representation of a string is simply the successive characters of the string. If slashification is on, the string is printed between double quotes, and any characters inside the string that need to be preceded by slashes are. Normally these are just double-quote and slash. Compatibly with Maclisp, carriage return is *not* ignored inside strings and vertical bars.

14.3.4 Printed Representation of Instances

If the instance has a method for the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure, and whether slashification is enabled. The object should print a suitable printed representation on the stream. (See the section "Flavors" in *Symbolics Common*

Lisp: Language Concepts. Instances are discussed there.) See the section "Printed Representation of Miscellaneous Data Types", page 289. Most such objects print as described there, except with additional information such as a name. Some objects print only their name when slashification is not in effect (when **princ**ed).

14.3.5 Printed Representation of Arrays That Are Named Structures

If the array has a named structure symbol with a **named-structure-invoke** property that is the name of a function, then that function is called on five arguments:

- The symbol **:print-self**
- The object itself
- The stream to print to
- The current *depth* of list structure
- Whether slashification is enabled

A suitable printed representation should be sent to the stream. This allows you to define your own printed representation for the array's named structures. See the section "Named Structures" in *Symbolics Common Lisp: Language Concepts.* If the named structure symbol does not have a **named-structure-invoke** property, the printed representation is like that for miscellaneous data types: a number sign and a less-than sign ("<"), the named structure symbol, the numerical address of the array, and a greater-than sign (">").

14.3.6 Printed Representation of Arrays That Are Not Named Structures

The printed representation of an array that is not a named structure contains the following elements, in order:

- A number sign and a less-than sign ("<")

- The "**art-**" symbol for the array type

- The dimensions of the array, separated by hyphens

- A space, the machine address of the array, and a greater-than sign (">")

14.3.7 Printed Representation of Miscellaneous Data Types

For a miscellaneous data type, the printed representation starts with a number sign and a less-than sign, the "**dtp-**" symbol for this data type, a space, and the octal machine address of the object. Then, if the object is a microcoded function, compiled function, or stack group, its name is printed. Finally, a greater-than sign is printed.

Including the machine address in the printed representation makes it possible to tell two objects of this kind apart without explicitly calling **eq** on them. This can be very useful during debugging. It is important to know that if garbage collection is turned on, objects are occasionally moved, and therefore their octal machine addresses are changed. It is best to shut off garbage collection temporarily when depending on these numbers.

None of the printed representations beginning with a number sign can be read back in, nor, in general, can anything produced by instances and named structures. See the section "What the Reader Recognizes", page 227. This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. But by setting the **si:print-readably** variable, you can make sure that what you are printing can indeed be read with the reader.

si:print-readably *Variable*

When **si:print-readably** is bound to **t**, the printer signals an error if there is an attempt to print an object that cannot be interpreted by **zl:read**. When the printer sends a **:print-self** or a **:print** message, it assumes that this error checking is done for it. Thus it is possible for these messages *not* to signal an error, if they see fit.

sys:printing-random-object *(object stream . keywords) body...* *Macro*

The vast majority of objects that define **:print-self** messages have much in common. This macro is provided for convenience, so that users do not have to write out that repetitious code. It is also the preferred interface to **si:print-readably**. With no keywords, **sys:printing-random-object** checks the value of **si:print-readably** and signals an error if it is not **nil**. It then prints a number sign and a less-than sign, evaluates the forms in *body*, then prints a space, the octal machine address of the object, and a greater-than sign. A typical use of this macro might look like:

```
(sys:printing-random-object (ship stream)
  (princ (typep ship) stream)
  (tyo #\space stream)
  (prin1 (ship-name ship) stream))
```

This might print #<**ship "ralph" 23655126**>.

The following keywords can be used to modify the behavior of **sys:printing-random-object**:

:no-pointer	This suppresses printing of the octal address of the object.
:typep	This prints the result of **(typep** *object***)** after the less-than sign. In the example above, this option could have been used instead of the first two forms in the body.

14.4 Controlling the Printed Representation of an Object

If you want to control the printed representation of an object, usually you make the object an array that is a named structure, or an instance of a flavor. See the section "Named Structures" in *Symbolics Common Lisp: Language Concepts.* See the section "Flavors" in *Symbolics Common Lisp: Language Concepts.* Occasionally, however, you might want to get control over all printing of objects in order to change in some way how they are printed. The best way to do this is to customize the behavior of **si:print-object**, which is the main internal function of the printer. All the printing functions, such as **print** and **princ**, as well as **zl:format**, go through this function. The way to customize it is by using the "advice" facility. See the special form **advise** in *Program Development Utilities.*

15. Output Functions

These functions all take an optional argument called *output-stream*, which is where to send the output. If unsupplied or **nil**, *output-stream* defaults to the value of ***standard-output***. If it is **t**, the value of ***terminal-io*** is used (that is, the interactive terminal). See the section "Introduction to Streams", page 5. Streams are documented in detail in that section.

write *object* &key *stream escape radix base circle pretty level length case gensym array readably array-length string-length bit-vector-length structure-contents abbreviate-quote* *Function*

The printed representation of *object* is written to the output stream specified by **:stream**, which defaults to the value of ***standard-output***.

The other keyword arguments specify values used to control the generation of the printed representation. Each defaults to the corresponding global variable: see ***print-escape***, ***print-radix***, ***print-base***, ***print-circle***, ***print-pretty***, ***print-level***, ***print-length***, ***print-case***, ***print-gensym***, ***print-array***, ***print-readably***, ***print-array-length***, ***print-string-length***, ***print-bit-vector-length***, ***print-structure-contents***, and ***print-abbreviate-quote***. Note that the printing of symbols is also affected by the value of the variable ***package***.

write returns *object*. For example:

```
(write "A simple string") => "A simple string"
"A simple string"
```

prinl *object* &optional *output-stream* *Function*

Outputs the printed representation of *object* to *stream*, with slashification. Roughly speaking, the output from **prinl** is suitable for input to the function **read**. **prinl** returns *object*. See the section "What the Printer Produces", page 288.

For example:

```
(prin1 "A simple string") => "A simple string"
"A simple string"
```

zl:prinl-then-space *object* &optional *output-stream* *Function*

Like **prinl** except that output is followed by a space. **zl:prinl-then-space** returns *object*. For example:

```
(zl:prin1-then-space "A simple string") => "A simple string"
 "A simple string"
```

prinl-to-string *object* *Function*

The object is printed as if by **prinl**, and the characters that would be output are made into a string, which is returned. For example:

```
(prin1-to-string '|red|) => "\"|red|\""
```

print *object* &optional *output-stream* *Function*

Like **prinl** except that output is preceded by a newline and followed by a space. **print** returns *object.* For example:

```
(print "A simple string") =>
"A simple string"
 "A simple string"
```

princ *object* &optional *output-stream* *Function*

Like **prinl** except that the output is not slashified. A symbol is printed as simply the characters of its print name, a string is printed without surrounding double quotes, and so on. The general rule is that output from **princ** is intended to look good to people, while output from **prinl** is intended to be acceptable to the function read. **princ** returns *object.*

```
(princ "A simple string") => A simple string
"A simple string"
```

princ-to-string *object* *Function*

The object is printed as if by **princ**, and the characters that would be output are made into a string, which is returned. For example:

```
(princ-to-string '|red|) => "|red|"
```

pprint *object* &optional *output-stream* *Function*

The printed representation of *object* is written to the *output-stream* using the pretty printer. The printed representation is preceded by a newline and escape characters are used as appropriate. **pprint** returns no values. For example:

```
(pprint "A simple string") =>
"A simple string"
```

write-byte *integer binary-output-stream* *Function*

Writes one byte, the value of *integer* to *binary-output-stream.* It is an error if *integer* is not of the type specified as the **:element-type** argument to **open** when the stream was created. **write-byte** returns *integer.*

write-char *character* &optional *output-stream* *Function*

Outputs *character* as a printing character to *output-stream*, and returns *character* as a character object. *character* must be a character object. For example:

```
(write-char #\a) => a
#\a
```

write-line *string* &optional *output-stream* &key (*start* **0**) *end* *Function*

Writes the characters of the specified substring of *string* to *output-stream*, followed by a newline. The **:start** and **:end** parameters delimit a substring of string. **write-line** returns *string*. For example:

```
(write-line "hello") => hello
"hello"

(setq stream (make-string-output-stream))
  => #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 35643762>

(write-line "two words" stream :start 4)
  => "two words"        ;returns the full string

(get-output-stream-string stream)
 => "words
"        ;writes the substring plus NEWLINE to the stream
```

write-string *string* &optional *output-stream* &key (*start* **0**) *end* *Function*

Writes the characters of the specified substring of *string* to *output-stream*, without a following newline. The **:start** and **:end** parameters delimit a substring of string. **write-string** returns *string*. For example:

```
(write-string "hello") => hello"hello"

(setq s (make-string-output-stream))
  => #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 14372772>

(write-string "two words" s :start 4)
  => "two words"        ;returns the full string

(get-output-stream-string s)
  => "words"            ;writes the substring to the stream
```

write-to-string *object* &key *escape radix base circle pretty level length* *Function*
case gensym array readably array-length
string-length pretty level length case gensym

array structure-contents

The object is printed as if by **write**, and the characters that would be output are made into a string, which is returned. The other keyword arguments specify values used to control the generation of the printed representation. See the function **write**, page 293.

For example:

```
(write-to-string '|red|) => "\"|red|\""
```

si:print-object *object prindepth slashify-p stream* &optional *which-operations* *Function*

Outputs the printed representation of *object* to *stream*, as modified by *prindepth* and *slashify-p*. This is the guts of the Lisp printer. When a stream's **:print** handler calls this function, it should supply the list **(:string-out)** for *which-operations*, to prevent itself from being called recursively. It can supply **nil** if it does not want to receive **:string-out** messages.

Advising this function is the way to customize the behavior of all printing of Lisp objects. See the special form **advise** in *Program Development Utilities*.

si:print-list *list prindepth slashify-p stream which-operations* *Function*

This is the part of the Lisp printer that prints lists. A stream's **:print** handler can call this function, passing along its own arguments and its own which-operations, to arrange for a list to be printed the normal way and the stream's **:print** hook to get a chance at each of the list's elements.

zl:tyo *char* &optional *stream* *Function*

Outputs the character *char* to *stream*.

fresh-line &optional *output-stream* *Function*

Outputs a newline only if the stream is not already at the start of a line. If for any reason this cannot be determined, then a newline is output anyway. This guarantees that the stream will be on a fresh line while consuming as little vertical space as possible. **fresh-line** returns **t** if it output a newline, otherwise it returns **nil**.

terpri &optional *output-stream* *Function*

Outputs a newline to *output-stream*, and returns **nil**. It is identical in effect to:

```
(write-char #\Newline output-stream)
```

zl:terpri &optional *stream* *Function*

Outputs a carriage return character to *stream.*

clear-output &optional *output-stream* *Function*

Some streams are implemented in an asynchronous, or buffered, manner. **clear-output** attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination. This is useful, for example, to abort a lengthy output to the terminal when an asynchronous error occurs. **clear-output** returns **nil.**

finish-output &optional *output-stream* *Function*

Some streams are implemented in an asynchronous, or buffered, manner. **finish-output** attempts to ensure that all output sent to *output-stream* has reached its destination, and only then returns **nil.**

force-output &optional *output-stream* *Function*

Some streams are implemented in an asynchronous, or buffered, manner. **force-output** initiates the emptying of any internal buffers, but returns **nil** without waiting for completion or acknowledgment.

15.1 Print Control Variables

There are a number of print variables that affect the performance of print functions.

print-level *Variable*

The variable ***print-level*** control how many levels of a nested data object will be printed. Its value can be either **nil** (the default), or any positive integer up to 2^{31}-1. This variable replaces **zl:prinlevel**, which is obsolete.

The entire object prints if

- the value of ***print-level*** is **nil**
- the value of ***print-level*** is equal to or greater than the number of levels in the object

If ***print-level*** is an integer, it indicates the maximum level to be printed. The object itself is level 0; its components (as for a list or vector) are level 1; and so on. If any part the object to be printed has components at or greater than the value of ***print-level***, then that part of the object is printed as simply #.

Examples:

```
(setq list '(a (b c) (d (e f) g))) => (A (B C) (D (E F) G))

(let ((*print-level* nil))
        (print list) nil) =>
(A (B C) (D (E F) G)) NIL

(let ((*print-level* 2))
        (print list) nil) =>
(A (B C) (D # G)) NIL

(let ((*print-level* 3))
        (print list) nil) =>
(A (B C) (D (E F) G)) NIL
```

print-length *Variable*

The variable ***print-length*** controls how many elements at a given level are printed. Its value can be either **nil** (the default), or any positive integer up to 2^{31}-1. This variable replaces **zl:prinlength**, which is obsolete.

