Release 5.0 Release Notes
 # 995050

March 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

1. Release 5.0: Introduction and Highlights

These notes accompany the release of Release 5.0. They describe changes made since Release 4.5. The changes are organized in the following sections. Within each section, the material is organized into incompatible changes, new features, and improvements.

Changes to the Lisp Language and Compiler in Release 5.0
This section describes changes relevant to the Lisp language and compiler. The biggest changes are these:

- Packages have completely changed for compatibility with Common Lisp. The **keyword** package is now separate from **user**, and it does not inherit from **global**. Files compiled in earlier systems will not work in Release 5.0 and should be recompiled.
- The procedure for logging in has changed as a result of a new network namespace system.
- The rubout handler has been renamed to the input editor and has been extensively changed.
- The readtable, reader, printer, and **open** have changed for compatibility with Common Lisp.
- The 3600 now supports IEEE-standard single- and double-precision floating point numbers.
- Several window system flavors and methods related to mouse input have been changed.
- Logical pathnames and translations have changed.
- Init file pathnames have been standardized.
- The first colon in a pathname now always delimits the host.

Changes to Networks in Release 5.0
This section describes changes in network implementation, interface, and protocols. The biggest change is the introduction of a network namespace system.

Changes to Utilities in Release 5.0
This section describes changes in what any other computer would call the operating system and utilities. This includes the

Debugger, the garbage collector, network support, and various system keyboard features. The most important changes are:

- Changes in the location of some keyboard keys on the 3600.
- A new system for yanking input in Zwei and the input editor.
- A new terminal program that incorporates Telnet and Supdup.
- A new utility, the Flavor Examiner, for finding information about flavors and methods.
- A new carry tape system.

Changes to the File System in Release 5.0
This section describes changes in the Lisp Machine file system. The major change is the introduction of accordion wildcards.

Changes to Zmacs in Release 5.0
This section describes changes in the Zmacs editor. The biggest change is the new yank system. See the document *Using the Input Editor*.

Changes to Zmail in Release 5.0
This section describes changes in the Zmail mail reading and sending program.

Changes to the FEP in Release 5.0
This section describes changes in the FEP. Release 5.0 requires FEP version number 17 or higher.

Release 5.0: Notes and Clarifications
This section contains explanations and clarifications of items that people found confusing in previous releases and documentation.

Release 5.0: Operations and Site Management
This section describes changes to the system and site configuration features of the system. These changes are important to the people who are responsible for the software at each site.

You can find all the incompatible changes by reading the first part of each section. A complete list of changes appears in the Table of Contents.

As in previous releases, many minor bugs have been fixed and performance in some areas has been improved. Only the more important or visible changes are mentioned here.

1.1 New Microcode in Release 5.0: 270 on 3600, 998 on LM-2

Release 5.0 world loads must be run with microcode version 270 on the 3600 and version 998 on the LM-2. The old world loads do not work with the new microcode, and the new world loads do not work with the old microcode.

2. Changes to the Lisp Language and Compiler in Release 5.0

2.1 Incompatible Changes to Lisp in Release 5.0

2.1.1 Changes to login

The **login** function has changed for Release 5 for compatibility with the new network naming scheme. The arguments are different. If you type a *user-name* argument that isn't the name of a known user in the network namespace, you are asked whether to supply a specific host to log into this time. Before **login** finishes, you are also prompted to add a user object to the network database, using **tv:edit-namespace-object**.

login *user-name* &key *host* (*load-init-file* **t**) *Function*

Note that although you enter the same user id for *user-name* as in previous releases, the user object that contains it now also contains the name of the host where your mail and init files reside. Therefore, you seldom need to supply a *host* argument to **login**. See the section "Network Database".

user-name is the name of a user. *host* is a particular host computer. If the value of *load-init-file* is **t**, as it is by default, the user's init file is loaded. If the value of *load-init-file* is **nil** the init file is not loaded.

You can log in as a registered user by not specifying a host, or you can log in to a specific host as a user on that host, not registered in the Lisp Machine namespace database.

If *host* requires passwords for logging in, you are asked for a password. When logging in to a TOPS-20 host, typing an asterisk before your password enables any special capabilities you may be authorized to use.

If anyone is logged into the machine already, **login** logs that user out before logging in *user-name*. See the function **logout**. **login** also runs the **login-initialization-list**. See the section "System Initialization Lists".

When **login** loads an init file, it looks for a file whose name depends on the host. See the section "Init File Naming Conventions". Init files should be written using **login-forms** so that **logout** can undo them. Usually, however, you cold boot the machine before logging in, to remove any traces of the previous user.

login returns **t**.

A typical use of **login** now looks like this:

```
(login 'djones)
```

If you supply an unknown user id and don't specify **:host**, you are given an opportunity to specify a particular host for the current login session, and to add the user object thus created to the network database (accomplished via **tv:edit-namespace-object**) for subsequent logins. You can instead select the Retry option, which is useful when the namespace server did not respond to your initial **login** request.

2.1.2 Changes to Packages

Packages have completely changed for Release 5.0. Formerly, packages were arranged in a hierarchy of *superpackages* and *subpackages*. This hierarchy no longer exists. Instead, symbols within a package are divided into *internal* and *external* symbols. One package can inherit the external symbols of another by *using* the second package. A package can also *import* or *export* symbols.

An important result of this reorganization is that the **keyword** package is no longer the same as **user**, and it does not inherit from **global** or any other packages. Thus, **foo** in the **user** package is no longer the same symbol as **:foo**, and **foo** in the **global** package is no longer the same symbol as **:foo**. The **fonts** package also no longer inherits from **global**.

You must change any symbols in your programs that are now in the wrong package. In particular, you must add package prefixes (colons) to symbols that are in the **global** or **user** package but should be in **keyword**, and you must remove package prefixes from symbols that are in **keyword** but should be in **global** or **user**.

You might have difficulty making these corrections because the editor currently signals an error when it parses a file in Lisp Mode that contains some symbols in the wrong package. To edit this kind of file, use Find File in Fundamental Mode (m-X).

All programs compiled in earlier systems should be recompiled in Release 5.0. The package information in code compiled in earlier systems is no longer valid. Unless your programs use symbol names that depend on the old package hierarchy, you should not have to rewrite programs to work in Release 5.0. Some functions and special forms have changed, but most changes are upward compatible.

Font files from previous releases load in Release 5.0, giving several warnings. You should then use the Font Editor to write files of type BFD. See the section "Default Font Format Now Bfd".

The *Lisp Machine Manual* chapter "Packages" has been rewritten. It describes both general changes to the package system and new functions, special forms, and condition flavors not documented here. See the document *Packages*.

2.1.2.1 New special form: defpackage

The special form **defpackage** replaces **package-declare**. **package-declare** still exists for compatibility with earlier systems.

See the special form **defpackage**.

2.1.2.2 New function: make-package

The function **make-package** replaces **pkg-create-package**. **pkg-create-package** still exists for compatibility with earlier systems.

See the function **make-package**.

2.1.2.3 intern, intern-local, intern-soft, and intern-local-soft return two values

The functions **intern**, **intern-local**, **intern-soft**, and **intern-local-soft** return two values instead of three. The second value is different but upward compatible.

See the function **intern**. See the function **intern-local**. See the function **intern-soft**. See the function **intern-local-soft**.

2.1.2.4 Optional argument to mapatoms-all and where-is eliminated

mapatoms-all and **where-is** no longer take an optional argument defaulting to the **global** package. They now always process all packages that are not invisible. The function **package-used-by-list** can help if you need to process only the subset of all packages that use some particular package.

See the function **mapatoms-all**. See the function **where-is**. See the function **package-used-by-list**.

2.1.3 Symbols in global and keyword packages with the same names

Before Release 5.0 the **keyword** package inherited from **global**. Symbols in the **global** and **keyword** packages with the same names were the same symbols. In Release 5.0 the keyword package does not inherit from **global**, and symbols in these packages with the same names are different symbols. Following is a partial list of symbols in the **keyword** package with the same names as symbols in **global**. Along with each symbol is the reason for its existence. Some other symbols that have the same names exist in both packages, but for these you would nearly always use the symbol in **global**.

Symbol	*Use*
:and	Method combination type
:append	Method combination type
:array	**defstruct** type, data type
:array-leader	**defstruct** type
:assoc	Choose-variable-values item type
:atom	Data type

:base	File attribute
:beep	Message to windows
:bitblt	Message to windows
:byte-size	**open** option
:cadr	#+ feature
:character	Data type
:close	Message to streams
:closure	Data type
:compile	**make-system** option
:cond	**trace** option
:defun-method	Function spec type
:delete	Message to pathnames (et al.)
:describe	Message to objects
:documentation	Property for **defvar** documentation
:equal	Message to (some) pathnames
:eval-when	**defstruct** option
:export	**defpackage** option
:fill-pointer	**make-array** option
:fix	Data type
:fixnum	Data type
:float	Data type
:flonum	Data type
:font	Internal; data type name is **font**
:funcall	Menu item type; constraint frames
:function	**trace** option
:get	Message to objects with property lists
:get-handler-for	Message to objects
:getl	Message to objects with property lists
:ibase	Obsolete file attribute
:import	**defpackage** option
:lambda-macro	Function spec type
:list	Method combination type; **defstruct** type
:list*	**defstruct** type
:make-array	**defstruct** option
:named-structure-symbol	**make-array** option
:nconc	Method combination type
:null	Data type
:open	Message to pathnames
:or	Method combination type
:otherwise	For **:case** method combination; use **otherwise** with **selectq**
:package	File attribute
:plist	Message to objects with property lists
:print	Message to streams
:progn	Method combination type
:prompt-and-read	Message to streams

:putprop	Message to objects with property lists
:random	Not a valid data type, but can be returned by one-argument **typep**
:read	Message to streams
:readline	**fquery** type
:remprop	Message to objects with property lists
:rubout-handler	Message to streams
:shadow	**defpackage** option
:shadowing-import	**defpackage** option
:step	**trace** option
:string-trim	**prompt-and-read** type
:symbol	Data type
:string	Data type
:tyi	Message to streams
:tyipeek	Message to streams
:tyo	Message to streams
:unbound-function	Internal to **who-calls** (not the special argument to it)

2.1.4 Symbols moved to or from global package

The following symbols were added to **global**:

block
byte
byte-position
byte-size
deallocate-whole-resource
defconstant
define-symbol-macro
defpackage
do-all-symbols
do-external-symbols
do-local-symbols
do-symbols
eql
export
find-all-symbols
import
keywordp
make-package
map-resource
mod
multiple-value-prog1
package-external-symbols
package-shadowing-symbols
package-use-list

package-used-by-list
pkg-add-relative-name
pkg-delete-relative-name
pkg-keyword-package
read-delimited-string
readline-or-nil
shadow
shadowing-import
tagbody
undefflavor
unexport
unuse-package
use-package
with-input-editing

The following symbols were added to **global** on the 3600 only:

%%arg-desc-quoted
%%arg-desc-rest-arg
%%q-fixnum
%%q-flonum
%%q-high-type
dfloat

The following symbols were removed from **global**:

◊g
◊p
bignum
def-open-coded
include
locative
open-code
page-table-area
pkg-contained-in
pkg-debug-copy
pkg-is-loaded-p
pkg-load
pkg-refname-alist
pkg-super-package
plane-ar-n
plane-as-n
process-class
screen-xgp-hardcopy
sg-area
sg-return-unsafe
small-flonum

source-file-name
supdup
telnet
unbind
with-resource

The following LM-2-specific symbols were removed from **global** on the 3600 only:

%%arg-desc-evaled-rest
%%arg-desc-fef-bind-hair
%%arg-desc-fef-quote-hair
%%arg-desc-quoted-rest
%%q-flag-bit
%%q-high-half
%%q-low-half
%24-bit-difference
%24-bit-plus
%24-bit-times
%activate-open-call-block
%allocate-and-initialize
%allocate-and-initialize-array
%arg-desc-evaled-rest
%arg-desc-fef-bind-hair
%arg-desc-fef-quote-hair
%arg-desc-quoted-rest
%assure-pdl-room
%divide-double
%float-double
%mar-high
%mar-low
%microcode-version-number
%multiply-fractions
%open-call-block
%p-deposit-field
%p-deposit-field-offset
%p-flag-bit
%p-mask-field
%p-mask-field-offset
%p-store-flag-bit
%remainder-double
%structure-boxed-size
%unibus-read
%unibus-write
%xbus-read
%xbus-write
***dif**
***plus**

***quo**
***times**
***unwind-stack**
ap-3
ar-3
art-32b
art-error
art-float
art-fps-float
art-half-fix
art-reg-pdl
art-special-pdl
art-stack-group-head
as-3
catch-all
cdr-error
clear-mar
dtp-array-header
dtp-array-pointer
dtp-entity
dtp-fef-pointer
dtp-free
dtp-header
dtp-instance-header
dtp-instance-variable-pointer
dtp-select-method
dtp-small-flonum
dtp-stack-group
dtp-symbol-header
dtp-trap
dtp-u-entry
enable-trapping
entity
entityp
fasd-update-file
fasl-append
fasload
font-next-plane
font-rasters-per-word
font-words-per-char
get-list-pointer-into-array
get-list-pointer-into-struct
get-locative-pointer-into-array
macro-compiled-program
mar-break
mar-mode

number-gc-on
print-error-mode
q-data-types
qc-file
qc-file-load
read-meter
return-next-value
set-current-band
set-current-microload
set-error-mode
set-mar
set-memory-size
small-float
small-floatp
swap-sv-of-sg-that-calls-me
swap-sv-on-call-out
trapping-enabled-p
write-meter
xstore

The following 3600-specific symbols were removed from **global** on the LM-2 only:

compiled-function-area
constants-area
control-tables
page-table-area
pname-area
property-list-area
stack-area
symbol-area
wired-control-tables

2.1.5 Keyword Symbols Are Self-evaluating

Keyword symbols now evaluate to themselves. You no longer have to quote them. The compiler takes account of this self-evaluation to produce efficient compiled code.

2.1.6 Functions moved from the si package to global: deallocate-whole-resource, map-resource

The functions **si:deallocate-whole-resource** and **si:map-resource** are now in the **global** package. These functions were previously undocumented.

deallocate-whole-resource *resource-name* *Function*

Deallocate all allocated objects of the resource specified by *resource-name*, returning them to the free-object list of the resource. You should use this function with caution. It marks all allocated objects as free, even if they are

still in use. If you call **deallocate-whole-resource** when objects are still in use, future calls to **allocate-resource** might allocate those same objects for another purpose.

map-resource *resource-name function* &rest *args* *Function*

Calls *function* once for every object in the resource specified by *resource-name*. *function* is called with the following arguments:

- The object
- **t** if the object is in use, or **nil** if it is free
- *resource-name*
- Any additional arguments specified by *args*

2.1.7 New special forms catch and throw replace *catch and *throw

The new special forms **catch** and **throw** are recommended for making nonlocal exits. ***catch** and ***throw** are supported for compatibility with earlier releases.

catch and **throw** differ from ***catch** and ***throw** mainly in the returned values: **catch** returns multiple values from its last body form when it exits normally, and **throw** causes **catch** and ***catch** to return multiple values that result from its second subform. You can use **catch** with ***throw** and ***catch** with **throw** (although this is not recommended). If control exits normally, the returned values depend on whether **catch** or ***catch** is used. If control exits abnormally, the returned values depend on whether **throw** or ***throw** is used.

The old Maclisp **catch** and **throw** macros are no longer supported on the LM-2. They were never supported on the 3600.

catch *tag body...* *Special Form*

catch is used with **throw** for nonlocal exits. **catch** first evaluates *tag* to obtain an object that is the "tag" of the catch. Then the *body* forms are evaluated in sequence, and **catch** returns the (possibly multiple) values of the last form in the body.

However, a **throw** or ***throw** form might be evaluated during the evaluation of one of the forms in *body*. In that case, if the throw "tag" is **eq** to the catch "tag" and if this **catch** is the innermost **catch** with that tag, the evaluation of the body is immediately aborted, and **catch** returns values specified by the **throw** or ***throw** form.

If the **catch** exits abnormally because of a **throw** form, it returns the (possibly multiple) values that result from evaluating **throw**'s second subform. If the **catch** exits abnormally because of a ***throw** form, it returns two values: the first is the result of evaluating ***throw**'s second subform, and the second is the result of evaluating ***throw**'s first subform (the tag thrown to).

On the LM-2 only, ***throw** and ***unwind-stack** cause the **catch** to return two additional values. If ***throw** is used, the third and fourth values are **nil**. If ***unwind-stack** is used, the third and fourth values are the third and fourth arguments to ***unwind-stack** (the active-frame-count and the action).

(catch 'foo *form*) catches a **(throw 'foo** *form*) but not a **(throw 'bar** *form*). It is an error if **throw** is done when no suitable **catch** exists.

The scope of the *tags* is dynamic. That is, the **throw** does not have to be lexically within the **catch** form; it is possible to throw out of a function that is called from inside a **catch** form.

On the LM-2 only, the values **t** and **nil** for *tag* are special: A **catch** whose tag is one of these values catches throws to any tag. These are for internal use only: **unwind-protect** uses **t**, and **catch-all** uses **nil**. The only difference between **t** and **nil** is in the error checking; **t** implies that after a "cleanup handler" is executed, control will be thrown again to the same tag. Thus, it is an error if a specific catch for this tag does not exist higher up the stack. With **nil**, the error check is not done.

Example:

```
(catch 'negative
  (mapcar (function (lambda (x)
                      (cond ((minusp x)
                             (throw 'negative x))
                            (t (f x)) )))
          y))
```

which returns a list of **f** of each element of **y** if they are all positive, otherwise the first negative member of **y**.

throw *tag form* *Special Form*

throw is used with **catch** to make nonlocal exits. It first evaluates *tag* to obtain an object that is the "tag" of the throw. It next evaluates *form* and saves the (possibly multiple) values. It then finds the innermost **catch** or ***catch** whose "tag" is **eq** to the "tag" that results from evaluating *tag*. It causes the **catch** or ***catch** to abort the evaluation of its body forms and to return all values that result from evaluating *form*. In the process, dynamic variable bindings are undone back to the point of the **catch**, and any **unwind-protect** cleanup forms are executed. An error is signalled if no suitable **catch** is found.

The scope of the *tags* is dynamic. That is, the **throw** does not have to be lexically within the **catch** form; it is possible to throw out of a function that is called from inside a **catch** form.

On the 3600, the value of *tag* cannot be the symbol **sys:unwind-protect-tag**; that is reserved for internal use. On the LM-2,

the values **t**, **nil**, and **0** for *tag* are reserved for internal use. At present you cannot use **t** and **nil** for *tag* on the 3600; this will be changed in a future release.

2.1.8 Nonkeyword form of make-array is obsolete

The nonkeyword form of **make-array** documented on page 113 of the *Lisp Machine Manual* is obsolete. All programs should use the form that takes keyword arguments in addition to the array dimensions.