The entire object prints if

- the value of ***print-length*** is **nil**
- The value of ***print-length*** is equal to or greater than the number of of components in any given level of the object

If ***print-length*** is an integer, it indicates the maximum number of components to be printed. If the object to be printed has components at or greater than the value of ***print-level***, then the object's structure name is printed. Note that the number of components begins with 0, so the entire

Examples:

```
(setq list '(a b (c) (d (e f) g))) => (A B (C) (D (E F) G))

(let ((*print-length* nil))
        (print list) nil) => (A B (C) (D (E F) G)) NIL

(let ((*print-length* 2))
        (print list) nil) => (A B ...) NIL

(let ((*print-length* 4))
        (print list) nil) => (A B (C) (D (E F) G)) NIL
```

print-abbreviate-quote *Variable*

The variable ***print-abbreviate-quote*** provides a way to print quoted forms in their short form. It is incorporated into ***print-pretty***, so the value of ***print-pretty*** must be **nil** in order for ***print-abbreviate-quote*** to have any effect.

Examples:

```
(let ((*print-abbreviate-quote* nil)
      (*print-pretty* nil))
        (print '(quote foo)) nil) => (QUOTE FOO) NIL

(let ((*print-abbreviate-quote* t)
      (*print-pretty* nil))
        (print '(quote foo)) nil) => 'FOO NIL

(let ((*print-abbreviate-quote* t)
      (*print-pretty* nil))
        (print '(function foo)) nil) => #'FOO NIL

(let ((*print-abbreviate-quote* t)
      (*print-pretty* nil))
        (print ''(foo ,@bar ,baz)) nil)
  => '(FOO ,@BAR ,BAZ) NIL
```

print-array *Variable*

The variable ***print-array*** is a boolean which controls whether the contents of arrays other than strings are printed. If the value of ***print-array*** is **nil**, then the array's structure name is printed in a concise form, such as #<ART-Q-4-2 270017201>, that identifies the array and gives the dimensions. If the value is **t**, non-string arrays are printed using #(, #*, or #*n*A syntax.

This variable replaces **si:prinarray**, which is obsolete.

print-array-length *Variable*

The variable ***print-array-length*** controls the number of objects in the array that will be printed. Its value can be either **nil** (the default), or any positive integer up to 2^{31}-1.

The entire array prints if

- the value of ***print-array-length*** is **nil**
- the value of ***print-array-length*** is equal to or greater than the length of the array to be printed

This variable is dependent on the value of the variable ***print-array***. If the value of ***print-array*** is **nil**, the array's structure name (which includes the array's length) is printed, no matter what the value of ***print-array-length*** is. The array's structure name is also printed when the array is longer than the integer value of ***print-array-length***.

Examples:

```
(setq array (make-array '(4 2) :initial-contents
        '((a b)
        (1 2)
        ("foo" "bar")
        (#\a #\b))))
  => #2A((A B) (1 2) ("foo" "bar") (#\a #\b))

(let ((*print-array-length* nil))
    (print array) nil)
  => #2A((A B) (1 2) ("foo" "bar") (#\a #\b)) NIL

(let ((*print-array-length* 2))
    (print array) nil)
  => #<ART-Q-4-2 10004306> NIL

(let ((*print-array-length* 8))
    (print array) nil)
 => #2A((A B) (1 2) ("foo" "bar") (#\a #\b)) NIL
```

print-base *Variable*

The value of this variable determines the radix in which the printer prints rational numbers (integers and ratios).

print-base can have any integer value from 2 to 36, inclusive; its default value is 10 (decimal radix). For values above 10, letters of the alphabet are used to represent digits above 9.

If no radix specifier is set (see ***print-radix***) integers in base ten are printed without a trailing decimal point.

If the value of ***print-base*** is a symbol that has a **si:princ-function** property (such as **:roman** or **:english**), the value of the property is applied to two arguments

- - of the number to be printed
- the stream to which to print it

This allows output in roman numerals and the like.

Examples:

```
(setq *print-base* ':roman)
(* 5 5) ==> XXV

(setq *print-base* ':english)
(* 5 5) ==> twenty-five
```

print-bit-vector-length *Variable*

The variable ***print-bit-vector-length*** controls the number of objects in the bit vector that will be printed. Its value can be either **nil** (the default), or any positive integer up to 2^{31}-1.

When the value of ***print-bit-vector-length*** is **nil**, ***print-bit-vector-length*** interacts with ***print-array***. Here is a table that shows the interactions:

print-bit-vector-length	***print-array***	Result
t	*	always print the bit vector
integer	*	print the bit vector if the value of ***print-bit-vector-length*** is equal to or greater than the length of the bit vector to be printed
nil	t	always print the bit vector
nil	nil	never print the bit vector

* means that the value of this variable does not affect the result

Examples:

```
(setq bit-vector (make-array 5 :element-type 'bit
        :initial-contents '(1 0 0 1 0))) => #*10010

(let ((*print-bit-vector-length* 2)
      (*print-array* t))
     (print bit-vector) nil)
  => #<ART-1B-5 10052423> NIL

(let ((*print-bit-vector-length* 5)
      (*print-array* t))
     (print bit-vector) nil)
  => #*10010 NIL
```

print-case *Variable*

The variable ***print-case*** controls the case in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used. The **read** function normally converts lowercase characters appearing in symbol names to their corresponding uppercase characters. This means that normally internal print names contain only uppercase letters. However, users may prefer to see output using lowercase or mixed case letters.

Lowercase characters in the internal print name are always printed in lowercase and are preceded by a single escape character or enclosed by multiple escape characters. Uppercase characters in the internal print name are printed in uppercase, lowercase, or in mixed case so as to capitalize words, according to the value of ***print-case***. The convention for what constitutes a "word" is the same as for the function **string-capitalize.**

The value of ***print-case*** must be one of the keywords **:upcase** (the default), **:downcase**, or **:capitalize.** This variable replaces **si:princase**, which is obsolete.

print-circle *Variable*

The variable ***print-circle*** controls whether or not the printer tries to detect cycles in the structure to be printed. When the value of ***print-circle*** is **nil** (the default), then the printing process proceeds by recursive descent. Attempts to print a circular structure may lead to looping behavior and failure to terminate.

When the value is non-**nil**, then the printer tries to detect cycles in the structure to be printed, and uses #*n*= and #*n*# syntax to indicate the circularities.

print-escape *Variable*

The variable ***print-escape*** controls whether or not the printer outputs escape characters. When the value of ***print-escape*** is **nil**, then escape characters are not output when an expression is printed. In particular, a symbol is printed by simply printing the characters of its print name. The function **princ** effectively binds ***print-escape*** to **nil**.

When the value is **t** (the default), then an attempt is made to print an expression in such a way that it can be read again to produce an **equal** structure. The function **prinl** effectively binds ***print-escape*** to **t**.

print-gensym *Variable*

The variable ***print-gensym*** controls whether the prefix #: is printed before symbols that have no home package. The prefix is printed if the value of ***print-gensym*** is non-**nil**. The initial value is **t**.

print-pretty *Variable*

The variable ***print-pretty*** controls the amount of whitespace output when printing an expression. When the value of ***print-pretty*** is **nil**, then only a small amount of whitespace is output. When the value is non-**nil**, the output is adjusted to be more readable. Common Lisp only uses the values **t** and **nil**. Symbolics has added the values **:code**, **:data**, **:plist** and **:alist**.

The permissible values are:

Value	Effect
nil	Disable pretty printing
t	Print in the default format (the default is **:code**)
:code	Prints lists as if they were lisp code (SCL extension)
:data	Prints lists with a format based on the first element (SCL extension)
:plist	Prints lists as property lists, with two elements per line (SCL extension)
:alist	Prints lists as association lists, giving a dotted cdr for each sublist, even when there is a proper list (SCL extension)

Examples:

```
(write '(defun defvar defparameter defflavor) :pretty t)
  => (DEFUN DEFVAR DEFPARAMETER
      DEFFLAVOR)
```

```
(write '(defun defvar defparameter defflavor) :pretty :data)
  => (DEFUN DEFVAR DEFPARAMETER DEFFLAVOR)


(write '((defun function)
         (defvar variable)
         (defflavor flavor))
       :pretty t)
  => ((DEFUN FUNCTION) (DEFVAR VARIABLE) (DEFFLAVOR FLAVOR))

(write '((defun function)
         (defvar variable)
         (defflavor flavor))
       :pretty :alist)
  => ((DEFUN . (FUNCTION))
      (DEFVAR . (VARIABLE))
      (DEFFLAVOR . (FLAVOR)))
```

print-pretty-printer *Variable*

The variable ***print-pretty-printer*** allows wholesale replacement of the pretty printer used by Common Lisp. Its value is a function, which will be called with three arguments: the object to be printed, the value of ***pretty-printer***, and the output stream. The default is the Common Lisp pretty printer.

print-radix *Variable*

If this variable is set to **t**, rational numbers are printed with a radix specifier indicating what radix the printer is using. (The current radix is controlled by the value of variable ***print-base***).

The default value of ***print-radix*** is **nil**.

The radix specifier has the general format

#nnrddddd

where *n* is an unsigned decimal integer in the range 2 - 36 (inclusive) representing the radix, and *ddddd* denotes the number in radix *n*.

When the value of ***print-base*** is 2, 8, or 16 (that is, binary, octal, or hexadecimal) the radix specifier is printed in the abbreviated form, #b, #o, #x, using lower case letters.

For printing integers, base ten is indicated by a trailing decimal instead of a leading radix specifier; for ratios, however, the specifier #10r is printed.

print-readably *Variable*

The variable ***print-readably*** is a boolean that signals an error if the object to be printed is not in a form that the reader will accept. This is useful for objects such as arrays and flavor instances which are not forms that the reader accepts.

```
(defflavor food () ()) => FOOD

(setq apple (make-instance 'food)) => #<FOOD 10074402>

(let ((*print-readably* nil)) (print apple) nil) =>
#<FOOD 10074402> NIL

(let ((*print-readably* t)) (print apple) nil)
Rebinding the following specials;
use Show Standard Value Warnings for details:
  *PRINT-PRETTY*, *PRINT-READABLY*, and GPRINT:*INSPECTING*

Error: Can't print #<FOOD 10074402> readably

SYS:PRINT-NOT-READABLE:
   Arg 0 (SI:OBJECT): #<FOOD 43123626>
s-A, <RESUME>:   Proceed without any special action
s-B, <ABORT>:    Return to Lisp Top Level in Dynamic Lisp Listener 1
→ Abort Abort
Return to Lisp Top Level in Dynamic Lisp Listener 1
Back to Lisp Top Level in Dynamic Lisp Listener 1.
```

print-string-length *Variable*

The variable ***print-string-length*** controls the number of string characters that will print. Its value can be either **nil** (the default), or any positive integer up to 2^{31}-1.

The entire string prints if

- the value of ***print-string-length*** is **nil**
- the value of ***print-string-length*** is equal to or greater than the length of the string to be printed

Only the structure name (which includes the string's length) is printed when the string is longer than the integer value of ***print-string-length***.

Examples:

```
(let ((*print-string-length* nil))
     (print "This is a very long string") nil)
  => "This is a very long string" NIL

(let ((*print-string-length* 4))
     (print "This is a very long string") nil)
  => #<ART-STRING-26 36450275> NIL

(let ((*print-string-length* 4))
     (print "chip") nil) => "chip" NIL
```

The **format** function can do anything any of the above functions can do and is very useful for producing nicely formatted text. See the function "**format**", page 309. **format** can generate a string or output to a stream.

The **grindef** function is useful for formatting Lisp programs. See the special form **grindef**, page 329.

See the special form **with-output-to-string**, page 139.

stream-copy-until-eof *from-stream to-stream* &optional *leader-size* *Function*

Inputs characters from *from-stream* and outputs them to *to-stream*, until it reaches the end-of-file on the *from-stream*. For example, if **x** is bound to a stream for a file opened for input, then **(stream-copy-until-eof x zl:terminal-io)** prints the file on the console.