See the function **make-array**.

2.1.9 string-length uses same coercion rules as string

string-length now uses the same rules as **string** in interpreting its argument as a string. In particular, it is an error for the argument to be a floating-point number, and **string-length** can now be used for an instance that handles the **:string-for-printing** message.

string-length *string* *Function*

string-length returns the number of characters in *string*. This function uses the same coercion rules as **string** in interpreting *string* as a string. **string-length** returns the **array-active-length** if *string* is a string, or the **array-active-length** of the pname if *string* is a symbol.

2.1.10 Change in type of array returned by string-append

Previously, when the first argument to **string-append** was an array, the result was an array of the same type. Now the result is an array of the same type as the argument with the greatest number of bits per element.

string-append &rest *strings* *Function*

Any number of strings are copied and concatenated into a single string. With a single argument, **string-append** simply copies it. The result is an array of the same type as the argument with the greatest number of bits per element. For example, if the arguments are arrays of type **art-string** and **art-fat-string**, an array of type **art-fat-string** is returned. **string-append** can be used to copy and concatenate any type of one-dimensional array. Example:

```
(string-append #/! "foo" #/!) => "!foo!"
```

2.1.11 Changes to Readtable, Reader, and Printer for Common Lisp

Changes have been made to the readtable, reader, and printer in preparation for the introduction of Common Lisp, which is not available in Release 5.0.

2.1.11.1 Reader Accepts Common Lisp Floating Point Exponents

The reader now accepts all exponent characters in Common Lisp.

Following is a summary of floating-point exponent characters and the way numbers containing them are read on the 3600 and LM-2.

Character	*3600*	*LM-2*
B or b	single-precision	flonum
D or d	double-precision	flonum
E or e	depends on value of **cl:*read-default-float-format***	depends on value of **cl:*read-default-float-format***
F or f	single-precision	flonum
L or l	double-precision	flonum
S or s	single-precision	small-flonum

2.1.11.2 New variable: cl:*read-default-float-format*

cl:*read-default-float-format* *Variable*

Controls how floating-point numbers with no exponent or an exponent preceded by "E" or "e" are read. Following is a summary of the way possible values cause these numbers to be read on the 3600 and LM-2:

Value	*3600*	*LM-2*
cl:single-float	single-precision	flonum
cl:double-float	double-precision	flonum
cl:short-float	single-precision	small-flonum
cl:long-float	double-precision	flonum

The default value is **cl:single-float**.

2.1.11.3 New descriptions: si:bitscale, si:digitscale, si:non-terminating-macro

Three syntax descriptions for characters have been added. **si:bitscale** and **si:digitscale** describe "left shift" operations on integers.
si:non-terminating-macro describes a character that can be a macro character or a symbol constituent. You can use **set-syntax-from-description** to set the syntax of characters to these descriptions.

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".

si: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 "".

si:macro A macro character. Do not use this; use **set-syntax-macro-char**.

si:circlecross The octal escape for special characters. In the initial readtable this is "⊗".

si:bitscale A character that causes the fixnum to its left to be doubled the number of times indicated by the fixnum to its right. In the initial readtable this is "_". See the section "What the Reader Accepts".

si:digitscale A character that causes the fixnum to its left to be multiplied by **ibase** the number of times indicated by the fixnum to its right. In the initial readtable this is "^". See the section "What the Reader Accepts".

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.)

These symbols will probably be moved to the standard keyword package at some point. *readtable* defaults to the current readtable.

2.1.11.4 New reader macro: #B

#B*rational* reads *rational* (an integer or a ratio) in binary (radix 2). Examples:

```
#B1101 <=> 13.
#B1100\100 <=> 3
```

2.1.11.5 New reader macro: #:

#:*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)**.

2.1.11.6 Printing Uninterned Symbols

By default, uninterned symbols print with a **#:** prefix. You can make them print without a prefix by evaluating **(setf (si:pttbl-uninterned-prefix readtable) "")**.

2.1.11.7 #/ and #\ now identical

The reader macros #/ and #\ are now identical. You can use either to read a printing character (such as **#\A**) or the name of a nonprinting character (such as **#/c-X**). Unless you are using Common Lisp, #/ and #\ continue to return integers to represent characters.

The editor, however, still distinguishes between these two syntaxes. #/ prevents the editor from treating the character immediately following as a special character; #\ does not. Thus, to include an open parenthesis character in your program, use #/(, not #\(.

See the section "Sharp-sign Abbreviations".

2.1.11.8 Reading and Printing Character Objects

For Common Lisp only, the reader macros #/ and #\ can create character objects, and a double-quoted string can create a string of characters, depending on a property of the readtable. Regardless of this, character objects print with a #\ prefix.

2.1.11.9 Ratios read in current ibase and print in current base

Two integers separated by \ are read as a ratio of the integers. Ratios are now read in the base that is the value of **ibase** and printed in the base that is the value of **base**, not in decimal.

2.1.11.10 New Rules for Reading Ambiguous Tokens

Some tokens could be read as either symbols or integers in a base larger than 10. The variables **si:*read-extended-ibase-unsigned-number*** and **si:*read-extended-ibase-signed-number*** determine how these tokens are read.

si:*read-extended-ibase-unsigned-number* *Variable*

Controls how a token that could be a number or a symbol, and does not start with a + or - sign, is interpreted when **ibase** is greater than ten.

nil	It is never a number.
t	It is always a number.
:sharpsign	It is a symbol at top level, but a number after **#X** or **#***n***R**.
:single	It is a symbol except immediately after **#X** or **#***n***R**.

The default value is **:single**.

si:*read-extended-ibase-signed-number* *Variable*

Controls how a token that could be a number or a symbol, and starts with a + or - sign, is interpreted when **ibase** is greater than ten.

nil	It is never a number.
t	It is always a number.
:sharpsign	It is a symbol at top level, but a number after **#X** or **#*n*R**.
:single	It is a symbol except immediately after **#X** or **#*n*R**.

The default value is **:sharpsign**.

Some tokens could be read as either integers or floating-point numbers. Such tokens are always read as integers. Thus, **1e0** is an integer if the value of **ibase** is at least 15. But **1.0e0** is always read as a floating-point number because of the decimal point.

2.1.12 Changes to make-syn-stream

The argument to **make-syn-stream** can now be a symbol or a locative. **make-syn-stream** returns an uninterned symbol.

make-syn-stream *symbol* *Function*

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 **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.

2.1.13 format directives ~@T and ~@* replace ~X and ~G

A new format directive, **~@T**, replaces **~X**, and the new directive **~@*** replaces **~G**. For the time being **~X** and **~G** still work as in previous releases, but in a future release they will become number-formatting directives for compatibility with Common Lisp.

~@T outputs a space. ***~n@T*** outputs *n* spaces.

~n@* "goes to" the *n*th argument (0 is the first). **~@*** or **~0@*** goes back to the first argument in *args*. Directives after a ***~n@**** will take sequential arguments after the one gone to. When within 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 **~***.

2.1.14 Changes to format:ochar

The **:sail** style keyword for **format:ochar** has been replaced by **:brief**, which has slightly different consequences.

format:ochar *character* &optional *style top-explain minwidth* *Function*
&rest *options*

format:ochar outputs *character* in one of three styles, selected by the *style* argument. *minwidth* and *options* control padding as usual.

If *style* is **:read**, **nil**, or not specified, then the character is printed using **#/** or **#** so that it could be read back in.

If *style* is **:editor**, then the output is in the style of the string "Meta-Rubout".

If *style* is **:brief**, a somewhat more abbreviated style is used in which "c-", "m-", and the like, are used to represent "Control" and "Meta", and shorter names for characters are also used when possible. See the section "The Character Set".

top-explain is useful with the **:editor** and **:brief** styles. It says that any character which has to be typed using the Symbol key should be followed by an explanation of how to type it. For example: **"α (Symbol-shift-A)"**.

2.1.15 Incompatible Changes to the Input Editor (Rubout Handler)

The input editor, formerly called the rubout handler, has been extensively redesigned. This section describes only the incompatible changes for Release 5.0. For new features: See the section "New Features Associated with the Input Editor (Rubout Handler)".

2.1.15.1 Changes to Input Editor User Interface

The history mechanism of the input editor has changed. Each stream that uses the input editor now maintains an infinite-length input history. A global kill history is maintained, shared with other streams that use the input editor and with Zwei-based editors. Commands for yanking input from these histories have changed. The histories are now emptied before cold booting or disk saving, not when logging out. See the document *Using the Input Editor*.

Formerly, when you yanked previous input to the input editor, that input was displayed without the final character. You could reevaluate a previous form by yanking it and then typing the final character.

Now the reader uses *activation characters*. The input editor displays yanked input fully. You can edit the input if you wish and "activate" it by pressing END from anywhere within the input you are editing. If you want to put an activation character into the input, "quote" it using c-Q.

readline and the Converse pop-up window also use activation. You can press RETURN anywhere within a line to activate it for **readline**. You can press END anywhere within a Converse pop-up reply to send the reply.

2.1.15.2 Changes to input editor options :do-not-echo, :pass-through, :prompt, :reprompt

:do-not-echo now implies that the character arguments are activation characters.

:do-not-echo *char1 char2 ...* *Option*

The characters *char1*, *char2*, and so on, are interpreted as activation characters. The comparison is done with =, 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.

:pass-through is now allowed only for characters with no modifier bits set.

:pass-through *char1 char2 ...* *Option*

The characters *char1*, *char2*, and so on 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**.

The second argument to the **:prompt** and **:reprompt** functions is now a keyword instead of a character.

:prompt *function-or-string* *Option*

If *function-or-string* is a function and it is time for the user to be prompted, *function-or-string* is called with two arguments. The first is a stream it may print on; the second is a keyword that indicates the origin of the function call:

Keyword	*Function called from*
:prompt	**:rubout-handler** method of **tv:stream-mixin**
:restore	**:restore-rubout-handler-buffer** method of **tv:stream-mixin**
:insert, :overwrite, :temporary	**:finish-typeout** method of **tv:stream-mixin**
:refresh	Body of the input editor

If *function-or-string* is a string and it is time for the user to be prompted, *function-or-string* is displayed as a prompt.

The difference between **:prompt** and **:reprompt** is that the latter does not call the prompt function or display the string 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 *function-or-string* *Option*

Like **:prompt** but calls *function-or-string* (if it is a function) or displays *function-or-string* (if it is a string) only when the 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.

2.1.15.3 New error flavors: sys:parse-error and sys:parse-ferror

The input editor no longer catches all flavors of error. It now catches only errors built on the flavors **sys:parse-error** and **sys:parse-ferror.**

sys:parse-error *Flavor*

This flavor is built on **error** and is the type of error caught by the input editor. This flavor accepts the init keyword **:correct-input.** If the value is **t,** which is the default, the input editor prints "Type RUBOUT to correct your input" and does not erase the message until a non-self-inserting character is typed. If the value is **nil,** no message is printed, and any typeout from the read function is erased immediately after the next character is typed. Syntax errors signalled by read functions should be built on top of this flavor.

sys:parse-ferror *Flavor*

This flavor is built on **sys:parse-error** and **ferror**. It accepts the init keywords **:format-string** and **:format-args** as well as **:correct-input**. This flavor exists for read functions that do not have a special flavor of error defined for them.

2.1.15.4 New function: sys:parse-ferror

sys:parse-ferror *format-string* &rest *format-args* *Function*

Signals an error of flavor **sys:parse-ferror**. *format-string* and *format-args* are passed as the **:format-string** and **:format-args** init options to the error object.

See the flavor **sys:parse-ferror**.

2.1.16 Changes to open

2.1.16.1 Changes to open option :direction

:direction :probe replaces **:direction nil**. New permissible values are **:probe-directory** and **:probe-link**. Various misspelled values such as **:in** that used to work for some file hosts are no longer supported for consistency.

:direction *Option*

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.
:probe	A "probe" opening; no data are to be transferred, and the file is being opened only to gain access to or change its properties. Returns the truename of the object at the end of a link or chain of links. (**probef** is usually preferable to an explicit probe opening.)
: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.
: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.

2.1.16.2 New open option: :estimated-length

:estimated-length *Option*

The value of the **:estimated-length** option may 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.

2.1.16.3 New open options: :if-exists and :if-does-not-exist

:if-exists determines what happens if the specified file already exists; **:if-does-not-exist** determines what happens if it does not exist. The ability to append files has been added.

:if-exists *Option*

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 **:newest**.

: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 **:newest**. File systems without version numbers may 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 doesn't 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 destructively 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.

:if-does-not-exist *Option*

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.

2.1.17 Changes to renamef and copyf

The following changes have been made to **renamef** and **copyf**:

- **renamef** now returns three values: the target pathname, the truename of the source pathname, and the truename of the target pathname. The **:rename** messages to streams and pathnames return the second and third of these values.

- For **renamef** and **copyf**, the target pathname defaults against the *link-opaque truename* of the source pathname.

- The **copyf** option **:copy-author** now defaults to **t**.

- A new **copyf** option, **:create-directories**, determines whether or not **copyf** tries to create a directory for the copy target.

renamef *file new-name* &optional *(error-p* **t***)* *Function*

renamef is a function for renaming one file to another. 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**. **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.

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">
```

copyf *from-path to-path* &key (*characters* **':default**) (*byte-size* **nil**) *Function*
(*copy-creation-date* **t**) (*copy-author* **t**)
(*report-stream* **nil**) (*create-directories* **':query**)

copyf is a function for copying 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 specs. *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**. **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**. 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 *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, **copyf** copies the creation date and author of the file.

Following is a description of the other options:

:characters Possible values:

:default	**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 will be opened for binary transfers. You must supply **:byte-size** when **:characters** is **nil** and the byte size is other than 16. Otherwise, **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 **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.

2.1.18 Changes to Host Determination in Pathnames

An important incompatible change has been made in the way the pathname system determines the host for a pathname being parsed. The first colon in a string to be parsed as a pathname now always delimits the host. You can "quote" embedded colons that are not intended to delimit the host by inserting a colon at the beginning of the string.

An upwardly compatible change has been made as well. From either a 3600 or an LM-2, you can now use the syntax "*host*|FEP*n*" to refer to a FEP file system on a remote 3600. *host* is the name of the host, and *n* is the disk unit number.

Following are the rules for host determination in a pathname.

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 which 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 may specify a host explicitly, by having the name of the host, followed by a colon, at the beginning of the string, or it may 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 "Defaults and Merging".

However, it is possible to have pathnames that 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**. The standard Lisp Machine user interface does not use such pathnames, but they may be used by particular programs, especially programs 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 one of these peculiar pathnames that have embedded colons *not* meaning hosts, 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 handful of "pseudo-" host names, which are recognized as host names even though they are not actually the names of hosts. They are "local", "FEP", "FEP*n*", and "*host*|FEP*n*".

"local" This pseudo-host 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" (3600 only) This pseudo-host 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*" (3600 only) 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 pseudo-host name refers to a FEP file system on a remote 3600. This may be used from LM-2's, as well as 3600's, to reference FEP file systems of remote 3600's. The syntax "*host*|FEP" is *not* acceptable: you may not access the "boot unit" of a remote 3600 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.

2.1.19 Meaning of argument changed for fs:parse-pathname

The meaning of the second, *with-respect-to* argument to **fs:parse-pathname** has changed. *with-respect-to* specifies a host against which to parse the first argument, *thing*. Formerly, if *thing* contained a host name and *with-respect-to* was not **nil**, an error was signalled when the two hosts were not the same. When they were the same, the host name was removed from *thing* before it was parsed. Now this is true only if *thing* is a Maclisp-style list.

Now, if *thing* is a string and *with-respect-to* is not **nil**, *thing* is taken as a true string for the host specified by *with-respect-to*. Host names are not removed from *thing* before it is parsed, and *thing* is parsed against the host specified by *with-respect-to*. Thus, when *with-respect-to* is not **nil**, *thing* should not contain host names.

This change was made necessary by the change in host naming conventions. See the section "Changes to Host Determination in Pathnames".

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

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

2.1.20 Arguments changed for fs:user-homedir and fs:init-file-pathname

The second, optional argument to **fs:user-homedir** has been removed. **fs:init-file-pathname** now has two optional arguments: the canonical type of the init file, and the host.

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 Lisp Machine it is the directory where the user keeps personal files such as init files and mail. This function returns a pathname without any name, type, or version component (those components are all **nil**).

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

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.

2.1.21 Init File Pathnames Standardized

The names of init files have been standardized, and init files are now of canonical type **:lisp** for source files and **:bin** and **:qbin** for compiled files. You must change the names of your init files for Release 5.0.

Init files are of canonical type **:lisp** for source files and **:bin** or **:qbin** 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 lispm 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.

2.1.22 :init canonical pathname type removed

The **:init** canonical type has been removed. Init files are now of canonical type **:lisp** for source files and **:bin** and **:qbin** for compiled files.

2.1.23 Changes to Logical Pathnames

2.1.23.1 Logical Pathname Name, Type, and Version Now Separated by Periods

The name, type, and version of logical pathnames are now separated by periods. Spaces are accepted on input for compatibility.

2.1.23.2 New Default Representations for Newest and Oldest Logical Pathname Versions

The default representation in a logical pathname for the **:newest** version is the string "NEWEST". The default representation for the **:oldest** version is the string "OLDEST". On input, ">" is accepted for **:newest** and "<" for **:oldest** for compatibility.

2.1.23.3 Logical Pathnames Now Hierarchical

Logical pathnames can now have hierarchical directories. Each directory level is separated by a semicolon.

2.1.23.4 Changes to Logical Pathname Translations

The procedure for translating logical to physical pathnames has changed to conform to the rules for wildcard pathname matching. Logical directory names in translation lists should now be terminated by semicolons (though **fs:set-logical-pathname-host** accepts logical directory names without semicolons).

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:

```
'(("L-BIN;" ">lmach>fasl>")
  ("L-COMPILER;" ">sys>l-compiler>")
  ("L-SYS;" ">lmach>")
  ("L-*;" ">lmach>*>")
  ("LMFS-PATCH;" ">sys>lmfs>patch>")
  ("*;" ">sys>*>")))
```

Note that logical directory names should be followed by semicolons (though **fs:set-logical-pathname-host** accepts names without semicolons).

The method of translating a logical pathname consists of matching it against each first element of each translation, in succession. The order in the list is thus very important. At the first match, the translated pathname is produced by sending the **:translate-wild-pathname** 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". See the section "Wildcard Directory Mapping".

Note that it is possible to have a translation that matches "everything else", as in the example below. In the presence of such a translation, it is impossible to have an undefined translation.

Back-translation is performed by searching the second elements of the translations list, and translating in the other direction.

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.

2.1.23.5 Flavor fs:undefined-logical-pathname-translation replaces fs:undefined-logical-pathname-directory

fs:undefined-logical-pathname-translation is a new condition flavor signalled when a logical pathname is referenced but has no translation to a physical pathname. It replaces **fs:undefined-logical-pathname-directory**.

fs:undefined-logical-pathname-translation *Flavor*

A logical pathname was referenced but is not defined. The **:logical-pathname** message returns the logical pathname. This flavor has a **:define-directory** proceed type, which prompts for a physical pathname whose directory component is the translation of the logical directory on the given host.

2.1.24 fs:make-logical-pathname-host replaces fs:add-logical-pathname-host

The function **fs:add-logical-pathname-host** is obsolete, though currently supported for compatibility. You should now use **fs:make-logical-pathname-host** to define a logical pathname host. This function loads the file sys:site;*host-name*.translations, which should contain a single form: a call to **fs:set-logical-pathname-host**.

fs:make-logical-pathname-host *host-name* *Function*

Defines *host-name*, which should be a string or symbol, to be the name of a logical pathname host. *host-name* should not conflict with the name of any existing host, logical or physical.

This function loads the file sys:site;*host-name*.translations. This file should contain a single form: a call to **fs:set-logical-pathname-host**. The file is always loaded into the **file-system** package. See the function **fs:set-logical-pathname-host**.

fs:make-logical-pathname-host not only loads this file but also arranges for the same 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 **make-system** on this system.

When a world load is taken to a new site, the translation file for each logical host that is defined in the current world is reloaded from the new site's sys:site; directory. This changes all logical pathnames to map into the new set of physical pathnames defined by the new site.

An **fs:make-logical-pathname-host** form often appears in the file

sys:site;*system-name*.system. **make-system** looks for this file when given the name of an unknown system. In addition to a call to **fs:make-logical-pathname-host**, this file should contain a call to **si:set-system-source-file**, which specifies the logical pathname of the file containing the **defsystem** form.

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")
```

2.1.25 Previously undocumented function: fs:set-logical-pathname-host

fs:set-logical-pathname-host creates a logical host and defines translations from logical directories on that host to physical directories on a physical host. A call to this function is the only form in the file sys:site;*logical-host*.translations, which is loaded by **fs:make-logical-pathname-host**.

fs:set-logical-pathname-host *logical-host* &key *physical-host translations no-translate* *Function*

fs:set-logical-pathname-host 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 *physical-host*. *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".