If *from-stream* supports the **:line-in** operation and *to-stream* supports the **:line-out** operation, then **stream-copy-until-eof** uses those operations instead of **:tyi** and **:tyo**, for greater efficiency. *leader-size* is passed as the argument to the **:line-in** operation.

beep &optional *beep-type (stream* **zl:terminal-io***)* *Function*

Tries to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the **:beep** operation, then this function sends it a **:beep** message, passing *type* along as an argument. Otherwise it just causes an audible beep on the terminal. *type* is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; *type* is currently ignored and should always be **nil**. See the message **:beep**, page 42.

zl:cursorpos &rest *args* *Function*

Exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages.

zl:cursorpos normally operates on the **zl:standard-output** stream; however, if the last argument is a stream or **t** (meaning **zl:terminal-io**), **zl:cursorpos** uses that stream and ignores it when doing the operations described below. Note that **zl:cursorpos** only works on streams that are capable of these operations, such as windows. A stream is taken to be any argument that is not a number and not a symbol, or a symbol other than **nil** with a name more than one character long.

(zl:cursorpos) => (*line* . *column*), the current cursor position.

(zl:cursorpos *line column***)** moves the cursor to that position. It returns **t** if it succeeds and **nil** if it does not.

(zl:cursorpos *op***)** performs a special operation coded by *op*, and returns **t** if it succeeds and **nil** if it does not. *op* is tested by string comparison, is not a keyword symbol, and can be in any package.

f	Moves one space to the right.
b	Moves one space to the left.
d	Moves one line down.
u	Moves one line up.
t	Homes up (moves to the top left corner). Note that **t** as the last argument to **zl:cursorpos** is interpreted as a stream, so a stream *must* be specified if the **t** operation is used.
z	Home down (moves to the bottom left corner).
a	Advances to a fresh line. See the **:fresh-line** stream operation.
c	Clears the window.
e	Clear from the cursor to the end of the window.
l	Clear from the cursor to the end of the line.
k	Clear the character position at the cursor.
x	**b** then **k**.

zl:exploden *x* *Function*

Returns a list of characters (as integers) that are the characters that would be typed out by **(princ** *x***)** (that is, the unslashified printed representation of *x*). Example:

```
(exploden '(+ /12 3)) => (#/( #/+ #/Space #/1 #/2 #/Space #/3 #/))
```

zl:explodec *x* *Function*

Returns a list of characters represented by symbols that are the characters that would be typed out by **(princ** *x***)** (that is, the unslashified printed representation of *x*). Example:

```
(explodec '(+ /12 3)) => (|(| + | | |1| |2| | | |3| |)|)
```

zl:explode *x* *Function*

Returns a list of characters represented by symbols that are the characters that would be typed out by **(prin1** *x***)** (that is, the slashified printed representation of *x*). Example:

```
(explode '(+ /12 3)) => (|(| + | | /| |1| |2| /| | | |3| |)|)
```

(Note that there are slashified spaces in the above list.)

sys:flatsize *x* *Function*

Returns the number of characters in the slashified printed representation of *x*. Example:

```
(flatsize '(+ /12 3)) => 12
```

sys:flatc *x* *Function*

Returns the number of characters in the unslashified printed representation of *x*. Example:

```
(flatsize '(+ /12 3)) => 10
```

16. Formatted Output

General formatted output is done using the **format** function. **format** uses a control string written in a special format specifier language to control the output format.

For simple tasks in which only the most basic format specifiers are needed, **format** is easy to use and has the advantage of brevity. For more complicated tasks, the format specifier language becomes somewhat obscure and hard to read, but has the advantage of being extremely powerful.

Additional tools are available for formatting Lisp code (as opposed to text and tables). See the section "Formatting Lisp Code", page 329.

format *destination control-string* &rest *args* *Function*

Produces formatted output. **format** outputs the characters of *control-string*, except that a tilde (~) introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies the kind of formatting desired. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

The output is sent to *destination*. If *destination* is **nil**, a string is created that contains the output; this string is returned as the value of the call to **format**. In all other cases **format** returns no interesting value (generally **nil**). If *destination* is a stream, the output is sent to it. If *destination* is t, the output is sent to ***standard-output***. If *destination* is a string with an array-leader, such as would be acceptable to **string-nconc**, the output is added to the end of that string.

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (:) and at-sign (@) modifiers, and a single character indicating the kind of directive. The alphabetic case of the character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"          ; This is an S directive with no parameters.
"~3,4:@s"     ; This is an S directive with two parameters, 3 and 4,
              ;    and both the colon and at-sign flags.
"~,4S"        ; The first prefix parameter is omitted and takes
              ;    on its default value, while the second is 4.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use **format** efficiently. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, such as the padding character in a right- or left-justifying operation. In this case a single quote (') followed by the desired character can be used as a prefix parameter, so that you do not have to know the decimal numeric values of characters in the character set. For example, you can use the following to print a decimal number in five columns with leading zeros.

`"~5,'0d"` instead of `"~5,48d"`

In place of a prefix parameter to a directive, you can put the letter **V**, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it does not have to be. This feature allows variable column-widths and the like. Also, you can use the character # in place of a parameter; it represents the number of arguments remaining to be processed.

Here are some relatively simple examples of how **format** is used.

```
(format nil "foo") => "foo"
(setq x 5)
(format nil "The answer is ~D." x) => "The answer is 5."
(format nil "The answer is ~3D." x) => "The answer is   5."

(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
(format nil "The character ~:@C is strange." #o1003)
     => "The character Meta-Beta (Symbol-shift-B) is strange."

(setq n 3)
(format nil "~D item~:P found." n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
     => "three dogs are here."
(format nil "~R dog~:*~[~1; is~:;s are~] here." n)
     => "three dogs are here."
(format nil "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
     => "Here are three puppies."
```

~A The next element from the *args* of the **format** function, any Lisp object, is printed without slashification (as by **princ**). **~:A** prints ( ) if the element is **nil**; this is useful when printing something that is always supposed to be a list. **~*n*A** inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. **~*mincol,colinc,minpad,padchar*A** is the full form of **~A**, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~S The next element from the *args* of the **format** function, any Lisp object, is printed with slashification (as by **prinl**). **~:S** prints ( ) if the element is **nil**; this is useful when printing something that is always supposed to be a list. **~*n*S** inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol,colinc,minpad,padchar***S** is the full form of **~S**, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~D The next element from the *args* of the **format** function, normally a number, is printed as a decimal integer. Unlike **print**, **~D** never puts a decimal point after the number. **~*n*D** uses a column width of *n*; spaces are inserted on the left if the number requires fewer than *n* columns for its digits and sign. If the number does not fit in *n* columns, additional columns are used as needed. **~*n*,*m*D** uses *m* as the pad character instead of space. The @ modifier causes the number's sign to be printed always; the default is to print it if only the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter can be used to change the character used as the comma. Thus the most general form of **~D** is ~*mincol,padchar,commachar***D**.

If the element is not an integer, it is printed in **~A** format and decimal base. Thus this directive can be used to print some list structure showing all fixnums in decimal.

~O The next element from the *args* of the **format** function, normally a number, is printed as an octal integer. **~*n*O** uses a column width of *n*; spaces are inserted on the left if the number requires fewer than *n* columns for its digits and sign. If the number does not fit in *n* columns, additional columns are used as needed. **~*n*,*m*O** uses *m* as the pad character instead of space. The @ modifier causes the number's sign to be printed always; the default is to print it only if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter can be used to change the character used as the comma. Thus the most general form of **~O** is ~*mincol,padchar,commachar***O**.

If the element is not an integer, it is printed in **~A** format and octal base. Thus this directive can be used to print some list structure showing all fixnums in octal.

~B Formats a number in binary. For example:

```
(format t "~B" 10.)
1010
NIL
```

~X In Common Lisp, formats a number in hexadecimal; in Zetalisp, prints spaces. For example:

```
(cl:format t "~X" 50.)
32
NIL

(format t "~X" 50.)

NIL
```

~F The next element from the *args* of the **format** function is printed in floating-point format. ~*n*F rounds the element to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed. If the magnitude of the element is too large or too small, it is printed in exponential notation. If the element is not a number, it is printed in **~A** format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired.

The Common Lisp version of ~F produces a different format. Examples:

```
(format nil "~2F" 5)  =>  "5.0"
(format nil "~4F" 5)  =>  "5.0"
(format nil "~4F" 1.5) => "1.5"
(format nil "~4F" 3.14159265)  =>  "3.142"
(format nil "~3F" 1e10)  =>  "1.0e10"

(format nil "~2F" 5) ==> #"5."
(format nil "~4F" 3.14159265) ==> #"3.14"
```

~E The next element from the *args* of the **format** function is printed in exponential format. ~*n*E rounds the element to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed. If the element is not a number, it is printed in **~A** format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired.

The Common Lisp version of ~E is not supported.

~$ The format for using it follows:

~rdig,ldig,field,padchar$

It expects a flonum argument. The modifiers for ~$ are all optional.

rdig	The number of digits after the decimal point. The default is 2.
ldig	The minimum number of digits before the decimal point. The default is 1. It pads on the left with leading zeros.
field	The full width of the field to print in. The default is the number of characters in the output. The field is padded to the left with *padchar* if necessary.
padchar	The character for padding the field if the field is wider than the number. The default is **#\space**.
:	The sign character is to be at the beginning of the field, before the padding, rather than just to the left of the number.
@	The number must always appear signed.

Examples:

```
(format t "~&Pi is ~$" (atan 0 -1))  =>
Pi is 3.14
(format t "~&Pi is ~8$" (atan 0 -1))  =>
Pi is 3.14159265
(format t "~&Pi is ~8,20:$" (atan 0 -1))  =>
Pi is +03.14159265
(format t "~&Pi is ~8,2,20$" (atan 0 -1))  =>
Pi is          03.14159265
(format t "~&Pi is ~8,,20,'x@$" (atan 0 -1))  =>
Pi is xxxxxxxxx+3.14159265
```

It uses free format (~@A) for very large values of the argument.

~C **(character** *arg***)** is put in the output, where *arg* is the next element from the *args* of the **format** function. *arg* is treated as a keyboard character and thus can contain extra modifier bits. The constants **char-control-bit, char-meta-bit, char-hyper-bit, char-super-bit,** and

char-bits return the modifier bits for characters. The modifier bits are printed first, represented as appropriate prefixes: `c-` for Control, `m-` for Meta, `c-m-` for Control plus Meta, `h-` for Hyper, `s-` for Super.

With the colon flag (~:C), the names of the modifier bits are spelled out (for example, "`Control-Meta-F`"), and nonprinting characters are represented by their names (for example, "`Return`") rather than being output as themselves.

With both colon and at-sign (~:@C), the colon-only format is printed, and then if the character requires the `SYMBOL` or `SHIFT` shift key(s) to type it, this fact is mentioned (for example, "`Symbol-1`"). This is the format used for telling the user about a key he or she is expected to press, for instance, in prompt messages.

For all of these formats, if the character is not a keyboard character but a mouse "character", it is printed as `Mouse-`, the name of the button, -, and the number of clicks.

With only an at-sign (~@C), the character is printed in such a way that the Lisp reader can understand it, using "#/" or "#\".

~◊ Takes a character as its argument and prints the name of the character inside a lozenge. The ~C directive does this with some characters, but ~◊ does it with all of them.

~(Format a string in lowercase. The ~(directive must be matched by a corresponding ~) directive. For example:

```
(format t "~(~S~)" 'fs:pathname)
fs:pathname
NIL

(format t "~S" 'fs:pathname)
FS:PATHNAME
NIL
```

~% A carriage return is written to the output. ~*n*% outputs *n* carriage returns. No argument is used. Simply putting a carriage return in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

~& The **:fresh-line** operation is performed on the output stream. Unless the stream knows that it is already at the front of a line, this outputs a carriage return. ~*n*& does a **:fresh-line** operation and then outputs *n-1* carriage returns.

~| Outputs a page separator character (#**page**). *~n|* does this *n* times. With a : modifier, if the output stream supports the **:clear-screen** operation this directive clears the screen; otherwise it outputs page separator character(s) as if no : modifier were present.

~~ Outputs a tilde. *~n~* outputs *n* tildes.

~? The next argument in *args* of the **format** function must be a string, and the argument after that must be a list. The string is processed as a **format** control string, with the elements of the list as the corresponding arguments. The processing of the format string containing ~? resumes when the processing of ~?'s string is finished.

If the @ modifier is supplied, the next argument in *args* must be a string; it is processed as part of the main format string, as if it were substituted for the ~@? directive.