Logical directory names should be terminated by semicolons, but **fs:set-logical-pathname-host** accepts names without semicolons. Host names can appear in the strings in the translations list, but each logical host in a string must refer to the same host as *logical-host*, and each physical host in a string must refer to the same host as *physical-host*. If the physical pathname is on a TOPS-20 or VMS device, you must include the host name (either explicitly or implicitly, with an initial colon) so that the device is not taken to be the host.

If *no-translate* is **nil** or unsupplied, 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** this mapping is suppressed, and some physical pathnames might not get the properties of the logical pathname. The consequences of this are unknown.

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.

Example:

Following is the contents of the file sys:site;cube.translations:

```
;;; -*- Mode: LISP; Package: FILE-SYSTEM -*-

(set-logical-pathname-host "cube"
  ':physical-host "pointer"
  ':translations '(("cube;" ">cube>")))
```

2.1.26 load-file-list obsolete

The function **load-file-list** is obsolete. Use **make-system** instead.

2.1.27 Change in arguments to print-herald

print-herald no longer accepts an optional argument of the stream for display; display now always goes to **standard-output**. Instead, it accepts two keyword arguments: **:as-if-band** is used by **disk-save**, and **:verbose** controls the display of system version numbers.

print-herald &key *as-if-band verbose* *Function*

Prints out the herald message to **standard-output**. The herald message is what the machine prints when it is cold booted. It shows you the name of the FEP file or partition for the current world load, any comment added to the herald, a measure of the physical memory and swapping space available, the versions of the systems that are running, the site name, and the machine's own host name.

:as-if-band is used by **disk-save** to supply the name of the FEP file or partition of the saved world. **:verbose** controls the system version information displayed: if **t**, the version numbers of all systems are displayed; if **nil**, the version numbers of only those systems that differ from the release are displayed.

2.1.28 Change in arguments to unadvise

The three subforms of **unadvise** are now independent. If *function* is **nil** but *class* or *position* is not, **unadvise** removes only the specified classes or positions of advice for all functions.

unadvise &optional *function class position* *Special Form*

Removes pieces of advice. None of its subforms are evaluated. *function* and *class* have the same meaning as they do in the function **advise**. *position* specifies which piece of advice to remove. It can be the numeric index (0 means the first one) or it can be the name of the piece of advice.

unadvise can remove more than one piece of advice if some of its arguments are missing or **nil**. The arguments *function*, *class*, and *position* all act independently. A missing value or **nil** means all possibilities for that aspect of advice. For example, the following form removes all **:before**, **:after**, and **:around** advice named **negative-arg-check** on the **factorial** function.

```
(unadvise factorial nil negative-arg-check)
```

In this example **unadvise** removes all **:around** advice on all functions in all positions with all names.

```
(unadvise nil :around)
```

In this example **unadvise** removes all classes of advice named **my-personal-advice** on all functions.

```
(unadvise nil nil my-personal-advice)
```

(unadvise) removes all advice on all functions, since *function*, *class*, and *position* take on all possible values.

2.1.29 Window System Changes Associated with Mouse Input

2.1.29.1 Flavors tv:any-tyi-mixin and tv:list-tyi-mixin obsolete

The flavors **tv:any-tyi-mixin** and **tv:list-tyi-mixin** are obsolete. The **:tyi** and **:tyi-no-hang** methods of **tv:stream-mixin** have been renamed to **:any-tyi** and **:any-tyi-no-hang**. The **:tyi**, **:tyi-no-hang**, **:list-tyi**, **:mouse-or-kbd-tyi**, and **:mouse-or-kbd-tyi-no-hang** methods of **tv:any-tyi-mixin** and **tv:list-tyi-mixin** have been moved to **tv:stream-mixin**. The **tv:any-tyi-mixin** and **tv:list-tyi-mixin** flavors still exist for compatibility, but they have no effect.

2.1.29.2 Changes to :tyi, :tyi-no-hang, :list-tyi, :mouse-or-kbd-tyi, and :mouse-or-kbd-tyi-no-hang methods of tv:stream-mixin

The **:tyi** method of **tv:stream-mixin** has been renamed to **:any-tyi**, and **:tyi-no-hang** has been renamed to **:any-tyi-no-hang**. The behavior of **:any-tyi** is somewhat more complex because of interactions with the input editor. If you want to receive input that might be integers (character codes) or blips (such as mouse clicks or activation blips), you should send these messages instead of **:tyi** and **:tyi-no-hang**. The **:tyi** and **:tyi-no-hang** methods now always discard blips; previously they did so only if **tv:any-tyi-mixin** was a component of the window flavor. You can send these messages if you want to receive only keyboard input. If you want to receive only blips, send the **:list-tyi** message.

The **:mouse-or-kbd-tyi** and **:mouse-or-kbd-tyi-no-hang** methods are of use to Zwei but probably not to any other programs.

:any-tyi &optional *eof-action* of **tv:stream-mixin** *Method*

Read and return the next character of input from the window, waiting if there is none. Where the character comes from depends on the value of the variable **rubout-handler**. Following is a summary of actions for each possible value of **rubout-handler**:

nil	If the input buffer contains unscanned input, take the next character from there. Otherwise, take the next character from the window's I/O buffer.
:read	If the input buffer contains unscanned input, take the next character from there. Otherwise, if an activation blip or character is present, return that. Otherwise, enter the input editor.
:tyi	Take the next character from the window's I/O buffer.

If *eof-action* is **nil**, an error is signalled when an end-of-file is encountered. Otherwise, the method returns **nil** when an end-of-file is encountered.

:any-tyi-no-hang &optional *eof-action* of **tv:stream-mixin** *Method*

Check the window's I/O buffer and return the next character if it is immediately available. If no characters are immediately available, return **nil**. It is an error to call this method from inside the input editor (that is, if the value of **rubout-handler** is not **nil**). *eof-action* is ignored. This is used by programs that continuously do something until a key is typed, then look at the key and decide what to do next.

:tyi &optional *eof-action* of **tv:stream-mixin** *Method*

If called from outside the input editor, this is the same as **:any-tyi**, except that only integers and **nil** can be returned. Blips are discarded, unless the first element of the blip is **:mouse-button** and the second element is **#\mouse-r-1**; in this case, the method pops up a system menu. If called from inside the input editor with **:full-rubout** specified and if an activation blip is read when the input buffer is empty, the method causes control to be returned from the input editor.

:tyi-no-hang &optional *eof-action* of **tv:stream-mixin** *Method*

This is like **:any-tyi-no-hang**, except that only integers and **nil** can be returned. Blips are discarded, unless the first element of the blip is **:mouse-button** and the second element is **#\mouse-r-1**; in this case, the method pops up a system menu.

:list-tyi of **tv:stream-mixin** *Method*

This is like **:any-tyi** except that it only returns blips and never returns integers. If it encounters any integers in the input stream, it discards them entirely (they are removed from the I/O buffer and the program never sees them).

:mouse-or-kbd-tyi of **tv:stream-mixin** *Method*

This is like **:any-tyi**, except that it returns two values, and it discards all blips but those whose first element is the symbol **:mouse**. In this case it returns the third element of the blip and the blip itself. Otherwise, if it sees an integer or **nil**, it returns that as both returned values. Blips whose first element is **:mouse** are produced by the user's clicking on the mouse while inside the editor. This method is used only by Zwei.

:mouse-or-kbd-tyi-no-hang of **tv:stream-mixin** *Method*

This is like **:any-tyi-no-hang**, except that it returns two values, and it discards all blips but those whose first element is the symbol **:mouse**. In this case it returns the third element of the blip and the blip itself. Otherwise, if it sees an integer or **nil**, it returns that as both returned values. Blips whose first element is **:mouse** are produced by the user's clicking on the mouse while inside the editor. This method is used only by Zwei.

2.1.29.3 Flavors tv:list-mouse-buttons-mixin and tv:kbd-mouse-buttons-mixin obsolete

The flavors **tv:list-mouse-buttons-mixin** and **tv:kbd-mouse-buttons-mixin** are obsolete. The facilities they provided for interpreting mouse clicks have been incorporated into the **:mouse-click** method of **tv:essential-mouse**. The flavors still exist for compatibility, but they have no effect.

2.1.29.4 Changes to :mouse-click method of tv:essential-mouse

The **:mouse-click** method of **tv:essential-mouse** now always sends blips to any window with an I/O buffer. The blips are of the form previously provided by **tv:list-mouse-buttons-mixin**. When the click is **#\mouse-r-1**, this method no longer pops up a system menu.

This change allows programs to receive blips from any windows, including Lisp Listeners, without having to define special flavors of window.

:mouse-click *buttons x y* of **tv:essential-mouse** *Method*

This method is called by the **:mouse-buttons** method of **tv:essential-mouse**, which is called by the default mouse handler when mouse buttons are pushed. *buttons* is an encoded integer representing the buttons pushed; use reader macros like **#\mouse-r-1** to handle these integers in your program. *x* and *y* represent the position of the mouse at the time of the click, in the window's outside coordinates.

If the click is **#\mouse-r-2**, the **:mouse-buttons** method pops up a system menu. Otherwise, if the window has an I/O buffer, **:mouse-click** sends it a blip of the form (**:mouse-button** *buttons window x y*). In addition, if the click is **#\mouse-l-1**, the window is selected.

:mouse-click methods are combined using **:or** combination, so the

:mouse-click method of **tv:essential-mouse** runs only if no earlier method handles the message (and all earlier methods return **nil**).

The following example illustrates the use of the **:any-tyi** message to receive both mouse and keyboard input to windows. It is a simple drawing program whose command loop accepts single keystroke or mouse click commands. This program does not require any special flavor of window in order to run. It runs using any window that can become the value of **terminal-io**.

```
(defun draw-help ()
  (send terminal-io ':clear-window)
  (format t "Click the left mouse button to draw a square.~@
             Click the middle mouse button to draw a circle.~@
             Click the right mouse button to draw a triangle.~@
             Type REFRESH to clear the screen.~@
             Type END to exit.~@
             Type HELP for documentation.~%"))

(defun draw ()
  (draw-help)
  (loop for command = (send terminal-io ':any-tyi)
        do (cond ((fixp command)
                  (selectq command
                    (#\refresh (send terminal-io ':clear-window))
                    (#\end (return))
                    (#\help (draw-help))
                    (t (beep))))
                 ((eq (car command) ':mouse-button)
                  (destructuring-bind (click nil x y) (cdr command)
                    (selectq click
                      (#\mouse-l-1 (send terminal-io ':draw-rectangle 20 20 x y))
                      (#\mouse-m-1 (send terminal-io ':draw-circle x y 10))
                      (#\mouse-r-1 (send terminal-io ':draw-triangle
                                         x y (- x 10) (+ y 20) (+ x 10) (+ y 20)))
                      (t (beep)))))
                 (t (beep)))))
```

2.1.29.5 Flavor tv:preemptable-read-any-tyi-mixin obsolete

The flavor **tv:preemptable-read-any-tyi-mixin** is obsolete. It has been replaced by the **:preemptable** input editor option. The flavor still exists for compatibility, but it has no effect. The **:preemptable-read** message is supported by **tv:stream-mixin** for compatibility.

: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.

2.1.30 :clear-screen, :clear-eol, and :clear-eof messages to windows renamed

The following messages to windows have been renamed:

- **:clear-screen** is now **:clear-window**
- **:clear-eol** is now **:clear-rest-of-line**
- **:clear-eof** is now **:clear-rest-of-window**

In Release 5.0, windows continue to accept the old messages for compatibility. The cold-load stream does not accept the new messages.

In a future release, **:clear-eof** will become a no-op so that the meaning of this message will be compatible with that for noninteractive streams. The cold-load stream will accept the new messages.

The **:clear-eof** message was renamed because it had two different meanings. For windows, it meant to clear the window from the cursor position to the bottom. For noninteractive streams, it meant to read the EOF indicator, so that data past the EOF could be read. The other two messages were renamed to be consistent with modern naming conventions.

:clear-window of **tv:sheet** *Method*

Erase the whole window and move the cursor position to the upper left corner of the window.

:clear-rest-of-line of **tv:sheet** *Method*

Erase from the current cursor position to the end of the current line; that is, erase a rectangle horizontally from the cursor position to the inside right edge of the window, and vertically from the cursor position to one line-height below the cursor position.

:clear-rest-of-window of **tv:sheet** *Method*

Erase from the current cursor position to the bottom of the window. In more detail, first do a **:clear-rest-of-line**, and then clear all of the window past the current line.

2.2 New Features in Lisp in Release 5.0

2.2.1 New function: eql

eql *x y* *Function*

eql returns **t** if is arguments are **eq**, or if they are numbers of the same type with the same value, or (in Common Lisp) if they are character objects that represent the same character. The predicate = compares the values of

two numbers even if the numbers are of different types. Use **equal** or **string-equal** to compare the characters of two strings.

Examples:

```
(eql 'a 'a) => t
(eql 3 3)   => t
(eql 3 3.0) => nil
(eql 3.0 3.0) => t
(eql #/a #/a) => t
(eql (cons 'a 'b) (cons 'a 'b)) => nil
(eql "foo" "FOO") => nil
```

The following expressions might return either **t** or **nil**:

```
(eql '(a . b) '(a . b))
(eql "foo" "foo")
```

In Zetalisp:

```
(eql 1.0s0 1.0d0) => nil
(eql 0.0 -0.0) => nil
```

2.2.2 New special form: defconstant

The special form **defconstant** is used to declare a named constant.

defconstant *variable initial-value [documentation]* *Special Form*

defconstant declares the use of a named constant in a program. *initial-value* is evaluated and *variable* set to the result. The value of *variable* is then fixed. It is an error if *variable* has any special bindings at the time the **defconstant** form is executed. Once a special variable has been declared constant by **defconstant**, any further assignment to or binding of that variable is an error.

The compiler is free to build assumptions about the value of the variable into programs being compiled. If the compiler does replace references to the name of the constant by the value of the constant in code to be compiled, the compiler takes care that such "copies" appear to be **eql** to the object that is the actual value of the constant. For example, the compiler may freely make copies of numbers, but it exercises care when the value is a list.

In Zetalisp, **defconstant** and **defconst** are essentially the same if the value is other than a number, a character, or an interned symbol. However, if the variable being declared already has a value, **defconst** freely changes the value, whereas **defconstant** queries before changing the value (unless the **defconstant** form is in a patch file). **defconstant** assumes that changing the value is dangerous because the old value might have been incorporated into compiled code, which would be out of date if the value changed.

In general, you should use **defconstant** to declare constants whose value is a

number, character, or interned symbol and is guaranteed not to change. An example is π. The compiler can optimize expressions that contain references to these constants. If the value is another type of Lisp object or if it might change, you should use **defconst** instead.

documentation, if provided, should be a string. It is accessible to the **documentation** function.

2.2.3 New special forms: block and tagbody

block is a primitive special form used with **return-from** for premature exit from a piece of code. **tagbody** is a primitive special form used with **go** for unstructured transfer of control. **prog**, **do**, and their variants are effectively constructed out of **let**, **block**, and **tagbody** forms.

block *name form...* *Special Form*

block evaluates each *form* in sequence and normally returns the (possibly multiple) values of the last *form*. However, **(return-from** *name value***)** or one of its variants (a **return** or **return-list** form) might be evaluated during the evaluation of some *form*. In that case, the (possibly multiple) values that result from evaluating *value* are immediately returned from the innermost block that has the same name and that lexically contains the **return-from** form. Any remaining forms in that block are not evaluated.

name is not evaluated. It must be a symbol.

The scope of *name* is lexical. That is, the **return-from** form must be inside the block itself (or inside a block that that block lexically contains), not inside a function called from the block.

do, **prog**, and their variants establish implicit blocks around their bodies; you can use **return-from** to exit from them. These blocks are named **nil** unless you specify a name explicitly.

For example, the following two forms are equivalent:

```
(cond ((predicate x)
       (do-one-thing))
      (t
       (format t "The value of X is ~S~%" x)
       (do-the-other-thing)
       (do-something-else-too)))

(block deal-with-x
  (when (predicate x)
    (return-from deal-with-x (do-one-thing)))
  (format t "The value of X is ~S~%" x)
  (do-the-other-thing)
  (do-something-else-too))
```

tagbody *tag-or-statement...* *Special Form*

The body of a **tagbody** form is a series of *tags* or *statements*. A *tag* is a symbol; a *statement* is a list. **tagbody** processes each element of the body in sequence. It ignores *tags* and evaluates *statements*, discarding the results. If it reaches the end of the body, it returns **nil**.

If a **(go** *tag***)** form is evaluated during evaluation of a *statement*, **tagbody** searches its body and the bodies of any **tagbody** forms that lexically contain it. Control is transferred to the innermost *tag* that is **eq** to the *tag* in the **go** form. Processing continues with the next *tag* or *statement* that follows the *tag* to which control is transferred.

The scope of the *tags* is lexical. That is, the **go** form must be inside the **tagbody** construct itself (or inside a **tagbody** form that that **tagbody** lexically contains), not inside a function called from the **tagbody**.

do, **prog**, and their variants use implicit **tagbody** constructs. You can provide *tags* within their bodies and use **go** forms to transfer control to the *tags*.

For example, the following two forms are equivalent:

```
(dotimes (i n) (print i))

(let ((i 0))
  (when (plusp n)
    (tagbody
      loop
      (print i)
      (setq i (1+ i))
      (when (< i n) (go loop)))))
```

2.2.4 New special forms: multiple-value-call and multiple-value-prog1

multiple-value-call and **multiple-value-prog1** are new special forms for returning multiple values. **multiple-value-call** evaluates forms and uses the (possibly multiple) values as arguments to a function. **multiple-value-prog1** is like **prog1** except that it can return multiple values from its first form.

multiple-value-call *function body ...* *Special Form*

multiple-value-call first evaluates *function* to obtain a function. It then evaluates all the forms in *body*, gathering together all the values of the forms (not just one value from each). It gives these values as arguments to the function and returns whatever the function returns.

For example, suppose the function **frob** returns the first two elements of a list of numbers:

```
(multiple-value-call #'+ (frob '(1 2 3)) (frob '(4 5 6)))
  <=> (+ 1 2 4 5) => 12.
```

(Defun frob (some-list)
  (values (car some-list) (cadr some-list)))

multiple-value-prog1 *first-form body...* *Special Form*

multiple-value-prog1 is like **prog1**, except that if its first form returns multiple values, **multiple-value-prog1** returns those values.

2.2.5 3600 Supports Ieee Single- and Double-precision Floating Point

The 3600 supports IEEE-standard single-precision and double-precision floating-point numbers. Single-precision floating-point numbers have a precision of 24 bits, or about 7 decimal digits. Their range is from 1.1754944e-38 to 3.4028235e38. Double-precision floating-point numbers have a precision of 53 bits, or about 16 decimal digits. Their range is from 2.2250738585072014d-308 to 1.7976931348623157d308.

Number objects exist that are outside the upper and lower limits of the ranges for single and double precision. Larger than the largest number is +1e= (or +1d= for doubles). Smaller than the smallest number is -1e= (or -1d= for doubles). Smaller than the smallest normalized positive number but larger than zero are the "denormalized" numbers. Some floating-point objects are Not-a-Number (NaN); they are the result of (// 0.0 0.0) (with trapping disabled) and like operations.

IEEE numbers are symmetric about zero, so the negative of every representable number is also a representable number (on the 3600 only). Zeros are signed in IEEE format, but +0.0 and -0.0 act the same arithmetically. For example:

```
(= +0.0 -0.0) => t
(plusp 0.0)   => nil
(plusp -0.0)  => nil
(zerop -0.0)  => t
(eq 0.0 -0.0) => nil
```

See the IEEE standard: Microprocessor Standards Committee, IEEE Computer Society, "A Proposed Standard for Binary Floating-Point Arithmetic: Draft 8.0 of IEEE Task P754," *Computer*, March 1981, pp. 51-62.

Some related functions have been added or extended. The mathematical functions, such as **sin** and **log**, have been modified to accept both single- and double-precision arguments.

2.2.5.1 float returns a single-precision number

float always coerces its argument to a single-precision floating-point number, even if the argument is a double-precision number.

float *x* *Function*

Converts any kind of number to a flonum on the LM-2 and to a single-precision floating-point number on the 3600. Note that, on the 3600, **float** reduces a double-precision argument to single precision.

2.2.5.2 New function: dfloat

dfloat converts its argument to a double-precision floating-point number.

dfloat *x* *Function*

(3600 only) Converts any kind of number to a double-precision floating-point number.

2.2.5.3 New data types: :single-float and :double-float

On the 3600, an object of type **:single-float** is a single-precision floating-point number. An object of type **:double-float** is a double-precision floating-point number. The **:float** data type is the union of these two types.

See the function **typep**.

2.2.5.4 floatp returns t for any floating-point number

floatp returns **t** if its argument is either a single-precision or a double-precision floating-point number.

floatp *arg* *Function*

floatp returns **t** if its argument is a floating-point number, that is, a flonum or a small flonum on the LM-2 or a single- or double-precision floating-point number on the 3600. Otherwise it returns **nil**.

2.2.5.5 New functions: sys:single-float-p, sys:double-float-p

sys:single-float-p and **sys:double-float-p** are predicates to distinguish between single- and double-precision floating-point numbers.

sys:single-float-p *arg* *Function*

(3600 only) Returns **t** if *arg* is a single-precision floating-point number, otherwise **nil**.

sys:double-float-p *arg* *Function*

(3600 only) Returns **t** if *arg* is a double-precision floating-point number, otherwise **nil**.

2.2.6 New function: mod

mod *x y* *Function*

The same as **remainder**, except that the returned value has the sign of the *second* argument instead of the first. When there is no remainder, the returned value is **0**.

Examples:

```
(mod -3 2)  => 1
(mod 3 -2)  => -1
(mod -3 -2) => -1
(mod 4 -2)  => 0
```

2.2.7 New functions: byte, byte-size, byte-position

byte creates a byte specifier; **byte-size** extracts the size field of a byte specifier; **byte-position** extracts the position field of a byte specifier.

byte *size position* *Function*

Creates a byte specifier for a byte *size* bits wide, *position* bits from the right-hand (least-significant) end of the word.

Example:

```
(ldb (byte 3 4) #o12345) => 6
```

byte-size *byte-specifier* *Function*

Extracts the size field of *byte-specifier*. You can use **setf** on this form:

```
(setq a (byte 3 4))
(setf (byte-size a) 2)
(byte-size a) => 2
```

byte-position *byte-specifier* *Function*

Extracts the position field of *byte-specifier*. You can use **setf** on this form:

```
(setq a (byte 3 4))
(setf (byte-position a) 2)
(byte-position a) => 2
```

2.2.8 New Metering Tools for the 3600

A new set of program counter (PC) metering tools is available on the 3600.

Program counter (PC) metering is a tool to allow the user to determine where time is being spent in a given program.

PC metering essentially produces a histogram. At regular intervals, the Front End Processor (FEP) causes the main processor to task switch to special microcode. This microcode looks up the macro PC that contains the virtual address of the macroinstruction that the processor is currently executing. If this virtual address falls outside the *monitored range*, the microcode increments a count of the number of PCs that missed the monitored range. If the address is within the monitored range, the microcode subtracts the bottom of the monitored range from the PC, leaving a word offset. It then divides the word offset by the number of words per *bucket* and uses that as an index into the *monitor array*. Next, it increments that indexed element of the monitor array. This can only measure statistically where the macro PC is pointing; for the results to be valid, a relatively large number of samples per bucket must be available. FEP version 13 samples at about 170 samples per second, so the PC monitoring with that version is probably valid only for sessions that take longer than five to ten seconds.

You specify some range of the program to be monitored. The range is specified by lower and upper bounding addresses, and compiled functions that lie between those

addresses are monitored. The range is divided into some number of buckets. The relative amount of time that the program spends executing in each bucket is measured.

The parameters you specify are the range of addresses to be monitored, the number of buckets, and an array with one word for each bucket.

Some of the metering functions deal with *compiled functions*. In this context a compiled function is either a compiled code object or an **art-16b** array, into which escape functions (small, internal operations used by the microcode) compile.

meter:make-pc-array *size* *Function*

Makes a PC array with *size* number of buckets. This storage is wired, so you probably do not want this to be more than about 64. pages, or **(* 64. sys:page-size)** words.

meter:monitor-all-functions *Function*

Changes the microcode parameters so that the monitor array refers to every possible function in the Lisp world at the time of the execution of **meter:monitor-all-functions**. This usually causes many functions to map into a single bucket, and is therefore useful in obtaining a first estimate of which functions are using a significant portion of the execution time.

meter:expand-range *start-bucket* &optional (*end-bucket start-bucket*) *Function*

Changes the microcode parameters so that the entire monitor array refers only to the functions previously contained within the range specified by *start-bucket* and *end-bucket*. *start-bucket* and *end-bucket* are inclusive bounds.

meter:report &optional *function-list* *Function*

Prints a summary of the data collected into the monitor array. You should not have to supply the *function-list* argument.

meter:start-monitor &optional (*clear* **t**) *Function*

Enables collection of PC data. If *clear* is not **nil**, the contents of the monitor array are cleared. If *clear* is **nil**, the array is not modified, so that the new samples are simply added to the old.

meter:stop-monitor *Function*

Disables further collection of PC data.

meter:print-functions-in-bucket *bucket* *Function*

Prints all the compiled functions that map into the specified *bucket*.

meter:list-functions-in-bucket *bucket* *Function*

Returns a list of all the compiled functions that map into the specified *bucket*.

meter:range-of-bucket *bucket* *Function*
Returns the virtual address range that maps into the specified *bucket*.

meter:with-monitoring *clear body...* *Macro*
Enables monitoring around the execution of *body*. If *clear* is not **nil**, clears the monitor array first. See the function **meter:start-monitor**.

meter:map-over-functions-in-bucket *bucket function* &rest *args* *Function*
Calls *function* for every compiled function in the specified *bucket*. The first argument to *function* should be the compiled function, and any remaining arguments are *args*.

meter:function-range *function* *Function*
Returns two values, the buckets that contain the first and last instructions of *function*.

meter:function-name-with-escapes *object* *Function*
If *object* is a compiled function, returns the function spec of the compiled function. Otherwise, returns **nil**.

2.2.9 New Meters for the LM-2

Five new microcode meters have been added for the LM-2.

sys:%tv-clock-counter *Meter*
Counts down every 60th of a second. When it reaches zero it resets from the **sys:%tv-clock-rate** meter and causes a sequence break if enabled.

sys:%count-disk-page-read-operations-in-transporter *Meter*
The number of page faults that went to the disk in the transporter (part of the garbage collector).

sys:%count-disk-page-read-operations-in-scavenger *Meter*
The number of page faults that went to the disk in the scavenger (part of the garbage collector).

sys:%transporter-run-time *Meter*
The number of microseconds spent in the transporter (part of the garbage collector).

sys:%scavenger-run-time *Meter*
The number of microseconds spent in the scavenger (part of the garbage collector).

2.2.10 New special form: define-symbol-macro

A symbol macro translates a symbol into a substitute form. When the Lisp evaluator is given a symbol, it checks whether the symbol has been defined as a symbol macro. If so, it evaluates the symbol's replacement form instead of the symbol itself. Use **define-symbol-macro** to define a symbol macro.

define-symbol-macro *name form* *Special Form*

This special form defines a symbol macro. *name* is a symbol to be defined as a symbol macro. *form* is a Lisp form to be substituted for the symbol when the symbol is evaluated. A symbol macro is more like a subst than a macro: *form* is the form to be substituted for the symbol, not a form whose evaluation results in the substitute form.

A symbol defined as a symbol macro cannot be used in the context of a variable. You cannot use **setq** on it, and you cannot bind it. You can use **setf** on it: **setf** substitutes the replacement form, which should access something, and expands into the appropriate update function. Example:

```
(define-symbol-macro foo (+ 3 bar))
(setq bar 2)
foo => 5
```

Here is a more complex example. Suppose you want to define some new instance variables and methods for a flavor. You want to test the methods using existing instances of the flavor. For testing purposes, you might use hash tables to simulate the instance variables, using one hash table per instance variable with the instance as the key. You could then implement an instance variable **x** as a symbol macro:

```
(defvar x-hash-table (make-hash-table))
(define-symbol-macro x (send x-hash-table ':get-hash self))
```

To simulate setting a new value for **x**, you could use **(setf x** *value*), which would expand into **(send x-hash-table ':put-hash self** *value*).

2.2.11 New function: undefflavor

undefflavor reverses the effect of a **defflavor**.

undefflavor *flavor-name* *Function*

Removes the flavor named by *flavor-name*.

2.2.12 New option for defflavor: :required-init-keywords

:required-init-keywords declares that some init keywords must be specified when making an instance of a flavor.

:required-init-keywords *Option*

The arguments are keywords. It is an error to try to make an instance of this flavor or any incorporating it without specifying these keywords as arguments to **make-instance** (or **instantiate-flavor**) or a **:default-init-plist** option in a component flavor. This error can often be detected at compile time.

2.2.13 New option for defflavor: :mixture

:mixture lets you define a family of flavors and use init options to select the member of the family to instantiate.

:mixture *Option*

Defines a family of related flavors. When **make-instance** (or **instantiate-flavor**) is called, it uses keywords in the init-plist to decide which flavor of the family to instantiate. Thus, init options can be used to select the flavor as well as instance-variable values.

The *ancestral* flavor is the one that includes the **:mixture** option in its **defflavor**. The flavors in the family are automatically constructed by mixing various mixins with the ancestral flavor. The names for the family members are chosen automatically. The name of such an automatically constructed flavor is a concatenation of the names of its components, separated by hyphens; however, obvious redundancies are removed heuristically.

defflavor of the ancestral flavor also defines the automatically constructed flavors. **compile-flavor-methods** of the ancestral flavor also compiles combined methods of the automatically constructed flavors.

The **:mixture** option has the following form:

(:mixture *spec spec ...***)**

Each *spec* is processed independently, and all the resulting mixins are mixed together. A *spec* may be any of the following:

(keyword mixin)
Add mixin if the value of *keyword* is **t**; add nothing if **nil**.

(keyword (value mixin) (value mixin) ...)
Look up the value of *keyword* in this alist and add the specified mixin.

(keyword mixin subspec subspec ...)

(keyword (value mixin subspec subspec ...) ...)
Subspecs take on the same forms as specs. Subspecs are processed only when the specified keyword has the specified value. Use them when there are interdependencies among keywords.

A *mixin* is one of the following:

symbol	The name of a flavor to be mixed in
nil	No flavor needs to be mixed in if the keyword takes on this value
string	This value is illegal: Signal an error with the string as the message

make-instance and **instantiate-flavor** check that the keywords are given with legal values.

Example:

```
(defflavor cereal-stream (...) (stream)
  ...
  (:init-keywords :characters :direction
                  :ascii :hang-up-when-close)
  (:mixture (:characters
             (t nil (:direction
                      (:in buffered-line-input-stream)
                      (:out buffered-output-character-stream))
                    (:ascii ascii-translating-character-stream))
             (nil nil (:direction (:in buffered-input-stream)
                                  (:out buffered-output-stream))
                      (:ascii "Ascii translation is not
                               meaningful for binary streams")))
            (:hang-up-when-close hang-up-when-close-mixin)))
```

Note the need for an **:init-keywords** declaration for any keywords that are used only in the **:mixture** declaration.

In this declaration, any kind of stream may have a **:hang-up-when-close** option. The **:characters** option does not itself add any mixins (hence the **nil**), but the processing of the **:direction** option depends on whether it is used with a character stream or a binary stream. The **:ascii** option is allowed only for character streams, and we specify an error message if it is used with a binary stream. If **:ascii** had not been mentioned in the **:characters nil** case, the keyword would have been ignored by **make-instance** on the assumption that an **:init** method was going to do something with it.

2.2.14 New format directives: ~→ and ~←

~→text~← is useful for indenting *text*.

~→text~← indents *text* at the cursor position that is current at the time of the ~→. A ~→ must be terminated with a ~←. ~→ and ~← can be nested like ~[~] and ~<~>. This directive is especially useful in making error messages indent properly.

Example:

```
(format t "~&Error:  ~→~A~←" "File not found
for FOO.LISP.1")
```

prints

```
Error:  File not found
        for FOO.LISP.1
```

~← terminates a ~→. It is undefined elsewhere.

2.2.15 New special form: format:defformat

format:defformat defines a new **format** directive. The function associated with the directive should send its output to the value of **format:*format-output***. Directives that were written in the old style and that send their output to **standard-output** still work, but you should begin to convert them to the new form.

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**:

```
(format t "~\foo:bar\" ...)
```

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 <"
```

2.2.16 New Features Associated with the Input Editor (Rubout Handler)

This section describes new features in the input editor and reading functions. For incompatible changes: See the section "Incompatible Changes to the Input Editor (Rubout Handler)".

2.2.16.1 New input editor options: :no-input-save, :activation, :command, :preemptable

The **:no-input-save** option causes the input editor not to save the scanned contents of the buffer on the input history.

: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. **yes-or-no-p** uses **:no-input-save** by default.

:activation allows a reading function (like **read** or **readline**) to recognize activation characters.

: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, **readline**, not the input editor, types a Newline at the end of the input buffer when RETURN, END, or LINE is pressed.

:command allows control to be returned from the input editor on reading special characters.

: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.

:preemptable allows control to be returned from the input editor on reading blips.

: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.

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 **si: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) (ldb-test %%kbd-super c))

(defun command-loop ()
  (loop
    do (multiple-value-bind (value flag)
           (si:read-or-end standard-input nil '((:command command-char-p)
                                                (:preemptable :blip)
                                                (:prompt "Command loop input: ")))
         (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))))))
```

2.2.16.2 New macro: with-input-editing

with-input-editing defines a context in which a reading function can be called using the input editor.

with-input-editing (*stream-var input-editor-options parameters keyword*) *body...* *Macro*

This macro 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 this macro instead of sending a **:rubout-handler** message directly.

All the "arguments" are optional. *stream-var* is a variable bound to the stream from which characters are read; if *stream-var* is not provided or is **nil**, **standard-input** is used. *input-editor-options*, if provided and not **nil**, should evaluate to a list of options suitable for the first argument to the **:rubout-handler** message.

keyword determines the activation characters for the input editor:

Value	*Activation characters*
nil	None
: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.

The macro defines an internal function with *body* as its body and (*stream-var* . *parameters*) as its argument list. When the function containing the **with-input-editing** form is called from outside the input editor, the stream is sent a **:rubout-handler** message if it handles it. The arguments to the **:rubout-handler** message are the list of *input-editor-options*, including any activation option determined by *keyword*; the internal function; and any arguments to the internal function. If the function containing the **with-input-editing** form is called from inside the input editor or if the stream does not handle the **:rubout-handler** message, the internal function is called instead.

parameters must be **nil** or a list of arguments and local variables that are defined outside the scope of the **with-input-editing** form but referenced within it. If a lexically external variable **x** is referenced within *body* but does not appear in the *parameters* list, the compiler issues the warning, "Lexical scoping not implemented for the variable **x**."