Examples:

```
(format nil "~? ~D" "<~A ~D>" '("Myname" 50.) 7) ==> "<Myname 50> 7"

(format nil "~@? ~D" "<~A ~D>" "Myname" 50. 7) ==> "<Myname 50> 7"
```

~<CR> Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

~* The next element in the *args* of the **format** function is ignored. *~n** ignores the next *n* arguments. **~:*** "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. *~n:** backs up *n* arguments. When within a ~{ construct, the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~@* *~n@** branches to the *n*th argument (0 is the first). **~@*** or **~0@*** goes back to the first argument in the *args* of the **format** function. Directives after a *~n@** take sequential arguments after the one that is the target of the branch. When within a ~{ construct, the branch is relative to the list of arguments being processed by the iteration. This is an "absolute branch". The directive for a relative branch is described elsewhere. See the function "**format**", page 309.

*~n***G** In Zetalisp, "goes to" the *n*th argument. ~0G goes back to the first

argument in the *args* of the **format** function. Directives after this one correspond to the sequence of arguments following the argument that is the target of ~G. Inside a ~{ construct, the "goto" is relative to the list of arguments being processed by the iteration.

This is an "absolute" goto; for a relative goto: See the function **format**, page 309.

The Common Lisp floating-point format specified by ~G is not supported.

~P If the next element in the *args* of the **format** function is not **1**, a lowercase s is output. ("P" is for "plural.") **~:P** does the same thing, after doing a **~:*** to back up one argument; that is, it prints a lowercase s if the last argument were not 1. **~@P** outputs "y" if the argument is 1 or "ies" if it is not. **~:@P** does the same thing but backs up first.

~T Spaces over to a given column. **~*n*,*m*T** outputs enough spaces to move the cursor to column *n*. If the cursor is already past column *n*, spaces are output to move it to column *n*+*mk*, for the smallest integer value *k* possible. *n* and *m* default to 1. Without the colon flag, *n* and *m* are in units of characters; with it, they are in units of pixels.

Note: This operation works properly only on streams that support the **:read-cursorpos** and **:set-cursorpos** stream operations. On other streams, any **~T** operation simply outputs two spaces.

When **format** is creating a string, **~T** works, assuming that the first character in the string is at the left margin.

~@T ~@T outputs a space. ~*n*@T outputs *n* spaces.

~R **~R** prints *arg* as a cardinal English number, for example, four. **~:R** prints *arg* as an ordinal number, for example, fourth. **~@R** prints *arg* as a Roman numeral, for example, IV. **~:@R** prints *arg* as an old Roman numeral, for example, IIII.

~*n***R** prints *arg* in radix *n*.

The full form is ~*radix,mincol,padchar,commachar***R**.

~*radix,n***R** uses a column width of *n*; spaces are inserted on the left if the number requires fewer than *n* columns for its digits and sign. If the number does not fit in *n* columns, additional columns are used as needed.

*~radix,n,m*R uses *m* instead of the space as the pad character.

The @ modifier causes the number's sign to be printed always; the default is to print it only if the number is negative.

The : modifier causes commas to be printed between groups of three digits; the *commachar* parameter can be used to change the character used as the comma.

~⊂str~⊃

~character-style⊂text~⊃ formats *text* in *character-style*. See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts*. The ~⊂ directive must be matched by a corresponding ~⊃ directive.

With the colon flag, `~:⊂...~⊃` binds the line-height of the output stream. See the macro **with-character-style** in *Programming the User Interface, Volume A*.

You can supply *character-style* parameter in the format control string as a single character, as in `~'i⊂...~⊃`. In that case, the character should be one of the following:

`'i`	**:italic**
`'b`	**:bold**
`'p`	**:bold-italic**
`'r`	**:roman**

You can also have the *character-style* parameter taken as an argument, using `~v⊂...~⊃`. In that case, it may be a character style face code, like **:italic**; or else something acceptable to **si:parse-character-style**, such as a list like **(:fix :italic nil)** or an actual character style object. See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts*.

For example:

```
(format t "Moose bites can be ~'i⊂very~⊃ nasty, mind you.")
Moose bites can be very nasty, mind you.
NIL

(format T "Half the square root of ~'i⊂two~⊃ is ~v⊂~s~⊃."
        '(:fix :bold :normal) (sind 45))
Half the square root of two is 0.7071068.
NIL
```

~[*str0***~;***str1***~;***...***~;***strn***~]**

This is a set of alternative control strings. The alternatives (called *clauses*) are separated by **~;** and the construct is terminated by **~]**. **~]** is undefined elsewhere. **~;** is also used as a separator in justification (~<) constructions but is undefined elsewhere.

For example:

```
"~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
   ~;Tiger ~;Yu-Hsiang ~]kitty"
```

Where *arg* is the next element from the *args* of the **format** function, the *arg*th alternative is selected; **0** selects the first. If a prefix parameter is given (that is, **~***n***[**), then the parameter is used instead of an argument (this is useful only if the parameter is "#"). If *arg* is out of range, no alternative is selected. After the selected alternative has been processed, the control string continues after the **~]**.

~[str0~;str1~;...~;strn~:;default~] has a default case. If the *last* **~;** used to separate clauses is instead **~:;**, then the last clause is an "else" clause, which is performed if no other clause is selected. For example:

```
"~[Siamese ~;Manx ~;Persian ~;Tiger ~
   ~;Yu-Hsiang ~:;Bad ~] kitty"
```

~[~tag00,tag01,...;str0~tag10,tag11,...;str1...~] allows the clauses to have explicit tags. The parameters to each **~;** are numeric tags for the clause that follows it. That clause is processed that has a tag matching the argument. If **~***a1,a2,b1,b2,...***:;** (note the colon) is used, the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, and so on. **~:;** with no parameters can be used at the end to denote a default clause. For example:

```
"~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z:;letter ~
   ~'0,'9:;digit ~:;other ~]"
```

~:[*false***~;***true***~]** selects the *false* control string if *arg* is **nil**, and selects the *true* control string otherwise.

~@[*true***~]** tests the argument. If it is not **nil**, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is **nil**, then the argument is used up, and the clause is not processed. For example:

```
(setq prinlevel nil prinlength 5)
(format nil "~@[ PRINLEVEL=~D~]~@[ PRINLENGTH=~D~]"
            prinlevel prinlength)
   =>  " PRINLENGTH=5"
```

The combination of **~[** and # is useful, for example, for dealing with English conventions for printing lists:

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~
           ~S~:;~@{~#[~1; and~] ~S~^,~}~].")
(format nil foo)
       =>  "Items: none."
(format nil foo 'foo)
       =>  "Items: FOO."
(format nil foo 'foo 'bar)
       =>  "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz)
       =>  "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux)
       =>  "Items: FOO, BAR, BAZ, and QUUX."
```

~{*str*~} This is an iteration construct. The corresponding argument of the **format** function should be a list, which is used as a set of arguments as if for a recursive call to **format**. (The terminator ~} is undefined elsewhere.)

The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes; if *str* uses up two arguments by itself, two elements of the list are used up each time around the loop.

If, before any iteration step, the list is empty, the iteration is terminated. Also, if a prefix parameter *n* is given, there will be at most *n* repetitions of processing of *str*. Here are some simple examples:

```
(format nil "Here it is:~{ ~S~}." '(a b c))
     => "Here it is: A B C."
(format nil "Pairs of things:~{ <~S,~S>~}." '(a 1 b 2 c 3))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step, one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format nil "Pairs of things:~:{ <~S,~S>~}."
            '((a 1) (b 2) (c 3)))
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument that is a list, all the remaining **format** arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs of things:~@{ <~S,~S>~}."
            'a 1 'b 2 'c 3)
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each must be a list. On each iteration, the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs of things:~:@{ <~S,~S>~}."
            '(a 1) '(b 2) '(c 3))
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it will not override an explicit prefix parameter of zero).

If *str* is empty, an argument is used as *str*. It must be a string and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(lexpr-funcall #'format stream string args)
(format stream "~1{~:}" string args)
```

This will use **string** as a formatting string. The ~**1**{ says it will be processed at most once, and the ~:} says it will be processed at least once. Therefore it is processed exactly once, using **args** as the arguments.

As another example, the **format** function itself uses **format-error** (a routine internal to the **format** package) to signal error messages, which in turn uses **zl:ferror**, which uses **format** recursively. **format-error** takes a string and arguments, like **format**, but also prints some additional information: if the control string in **ctl-string** actually is a string (it might be a list), it prints the string and a small arrow showing where in the processing of the control string the error occurred. The variable **ctl-index** points one character after the place of the error.

```
(defun format-error (string &rest args)
  (if (stringp ctl-string)
      (ferror nil "~1{~:}~%~VT↓~%~3X/"~A/"~%"
              string args (+ ctl-index 3) ctl-string)
      (ferror nil "~1{~:}" string args)))
```

This first processes the given string and arguments using **~1{~:}**, then tabs a variable amount for printing the down-arrow, then prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
         must be a number
                    ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
...
```

~< *~mincol,colinc,minpad,padchar<text~>* justifies *text* within a field at least *mincol* wide. *text* can be divided into segments with ~;; the spacing is evenly divided between the text segments. The terminator ~> is undefined elsewhere.

With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified.

The **:** modifier causes spacing to be introduced before the first text segment. The @ modifier causes spacing to be added after the last. *Minpad*, default **0**, is the minimum number of *padchar* padding characters (default is the space character) to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to **1**. *mincol* defaults to **0**. For example:

```
(format nil "~10<foo~;bar~>")          => "foo    bar"
(format nil "~10:<foo~;bar~>")         => "  foo  bar"
(format nil "~10:@<foo~;bar~>")        => "  foo bar "
(format nil "~10<foobar~>")            => "    foobar"
(format nil "~10:<foobar~>")           => "    foobar"
(format nil "~10@<foobar~>")           => "foobar    "
(format nil "~10:@<foobar~>")          => "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.59023) => "$******2.59"
```

Note that *text* can include format directives. The last example illustrates how the ~< directive can be combined with the ~f directive to provide more advanced control over the formatting of numbers.

Here are some examples of the use of ~^ within a ~< construct. ~^ eliminates the segment in which it appears and all following segments if there are no more arguments.

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
       => "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
       => "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
       => "FOO   BAR   BAZ"
```

If a segment contains a ~^, and **format** runs out of arguments, it stops there instead of getting an error, and it as well as the rest of the segments are ignored.

If the first clause of a ~< is terminated with ~:; instead of ~;, it is used in a special way. All the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, the text segment for the first clause is output before the padded text. The first clause should contain a carriage return (~%). The first clause is always processed, and so any arguments to which it refers are used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the ~:; has a prefix parameter *n*, the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the following control string can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ";; ".

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

The prefix parameter **1** in ~**1:;** accounts for the width of the comma that will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second prefix parameter, it is used as the width of the line, overriding the natural line width of the output stream. To make the preceding example use a line width of 50, you would write:

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then **format** sees whether the stream handles the **:size-in-characters** message. If it does, then **format** sends that message and uses the first returned value as the line length in characters. If it does not, **format** uses **95.** as the line length.

Rather than using this complicated syntax, you can often call the function **format:print-list**.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the ~< case, the formatting is performed, but no more segments are processed before doing the justification. The ~^ should appear only at the beginning of a ~< clause, because it aborts the entire clause. ~^ can appear anywhere in a ~{ construct.

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a **V** parameter.

If ~^ is used within a ~:{ construct, it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use **~:^**.

~→ ~→*text*~← indents *text* at the cursor position that is current at the time of the ~→. A ~→ must be terminated with a ~←, which is undefined elsewhere. ~→ and ~← can be nested like **~[~]** and ~<~>; if they are nested, the indention of an inner pair is relative to the margin set by the pair containing it. A numeric argument, if supplied, specifies how far to indent. This directive is especially useful in making error messages indent properly. For example:

```
(format t "~&Error:  ~→~A~←" "File not found
for FOO.LISP.1")
```

prints

```
Error:  File not found
        for FOO.LISP.1
```

~Q An escape to arbitrary user-supplied code. *arg* is called as a function; its arguments are the prefix parameters to **~Q**, if any. *args* can be passed to the function by using the **V** prefix parameter. The function can output to ***standard-output*** and can look at the variables **format:colon-flag** and **format:atsign-flag**, which are **t** or **nil** to reflect the **:** and **@** modifiers on the **~Q**. For example,

```
(format t "~VQ" foo bar)
```

is a fancy way to say

```
(funcall bar foo)
```

and discard the value. Note the reversal of order; the **V** is processed before the **Q**.

~\date\ Prints its argument as a date and time, assuming the argument is a universal time. It uses the function **time:print-universal-date**.