The following example defines a reading function.

```
(defun read-number
       (&optional (stream standard-input) input-editor-options
        input-radix or-nil)
  (with-input-editing (stream input-editor-options (input-radix or-nil)
                              :line)
    (loop with number
          with ibase = (or input-radix ibase)
          for string = (readline-or-nil stream)
          do (cond ((not string)
                    (if or-nil (return nil)))
                   ((numberp (setq number (ignore-errors
                                            (read-from-string string))))
                    (return number))
                   (t (sys:parse-ferror
                        "A~:[~; decimal~] number~:[~;, or <Return> for none,~] ~
                        is required"
                        (= ibase 10.) or-nil))))))
```

2.2.16.3 New function: read-delimited-string

read-delimited-string allows reading from a stream until a delimiter character is encountered.

read-delimited-string &optional (*delimiters* **#\end**) (*stream* **standard-input**) (*eof* **nil**) (*input-editor-options* **nil**) &rest (*make-array-args* **'(100. :type 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 may 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 noninteractive streams. *input-editor-options* are passed on as the first argument to the **:rubout-handler** message, after having an **:activation** entry prepended. *make-array-args* are arguments to be passed to **make-array** when constructing the string to return.

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**, **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 *read-delimited-string*: See the section "Examples of Use of the Input Editor".

2.2.16.4 New optional argument to read

A list of input editor options can now be passed to **read** as an optional argument.

read &optional (*stream* **standard-input**) *eof-option* *Function*
input-editor-options

read 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".

(This function can take its arguments in the other order, for Maclisp compatibility only.)

2.2.16.5 New function: si:read-or-end

read-or-end &optional (*stream* **standard-input**) *eof-option* *input-editor-options* *Function*

This function is like **read**, 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, END has the same meaning as for **read**. *eof-option* has the same meaning as for other reading functions. *input-editor-options* are passed to the input editor if the stream supports it.

The **:expression-or-end** and **:eval-form-or-end** options for **prompt-and-read** invoke **si:read-or-end**.

2.2.16.6 readline and readline-trim return additional values

readline and **readline-trim** now return four values: a string; an *eof* flag; the delimiter that terminated the string; and any numeric argument given the delimiter.

readline &optional (*stream* **standard-input**) *eof-option* *input-editor-options* *Function*

readline 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 **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, **readline** calls **read-delimited-string**, and *input-editor-options* is passed as the list of options to the input editor.

readline returns four values:

- The line as a character string, without the Newline character.
- 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.
- The character that delimited the string.
- Any numeric argument given the delimiter character.

See the function **read-delimited-string**.

readline-trim &optional (*stream* **standard-input**) *eof-option* *input-editor-options* *Function*

readline-trim trims leading and trailing whitespace from string input. "Whitespace" means spaces, tabs, or newlines. It takes the same arguments as the normal **readline** and returns the same four values.

Examples:

```
(readline-trim)   exciting option   RETURN =>
"exciting option"
NIL
141
NIL

(readline-trim)RETURN =>
""
NIL
141
NIL
```

The **:string-trim** option for **prompt-and-read** and **tv:choose-variable-values** uses **readline-trim**.

2.2.16.7 New function: readline-or-nil

readline-or-nil returns a string trimmed of white space, or else **nil** if the string is empty.

readline-or-nil &optional (*stream* **standard-input**) *eof-option* *Function*
input-editor-options

Like **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** choose-variable-values keyword use **readline-or-nil**.

See the function **readline-trim**.

2.2.16.8 New methods of tv:stream-mixin: :start-typeout, :finish-typeout, :rescanning-p, :force-rescan, :replace-input, :read-bp

Six new methods have been added to **tv:stream-mixin** for communication between the input editor and sophisticated reading functions that offer typeout and completion. The methods are **:start-typeout**, **:finish-typeout**, **:rescanning-p**, **:force-rescan**, **:replace-input**, and **:read-bp**.

:start-typeout *type* &optional *spacing* of **tv:stream-mixin** *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 **break**.
:temporary	Typeout appears after the current input and is erased after the user types a character.

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.

:finish-typeout &optional *spacing erase?* of **tv:stream-mixin** *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 **tv:stream-mixin** *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 **tv:stream-mixin** *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* of **tv:stream-mixin** *Method*

This message can be sent by a read function that uses the input editor to provide completion of the current input. *n-chars* is the number of characters to be removed from the end of the input buffer and erased from the screen. The substring of *string* determined by *begin* and *end* is then displayed on the screen. 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* will be seen as input.

:read-bp of **tv:stream-mixin** *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**.

2.2.16.9 New variable: tv:rh-typeout-default

tv:rh-typeout-default controls the style of typeout performed by the input editor.

tv:rh-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 **tv:stream-mixin**. These are **:insert**, **:overwrite**, **:append**, and **:temporary**. The default value is **:insert**.

2.2.16.10 Using the Input Editor: Examples

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** macro. 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 (stream '((:prompt "Type two lines: ")) () :end-activation)
    (list (readline stream) (readline stream))))
```

read-n-lines-1 is like **read-two-lines** except that you specify the number of lines to be read using the **n-lines** argument. This example illustrates passing a parameter into the body of the **with-input-editing** form.

```
(defun read-n-lines-1 (n-lines &optional (stream standard-input))
  (with-input-editing (stream '((:prompt "Type some lines: ")) (n-lines) :end-activation)
    (loop repeat n-lines collect (readline stream))))
```

read-n-lines-2 shows a different way of passing the **n-lines** parameter into the **with-input-editing** body. It uses a prompt function instead of a string to generate the prompt, and it passes the **n-lines** parameter to that function.

```
(defvar *n-lines*)

(defun read-n-lines-prompt (stream ignore)
  (format stream "Type ~R line~:P:~%" *n-lines*))

(defun read-n-lines-2 (*n-lines* &optional (stream standard-input))
  (with-input-editing (stream '((:prompt read-n-lines-prompt)) () :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 (stream '((:prompt "Type a sentence:  ")))
    (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 **sys:parse-ferror** to signal an error caught by the input editor.

```
(defun read-sentence-2 (&optional (stream standard-input))
  (with-input-editing (stream '((:prompt "Type a sentence:  ")) () :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 **sys: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) input-editor-options)
  (with-input-editing (stream input-editor-options () :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)))))))
```

2.2.17 New macro: sys:with-open-file-search

sys:with-open-file-search is like **with-open-file**, but it searches for a file with one of the types in a list of file types. **load** uses this to search first for a binary file and then for a source file.

sys:with-open-file-search (*stream-variable* *Macro*
(*operation defaults auto-retry*)
(*type-list-function pathname* . *type-list-args*) .
open-options) *body...*

sys:with-open-file-search performs a **with-open-file**, searching for a file with one of the types in a list of file types. **load** uses this macro 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 macro 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.

2.2.18 New condition flavor: fs:multiple-file-not-found

fs:multiple-file-not-found is signalled when none of a number of possible files is found. This condition is signalled by **sys:with-open-file-search** when it fails to find any file.

fs:multiple-file-not-found *Flavor*

None of a number of possible files was found. This flavor is built on **fs:file-lookup-error**. It is signalled when **load** is not given a specific file type but cannot find either a source or a binary file to load.

The flavor allows three init keywords of its own. These are also the names of messages that return the following:

:operation	The operation that failed
:pathname	The pathname given to the operation
:pathnames	A list of pathnames that were sought unsuccessfully

The condition has a **:new-pathname** proceed type to prompt for a new pathname.

2.2.19 New condition flavor: fs:rename-across-hosts

fs:rename-across-hosts *Flavor*

The hosts of the initial and target pathnames are not the same. This flavor is built on **fs:rename-failure**.

2.2.20 New variable: fs:*remember-passwords*

fs:*remember-passwords*, if not **nil**, causes the first password for each file access path to be remembered and suppresses prompting for passwords when the same person uses that access path again.

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.

2.2.21 New function: si:patch-loaded-p

si:patch-loaded-p is a predicate that tells whether the loaded version of a system is at or past a specified patch level.

si:patch-loaded-p *major-version minor-version* *Function*
&optional *(system* **"System"***)*

A predicate that tells whether the loaded version of *system* is past (or at) the specified patch level. Returns **t** if:

- the major version loaded is *major-version* and the minor version loaded is greater than or equal to *minor-version*

- the major version loaded is greater than *major-version*

Otherwise, the function returns **nil**.

2.2.22 New functions: si:make-process-queue, si:process-enqueue, si:process-dequeue, si:process-queue-locker, si:reset-process-queue

A process queue is a new facility for round-robin locking. Each process that requests a lock via a queue enters itself on the queue if the lock is not free. Processes are given a chance to seize the lock in the order in which they request it.

si:make-process-queue *name size* *Function*

Makes and returns a queue for processes requesting a lock. *name* is an external name for the queue and is used only in printing the queue. *size* is the size of the queue. This is the maximum number of processes that will be guaranteed to lock the queue in exact requesting order.

si:process-enqueue *queue* &optional *queue-value* (*whostate* **"Lock"**) *Function*

Locks *queue*. *queue-value* is an object to enter on the queue; if *queue-value* is **nil** or unsupplied, the object is the current process. If *queue* is empty, seizes the lock immediately by inserting *queue-value* on the queue and returning. If *queue* is not full but other processes are on the queue waiting for the lock to be free, inserts *queue-value* at the end of the queue, waits for the lock to be free, and then seizes the lock by returning. If *queue* is full, waits until *queue* is not full and tries again to seize the lock. *whostate* is displayed in the status line while waiting to seize the lock. Signals an error if *queue-value* has already seized the lock.

si:process-dequeue *queue* &optional *queue-value* (*error-p* **t**) *Function*

Unlocks *queue*. *queue-value* is an object on the queue. If *queue-value* is **nil** or unsupplied, it is the current process; if not **nil**, it should be the same as the *queue-value* given to the matching call to **si:process-enqueue**. If *queue-value* has the lock, unlocks the lock by removing *queue-value* from *queue* and giving the next process on the queue a chance to seize the lock. If *queue-value* does not have the lock and *error-p* is not **nil**, signals an error.

si:process-queue-locker *queue* *Function*

Returns the *queue-value* for the process that holds the lock on *queue*, or **nil** if the lock is free.

si:reset-process-queue *queue* *Function*

Unlocks *queue* and removes all processes on the queue.

2.2.23 New function: applyhook

applyhook provides a hook into **apply**, much as **evalhook** provides a hook into **eval**. It is useful for printing debugging information about **apply** operations.

applyhook *Variable*

When the value of this variable is not **nil** and **eval** calls **apply**, **applyhook** is bound to **nil** and the function that was its value is applied to two arguments: the function that **eval** gave to **apply** and the list of arguments to that function. The value it returns is returned from the evaluator.

applyhook *function args evalhook applyhook* *Function*

function is applied to *args* with **evalhook** lambda-bound to the function *evalhook* and with **applyhook** lambda-bound to the function *applyhook*. Like the **evalhook** function, this bypasses the first place where the relevant hook would normally be triggered. Either of the last two arguments can be nil.

The function **evalhook** now takes an optional third argument, an *applyhook* function to be called by **eval** after **eval** has evaluated the arguments to a function.

evalhook *form evalhook* &optional *applyhook* *Function*

evalhook is a function that helps exploit the **evalhook** feature. The *form* is evaluated with **evalhook** lambda-bound to the function *evalhook*. The checking of **evalhook** is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger.

```
Example:
;; This function evaluates a form while printing debugging
;; information.
(defun hook (x)
   (terpri)
   (evalhook x 'hook-function))

;; Notice how this function calls evalhook to evaluate the
;;  form f, so as to hook the subforms.
(defun hook-function (f)
   (let ((v (evalhook f 'hook-function)))
     (format t "form: ~s~%value: ~s~%" f v)
     v))

;; This isn't a very good program, since if f returns multiple
;; values, it will not work.
```

The following output might be seen from **(hook '(cons (car '(a . b)) 'c))**:

```
form: (quote (a . b))
value: (a . b)
form: (car (quote (a . b)))
value: a
form: (quote c)
value: c
(a . c)
```

Normally after **eval** has evaluated the arguments to a function, it calls the function. If *applyhook* exists, however, **eval** calls the hook with two arguments: the function and its list of arguments. The values returned by the hook constitute the values for the form. The hook could use **apply** on its arguments to do what **eval** would have done normally. This hook is active for special forms as well as for real functions.

Whenever either an evalhook or applyhook is called, both hooks are bound off. The evalhook itself can be **nil** if only an applyhook is needed.

applyhook catches only **apply** operations done by **eval**. It does not catch **apply** called in other parts of the interpreter or **apply** or **funcall** operations done by other functions such as **mapcar**. In general, such uses of **apply** can be dealt with by intercepting the call to **mapcar**, using the applyhook, and substituting a different first argument.

The argument list is like an &rest argument: it might be stack-allocated but is not guaranteed to be. Hence you cannot perform side-effects on it and you cannot store it in any place that does not have the same dynamic extent as the call to *applyhook*.

2.2.24 New variable: gc-on

gc-on *Variable*

The value of this variable is **t** when the garbage collector is turned on and **nil** when it is turned off. **gc-on** is useful in finding out whether the garbage collector has turned itself off (as it does when not enough free space remains to be able to complete a copying garbage collection).

2.2.25 New initialization list: :after-full-gc

si:full-gc runs the **:after-full-gc** initialization list after it collects all the garbage. It runs the previously undocumented **:full-gc** initialization list before it collects the garbage.

See the section "System Initialization Lists".

2.2.26 New variable: dbg:*debug-io-override*

dbg:*debug-io-override* can be used to direct the Debugger to a stream other than that designated by **debug-io**. This is useful mainly in complex debugging using the cold-load stream.

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

If the value of this variable is **nil** (the default), the Debugger uses the stream that is the value of **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.

2.2.27 New message to conditions: :special-command-p

You can send the **:special-command-p** message to a condition object to determine whether a command is one of the Debugger special commands for that object.

:special-command-p *command-type* of **condition** *Method*

Returns **t** if *command-type* is a valid Debugger special command for this condition object; otherwise, returns **nil**.

2.2.28 New macro: tv:with-mouse-grabbed-on-sheet

tv:with-mouse-grabbed-on-sheet grabs the mouse and confines it to a window. This is usually preferable to using **tv:with-mouse-grabbed**.

tv:with-mouse-grabbed-on-sheet (*sheet*) *body...* *Macro*

Evaluates *body* with the mouse grabbed and confined to *sheet*. During execution the variables **tv:mouse-x** and **tv:mouse-y** are relative to the window's outside coordinates. The default value of *sheet* is **self**.

2.2.29 New variable: tv:cold-load-stream-old-selected-window

This variable tells you which window was selected at the time you entered the cold load stream.

tv:cold-load-stream-old-selected-window *Variable*

At a cold-load stream break, the value of this variable is the value of **tv:selected-window** at the time you entered the cold-load stream.

2.2.30 New flavor: tv:margin-space-mixin

tv:margin-space-mixin lets you leave some blank space in the margins of a window. You can use the **:space** init option and the **:space** and **:set-space** messages to determine the amount of blank space to be left.

tv:margin-space-mixin *Flavor*

This flavor provides a margin item that just leaves some blank space. It might be useful if you're using scroll bars, and you want to leave a little white space between the scroll bar and the inside of the window.

:space (for **tv:margin-space-mixin**) *Init Option*

Initializes the amount of blank space in the margins of the window. Possible values:

nil	No space
t	One pixel blank in each of the four margins
n	*n* pixels of space in each of the four margins (*n* is an integer)

(*left top right bottom*)

left pixels blank in the left margin, *top* pixels blank in the top margin, and so on (values are integers)

:space of **tv:margin-space-mixin** *Method*

Returns a list of four elements, (*left top right bottom*). These are integers representing the number of pixels of blank space in the four margins of the window.

:set-space *new-space* of **tv:margin-space-mixin** *Method*

Specifies the amount of blank space to be left in the margins of the window. Possible values of *new-space*:

nil	No space
t	One pixel blank in each of the four margins
n	*n* pixels of space in each of the four margins (*n* is an integer)

(*left top right bottom*)

left pixels blank in the left margin, *top* pixels blank in the top margin, and so on (values are integers)

2.2.31 New font: fonts:cptfonti

A new tv font, **fonts:cptfonti**, is available. This is a fixed-width italic font of the same width and shape as **fonts:cptfont**, the default screen font. It is most useful for italicizing running text along with **fonts:cptfont**.

2.2.32 New Choose-variable-values Keywords

The following choose-variable-values keywords are new in Release 5.0 or were previously undocumented: **:choose-multiple, :string-or-nil, :decimal-number, :decimal-number-or-nil, :date-or-never, :past-date, :time-interval-or-never, :pathname, :pathname-or-nil, :pathname-list, :host, :host-list, :pathname-host, :keyword-list,** and **:font-list**.

Keyword *Action*

:choose-multiple *values-list print-function*
This type takes arguments like **:assoc** but permits the user to choose more than one element in the values-list. The variable is set to a list of all the values chosen.

:string-or-nil This value is a string or **nil** if the user just presses RETURN, LINE, or END.

:decimal-number This value is a decimal number, read and printed in radix 10.

:decimal-number-or-nil
This value is a decimal number, read and printed in radix 10, or **nil** if the user just presses RETURN, LINE, or END.

:date-or-never This value is a universal date-time or **nil** if the user types "never". An ambiguous date is interpreted as being in the future.

:past-date The value is a universal date-time. An ambiguous date is interpreted as being in the past.

:time-interval-or-never
The value is an integer representing the number of seconds in a time interval, or **nil** if the user types "never". The interval is read and printed as either "never" or alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years.

:pathname The value is a pathname, represented as a string. The pathname read is merged with the defaults in **fs:*default-pathname-defaults*** and has a default version of **:newest**.

:pathname-or-nil The value is a pathname, represented as a string, or **nil** if the user just presses RETURN, LINE, or END. The pathname read is merged with the defaults in **fs:*default-pathname-defaults*** and has a default version of **:newest**.

:pathname-list The value is a list of pathnames, read as a series of pathnames separated by commas and optional spaces, and merged with the defaults in **fs:*default-pathname-defaults***. The default version is **:newest**. The list is printed as a series of pathnames separated by commas and spaces.

:host The value is a network host, read and printed as the name of the host.

:host-list The value is a list of network hosts, read as a series of host names separated by commas or spaces, and printed as a series of host names separated by commas and spaces.

:pathname-host The value is a pathname host, read and printed as the name of the host. The name can be "local", "sys", or the name of another logical host as well as the name of a physical host.

:keyword-list The value is a list of symbols in the **keyword** package, read as a series of symbol names separated by commas or spaces, and printed as a series of symbol names separated by spaces. Symbol names are read and printed without package prefixes (that is, not preceded by colons).

:font-list The value is a list of fonts, read as a series of font names separated by commas or spaces, and printed as a series of font names separated by commas and spaces. Font names are read and printed without package prefixes (that is, not preceded by **fonts:**).

2.3 Improvements to Lisp in Release 5.0

2.3.1 Previously undocumented special form: destructuring-bind

destructuring-bind *variable-pattern data body ...* *Special Form*

destructuring-bind binds variables to values, using **defmacro**'s destructuring facilities, and evaluates the body forms in the context of those bindings.