```
(format nil "Today is ~\date\" (time:get-universal-time))
=> "Today is Tuesday the fourteenth of May, 1985; 3:07:05 pm"
```

~\time\ Prints its argument as a time, assuming the argument is a universal time. It uses the function **time:print-universal-time**.

```
(format nil "Today is ~\time\" (time:get-universal-time))
"Today is 5//14//85 15:08:41"
```

~\datime\

Prints the current time of day. It does not take an argument. It uses the function **time:print-current-time**.

```
(format nil "Today is ~\datime\")
"Today is 5//14//85 15:19:06"
```

~\time-interval\

Prints the length of a time interval. It uses the function **time:print-interval-or-never.**

```
(setq a (time:get-universal-time))
...
(format nil "It is ~\time-interval\ since I set this variable"
        (- (time:get-universal-time) a))
"It is 1 hour 5 minutes 9 seconds since I set this variable"
```

You can use the special form **format:defformat** to define your own directives.

format:defformat *directive (arg-type) arglist body ...* *Special Form*

Defines a new **format** directive.

directive is a symbol that names the directive. If *directive* is longer than one character, it must be enclosed in backslashes in calls to **format**:

```
(format t "~\foo\" ...)
```

directive is usually in the **format** package; if it is in another package, the user must specify the package in calls to **format**. For example, we've defined a format directive called **si:keystroke** that prints out the short names for all characters.

```
(defun gtest ()
  (loop for (name char) in '(("Space" #\space)
                             ("c-Space" #\c-space)
                             ("Tab" #\tab)
                             ("Page" #\page)
                             ("Left" #\mouse-L)
                             ("c-Left" #\c-mouse-L)
                             ("A" #\A)
                             ("c-A" #\c-A))
        do
        (format t "~%~A: ~C, ~\\keystroke\\" name char char))) =>
Space:  , Space
c-Space: c- , c-Space
Tab:    , Tab
Page: , Page
Left: Mouse-L, Mouse-L
c-Left: c-Mouse-L, c-Mouse-L
A: A, A
c-A: c-A, c-A
NIL
```

format:defformat defines a function to be called when **format** is called using *directive*. *body* is the body of the function definition. *arg-type* is a keyword that determines the arguments to be passed to the function as *arglist*:

:no-arg	The directive uses no arguments. The function is passed one argument, a list of parameters to the directive. The value returned by the function is ignored.
:one-arg	The directive uses one argument. The function is passed two arguments: the argument associated

with the directive and a list of parameters to the directive. The value returned by the function is ignored.

:multi-arg The directive uses a variable number of arguments. The function is passed two arguments. The first is a list of the first argument associated with the directive and all the remaining arguments to **format**. The second is a list of parameters to the directive. The function should **cdr** down the list of arguments, using as many as it wants, and return the tail of the list so that the remaining arguments can be given to other directives.

The function can examine the values of **format:colon-flag** and **format:atsign-flag**. If **format:colon-flag** is not **nil**, the directive was given a : modifier. If **format:atsign-flag** is not **nil**, the directive was given a @ modifier.

The function should send its output to the stream that is the value of **format:*format-output***.

Here is an example of a **format** directive that takes one argument and prints a number in base 7:

```
(format:defformat format:base-7 (:one-arg) (argument parameters)
  parameters                                     ;ignored
  (let ((base 7))
    (princ argument format:*format-output*)))
```

Now:

```
(format nil "> ~\base-7\ <" 8)  =>  "> 11 <"
```

format:print-list *destination element-format-string list* *Function*
&optional (*separator-format-string* ", ")
(*start-line-format-string* "   ")
(*tilde-brace-options* "")

Provides a simpler interface for the specific purpose of printing comma-separated lists with no list element split across two lines.

The *destination* argument tells where to send the output; it can be **t**, **nil**, a string suitable for **zl:string-nconc**, or, as with **format**, a stream.

element-format-string is a **format** control string specifying how to print each element of *list*. It is used as the body of an iteration construction (as in ~{*element-format-string*~}). See the function "**format**", page 309.

separator-format-string, which defaults to ", " (comma, space), is a string that is placed after each element except the last. **format** control directives are allowed in this string but should not take arguments from the *list.*

start-line, which defaults to three spaces, is a **format** control string that is used as a prefix at the beginning of each line of output except the first.

tilde-brace-options is a string inserted before the opening brace ({) of the iteration construct. It defaults to the null string but allows you to insert a colon or at-sign. The line width of the stream is computed in the same way as with the ~{str} **format** directive. It is not possible to override the natural line width of the stream.

sys:with-indentation (*stream-var relative-indentation*) &body *body* *Macro*

Within the body of **sys:with-indentation**, any output to *stream-var* is preceded by a number of spaces. At every recursion, the additional indentation is specified by *relative-indentation.* The macro does not work this way with the **:item** message used to display mouse-sensitive items; the items appear, but without indentation. (See the section "Interactive Streams and Mouse-Sensitive Items" in *Programming the User Interface, Volume B.*)

```
(defun traced-factorial (n)
  (format t "~%Argument:  ~D" n)
  (sys:with-indentation (standard-output 2)
    (let ((value (if (≤ n 1)
                     1
                     (* n (traced-factorial (1- n))))))
      (format t "~%Value:  ~D" value)
      value)))