First *data* is evaluated. If *variable-pattern* is a symbol, it is bound to the result of evaluating *data*. If *variable-pattern* is a t the result of evaluating *data* should be a tree of the same shape. The trees are disassembled, and each variable that is a component of *variable-pattern* is bound to the value that is the corresponding element of the tree that results from evaluating *data*. If not enough values are supplied, the remaining variables are bound to **nil**. If too many values are supplied, the excess values are ignored. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned.

As with the pattern in a **defmacro** form, *variable-pattern* actually resembles the **lambda**-list of a function; it can have **&**-keywords. See the section "Advanced Features of **defmacro**".

Example:

```
(destructuring-bind (a (b) &optional (c 'd))
                    '((x y) (z))
  (values a b c))
```

returns **(x y)**, **z**, and **d**.

2.3.2 Invisible blocks in progs and dos

You can now make a block invisible to **returns** in any kind of **prog** or **do** form by including immediately within it the form **(declare (invisible-block t))**. This feature is intended for macro expansions, not for user code.

See the special form **do-named**.

2.3.3 Previously undocumented function: clear-resource

clear-resource causes **allocate-resource** to ignore existing objects in the resource and make new objects when called.

clear-resource *resource-name* *Function*

Forget all of the objects being remembered by the resource specified by *resource-name*. Future calls to **allocate-resource** create new objects. This function is useful if something about the resource has been changed incompatibly, such that the old objects are no longer usable. If an object of the resource is in use when **clear-resource** is called, an error is signalled when that object is deallocated.

2.3.4 Multidimensional Arrays on the 3600 Remember Actual Dimensions

Arrays of more than one dimension on the 3600 now store their dimensions. This allows multidimensional indirect arrays to have *conformal indirection*. A new **make-array** option, **:displaced-conformally**, has been added. The window system now uses conformal indirect arrays for screen arrays.

Multidimensional arrays on the 3600 remember their actual dimensions, separately from the magic numbers by which to multiply the subscripts before adding them together to get the index into the array.

As a result of this, multidimensional indirect arrays on the 3600 can have *conformal indirection*. If A is indirected to B, and they do not have the same width, then normally the part of B that is shared with A does not have the same shape as A. If conformal indirection is used, then it does have the same shape and there are gaps between the rows of A. For example:

```
(setq b (make-array '(10. 20.)))
(setq a (make-array '(3 5) ':displaced-to b ':displaced-index-offset 12.))
```

Now:

```
(aref a 1 0) = (aref b 3 1) and (aref a 1 1) = (aref b 6 1).
```

In contrast:

```
(setq a (make-array '(3 5) ':displaced-to b
                           ':displaced-index-offset 12.
                           ':displaced-conformally t))
```

(aref a 1 0) = (aref b 3 1) still, but (aref a 1 1) = (aref b 3 2). Each row of A corresponds to part of a row of B, always starting at the same column (2).

A graphic illustration:

```
(setq a (make-array '(6 20.))
      b (make-array '(3 5) ':displaced-to a ':displaced-index-offset 22.)
      c (make-array '(3 5) ':displaced-to a ':displaced-index-offset 22.
                           ':displaced-conformally t))
```

```
 Normal case                    Conformal case
 0                   19         0                   19
 +--------------------+         +--------------------+
0|aaaaaaaaaaaaaaaaaaaa|        0|aaaaaaaaaaaaaaaaaaaa|
 |aaBBBBBBBBBBBBBBBaaa|         |aaCCCCCaaaaaaaaaaaaa|
 |aaaaaaaaaaaaaaaaaaaa|         |aaCCCCCaaaaaaaaaaaaa|
 |aaaaaaaaaaaaaaaaaaaa|         |aaCCCCCaaaaaaaaaaaaa|
 |aaaaaaaaaaaaaaaaaaaa|         |aaaaaaaaaaaaaaaaaaaa|
5|aaaaaaaaaaaaaaaaaaaa|        5|aaaaaaaaaaaaaaaaaaaa|
 +--------------------+         +--------------------+
```

Arrays are stored in column-major order, so the units in which the index-offset is measured should be read first from left to right and then from top to bottom.

The meaning of **adjust-array-size** for conformal indirect arrays is undefined.

The window system now uses conformal indirect arrays for its screen arrays. This means that on the 3600 the bit-array in which a window saves its bits when it is not visible no longer has to be the full width of the screen; now it is just the width of the window, rounded up to the next multiple of 32 bits. On the LM-2, screen arrays and bit-save arrays are still the full width of the screen.

Some associated internal changes for both the 3600 and the LM-2:

- The **locations-per-line** instance variable changes when expose and deexpose happen
- The arguments to the **:create-screen-array** message to screens have changed
- The **:adjust-screen-array** message to sheets replaces the **:redirect-screen-array** message to screens
- New message to screens: **:inferior-screen-array-adjusted**

Screen arrays no longer use multilevel indirection; the screen array of a nonscreen sheet always indirects either to a bit-save array or to the screen array of its screen. The screen array of a screen is always a displaced array to the hardware screen buffer.

2.3.5 New options for make-plane: :initial-dimensions, :initial-origins

make-plane has two new keyword options: **:initial-dimensions** and **:initial-origins**.

make-plane *rank* &rest *options* *Function*

Creates and returns a plane. *rank* is the number of dimensions. *options* is a list of alternating keyword symbols and values. The allowed keywords are:

:type The array type symbol (for example, **art-1b**) specifying the type of the array out of which the plane is made.

:default-value
The default component value.

:extension
The amount by which to extend the plane. See the section "Planes".

:initial-dimensions
A list of dimensions for the initial creation of the plane. You might want to use this option to create a plane whose first dimension is a multiple of 32, so you can use **bitblt** on it. Default: the result returned by **(make-list** *rank* **':initial-value 1)**.

:initial-origins
A list of origins for the initial creation of the plane. Default: the result returned by **(make-list** *rank* **':initial-value 0)**.

Example:

```
(make-plane 2 ':type 'art-4b ':default-value 3)
```

creates a two-dimensional plane of type **art-4b**, with default value **3**.

2.3.6 New optional arguments to string-upcase and string-downcase

string-upcase and **string-downcase** now take three optional arguments: a starting index and limit for changing case in substrings, and an indicator of whether the string should be copied or modified directly.

string-upcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*

If *copy-p* is not **nil**, returns a copy of *string*, with lowercase alphabetic characters replaced by the corresponding uppercase characters. If *copy-p* is **nil**, uppercases characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin uppercasing characters. If *to* is supplied, it is used in place of **(array-active-length** *string***)** as the index one greater than the last character to be uppercased.

string-downcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*

If *copy-p* is not **nil**, returns a copy of *string*, with uppercase alphabetic

characters replaced by the corresponding lowercase characters. If *copy-p* is **nil**, lowercases characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin lowercasing characters. If *to* is supplied, it is used in place of **(array-active-length** *string***)** as the index one greater than the last character to be lowercased.

2.3.7 Previously undocumented function: string-compare

string-compare compares two strings using dictionary order and returns a number that depends on whether or not the strings are equal.

string-compare *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
Compares the characters of *string1* starting at *idx1* and ending just below *lim1* with the characters of *string2* starting at *idx2* and ending just below *lim2*. The comparison is in alphabetical order. *lim1* and *lim2* default to the lengths of the strings. **string-compare** returns:

- a positive number if *string1* > *string2*
- zero if *string1* = *string2*
- a negative number if *string1* < *string2*

If the strings are not equal, the absolute value of the number returned is one more than the index (in *string1*) at which the difference occurred.

string-compare uses the same rules as **string** in coercing *string1* and *string2* into strings.

2.3.8 3600 select-methods handle :operation-handled-p and :send-if-handles

Select-methods on the 3600 now handle the **:operation-handled-p** and **:send-if-handles** messages. Methods for these messages are generated automatically when the **defselect** form is evaluated.

See the special form **defselect**. See the message **:operation-handled-p**. See the message **:send-if-handles**.

2.3.9 Compiler Performs Style Checking on All Forms

The compiler no longer fails to perform style checking on the results of macro expansions and optimizations.

The compiler performs style checking on all forms. Style checking is implemented by the **compiler:style-checker** property on a symbol; the value of the property is called on all forms whose **car** is that symbol, except those immediately enclosed in **inhibit-style-warnings**.

2.3.10 sys:dump-forms-to-file always puts package attribute into binary file

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 **user** package. 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.

2.3.11 Previously undocumented macro: swapf

swapf exchanges the value of one generalized variable with that of another.

swapf *a b* *Macro*

Exchanges the value of one generalized variable with that of another. *a* and *b* are access-forms suitable for **setf**. The returned value is not defined. All the caveats that apply to **incf** apply to **swapf** as well: Forms within *a* and *b* may be evaluated more than once. Examples:

```
(swapf a b)
   ==> (setf a (prog1 b (setf b a)))
   ==> (setq a (prog1 b (setq b a)))

(swapf (car (foo)) (car (bar)))
   ==> (setf (car (foo)) (prog1 (car (bar)) (setf (car (bar)) (car (foo)))))
   ==> (rplaca (foo) (prog1 (car (bar)) (rplaca (bar) (car (foo)))))
```

Note that in the second example the functions **foo** and **bar** are called twice.

2.3.12 Compiler now warns about implicit progns in loops

An expression in a **loop** clause can be a single form or a series of forms that constitute an implicit **progn**. When an implicit **progn** appears, it is often an error caused by omitting a **do**. Because this error is so frequent, the intentional use of implicit **progns** in most clauses is considered obsolete and dangerous. If you intend to use a **progn**, use an explicit one. To help you detect implicit **progns**, the compiler now issues a warning whenever it encounters one in a context where it is likely to be a mistake.

Consider the following example:

```
(defun frob (list)
  (loop for thing in list
        collect (string thing)
        (format t "~&~A" thing)))
```

The returned value is a list of **nils**, one for each element of *list*. The author most likely intended to return a list of the elements of *list*, coerced to strings, but omitted a **do** before the **(format ...)** form. When this definition is compiled, the compiler issues this warning:

```
For function FROB:
  The use of multiple forms with an implicit PROGN in this context is
  considered obsolete, but is still supported for the time being.
  If you did not intend to use multiple forms here, you probably
  omitted a DO.  If the use of multiple forms was intentional, put a
  PROGN in your code.
  The offending clause -- LIST (STRING THING) (FORMAT T "~&~A" THING)
```

2.3.13 Some Methods Can Use Combination Type as Method Type

Methods used with combination types that formerly allowed only untyped methods can now use the combination type as the method type.

Methods used with **:progn**, **:append**, **:nconc**, **:and**, **:or**, **:list**, **:inverse-list**, and **:pass-on** combination types can use the combination type as the method type. This is useful in documenting how the method is used.

In the following example, **(:method foo :or :find-frabjous-frob)** could have been defined as **(:method foo :find-frabjous-frob)**. The only difference is one of style: Using **:or** as the method type makes it clear that the methods are combined using **:or** combination.

```
(defflavor foo (frobl) (bar)
  (:method-combination (:or :base-flavor-last :find-frabjous-frob)))

(defmethod (foo :or :find-frabjous-frob) (type)
  (dolist (frob frobl)
    (when (send frob ':frabjous-p type)
      (return frob))))
```

2.3.14 Previously undocumented reader macro: #| and |#

#| begins a comment for the Lisp reader, and |# ends one.

#| 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.

2.3.15 New function to be called by reader macros: si:read-recursive

Reader macros that call a read function should now call **si:read-recursive** instead of **read**.

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 **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 may be called only from inside a **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))
```

2.3.16 New optional arguments to read-from-string

read-from-string now takes two new optional arguments for specifying a substring. These are indices for the first character to be read and the character after the last one to be read.

read-from-string *string* &optional (*eof-option* **'si:no-eof-option**) (*start* **0**) *end* *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 will 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 **(array-active-length** *string***)** as the integer that is one greater than the index of the last character to be read.

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
```

2.3.17 Changes to prompt-and-read

Changes have been made to many **prompt-and-read** options.

- New options: **:character**, **:date**, **:time-interval-or-never**, **:expression-or-end**, **:pathname-or-nil**, **:string-list**, **:delimited-string**, **:delimited-string-or-nil**, **:host**, **:host-list**, **:pathname-host**, **:keyword-list**, **:font-list**.

- When options accept a single line of text as input, the line can be terminated by RETURN, LINE, or END. These are activation characters, so they can be typed anywhere in the line.

- The **:number** option accepts keyword-value pairs that determine the base in which the number is read and whether or not **nil** can be returned.

- The **:pathname** option accepts a **:default-version** keyword. If not specified, the default version is **:newest**.

prompt-and-read *type* &optional *format-string* &rest *format-args* *Function*

prompt-and-read prompts the user, with *format-string* and its arguments as the prompt. It uses **format** to **query-io** to produce the prompt; it reads from the **query-io** stream, calling the reading function associated with the *type* keyword. If *format-string* is not specified, it generates a prompt appropriate to *type*. The *type* argument can be a list in which the first element is the type keyword and the rest are keyword/value pairs to serve as arguments to the reading function. **prompt-and-read** returns whatever the reading function returns.

This is an appropriate function to call for collecting input from the user. Its main advantages are that it does type checking on the input the user types and that it takes care of redisplaying the prompt at appropriate times (for example, after the screen has been refreshed or after a notification arrives).

```
(prompt-and-read ':number "Please enter a number: ") =>
Please enter a number: 4
4
(prompt-and-read ':string "Please enter a string: ") =>
Please enter a string: 4
"4"
```

It expects to collect input of type *type*, where *type* is a keyword. It handles the following types of input:

Option	*Action*
:eval-form	Reads a Lisp form. Evaluates it and returns the first value. Asks for confirmation of nonconstant values. The Debugger uses this to prompt for a form to evaluate.
:eval-form-or-end	Reads a Lisp form or just END. Evaluates it and returns the first value for a form. Returns two values, **nil** and **:end**, for END. Asks for confirmation of nonconstant values. The Debugger uses this to prompt for a form to evaluate.
:expression	Reads a Lisp expression. (It returns the expression without evaluating it.)
:expression-or-end	Reads a Lisp expression or just END. It returns the expression without evaluating it. If the user just presses END, it returns two values, **nil** and **:end**.
:character	Reads and returns a character. The returned value is a character code (an integer).

(**:number :input-radix** *radix* **:or-nil** *or-nil-value*)
Reads and returns a number, terminated by RETURN, LINE, or END. If **:input-radix** is specified, the number is read in radix *radix*; otherwise, it is read in the current **ibase**. If **:or-nil** is specified with a value of **t**, it returns **nil** if the user just presses RETURN, LINE, or END.

:number-or-nil The same as **(:number :or-nil t)**.

:decimal-number The same as **(:number :input-radix 10.)**.

:decimal-number-or-nil
The same as **(:number :input-radix 10. :or-nil t)**.

(**:date :past-p** *past-p-value* **:never-p** *never-p-value*)
Reads and returns a date, terminated by RETURN, LINE, or END. The returned date is a universal-time integer of the form returned by **time:parse-universal-time**. If **:past-p** is specified with a value of **t**, an ambiguous date is interpreted as being in the past; otherwise, it is interpreted as being in the future. If **:never-p** is specified with a value of **t**, it returns **nil** if the user types "never".

:past-date The same as **(:date :past-p t)**.

:date-or-never The same as **(:date :never-p t)**.

:past-date-or-never
The same as **(:date :past-p t :never-p t)**.

:time-interval-or-never
Reads a time interval, terminated by RETURN, LINE, or END. The interval must be either "never" or alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years. It returns **nil** if the user types "never". Otherwise, it returns an integer representing the number of seconds in the time interval.

Example:

```
(prompt-and-read ':time-interval-or-never)
Enter a time interval, or "never": 1 day 2 hrs 13 min =>
94380.
```

(**:pathname :default** *defaults* **:default-version** *version*)
Reads a pathname, terminated by RETURN, LINE, or END, merging it with defaults. If **:default** is not specified, the defaults are the value of **fs:*default-pathname-defaults***. If **:default** is specified, its value should be suitable as the second argument to **fs:merge-pathnames**: a pathname, a pathname string, or an alist of hosts and pathnames of the sort that is the value of

fs:*default-pathname-defaults*. If **:default-version** is not specified, the default version is **:newest**. If **:default-version** is specified, its value should be an integer or keyword suitable as the third argument to **fs:merge-pathnames**. It returns the merged pathname.

Example:

```
(prompt-and-read '(:pathname :default ,my-defaults-alist)
                 "Enter a name (default is ~A) "
                 (fs:default-pathname my-defaults-alist))
```

(:pathname-or-nil :default *defaults* **:default-version** *version*)
Like **:pathname**, except that the returned value depends on what the user types:

User types	*Returned value*
A string	Merged pathname
RETURN or LINE only	Default pathname
END only	**nil**

:pathname-list Reads a series of pathnames, separated by commas and terminated by RETURN, LINE, or END. Merges the pathnames with the defaults in **fs:*default-pathname-defaults*** and with a default version of **:newest**. It returns a list of the merged pathnames.

:string Reads a string terminated by RETURN, LINE, or END. It returns the empty string when the string is empty.

:string-or-nil Reads a string terminated by RETURN, LINE, or END. It trims any leading or trailing white space. It returns **nil** when the string is empty.

:string-trim Reads a string terminated by RETURN, LINE, or END. It trims any leading or trailing white space. It returns the empty string when the string is empty.

:string-list Reads a series of strings separated by commas and terminated by RETURN, LINE, or END. It returns a list of the strings.

(:delimited-string :delimiter (*delimiter-1 delimiter-2 ...*) **:buffer-size** *size*)
Reads characters until one of the delimiters is typed. The delimiters are set up as activation characters. If no delimiters are specified, the default is **#\end**. If **:buffer-size** is specified, an initial buffer of size *size* characters is allocated; otherwise, the initial size is 100. characters. It returns the empty string when the string is empty.

(:delimited-string-or-nil :delimiter (*delimiter-1 delimiter-2...*) **:buffer-size** *size*)
Like **:delimited-string**, except that it returns **nil** when the string is empty.

(:host :default *default*)
Reads the name of a network host, terminated by RETURN, LINE, or END. If **:default** is specified, it should be the name of a host as a symbol or string. If **:default** is specified and the user just presses RETURN, LINE, or END, it returns the host specified by **:default**. Otherwise, it returns the host whose name the user types.

(:host-list :chaos-only *chaos-only*)
Reads a series of names of network hosts, separated by spaces or commas, and terminated by RETURN, LINE, or END. If **:chaos-only** is not **nil**, each host must have a Chaos address. It returns a list of the hosts whose names the user types.

(:pathname-host :default *default*)
Reads the name of a pathname host, terminated by RETURN, LINE, or END. The name can be "local", "sys", or the name of another logical host as well as the name of a physical host. If **:default** is specified, it should be the name of a pathname host as a string. If **:default** is specified and the user just presses RETURN, LINE, or END, it returns the host specified by **:default**. Otherwise, it returns the host whose name the user types.