(traced-factorial 5)
```

```
Argument:  5
  Argument:  4
    Argument:  3
      Argument:  2
        Argument:  1
        Value:  1
      Value:  2
    Value:  6
  Value:  24
Value:  120
120
```

17. Formatting Lisp Code

grindef &rest *fcns* *Special Form*

Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed:

- The **quote** special form is represented with the ' character.
- Displacing macros are printed as the original code rather than the result of macro expansion.
- The code resulting from the backquote (') reader macro is represented in terms of '.

The subforms to **grindef** are the function specs whose definitions are to be printed; ordinarily, **grindef** is used with a form such as **(grindef foo)** to print the definition of **foo**. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as **defun** special forms, and values are printed as **setq** special forms.

If a function is compiled, **grindef** says so and tries to find its previous interpreted definition by looking on an associated property list. See the function **uncompile** in *Program Development Utilities*. This works only if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a BIN file, **grindef** does not find the interpreted definition and cannot do anything useful.

With no subforms, **grindef** assumes the same arguments as when it was last called.

zl:grind-top-level *exp* &optional (*grind-width* **nil**) (*grind-real-io* **zl:standard-output**) (*grind-untyo-p* **nil**) (*grind-displaced* **'si:displaced**) (*terpri-p* **t**) (*grind-notify-fun* **nil**) (*loc* **(ncons exp)**) *Function*

Pretty-prints *obj* on *stream*, inserting up to *width* characters per line. This is the primitive interface to the pretty-printer. Note that it does not support variable-width fonts. If the *width* argument is supplied, it is how many characters wide the output is to be. If *width* is unsupplied or **nil**, **zl:grind-top-level** tries to determine the "natural width" of the stream by sending a **:size-in-characters** message to the stream and using the first returned value. If the stream does not handle that message, a width of **95.** characters is used instead.

The remaining optional arguments activate various features and usually should not be supplied. These options are for internal use by the system, and are documented here only for completeness. If *untyo-p* is **t**, the **:untyo** and **:untyo-mark** operations are used on *stream*, speeding up the algorithm somewhat. *displaced* controls the checking for displacing macros; it is the symbol that flags a place that has been displaced, or **nil** to disable the feature. If *terpri-p* is **nil**, **zl:grind-top-level** does not advance to a fresh line before printing.

If *notify-fun* is non-**nil**, it is a function of three arguments and is called for each "token" in the pretty-printed output. Tokens can be atoms, open and close parentheses, and reader macro characters such as **'**. The arguments to *notify-fun* are the token, its "location" (see next paragraph), and **t** if it is an atom or **nil** if it is a character.

loc is the "location" (typically a cons) whose **car** is *obj*. As the grinder recursively descends through the structure being printed, it keeps track of the location where each thing came from, for the benefit of the *notify-fun*, if any. This makes it possible for a program to correlate the printed output with the list structure. The "location" of a close parenthesis is **t**, because close parentheses have no associated location.

18. The Serial I/O Facility

18.1 Introduction to Serial I/O

3600-family computers have a serial input/output facility, which uses the EIA RS-232 protocol to receive and transmit serial data. Many computer peripherals can communicate using the RS-232 protocol, and so can be connected to the 3600-family computer through this facility. This chapter explains the capabilities of the facility, gives a brief description of the hardware performing the serial I/O and how to interface to it, and describes the 3600-family software driving that hardware.

Before reading this chapter, you should be familiar with the basic concepts of serial data communication, including the RS-232 standard. You should also be familiar with Symbolics Common Lisp, which is the systems programming language for the 3600-family computer. In particular, you should understand what *streams* are. See the section "Streams", page 3.

18.2 Hardware Description for Serial I/O

This section gives a brief description of the hardware that performs serial I/O on 3600-family computers. You do not have to understand everything in this section to use the serial I/O facility.

18.2.1 Overview of Serial I/O Hardware

Symbolics 3600-family computers support four serial I/O ports. Three ports are located on the bulkhead at the back of the processor. The fourth port is located in the rear of the console. The console serial port requires the new FEP system, Release 6.1 software, and the new console EPROM to be installed.

The external data communication signals appear on RS-232 25-pin D-type connectors. All serial I/O communication is controlled by the 3600-family computer's FEP.

The RS-232 protocol provides for communication between Data Circuit Terminating Equipment (DCEs, also known as "data sets"; for example, modems), and Data Terminal Equipment (DTEs, also known as "data terminals"; for example, computer terminals, computers, or most devices that use serial lines).

The single console port is configured differently than the three ports on the bulkhead:

Each of the three ports on the bulkheads of the 3600-family computer is a DTE. You can connect a bulkhead serial port directly to a DCE, but if you want to connect the serial port to a DTE, you must supply a *null modem.*

In contrast, the console port is a DCE. You can connect the console serial port directly to a DTE, but if you want to connect the serial line to a DCE, you must supply a *null terminal.*

For example, a Kanji tablet is configured as a DCE. You can connect a Kanji tablet directly to a bulkhead port (a DTE), but you must supply a null terminal to connect a Kanji tablet to the console port (a DCE).

18.2.2 Console Serial I/O Port

The external data communication signals appear on one female RS-232 25-pin D-type connector in the rear of the console of Symbolics 3600-family computers. The console serial I/O port is labelled "RS-232".

The correspondence between connector pins and RS-232 signals is given in Table 1.

Console connector pin	*RS-232 signal*
2	Transmitted Data [Input]
3	Received Data [Output]
4	RTS (Request To Send) [Input]
5	CTS (Clear To Send) [Output]
6	DSR (Data Set Ready) [Output]
8	DCD (Data Carrier Detect) [Output]
20	DTR (Data Terminal Ready) [Input]
1	Chassis Ground
7	Signal Ground

Table 1. Assignment of RS-232 Signals to Pins

To build a cable that includes a null terminal for asynchronous communications, follow the wiring instructions in Table 2. Both ends of the cable should be male 25-pin RS-232 connectors.

Pin at back of the console (DCE)		*Pin at remote end (DTE)*	
2	Receive data (from DCE)	3	Transmit data (from DTE)
3	Transmit data (to DCE)	2	Receive data (to DTE)
4	Request to send (to DCE)	5	Clear to send (to DTE)
5	Clear to send (from DCE)	4	Request to send (from DTE)
7	Signal ground	7	Signal ground
8	Carrier detect (from DCE)	20	Data Terminal ready (from DTE)
18	Special DTE-DSR input	6	Data set ready (to DTE)
19	Special DTE-RI input	22	Ring Indicator (to DTE)
20	Data terminal ready (to DCE)	8	Carrier detect (to DTE)

Table 2. Assignment of RS-232 Signals to Pins in Asynchronous Null Terminals

18.2.3 Bulkhead Serial I/O Ports

The external data communication signals appear on three RS-232 25-pin D-type connectors on the rear bulkhead (in the back of the processor).

The gender and labeling of these connectors varies with the processor model:

- The 3600 I/O bulkhead presents 3 female connectors labelled "EIA 1", "EIA 2", and "EIA 3". (The male connector labelled "EIA 4" is not a serial port at all, but the connection to an inboard Vadic VA3450 modem, if present. See the section "Physical Connection to the Dial Network".)

- The 3670 I/O bulkhead presents 3 male connectors labelled "EIA 1", "EIA 2", and "EIA 3".

- The 3640 I/O bulkhead presents 3 male connectors labelled "SERIAL 1", "SERIAL 2", and "SERIAL 3".

The correspondence between connector pins on the rear bulkhead and RS-232 signals is given in Table 1.

Rear bulkhead connector pin	*RS-232 signal*
2	Transmitted Data [Output]
3	Received Data [Input]
4	RTS (Request To Send) [Output]
5	CTS (Clear To Send) [Input]
6	DSR (Data Set Ready) [Input]
8	DCD (Data Carrier Detect) [Input]
20	DTR (Data Terminal Ready) [Output]
1	Chassis Ground
7	Signal Ground

Table 3. Assignment of RS-232 Signals to Pins

To build a cable that includes a null modem for asynchronous communications, follow the wiring instructions in Table 2.

One side	*Other side*	*RS-232 signal*
3	2	Data Out (from data set to terminal)
2	3	Data In (from terminal to data set)
5	4	RTS (Request To Send)
4	5	CTS (Clear To Send)
20	6	DSR (Data Set Ready)
20	8	DCD (Data Carrier Detect)
6	20	DTR (Data Terminal Ready)
8	20	DTR (Data Terminal Ready)
1	1	Chassis Ground
7	7	Signal Ground

Table 4. Assignment of RS-232 Signals to Pins in Asynchronous Null Modems

Note that this null modem is suitable only for asynchronous communications; a synchronous null modem is considerably more complex.

When using the 3600-family computer with a device that does not supply RS-232 modem control signals, it is necessary to supply Clear To Send and Data Carrier Detect inputs to the 3600-family computer, for example by jumpering pin 4 to pin 5, and pins 6, 8, and 20 together. This should be done in the cable or in the device connector, *not* in the 3600-family computer's connector or inside the 3600-family computer.

18.3 The Serial I/O Stream

The function of the serial I/O facility is to receive and transmit data over a serial communications channel. The unit of communication is the *character*; each character is represented as a binary number. The facility has two independent parts: a *receiver*, which receives a sequence of characters from the external device, and a *transmitter*, which transmits a sequence of characters to the external device.

A Symbolics Common Lisp program uses the facility through an I/O stream. The output operations, such as **:tyo**, send characters to the transmitter and from there to the external device; the input operations, such as **:tyi**, read characters from the receiver, which gets them from the external device. In addition to regular I/O operations, the serial I/O stream also supports special operations that examine and alter parameters of the serial I/O facility. To perform serial I/O, a program should first get the serial I/O stream by calling the function **si:make-serial-stream**, setting up the parameters of the serial I/O facility as it needs them; then it can use normal stream operations to read and write characters. When the program is done with the serial I/O stream, it should close it; programs that use the serial I/O stream should include an **unwind-protect** form whose cleanup handler closes the stream. The **with-open-stream** special form is a good way to do this when the entire lifetime of the stream is to be enclosed in the body of one Symbolics Common Lisp form. Closing the stream frees up a buffer in main memory and disables interrupts.

The serial I/O stream is different from most streams in that the characters you send to it and get from it are probably *not* interpreted as being in the Symbolics character set. Of course, the interpretation of the characters depends completely on the external device, but most devices that are likely to use serial communications use the standard ASCII character set. You can tell the stream whether or not to convert between ASCII characters and Symbolics characters.

The serial I/O stream is also different from some streams in being buffered on the output side. If you send characters to the serial stream using, for example, **:tyo** or **:string-out**, the characters are placed into a buffer for eventual transmission over the serial line. They are not actually transmitted until the buffer fills up, the serial stream is closed, or a **:force-output** operation is done on the stream.

The **:force-output** option to **si:make-serial-stream** causes characters to be transmitted immediately; this makes the serial stream easier to use but degrades its performance.

The serial I/O stream has several parameters. Each parameter is denoted by a keyword symbol. These keywords are passed to the **si:make-serial-stream** function and to the **:get** and **:put** operations to specify which parameter the caller is interested in. (Some parameters make sense only when creating a stream, or affect the flavor of the stream; these parameters are not valid for **:get** and **:put**.) For descriptions of the parameters: See the section "Parameters for Serial I/O", page 337.

si:make-serial-stream &rest *options* *Function*

Initializes the serial I/O facility and returns the serial I/O stream.

options are alternating keyword symbols, naming parameters, and initial values for those parameters. They let you initialize parameters when you start using the serial I/O stream. You can change most of them later with the **:put** operation.

si:make-serial-stream, which accesses a serial line, causes the accessing process to wait if all ports are in use. The command `c-m-SUSPEND` allows you to invoke a restart handler to close a line that you believe has been left open by mistake.

For documentation of parameters for serial I/O: See the section "Parameters for Serial I/O", page 337.

The serial I/O stream supports all standard stream operations. Of the optional input operations, it supports **:listen** and **:clear-input**; the latter is relevant because input from the serial port is buffered. There is also a **:reset** operation, which resets the state of the hardware and the FEP. The **:tyi-no-hang** special-purpose operation is supported as well. The **:force-output** and **:finish** optional output operations are supported, since output is buffered.

The serial I/O stream also supports two nonstandard operations: **:get** and **:put**. These two operations respectively allow you to examine and alter various properties of the serial I/O facility. The names of these operations are intended to suggest the **zl:get** and **zl:putprop** functions in Symbolics Common Lisp.

:get *parameter* of **si:serial-stream** *Method*

parameter should be one of the symbols that name parameters of the serial I/O facility. This message returns the value of that parameter. See the section "Parameters for Serial I/O", page 337.

:put *parameter value* of **si:serial-stream** *Method*

parameter should be one of the symbols that name parameters of the serial

I/O facility. The value of that parameter is set to *value*. See the section "Parameters for Serial I/O", page 337.

If you are using serial I/O streams, you might also be interested in the remote login facilities:

See the section "Using the Remote Login Facilities" in *Networks*.
See the function **neti:enable-serial-terminal** in *Networks*.

18.4 Parameters for Serial I/O

This section lists all parameters of the serial I/O facility. For each parameter, it lists the keyword symbol, the meaning of the parameter, and the default value. A few parameters can be examined but not altered; they are so marked in their descriptions. Parameters whose functions are similar are grouped together.

Parameters from the following group are used *only* when the stream is being created, as arguments to **si:make-serial-stream**. You cannot use the **:put** operation with them, and you can use the **:get** operation only with **:unit**.

:unit	This parameter says which of the serial ports to create a stream to. Its value can be **1**, **2**, or **3** to indicate one of the three bulkhead ports (each of which is a DTE); or **0** to indicate the serial I/O port located at the back of the console (a DCE). The default is **2**. For more information on the serial I/O ports: See the section "Overview of Serial I/O Hardware", page 331.
:ascii-characters	If the value of this parameter is **t**, the serial stream is a Zetalisp character stream. The characters are translated from ASCII to the Symbolics internal character set on input, and to ASCII on output. If the value of this parameter is **nil**, the serial stream is a binary stream. Binary streams do not support **:line-out**, **:fresh-line**, and similar messages. The default is **nil**.
:flavor	The value of this parameter is the flavor of stream to create. Normally, the value is computed automatically, based on the values of the **:ascii-characters** and **:force-output** parameters; this parameter is needed only if you want to use some special flavor that includes the serial stream flavors and other mixins.
:force-output	If the value of this is **t**, a **:force-output** stream operation is done after every **:tyo** and every **:string-out**. If it is **nil** (the default), output is not transmitted until the output buffer fills up, a **:force-output** is done explicitly, or the stream is closed (and the close mode is not **:abort**). The nonforcing mode is

usually more efficient, although efficiency depends on the application.

The following group of parameters controls the format of the transmitted characters. It is important to set the parameters to be compatible with the external device, or else proper communication is impossible. These parameters apply to both the transmitter and the receiver.

:mode	The kind of communications protocol used over the port. The two possible values are **:asynchronous**, for asynchronous operation, and **:hdlc**, for the HDLC-like bit-stuffing protocol. See the section "HDLC Serial I/O", page 344. The default is **:asynchronous**.
:baud	The data transmission rate, in bits per second. This should be one of the following integers (in decimal): **300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200**. The default is **1200**.
:number-of-data-bits	The number of bits in each character. This should be one of the following fixnums: **5**, **6**, **7**, or **8**. The default is **7**.
:parity	The kind of parity bit that should be sent. If the value of this parameter is **nil**, no parity bit is sent. If it is **:even**, even parity is transmitted. If it is **:odd**, odd parity is transmitted. The default is **:even**. This parameter also controls what kind of parity checking is done on received characters.
:number-of-stop-bits	The number of "stop" bits transmitted after each character. It should be one of the following numbers: **1**, **1.5**, or **2**. The default is **1**.

The following parameters control error checking in the receiver. After a character is read by an input stream operation, the stream checks for error conditions detected by the receiver when the character arrived. If any of the enabled error conditions occurred, the stream signals an error.

:check-parity-errors	If the value of this parameter is **nil**, parity errors are ignored; if it is **t**, a parity error causes an error to be signaled when the character is read. The default is **nil**. A parity error occurs when the parity of the data bits disagrees with the value of the received parity bit. This never happens if parity checking is not being used, that is, if the **:parity** option is **nil**.

:input-error-character
The value is a character to be substituted for any input character in which a parity error is detected. This is independent of the **:check-parity-errors** flag. If the value is **nil** (the default), the character is left alone.

:check-over-run-errors
If the value of this parameter is **nil**, over-run errors are ignored; if it is **t**, then an over-run error causes an error to be signaled when the character is read. The default is **nil**. An over-run error occurs if input arrives faster than it can be read.

:check-framing-errors
If the value of this parameter is **nil**, framing errors are ignored; if it is **t**, then a framing error causes an error to be signaled when the character is read. The default is **nil**. A framing error occurs when the "stop" bit (the bit after all the data bits, and after the parity bit if parity is being checked) is not **1**. This indicates a line error, a baud rate mismatch between the external device and the receiver, or the sending of a "break".

The following parameters to **si:make-serial-stream** deal with the "modem control" signals (signals other than Data In and Data Out) defined by the RS-232 protocol. Note that the interpretation of the parameters differs for the three bulkhead serial ports and the single console serial port. This difference reflects the different hardware configurations: The bulkhead serial ports (**:unit 1**, **2**, and **3**) are DTEs, whereas the console serial port (**:unit 0**) is a DCE. For more information on the serial I/O ports: See the section "Overview of Serial I/O Hardware", page 331.

:carrier-detect
For the bulkhead serial ports: If the value of this parameter is **t**, the external device is asserting the DCD ("data carrier detect") signal; otherwise it is not. This parameter can be examined but not altered.

For the console serial port: If the value of this parameter is **t**, the external device is asserting the DTR ("data terminal ready") signal; otherwise it is not. This parameter can be examined but not altered.

:clear-to-send
For the bulkhead serial ports: If the value of this parameter is **t**, the external device is asserting the CTS ("clear to send") signal; otherwise it is not. This parameter can be examined but not altered.

For the console serial port: If the value of this parameter is **t**, the external device is asserting the RTS ("request to send") signal; otherwise it is not. This parameter can be examined but not altered.

:request-to-send For the bulkhead serial ports: If the value of this parameter is **t**, assert the RTS ("request to send") signal; otherwise do not. The default is **nil**.

For the console serial port: If the value of this parameter is **t**, assert the CTS ("clear to send") signal; otherwise do not. The default is **nil**.

:data-terminal-ready
For the bulkhead serial port: If the value of this parameter is **t**, assert the DTR ("data terminal ready") signal; otherwise do not. The default is **nil**.

For the console serial port: If the value of this parameter is **t**, assert the DCD ("data carrier detect") signal; otherwise do not. The default is **nil**.

The following parameters control the use of the XON/XOFF protocol.

:xon-xoff-protocol If this is **t**, output to the serial stream is flow-controlled using the ASCII XON/XOFF (Control-S/Control-Q) protocol. While the stream is transmitting characters, it checks the receiver to see if any characters have arrived. If an ASCII XOFF or Control-S character (octal 23, decimal 19) has arrived, transmission is stopped. Then the stream reads characters from the receiver until an ASCII XON or Control-Q character (octal 21, decimal 17) arrives, and then proceeds with the transmission.

This feature allows the external device to limit the rate at which characters are transmitted to it by the serial I/O facility. The default is **nil** (XON/XOFF feature not enabled).

Interpretation of incoming XON/XOFF signals is done at interrupt level in the FEP, and is therefore quite fast. After an XOFF is received, the 3600-family computer ceases transmission after two or three characters (buffered in the multiprotocol chip).

:output-xoff-character
The value is a character that is used to control flow of data from the Symbolics computer to the external device. It is used to suspend the flow of data when the **:xon-xoff-protocol** parameter is set. The default is **#o023**.

:output-xon-character
The value is a character that is used to control flow of data from the Symbolics computer to the external device. It is used

to resume the flow of data when the **:xon-xoff-protocol** parameter is set. The default is **#o021**.

:generate-xon-xoff

If the value of this parameter is **t**, then the serial port generates XON and XOFF controls itself. This can be used to accept input at high speed from devices that understand the XON/XOFF protocol. The default is **nil**.

The XON and XOFF characters are transmitted directly by the FEP, so the response time is excellent. After the FEP transmits an XOFF, the device is required to cease transmission after no more than about 100 characters, so the device is not required to act very quickly.

:input-xoff-character

The value is a character that is used to control flow of data from the external device to the Symbolics computer. It is sent by the Symbolics computer to suspend the flow of data when the **:generate-xon-xoff** flag is set. The default is **#o023**.

:input-xon-character

The value is a character that is used to control flow of data from the external device to the Symbolics computer. It is sent by the Symbolics computer to resume the flow of data when the **:generate-xon-xoff** flag is set. The default is **#o021**.

18.5 Simple Examples: Serial I/O

The following two examples illustrate the use of the serial I/O facility. For further information on the function **si:make-serial-stream** and its parameters:

See the section "The Serial I/O Stream", page 335.
See the section "Parameters for Serial I/O", page 337.

Both examples below assume that the serial I/O port numbered 1 is hooked to an ASCII computer terminal operating on a normal RS-232 asynchronous connection at 300 baud, with one stop bit and odd parity. It types the characters "Hello there." on the terminal. A null modem is used between the serial port and the terminal, because both ends are acting as DTEs.

The first example illustrates creating a serial stream, saving the result in a variable, sending output to the stream, and closing the stream:

```
(setq ss (si:make-serial-stream
           :unit 1
           :baud 300
           :ascii-characters t
           :number-of-stop-bits 1
           :parity :odd
           :force-output t))
(send ss :string-out "Hello there.")
(close ss)
```

The second example uses **:with-open-stream**, thereby enclosing the entire lifetime of the serial stream in the body of one Symbolics Common Lisp form:

```
(defun type-greeting-message ()
  (with-open-stream (stream (si:make-serial-stream
                              :unit 1
                              :baud 300
                              :ascii-characters t
                              :number-of-stop-bits 1
                              :parity :odd))
    (send stream :string-out "Hello there.")))
```

You can also use the function **neti:enable-serial-terminal** to enable a terminal to communicate with a Symbolics computer: See the function **neti:enable-serial-terminal** in *Networks.*

18.6 Troubleshooting: Serial I/O

If you have trouble making your device communicate with the 3600-family computer through a serial port, there are several things to try:

- Make sure that the baud rate, the number of data bits, the parity checking, and the number of stop bits are set the same way on the device as they are in your serial stream parameters.

- Make sure that the device is connected to the proper serial port. The bulkhead serial ports are labelled "EIA1" (or on 3640s, "SERIAL 1"), "EIA2" ("SERIAL 2"), and "EIA3" ("SERIAL 3"). The console serial port is labelled "RS-232". You must use the port corresponding to the value of the **:unit** keyword to **si:make-serial-stream.** The default value is **2**, so if you do not specify anything, the "EIA2" ("SERIAL 2") connector is the appropriate one.

- If you are using any one of the three bulkhead serial ports (units **1**, **2**, or **3**, you can connect the port directly to a DCE device. However, if the device is a DTE, make sure that there is a null modem between your device and the serial connectors. Since most devices are DTEs, the null modem is probably necessary.

- If you are using the console serial port (unit **0**) you can connect the port directly to a DTE device. However, if the device is a DCE, make sure that there is a null terminal between your device and the serial connectors.
- Try using a different port. Remember both to plug your device into a different connector, and to change the program to specify a different value for the **:unit** keyword.

18.7 Notes on Serial I/O

The receiver is implemented using the 3600-family computer's general front end processor (FEP) "channel" facility. When a character arrives at the serial port, the FEP buffers it and transfers it to the 3600-family computer over a "channel". Therefore, it is not necessary for the program doing input from the stream to read in characters as quickly as they arrive from the external device. The **:clear-input** operation to the serial stream resets this buffer (including the buffers in Symbolics Common Lisp, and the buffers in the FEP). The buffering capacity is about 500 characters. If the buffer is full and another character arrives, an over-run error occurs; if the **:check-over-run-errors** parameter is used, this is reflected by the signalling of an error.

A useful debugging technique is to create a serial stream with the desired parameters and set a variable (say, **s**) to it, and do:

```
(stream-copy-until-eof s standard-output)
```

This prints received characters on the screen until you type `c-ABORT`. This technique works only with the **:number-of-data-bits** parameter set to **7**, so that the Symbolics computer does not see the ASCII parity bit. Unless character set translation is enabled (via the **:ascii-characters** parameter), ASCII control characters, including carriage return and line feed, are displayed as special symbols, such as circle-cross or delta, because of the differences between the Symbolics character set and ASCII. See the section "The Character Set", page 355.

18.8 HDLC Serial I/O

The 3600 family supports synchronous serial I/O using HDLC-like bit-stuffing protocols. The CCITT-16 CRC polynomial is used.

This facility requires that the computer be running with FEP version 14 or later. Also, some older 3600s might require that a special adapter cable be connected to serial port 1. Baud rates of 9600 or lower are recommended.

An HDLC stream is a stream of flavor **si:serial-hdlc-stream.** Use the function **si:make-serial-stream** to make one of these streams. HDLC can be used only on serial port 1, so you must supply a **:unit** argument to **si:make-serial-stream** with a value of 1 (it defaults to 2). HDLC streams accept **:read-frame** and **:write-frame** messages.

si:serial-hdlc-stream *Flavor*
An HDLC serial I/O stream. This flavor is built on **si:serial-binary-stream** and **si:serial-hdlc-mixin.**

:read-frame *string* &optional (*start* **0**) *end* of **si:serial-hdlc-mixin** *Method*
Reads an HDLC frame into *string.* Returns the length actually read.

:write-frame *string* &optional (*start* **0**) *end* of **si:serial-hdlc-mixin** *Method*
Writes *string* as an HDLC frame. This method never calls **process-wait** and can be used in a simple process. If insufficient buffers are available, it returns a form that evaluates to **t** when buffers become available.

18.9 Using the Terminal Program with Hosts Connected to the Serial Line

You can connect a 3600-family machine to another host via the serial line. Specifically, you can use the terminal program to communicate with another host when the 3600-family computer's serial line is connected to a terminal port on the other host.

The network system treats the set of hosts connected to the serial lines of a 3600-family computer as a special network, a *pseudonet.* Before you can use the terminal program to talk to another host over the serial line, you must use the **tv:edit-namespace-object** or the Edit Namespace Object command to create this network and assign an address on that network to the 3600-family computer. You might want to create or modify the remote host as well.

1. Create the network. Give it a **name** attribute associated with the 3600-family computer and a **type** attribute of **serial-pseudonet.**

In the following example, Merrimack is the name of the 3600-family computer:

```
NETWORK MERRIMACK-SERIAL
TYPE SERIAL-PSEUDONET
```

2. Add an entry to the **address** attribute of the 3600-family computer to specify that the 3600-family computer is connected to the new network. Each **address** entry is usually a pair of the form (*network address*). By convention, the 3600-family computer is assigned address 0 on a serial pseudonet. Following is an example of a new **address** entry for the 3600-family computer Merrimack:

```
ADDRESS MERRIMACK-SERIAL 0
```

3. If the line rate of the serial line is other than 9600 baud, supply a **peripheral** entry for the 3600-family computer giving the correct baud rate. The peripheral type is **serial-pseudonet**, and the **unit** attribute is the unit number of the serial line. Following is an example of a **peripheral** entry for the 3600-family computer:

```
PERIPHERAL SERIAL-PSEUDONET UNIT 2 BAUD 4800
```

4. If you want the terminal program to start out simulating one of the supported terminal types, add a **terminal-type** attribute to the peripheral. Currently supported terminal types are the VT100 and Ann Arbor Ambassador. For example, to make the terminal program simulate an Ambassador, add to the 3600-family computer a **peripheral** entry of this form (note that the entry must actually be on one line):

```
PERIPHERAL SERIAL-PSEUDONET UNIT 2 BAUD 9600
TERMINAL-TYPE Ambassador
```

You can now use the terminal program to connect to the remote host. At the "Connect to host:" prompt, you must supply an address of the form MERRIMACK-SERIAL|2. If you want to type a name or nickname of the remote host instead, add **address** and **service** entries for the remote host's namespace object. If the remote host does not exist in the network database, use the Edit Namespace Object command or the function **tv:edit-namespace-object** to create it.

For the **address** entry, specify the serial pseudonet and an address that corresponds to the unit number of the serial line to which the host is connected. The **service** entry is a triple of the form (*service medium protocol*). For the regular host login server, *service* is **login**, *medium* is **serial-pseudonet**, and *protocol* is **tty-login**. Following is an example of **address** and **service** entries for the remote host Blue connected to the 3600-family computer Merrimack:

```
HOST BLUE
SYSTEM-TYPE TENEX
ADDRESS MERRIMACK-SERIAL 2
SERVICE LOGIN SERIAL-PSEUDONET TTY-LOGIN
```

You can also use the serial line to connect to servers other than normal login on a remote host. You must add a **service** entry for the remote host to specify the kind of service, the **serial-pseudonet** medium, and the protocol that the remote host uses. You must also add an **address** entry on the serial pseudonet for the remote host. In the **address** entry, specify the address in the form *protocol=unit* instead of just *unit*. Following are examples of **address** and **service** entries for a file server using protocol **myftp** on remote host Blue:

```
HOST BLUE
SYSTEM-TYPE TENEX
ADDRESS MERRIMACK-SERIAL MYFTP=2
SERVICE FILE SERIAL-PSEUDONET MYFTP
```

For information on the Terminal program: See the section "Connecting to a Remote Host Over the Network" in *Networks*.

For information on network and host attributes: See the section "Namespace System Object Definitions" in *Networks*.

For information on services, media, and protocols: See the section "Symbolics Generic Network System" in *Networks*.

19. Writing Programs That Use Magnetic Tape

19.1 The tape:make-stream Function

tape:make-stream *Function*

tape:make-stream is used to create streams that read or write magnetic tape. It handles both cartridge and industry-compatible tape. With **tape:make-stream**, you can access tape on the local machine, or on any machine with a tape server.

tape:make-stream creates a stream. **with-open-stream** and other standard tools for managing streams should be used to ensure proper closing of a stream made with **tape:make-stream**.

Tape streams accept (for output) and return (as input) 8-bit characters. Normal stream messages can be used to tape streams. See the section "Streams", page 3. There are a few other messages: See the section "Messages to Tape Streams", page 350.

tape:make-stream takes a large number of optional keyword arguments:

:host	The host on which the tape drive to be used is located. This can be a string or a host object. The keyword **:local** is also accepted for the local host. If this argument is not provided, **tape:make-stream** prompts for the name of the host. The host must already be registered in the network database for supporting TAPE service.
:unit	The identifier of the tape drive on the selected host that is to be used. Hosts having only one tape drive generally do not require this information. The value of this argument is generally a character string. "" or **nil** specifies "don't care", which is the usual value.
:reel	The name of the tape reel to be mounted. This information is needed by tape servers that have operators, who need to know the name of a tape in order to mount it. It is also needed by servers who have tape access control systems. Currently (Release 5.0) no such servers are supported. "" or **nil**, the usual default, means "don't care".
:direction	Specifies whether reading, writing, or intermixed reading

and writing are to be performed. The valid values of this argument are thus **:input**, **:output**, and **:bidirectional**, respectively.

:input-stream-mode
This argument, which is only valid if the **:direction** argument is **:input** or **:bidirectional**, controls whether record boundaries, on input, are reflected to you. The default is **t**, meaning that they are *not*. It is not meaningful for cartridge tapes: record boundaries are never visible to the user of cartridge tape.

In *input stream mode* (a value of **t**), input bytes are transferred from the tape records to you until a file mark (tape mark, EOF) is encountered, at which time you see an end-of-file in your stream.

In *input record mode* (a value of **nil**), input bytes are transferred from the tape records to you until a record boundary, at which time you see an end-of-file in your stream. To progress beyond the record boundary, the message **:discard-current-record** must be sent to the stream.

:record-length Controls the maximum length, in bytes, of tape records. This is ignored for cartridge tape. For reading, it must provide for the largest record to be read. Not all input records need be this long, although in some cases the server decides whether to allow records of other than this size. See also the keywords **:minimum-record-length** and **:minimum-record-length-granularity**. The default is 4096.

:density Density of the tape in bits per inch. This is ignored for cartridge tape. The default is 1600 for servers that have the capability of multiple densities.

:pad-char A number that is the single character with which to pad records when short records are padded. (This is ignored for cartridge tape.) The default pad character is 0. For compatibility with previous releases, supplying this argument and *not* supplying a value for either **:minimum-record-length** or **:minimum-record-length-granularity** implies a value of **:full** for **:minimum-record-length**.

:minimum-record-length

A number that is the minimum record length, in bytes, to which all output records will be padded. (This is ignored for cartridge tape.) This ability is present because many tape controllers cannot read records shorter than some minimum. Arguments to this keyword can be:

not supplied	If this argument is not supplied, a value of 64 is assumed.
integer	Some number smaller than the value of the **:record-length** argument. Short records are padded with 0, or the value of the **:pad-char** argument, if that is supplied.
:full	All records are padded to their maximum length, namely, the value of the **:record-length** argument. Short records are padded with 0, or the value of the **:pad-char** argument, if that is supplied.
nil	The Lisp Machine does not enforce any minimum record length. The tape server and/or the tape hardware on that server might enforce some minimum of its own.

:minimum-record-length-granularity

An integer, or **nil**, establishing a *granularity*, or enforced integral divisor, for the length of all tape records written. If non-**nil**, all records written are padded (with 0, or the value of the **:pad-char** argument, if that is supplied) to be multiples of this number in length. This value is ignored for cartridge tape. It is also ignored if short records are not to be written, that is, **:minimum-record-length** is given as **:full** or the same as **:record-length**.

All Lisp Machine tape applications (LMFS and distribution dumpers and carry tape) enforce a granularity of 4.

:prompt This is an optional string that is formatted into

tape:make-stream's prompt for a host name, if one is issued. It should describe the tape to be mounted in terms of the application program running. For instance, if this string is supplied as "**billing master**", **tape:make-stream** might prompt

```
Type name of tape host for billing master:
```

:no-bot-prompt Normally, **tape:make-stream** notices if the tape is offline, or not at BOT (beginning-of-tape) when it is called. If the tape is offline, **tape:make-stream** queries you to wait for it to become ready. If the tape is not at BOT, **tape:make-stream** queries you about rewinding it. Supplying a non-**nil** value for **:no-bot-prompt** suppresses these checks, allowing you to handle these exigencies in any way you choose. The message **:bot-p** can be sent to a tape stream to determine if it is at BOT, and **:check-ready** to wait for a tape to become ready.

:norewind Normally, **tape:make-stream** rewinds the tape at the time the stream is closed. Supplying a non-**nil** value for **:norewind** suppresses this behavior.

:lock-reason Another optional string describing the application. This string is used in error messages sent to other users who try to access the tape drive you are using. For instance, if it is supplied as "**daily billing run**", another user might see a message like:

```
Cannot mount tape:
Drive 0 in use by daily billing run.
```

19.2 Messages to Tape Streams

The following messages to tape streams are important. Tape streams, of course, also support standard stream messages appropriate to input or output streams. See the section "Streams", page 3.

These are the messages relevant to any kind of tape stream:

:close (&optional (*abort-p* **nil**))

Closes the stream. Normally, causes a rewind, and all the operations associated with **:rewind** (see the description of **:rewind**) to take place. The **:norewind** argument suppresses this rewind, although, for an output stream, buffered output is written, along with two EOFs. The tape is left positioned between the two EOFs, for industry-compatible tape, or after them, for cartridge tape.

:rewind Rewinds the tape. For input streams, buffered input is discarded before the rewind. For output streams, buffered output is written out, possibly padded, according to the current padding parameters, and then two EOFs written, before the rewind. No read-ahead is performed. This message does *not* wait for the rewind to complete.

:await-rewind Waits for a previously started rewind to complete.

:set-offline A **:rewind** is done, and the tape is set offline, or unloaded, as befits the controller and drive. The setting of the tape offline does not wait for the rewind to complete.

:clear-error If a tape error occurs, and is handled by you, you must send this message before attempting to continue using the stream. Otherwise, it remains in the error state, where it can only be closed.

:skip-file (&optional (*n* **1**))
Skips to, and past, a file mark (EOF). *n* is how many to skip, and can be negative, indicating backward motion. For input streams, all buffered input is discarded before the motion. For output streams, this operation is not valid unless the last thing written was an EOF, not a data record. Cartridge tape cannot skip backward. Forward motion is not allowed immediately after output.

:host-name The name of the host on which the tape is mounted.

:bot-p Returns **t** if the tape is at BOT (beginning of tape), and **nil** if not.

:check-ready Checks to make sure the tape drive is ready, and informs you, waiting interactively, if not.

These are the messages specifically relevant to tape input streams. Most of them are relevant only to input record mode, which is the mode requested by a value of **nil** for **:input-stream-mode**. See the description of the **:input-stream-mode** argument to the function **tape:make-stream**.

:clear-eof This clears the EOF state that results from reading an EOF mark. When an EOF is encountered, all character-reading operations encounter an end-of-file indication until **:clear-eof** is sent. This is needed in input stream mode as well as input record mode.

:discard-current-record

This discards the remainder of the current record, when in input record mode, and allows reading the next record. This message must be issued to progress past a record boundary in input record mode, even if all of the bytes in the record have been read. This is meaningless for cartridge tape.

:record-status (&optional (*error-p* **t**))

This is only valid in input record mode, and meaningless for cartridge tape. This call is only valid at the beginning of a record, that is, if no bytes have been read from the current record. It describes, via its return value, the record that is about to be read by the user. Here are the possible values:

an error object	The next record cannot be read, due to error. An error object is returned. If *error-p* is **t**, which is the default, an error is signalled in this case, instead of an error object being returned.
integer	The length of a good record, in bytes.
:eof	The next record is not a record at all, but an EOF (a file mark).

These are the messages relevant to tape output streams:

:write-eof Writes an EOF (a file mark). If a record is being built, it is written out. Whether or not it is padded depends upon the values of the arguments **:minimum-record-length** and **:minimum-record-length-granularity**.

:force-output Writes out any record being buffered. Whether or not it is padded depends upon the values of the arguments **:minimum-record-length** and **:minimum-record-length-granularity**. This is the normal way to end a record when record boundaries are significant, or short records are written. Otherwise, records are written when they are full.

:write-error-status (&optional *error-p*)

Verifies that all records have been written correctly. Tape streams often buffer many records ahead. **:write-error-status** waits for all buffered I/O to complete. If there was no error, **nil** is returned. If there was an error, an error object is returned describing the error. If *error-p* is non-**nil**, an error is signalled instead. If the error is end of tape, however, and **error-p** is **nil**, **:end-of-tape** is returned.

19.3 Tape Error Flavors

tape:tape-error *Flavor*

This set includes all tape errors. This flavor is built on **error**.

tape:mount-error *Flavor*

A set of errors signalled because a tape could not be mounted. This includes problems such as no ring and drive not ready. Normally, **tape:make-stream** handles these errors and manages mount retry. This flavor is built on **tape:tape-error**.

tape:tape-device-error *Flavor*

A hardware data error, such as a parity error, controller error, or interface error, occurred. This flavor has **tape:tape-error** as a **:required-flavor**.

tape:end-of-tape *Flavor*

The end of the tape was encountered. When this happens on writing, the tape usually has a few more feet left, in which the program is expected to finish up and write two end-of-file marks. Normally, closing the stream does this automatically. Whether or not this error is ever seen on input depends on the tape controller. Most systems do not see the end of tape on reading, and rely on the software that wrote the tape to have cleanly terminated its data, with EOFs.

This flavor is built on **tape:tape-device-error** and **tape:tape-error**.

Appendix A
The Character Set

Characters in the Symbolics standard character set whose codes are less than 200 octal (with the 200 bit off), and only those, are "printing graphics"; when output to a device they are assumed to print a character and move the "cursor" one character position to the right. (All software provides for variable-width character styles, so the term "character position" should not be taken too literally.)

Characters in the range of 200 to 236 inclusive are used for special characters. Character 200 is a "null character", which does not correspond to any key on the keyboard. The null character is not used for anything much. Characters 201 through 236 correspond to the special function keys on the keyboard such as RETURN. Some characters are reserved for future expansion.

It should never be necessary for a user or a source program to know these numerical values. Indeed, they are likely to be changed in the future. There are symbolic names for all characters; see below.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Symbolics characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding, and the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, will not be the same measured in original Symbolics characters as it will measured in the encoded ASCII characters.

In TOPS-20, Tenex, and ITS, in the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (delete/vt), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a "quoting character" for these codes. Codes 210 (overstrike), 211 (tab), 212 (line), and 214 (page), are converted to their ASCII cognates, namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

Most of the special characters do not normally appear in files (although it is not forbidden for files to contain them). These characters exist mainly to be used as "commands" from the keyboard.

A few special characters, however, are "format effectors" which are just as legitimate as printing characters in text files. The following is a list of the names and meanings of these characters:

Return	The "carriage return" character which separates lines of text. Note that the PDP-10 convention that lines are ended by a pair of characters, "carriage return" and "line feed", is not used.
Page	The "page separator" character which separates pages of text.
Tab	The "tabulation" character which spaces to the right until the next "tab stop". Tab stops are normally every 8 character positions.

The Space character is considered to be a printing character whose printed image happens to be blank, rather than a format effector.

There are some characters which are not typeable as keys on a Symbolics 3600 console, even though there are codes and names for such characters. Those characters are:

```
205 Macro            220 Stop-Output      231 Hand-Up
216 Quote            223 Status           233 Hand-Left
217 Hold-Output      230 Roman-IV         234 Hand-Right
```

The Symbolics standard character set consists of mappings for the octal codes 000-241. The codes 242-377 are unused in this character set. The names of the characters are in the table in sys:io;rddefs.lisp. Here is a table of the code mappings:

000	· Center-Dot	040	Space	100	@	140	‘
001	↓ Down-Arrow	041	!	101	A	141	a
002	α Alpha	042	"	102	B	142	b
003	β Beta	043	#	103	C	143	c
004	∧ And-sign	044	$	104	D	144	d
005	¬ Not-sign	045	%	105	E	145	e
006	ε Epsilon	046	&	106	F	146	f
007	π Pi	047	’	107	G	147	g
010	λ Lambda	050	(Open	110	H	150	h
011	γ Gamma	051	) Close	111	I	151	i
012	δ Delta	052	*	112	J	152	j
013	↑ Up-Arrow	053	+ Plus-sign	113	K	153	k
014	+ Plus-Minus	054	,	114	L	154	l
015	⊕ Circle-Plus	055	- Minus-sign	115	M	155	m
016	∞ Infinity	056	.	116	N	156	n
017	∂ Partial-Delta	057	/	117	O	157	o
020	⊂ Left-Horseshoe	060	0	120	P	160	p
021	⊃ Right-Horseshoe	061	1	121	Q	161	q
022	∩ Up-Horseshoe	062	2	122	R	162	r
023	∪ Down-Horseshoe	063	3	123	S	163	s
024	∀ Universal-Quantifier	064	4	124	T	164	t
025	∃ Existential-Quantifier	065	5	125	U	165	u
026	⊗ Circle-X	066	6	126	V	166	v
027	↔ Double-Arrow	067	7	127	W	167	w
030	← Left-Arrow	070	8	130	X	170	x
031	→ Right-Arrow	071	9	131	Y	171	y
032	≠ Not-Equals	072	:	132	Z	172	z
033	◊ Lozenge	073	;	133	[	173	{
034	≤ Less-Or-Equal	074	< Less-sign	134	\	174	|
035	≥ Greater-Or-Equal	075	= Equal-sign	135	]	175	}
036	≡ Equivalence	076	> Greater-sign	136	^	176	~
037	∨ Or-sign	077	?	137	_	177	∫ Integral

200	Null	210	Back-Space	220	Stop-Output	230	Roman-IV
201	Suspend	211	Tab	221	Abort	231	Hand-Up
202	Clear-Input	212	Line	222	Resume	232	Scroll
203	Reserved	213	Refresh	223	Status	233	Hand-Left
204	Function	214	Page	224	End	234	Hand-Right
205	Macro	215	Return	225	Square	235	Select
206	Help	216	Quote	226	Circle	236	Network
207	Rubout	217	Hold-Output	227	Triangle	237	Escape
240	Complete						
241	Symbol-Help						

Index