:keyword-list Reads a series of names of symbols to be interned in the **keyword** package, separated by spaces or commas, and terminated by RETURN, LINE, or END. The symbol names should not have package prefixes (that is, they should not be preceded by colons). It returns a list of keyword symbols whose names the user types.

:font-list Reads a series of names of fonts, separated by spaces or commas, and terminated by RETURN, LINE, or END. The font names should not have package prefixes (that is, they should not be preceded by **fonts:**), and they must be names of known fonts. It returns a list of fonts whose names the user types.

Streams are permitted to have a handler for **:prompt-and-read** messages. The **prompt-and-read** function first determines whether the **query-io** stream handles the **:prompt-and-read** message. If so, it sends a **:prompt-and-read** message with its own arguments on to the stream. The stream returns several values. The first value the stream returns says

whether or not it wants to handle the interaction with the user itself. It returns **nil** to indicate that it declines to handle the message, in which case the **prompt-and-read** function continues its normal action of prompting the user. When the first value is not **nil**, the **prompt-and-read** function returns the rest of the values to its caller.

2.3.18 Previously Undocumented Feature: Coroutine Streams

Coroutine streams are a means of using output from one function as input to another, and vice versa. Functions are provided that construct two coroutine streams, each associated with a separate stack group but sharing a common I/O buffer.

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 out of 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 **si: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 **si: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 **si: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. The Lisp Machine does 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.

si: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 **si:make-coroutine-bidirectional-stream** is called. **si: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 **si: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 **si: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.

2.3.19 format ~\ directives can have package prefixes

Format directives enclosed in backslashes can now have package prefixes. If they have none, they refer to symbols in the **format** package.

See the special form **format:defformat**.

2.3.20 Wildcard Directory Mapping Available

True wildcard mapping of directories is now supported. This facility is used by functions that copy and rename files. It allows you to copy or rename entire subtrees.

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".

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 may 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". 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".

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 may 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

2.3.21 Previously undocumented function: describe-system

describe-system is a useful function for finding information about a system.

describe-system *system-name* &key *(show-files* **t***)* *Function*
(show-transformations **t***)*

Displays useful information about the system named *system-name*. This includes the name of the system source file, the system package default if any, and component systems. For a patchable system, **describe-system** displays the system version and status, a typical patch file name, the sites maintaining the system, and, if the user wants, a listing of patches. If **:show-files** is **t**, it displays the history of the files in the system. Other possible values are **nil** (do not show file history) and **:ask** (ask the user). If **:show-transformations** is **t**, it displays the transformations required to make the system. Other possible values are **nil** (do not display transformations) and **:ask** (ask the user).

2.3.22 Improvements to make-system: error-restart, selective transformations

make-system now has an **error-restart** that reinvokes the **make-system** itself. It probes for files that have changed since the **make-system** was started.

make-system also has a new possible answer, "S" (selective), to its request for confirmation of the list of transformations to be performed. If the user answers "S", **make-system** proceeds as if the **:selective** option had been specified, asking for confirmation of each individual transformation.

See the function **make-system**.

2.3.23 Second argument to si:install-microcode now optional

The second argument to **si:install-microcode**, *to-file-or-version*, is now optional. This argument defaults to a file on FEP:> and rarely needs to be supplied.

si:install-microcode *from-file-or-version* &optional *to-file-or-version* *Function*
(3600 only) Installs microcode from a system file into a file in the FEP file system.

from-file-or-version is a microcode version number (in decimal). The file resides in the logical directory sys:l-ucode;.

to-file-or-version rarely needs to be supplied. It defaults to a file on FEP:> (the root directory of the boot disk) whose name is based on the microcode name and version. If supplied, *to-file-or-version* is either a pathname (string) of a file on FEP:>, or an integer *n*, which stands for the file TMC5-MIC.MIC.n on FEP:>.

2.3.24 Change in argument to process-wait-with-timeout

The *interval* argument to **process-wait-with-timeout** can now be **nil**. If so, **process-wait-with-timeout** waits indefinitely for the application of *function* to *arguments* to return something other than **nil**.

process-wait-with-timeout *whostate time function* &rest *args* *Function*

This is a primitive for waiting. It applies *function* to *arguments* until the function returns something other than **nil** or until the interval times out. *interval* is a time in 60ths of a second. When the process times out, **process-wait-with-timeout** returns **nil**. When the function returns something other than **nil** within the interval, **process-wait-with-timeout** returns **t**.

If *interval* is **nil**, **process-wait-with-timeout** waits indefinitely for the application of *function* to *arguments* to return something other than **nil**. This behavior is the same as that of **process-wait**.

2.3.25 New option for si:sb-on: :mouse (3600 only)

The **:mouse** option for **si:sb-on** on the 3600 causes sequence breaks when the mouse moves. This option is on by default.

si:sb-on &optional *when* *Function*

si:sb-on controls what events cause a sequence break, that is, when rescheduling occurs. The following keywords are names of events that can cause a sequence break.

:clock This event happens periodically based on a clock. The default period is one second. For the 3600, the period is the value of the variable **si:sequence-break-interval**, an integer representing the number of microseconds in the period (default 1000000.). For the LM-2, see the meter **sys:%tv-clock-rate**. This event is enabled by default.

:disk (3600 only) A sequence break happens whenever the disk hardware/firmware decides to wake up the wired disk system. This might occur with every disk I/O operation or after several have been completed. This event is always enabled; you cannot turn it off. However, these sequence breaks do not cause rescheduling.

:mouse (3600 only) Happens when the mouse moves. Sixty times per second it tests the variable **tv:mouse-wakeup**, which is set by the FEP. Causes a sequence break if the value is not **nil**. This event is enabled by default.

:unibus (LM-2 only) Happens when a character is received from the keyboard. Actually, a sequence break happens

whenever input is received from any UNIBUS channel that has a flag bit set. This event is disabled by default.

:keyboard (LM-2 only) Same as **:unibus**.

:chaos (LM-2 only) Happens when a packet is received from the Chaosnet, or transmission of a packet to the Chaosnet is completed. This event is disabled by default.

Since the keyboard and Chaosnet are heavily buffered, there is no particular advantage to enabling the **:keyboard** and **:chaos** events, unless the **:clock** event is disabled.

With no argument, **si:sb-on** returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, **nil** (which disables sequence breaks entirely since it is the empty list), or a number that is the internal mask, not documented here.

2.3.26 New format for trace output

The default format for **trace** output has changed.

Example of the old style:

```
(1 ENTER FACT (4.))
  (2 ENTER FACT (3.))
    (3 ENTER FACT (2.))
      (4 ENTER FACT (1.))
        (5 ENTER FACT (0.))
        (5 EXIT FACT 1.)
      (4 EXIT FACT 1.)
    (3 EXIT FACT 2.)
  (2 EXIT FACT 6.)
(1 EXIT FACT 24.)
```

Example of the new style:

```
1 Enter FACT 4.
| 2 Enter FACT 3.
|   3 Enter FACT 2.
|   | 4 Enter FACT 1.
|   |   5 Enter FACT 0.
|   |   5 Exit FACT 1.
|   | 4 Exit FACT 1.
|   3 Exit FACT 2.
| 2 Exit FACT 6.
1 Exit FACT 24.
```

You can use the variables **si:*trace-columns-per-level***, **si:*trace-bar-p***, **si:*trace-bar-rate***, and **si:*trace-old-style*** to control the format of **trace** output.

si:*trace-columns-per-level* *Variable*

For **trace** output, controls the number of columns of indentation that are added for each level of function call. The value must be an integer. The default is 2.

si:*trace-bar-p* *Variable*

For **trace** output, controls whether columns of vertical bars are printed. If the value is not **nil**, they are printed; otherwise, spaces are printed instead of the vertical bars. The default is **t** (print the bars).

si:*trace-bar-rate* *Variable*

When **si:*trace-bar-p*** is not **nil**, columns of vertical bars are printed in **trace** output for every *n* levels of function call, where *n* is the value. The value must be an integer. The default is 2.

si:*trace-old-style* *Variable*

If not **nil**, the old, Maclisp-compatible form of printing **trace** output is used. The default is **nil** (use the new style).

2.3.27 Recursion in Bound and Default Handlers Eliminated

Condition handlers bound by **condition-bind**, **condition-bind-default**, and related special forms no longer cause infinite recursion when they signal the same condition they are handling.

While a bound or default handler is executing, that handler and all handlers inside it are removed from the list of bound or default handlers. This is to prevent infinite recursion when a handler signals the same condition that it is handling, as in the following simplistic example:

```
(condition-bind ((error '(lambda (x) (ferror "foo"))))
  (ferror "foo"))
```

If you want recursion, the handler should bind its own condition.

2.3.28 :proceed methods can now return nil

It is no longer an error for a **:proceed** method to return **nil** as its first value.

A **:proceed** method can return a first value of **nil** if it declines to proceed from the condition. If a **nil** returned by a **:proceed** method becomes the return value for a **condition-bind** handler, this signifies that the handler has declined to handle the condition, and the condition continues to be signalled. When the **:proceed** message was sent by the Debugger, the Debugger prints a message saying that the condition was not proceeded, and it returns to its command level. This might be used by an interactive **:proceed** method that gives the user the opportunity either to proceed or to abort; if the user aborts, the method returns **nil**. Returning **nil** from a **:proceed** method should not be used as a substitute for detecting earlier (such as

when the condition object is created) that the proceed type is inappropriate for that condition.

2.3.29 New clause for condition-call: :no-error

condition-call and **condition-call-if**, like **condition-case**, can now take a **:no-error** clause as the final clause.

As a special case, *predicate-m* (the last one) can be the special symbol **:no-error**. If *form* is evaluated and no error is signalled during the evaluation, **condition-case** executes the **:no-error** clause instead of returning the values returned by *form*. The variables *vars* are bound to the values produced by *form*, in the style of **multiple-value-bind**, so that they can be accessed by the body of the **:no-error** case. Any extra variables are bound to **nil**.

2.3.30 New message to arithmetic errors: :operands

All arithmetic errors (built on **sys:arithmetic-error**) now handle the **:operands** message. On the 3600, this returns a list of the operands in the operation that caused the error. On the LM-2, this message nearly always returns **nil**.

See the flavor **sys:arithmetic-error**.

2.3.31 Change in Debugger special command for fs:directory-not-found

The condition flavor **fs:directory-not-found** now has two Debugger special commands: **:create-directory**, to create only the lowest level of directory, and **:create-directories-recursively**, to create any missing superiors as well.

fs:directory-not-found *Flavor*

The directory of the file was not found or does not exist. This means that the containing directory was not found. If you are trying to open a directory, and the actual directory you are trying to open is not found, **fs:file-not-found** is signalled. This flavor is built on **fs:file-lookup-error**.

This flavor has two Debugger special commands: **:create-directory**, to create only the lowest level of directory, and **:create-directories-recursively**, to create any missing superiors as well.

2.3.32 New optional argument to gc-immediately

gc-immediately now takes an optional argument, **nil** by default. If it is not **nil**, **gc-immediately** does garbage collection without querying, regardless of how much space is left.

gc-immediately &optional *no-query* *Function*

gc-immediately does nonincremental garbage collection, taking less space and less total time than an incremental gc, but running continuously in the

process calling it, until the garbage collection is complete. The main advantage of this compared to incremental gc is that it requires less free space and hence can succeed where an incremental gc would fail because virtual memory was too full.

If *no-query* is not **nil**, **gc-immediately** commences garbage collection without asking any questions, regardless of how much space is available.

You should call this rather than **si:full-gc** (unless you are compressing a band). The difference is that **gc-immediately** does not lock out other processes, does not run various **full-gc** initializations, and does not affect the static areas.

Suppose garbage collection has already started, that the flip has occurred but not all good data have been copied out of old space. **gc-immediately** then copies the rest of the good data but does not flip again.

2.3.33 New optional arguments to print-notifications

print-notifications now takes optional arguments that allow you to print only part of the notification history.

print-notifications &optional (*from* **0**) *Function*
(*to* **(1- (length tv:notification-history))**)

Reprints any notifications that have been received. The difference between notifications and sends is that sends come from other users, while notifications are asynchronous messages from the Lisp Machine system itself. If *from* or *to* is specified, prints only part of the notifications list.

Example: `(print-notifications 0 4)` prints the five most recent notifications.

2.3.34 :draw-filled-in-circle uses same algorithm as :draw-circle

Previously, the **:draw-filled-in-circle** method of **tv:graphics-mixin** used a different algorithm from that of the **:draw-circle** method. This algorithm was slower and sometimes resulted in inaccurate displays when the two methods were used together. **:draw-filled-in-circle** now uses the same algorithm as **:draw-circle**.

:draw-circle *center-x center-y radius* &optional *alu* of **tv:graphics-mixin** *Method*

Draw the outline of a circle specified by its center and radius.

:draw-filled-in-circle *center-x center-y radius* &optional *alu* of **tv:graphics-mixin** *Method*

Draw a filled-in circle specified by its center and radius.

2.3.35 Previously undocumented variables: sys:mouse-x-scale-array and sys:mouse-y-scale-array (LM-2 only)

sys:mouse-x-scale-array and **sys:mouse-y-scale-array** are variables on the LM-2 whose values are arrays used in mouse scaling. These arrays determine the relation between motion of the mouse on the table and motion of the mouse cursor on the screen. That relation can vary with the speed of the mouse. You can use these variables to speed up or slow down the motion of the mouse cursor caused by corresponding motion of the mouse.

sys:mouse-x-scale-array *Variable*

(LM-2 only) The value of this variable is an array that, along with the array that is the value of **sys:mouse-y-scale-array**, can be used to control mouse scaling. These arrays determine the relation between the rates of motion of the mouse on the table and the mouse cursor on the screen. This relation can be nonlinear and can vary with the speed of the mouse. For example, fast mouse motion can move the cursor a distance that is proportionally greater than slow mouse motion.

Scaling is computed as follows. The even-numbered elements of **sys:mouse-x-scale-array** are compared with the value of **sys:mouse-x-speed**, and the even-numbered elements of **sys:mouse-y-scale-array** are compared with the value of **sys:mouse-y-speed**. **sys:mouse-x-speed** and **sys:mouse-y-speed** are the x- and y-components of the mouse speed on the table, typically in units of hundredths of an inch per second.

For each array, the first even array element that is greater than the mouse speed causes its corresponding odd-numbered array element to be multiplied by the mouse motion on the table and then divided by 1024 (decimal). The result is the mouse motion on the screen. Appropriate care is taken to save the fractions for the next computation.

The default array setup code is as follows:

```
;; Set the X scale to 2/3 and the Y scale to 3/5.
;; Disable speed-dependent scaling.
(aset #o37777777 sys:mouse-x-scale-array 0)
(aset (// (lsh 2 10.) 3) sys:mouse-x-scale-array 1)
(aset #o37777777 sys:mouse-y-scale-array 0)
(aset (// (lsh 3 10.) 5) sys:mouse-y-scale-array 1)
```

The following code provides for simple scaling of motion for the Hawley mouse. The microcode knows specially about each array. You may store into each array, but you may not replace it with a new array or use **adjust-array-size** on it.

```
;;; Aids to trying speed-dependent scaling
;;; Specs are scale-factor speed-break
;;; No attempt to treat X and Y differently
;;; Args of (1 80. 2) seem to be about right for the Hawley mouse
(defun mouse-speed-hack (&rest specs)
  (loop for (scale speed) on specs by 'cddr
        for i from 0 by 2
        do (aset (or speed #o37777777) sys:mouse-x-scale-array i)
        (aset (or speed #o37777777) sys:mouse-y-scale-array i)
        (aset (// (fix (* 2 scale 1024.)) 3)
              sys:mouse-x-scale-array (1+ i))
        (aset (// (fix (* 3 scale 1024.)) 5)
              sys:mouse-y-scale-array (1+ i))))

(defun hawley-mouse-hack ()
  (mouse-speed-hack 1 80. 2))
```

The corresponding variables **tv:mouse-x-scale-array** and **tv:mouse-y-scale-array** exist on the 3600, but in this release they have no effect.

sys:mouse-y-scale-array *Variable*

(LM-2 only) The value of this variable is an array that, along with the array that is the value of **sys:mouse-x-scale-array**, can be used to control mouse scaling. See the variable **sys:mouse-x-scale-array**.

2.3.36 New optional argument to tv:mouse-wait

tv:mouse-wait now takes another optional argument, a string to be displayed in the status line while waiting for the status of the mouse to change.

tv:mouse-wait &optional (*old-mouse-x* **tv:mouse-x**) *Function*
(*old-mouse-y* **tv:mouse-y**)
(*old-mouse-buttons* **tv:mouse-last-buttons**)
(*whostate* **"Mouse"**)

This function waits for any of the variables **tv:mouse-x**, **tv:mouse-y**, or **tv:mouse-last-buttons** to become different from the values passed as arguments. While waiting, *whostate* is displayed in the status line. To avoid timing errors, your program should examine the values of the variables, use them, and then pass in the values that it examined as arguments to **tv:mouse-wait** when it is done using the values and wants to wait for them to change again. It is important to do things in this order, or else you might fail to wake up if one of the variables changed while you were using the old values and before you called **tv:mouse-wait**.

2.3.37 New flavors: tv:truncatable-lines-mixin, tv:truncating-lines-mixin

tv:truncatable-lines-mixin causes text to be truncated at the right edge of the window, but only if the window's "truncate line out" flag is set. **tv:truncating-lines-mixin** initializes this flag to on so that truncation actually happens. These flavors replace the obsolete flavor **tv:line-truncating-mixin**, which, despite its name, did not initialize the "truncate line out" flag to on. You can use the new messages **:truncate-line-out** and **:set-truncate-line-out** to read and set this flag. The flavor **tv:truncating-window** is now built on **tv:truncating-lines-mixin**.

tv:truncatable-lines-mixin *Flavor*

If you mix in this flavor and the window's *truncate line out* flag is on, typeout does not wrap around when lines are too long. That is, when the cursor is near the right-hand edge of the window and an attempt is made to type out a character, the character is not typed out; text is truncated at the edge of the window. When the truncate line out flag is turned off, this flavor has no effect.

tv:truncating-lines-mixin *Flavor*

When this flavor is mixed in, lines of output that are too long to fit inside the window do not wrap around but are truncated at the edge of the window. This flavor is built on **tv:truncatable-lines-mixin**. It initializes the window's truncate line out flag to be on.

tv:truncating-window *Flavor*

This flavor is built on **tv:window** with **tv:truncating-lines-mixin** mixed in. If you instantiate a window of this flavor, it will be like regular windows of flavor **tv:window** except that lines will be truncated instead of wrapping around.

:truncate-line-out of **tv:sheet** *Method*

Returns **t** if the window's truncate line out flag is set, or **nil** if it is not.

:set-truncate-line-out *new-value* of **tv:sheet** *Method*

Sets the value of the window's truncate line out flag. If *new-value* is **t** the flag is turned on; if **nil**, it is turned off.

2.3.38 New variable: tv:*mouse-modifying-keystates*

In previous releases you could use the variables **tv:mouse-double-click-time** and **tv:*mouse-incrementing-keystates*** to replace double clicks with shifted clicks.

Now mouse characters — characters with the **%%kbd-mouse** bit set to 1 — can be modified with the modifier keys CONTROL, META, SUPER, and HYPER, just as keyboard characters can. Which of these keys modify mouse characters depends on the value of the variable **tv:*mouse-modifying-keystates***.

You can use **login-forms** in an init file to set the variables **tv:mouse-double-click-time**, **tv:*mouse-incrementing-keystates***, and **tv:*mouse-modifying-keystates*** and customize the behavior of the mouse.

tv:mouse-double-click-time *Variable*

The maximum period of time (in microseconds) between mouse clicks for which the clicks are interpreted as a double click instead of two single clicks. Default: **200000** (decimal). If you set this to **nil**, disabling double clicking entirely, mouse response time improves slightly.

tv:*mouse-incrementing-keystates* *Variable*

A list of names of keys, acceptable to **tv:key-state**. If one or more of these keys are pressed, single mouse clicks are interpreted as double clicks. Default: **(:shift)**.

tv:*mouse-modifying-keystates* *Variable*

A list of names of keys, acceptable to **tv:key-state**. If one or more of these keys are pressed, sets the corresponding modifier bits in the mouse character. Default: **(:control :meta :super :hyper)**. If a key appears as an element of both this list and the list that is the value of **tv:*mouse-incrementing-keystates***, the modifier bit is set and the click is interpreted as a double click.

2.3.39 Shifted Mouse Clicks Can Now Be Used for Editor Commands

Mouse characters can now be modified with modifier keys. See the variable **tv:*mouse-modifying-keystates***. The editor considers each modified mouse click to be a separate command. You can bind commands to particular modified mouse clicks. You can also use Install Mouse Macro (m-X) with modified mouse clicks to increase the number of mouse macros available.

You can put a form in an init file to install a Zwei command on a modified mouse click. Note that in the following example, the mouse click marks the paragraph that surrounds point, not the paragraph under the mouse cursor:

```
(login-forms
  zwei:(set-comtab
         *standard-comtab*                    ;in standard command table
         '(#\Hyper-Mouse-L com-mark-paragraph)
         ))
```

2.3.40 Previously undocumented functions: tv:add-to-system-menu-programs-column, tv:add-to-system-menu-create-menu

tv:add-to-system-menu-programs-column lets you add an entry to the Programs column of the system menu. **tv:add-to-system-menu-create-menu** lets you add an entry to the menu that appears when you click on [Create] in the system menu or in the Edit Screen menu.

tv:add-to-system-menu-programs-column *name form documentation* &optional *after* *Function*

Adds a program to the Programs column of the system menu. *name* is a string, the name to appear in the menu. *form* is a form to evaluate, in its own process, when the program is selected; often this is a call to **tv:select-or-create-window-of-flavor**. *documentation* is mouse documentation for the menu item. *after* determines the position of the new program name in the Programs column:

nil	Bottom of the column
t	Top of the column
string	After the program named *string* that is now in the menu

Example:

```
(tv:add-to-system-menu-programs-column
  "Hardcopy" '(press:hardcopy-via-menus nil t)
  "Print files on the hardcopy printer")
```

tv:add-to-system-menu-create-menu *name flavor documentation* &optional *after* *Function*

Adds an entry to the menu that appears when you click on [Create] in the system menu or in the Edit Screen menu. *name* is a string, the name of the menu item. *flavor*, a flavor name, is the flavor of window that is created when the menu item is selected. *documentation* is mouse documentation for the menu item. *after* determines where in the [Create] menu the item should appear:

nil	Bottom of the menu
t	Top of the menu
string	After the item named *string* that is now in the menu

Example:

```
(tv:add-to-system-menu-create-menu
  "Concept Editor" 'crl:concept-editor
  "Edit the representation of a concept in the CRL system")
```

tv:select-or-create-window-of-flavor *find-flavor* &optional *(create-flavor find-flavor)* *Function*

Selects the most recently selected window of flavor *find-flavor*. If no window of that flavor exists, makes a window of flavor *create-flavor* and selects it.

2.3.41 Argument to :menu type menu items can be a menu or a form

The "argument" to **:menu** type menu items — the specifier for the submenu — can now be a menu or a form. Previously it could be a menu or a symbol.

2.3.42 Clicking Middle Edits Current String in Choose-variable-values Windows

In a choose-variable-values window, clicking middle on a string that is displayed as a value now lets you edit that string.

2.3.43 tv:scroll-maintain-list init function can take arguments

Previously, the init function specified as an argument to **tv:scroll-maintain-list** was itself called with no arguments. Now you can use an optional &rest argument to **tv:scroll-maintain-list** to specify arguments to be passed to the init function at redisplay time.

tv:scroll-maintain-list *init-fun item-fun* &optional *per-element-fun stepper-fun compact-p pre-proc-fun* &rest *init-args* *Function*

Constructs and returns a list item that updates itself when the scroll window is asked to redisplay. Takes the following arguments:

init-fun — The init function that will be called at redisplay time to provide a representation of the set of objects to be displayed.

init-args — Arguments to be passed to *init-fun* when called at redisplay time.

item-fun — The item function, to be applied to each object of yours to produce a display item.

per-element-fun — A function to be put in the list item plist of the list item as the **:function** function.

stepper-fun — The function that is called on the set of objects and all "rest"s of the set. It is expected to return three values: the next element, the "rest" of the set, and **t** if it has returned the last element of the set. If not given, *stepper-fun* defaults to **tv:scroll-maintain-list-stepper**, a function that handles ordinary lists.

compact-p — An optional flag that causes **tv:scroll-maintain-list** to copy the list it builds at each redisplay into a special area for such lists, in order to optimize paging performance. The list so constructed will be stored in compact (that is, cdr-coded) form.

pre-proc-fun — A function to be put in the list item plist of the list item

as the **:pre-process-function** function. If not given, *pre-proc-fun* defaults to **tv:scroll-maintain-list-update-function**.

Following is a simple example:

```
(tv:scroll-maintain-list #'(lambda (instance)     ;The init function
                             (send instance ':value-list))
                         #'(lambda (value)      ;The item function
                             (tv:scroll-parse-item
                               '(:string ,(format nil "~S" value))))
                         nil nil nil nil
                         self)                  ;Argument to init function
```

3. Changes to Networks in Release 5.0

3.1 Incompatible Changes to Networks in Release 5.0

3.1.1 Network Namespace System

Release 5.0 implements a new network database. This database is a collection of objects known to networks, such as hosts, users, networks, printers, sites, and classes of objects. Objects of different classes can have the same names. To eliminate naming conflicts when different sites are linked by long-distance networks, the database is divided into namespaces, or mappings of names of objects to objects.

The database is maintained by database servers. A namespace editor exists to add objects to the database or change their properties. The user interface to the editor is the function **tv:edit-namespace-object**. The system also includes means of defining protocols and media, and defining and invoking network services.

For more information on the changes to networks in Release 5.0: See the section "Network Database". See the section "The Lisp Machine Generic Network System". See the section "Interfacing to the Network System".

3.1.2 chaos:stream, chaos:close, and chaos:finish renamed

The following functions have been renamed in the Chaosnet implementation.

All the known places in the system that use these have been updated. The old function names are still shadowed in the **chaos:** package in order to cause undefined function errors during compilation instead of calling an incompatible function.

Old name	*New name*
chaos:stream	**chaos:make-stream**
chaos:close	**chaos:close-conn**
chaos:finish	**chaos:finish-conn**

3.1.3 neti:reset, neti:enable, and neti:disable replace chaos:reset, chaos:enable, and chaos:disable

The functions **chaos:reset** and **chaos:enable** have been replaced with **neti:reset** and **neti:enable**, which reset and enable the entire network system. If you call **neti:reset** and then want to turn the network back on, you must now call **neti:enable** to do so. Formerly, **chaos:reset** turned the network back on after resetting it.

chaos:disable has been replaced by **neti:disable.**

neti:reset *Function*

Resets the local networks. Disables and then resets the interfaces. After using **neti:reset** you must call **neti:enable** if you want to turn the network back on.

neti:enable *Function*

Enables the local networks and interfaces.

neti:disable *Function*

Disables the local networks and interfaces. If you want to reset the local networks and interfaces and then turn them back on, you should call **neti:reset** and then **neti:enable.**

3.1.4 Changes to chaos:open-stream

In general, applications should use the service mechanism instead of Chaos-specific routines. See the section "The Lisp Machine Generic Network System".

The function **chaos:open-stream** accepts **nil** as the host argument to mean issue a *listen* for the contact name (as opposed to a request at a specific host). The stream that is returned is still in the *RFC Received* state. The following messages can be sent to the stream:

:foreign-host Returns the host object of the host requesting the connection.

:accept Accepts the connection.

:reject &optional *reason*
Rejects the connection with *reason.*

3.1.5 chaos:send-unc-pkt automatically returns the packet to the free pool

chaos:send-unc-pkt does an implicit **chaos:return-pkt,** which returns the packet to the free pool at the appropriate time. The user is not allowed to reuse this packet. This is an incompatible change.

The documentation for **chaos:send-pkt** in the *Chaosnet* document failed to mention that **chaos:send-pkt** automatically returns the packet via **chaos:return-pkt.** The code for this function remains unchanged; this is a clarification.

3.2 New Features in Networks in Release 5.0

3.2.1 New function: chaos:conn-finished-p

chaos:conn-finished-p *conn* *Function*

A predicate that returns something other than **nil** if all data that have been output have been received *and* acknowledged by the foreign side of the connection.

3.2.2 Changes to VMS Chaosnet

The following changes have been made to VMS Chaosnet:

- The NCP writes the CHNCP.GSF global section, which contains the connection database. The SHOWNCP utility displays the data in CHNCP.GSF.
- An NCP internal routing table server allows routing tables to be examined dynamically.
- The FILE server now supports pathname completion and directory creation.
- When transferring binary files to a VMS host, the files are written in the following way: If the file is a "QFASL" file, it is written as an RMS sequential file with variable-length records whose maximum size is 2048 bytes. Note that only the last record can be less than 2048 bytes long. If is not a "QFASL" file, it is written as an RMS sequential file with fixed-length, 512-byte records. A "QFASL" file is any file that has to remember its length in bytes exactly; some examples are Lisp Machine .BIN and .QBN files. An example of a non-"QFASL" binary file is a VMS executable file (.EXE).
- CFTP now supports GET and SEND /BINARY=*byte-size*. It also supports the GET/VAR feature. This tells CFTP, when transferring binary files, to write RMS sequential files with variable-length records whose maximum size is 2048 bytes. Note that only the last record can be less than 2048 bytes long. Without /VAR, CFTP writes RMS sequential files with fixed-length, 512-byte records.
- A TELNET user exists. You can invoke it with the DCL "TELNET" foreign command.

3.2.3 Changes to Serial I/O: Parity Recovery and Xon/Xoff Character Setting

Two features, serial parity recovery and XON/XOFF character setting, have been added to serial I/O. The new parameters (at either initialization or **:put**) are:

: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.

:output-xoff-character
The value is a character that is used to control flow of data from the Lisp Machine 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 Lisp Machine 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.

:input-xoff-character
(3600 only) The value is a character that is used to control flow of data from the external device to the Lisp Machine. It is sent by the Lisp Machine to suspend the flow of data when the **:generate-xon-xoff** flag is set. The default is #o023.

:input-xon-character
(3600 only) The value is a character that is used to control flow of data from the external device to the Lisp Machine. It is sent by the Lisp Machine to resume the flow of data when the **:generate-xon-xoff** flag is set. The default is #o021.

3.2.4 Hdlc Serial I/O on the 3600

In Release 5.0, the 3600 supports synchronous serial I/O using HDLC-like bitstuffing protocols. The CCITT-16 CRC polynomial is used.

This facility requires that the 3600 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.

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 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**.

si:make-serial-stream &rest *options* *Function*
This function initializes the serial I/O facility and returns the serial I/O stream.

The *options* argument is an alternating list of keyword symbols naming parameters, and initial values for those parameters. This lets you initialize

parameters when you start using the serial I/O stream. You can change most of them later with the **:put** operation.

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 on the 3600: See the section "Parameters: 3600 Serial I/O".

: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.

3.2.5 Interface to the Vadic Modem

This is the low-level interface to the Vadic modem. To open a connection with the Vadic modem do:

```
(si:make-serial-stream ':flavor 'si:modem ':phone-number
  number-to-dial other-serial-options)
```

The system uses the autodialer to dial the given number and return a serial stream if it succeeds. If it fails, it signals an error based on **si:modem-error**.

4. Changes to Utilities in Release 5.0

4.1 Incompatible Changes to Utilities in Release 5.0

4.1.1 Default Font Format Now Bfd

The Font Editor now reads and writes BFD format fonts by default. These fonts are stored in files of type BFD, as are the system TV and LGP-1 fonts. You can still read and write font files of other types, such as BIN files on the 3600 and QBIN files on the LM-2. See the section "Changes to Font Editor File Commands".

4.1.2 Changes to Font Editor File Commands

Pressing R in FED translates into "read any font file". Pressing W translates into "write any font file". FED determines what kind of file to read or write based on the (canonical) type of the pathname you type. The default type is BFD. Clicking left on [Read File] is equivalent to pressing R, and clicking left on [Write File] is equivalent to pressing W.

The file type defaults from the (canonical) type of the pathname presented as the default. For example, if you type foo.bfd, you read or write a BFD file, whereas if you type foo.bin, you read or write a BIN file. FED complains if you supply a file type that is not a valid font file type for the machine you are using.

Clicking right on [Read File] or [Write File] gives you a menu of file types to use as the default and the prompt. Holding down the CTRL key while pressing R or W or while clicking left on [Read File] or [Write File] makes the default type BFD. Holding down the META key while using one of these commands makes the default type BIN on the 3600 and QBIN on the LM-2.

4.1.3 Changes to FUNCTION C, FUNCTION M, and FUNCTION Q

FUNCTION C, FUNCTION M, and FUNCTION Q have been changed to provide easier and more consistent control over which windows they affect. The operation of these keystrokes is as follows:

FUNCTION C — Controls the black-on-white state of the entire screen.
Arguments:

None	Toggle
0	Black-on-white
1	White-on-black

FUNCTION c-C — Controls the black-on-white state of the selected window.
Arguments:

	None	Toggle
	0	Same state as main screen's
	1	Opposite of state of main screen

FUNCTION m-C — Controls the black-on-white state of the mouse documentation line. Arguments:

None	Toggle
0	Same state as main screen's
1	Opposite of state of main screen

FUNCTION M — Controls global MORE processing. Arguments:

None	Toggle
0	Turn off MORE processing
1	Turn on MORE processing

FUNCTION c-M — Controls MORE processing for the selected window. Arguments: same as those for FUNCTION M.

FUNCTION Q — Hardcopies the entire screen on the default screen hardcopy device. Arguments: none.

FUNCTION c-Q — Hardcopies the selected window on the default screen hardcopy device. Arguments: none.

FUNCTION m-Q — Hardcopies the entire screen, without the status line and mouse documentation line, on the default screen hardcopy device. Arguments: none.

4.2 New Features in Utilities in Release 5.0

4.2.1 New feature: Flavor Examiner (SELECT X)

The Flavor Examiner is available via SELECT X or the system menu. This is strictly an interim program; it is supported fully in Release 5 but will eventually be incorporated into the Inspector.

Use the HELP command to learn how to use this new feature.

4.2.2 New terminal program (SELECT T)

The new terminal program incorporates the functions of the former Telnet and Supdup programs. It is available via SELECT T. Because it uses the generic network system, it allows access (in the presence of appropriate gateways) via autodialers to dialups, as well as direct Chaosnet and TCP through a gateway.

The prompt is still Connect to host:. To this you simply type the name of any host. (For information on naming of hosts, setting up host databases, declaring host addresses, and supported login services: See the section "Network Database". See the section "The Lisp Machine Generic Network System".)

The network system picks the best login service supported by the host and the optimum route to it. You can no longer specify a particular gateway and special contact name or port using ◊ and /. Such control arguments and new higher-level ones (such as a particular protocol to use, rather than the default) will be added to the terminal program when the system includes a command processor.

Once connected, you can give commands by pressing NETWORK followed by another character. The following commands are available:

NETWORK A	Send an ATTN (in Telnet, a new Telnet "Interrupt Process")
NETWORK D	Disconnect
NETWORK L	Log out of remote host and break the connection
NETWORK Q	Disconnect and deselect this window (Quit)
NETWORK M	Toggle MORE processing

You can issue more complicated commands using the extended command, NETWORK X. This command would use a command processor; in the interim (Release 5.0), this command uses the choose-variable-values facility. With NETWORK X you can change the following:

- the escape character
- whether characters overstrike or erase
- whether MORE processing is enabled
- in the case of Telnet, whether Imlac terminal codes are interpreted in host output

These were all formerly single-letter commands. A facility also exists for logging host output to a wallpaper file.

You can no longer press NETWORK at the Connect to host: prompt to issue a command before connecting. A command processor will fill this role in a future release.

4.2.3 Show Hardcopy Status (m-X) replaces chaos:print-lgp-queue

The Zwei command Show Hardcopy Status (m-X) replaces **chaos:print-lgp-queue** as the means of viewing the print queue.

The command prompts for the name of a printer. You can specify the queue of a particular printer by typing the name of the printer, or you can see the queues for all printers by pressing RETURN.

4.3 Improvements to Utilities in Release 5.0

4.3.1 Font Editor and Inspector use ESCAPE to evaluate forms

The Font Editor (FED) and the Inspector now use the ESCAPE key (on the 3600) and the ALTMODE key (on the LM-2) to evalute a Lisp form. Previously, those programs used the QUOTE key for this function. QUOTE is still accepted for compatibility on the LM-2; this key does not exist on the 3600.

4.3.2 Debugger c-M creates a process

The c-M command in the Debugger has been changed to create a new process. While you are editing the message, you use FUNCTION S to switch back to the Debugger, use its commands to investigate the state of the Lisp environment, and then switch back to the mail process to continue editing the message.

4.3.3 m-SUSPEND selects frame with break read function for Debugger

The behavior of m-SUSPEND in break loops has changed. The **break** read function now intercepts m-SUSPEND itself. When you press m-SUSPEND to enter the Debugger at the beginning of a line in a break loop, the current frame in the Debugger is now the frame that contains the **break** read function. The difference between m-SUSPEND and c-m-SUSPEND is that when you use m-SUSPEND at the beginning of a line, you enter the Debugger with a current frame that is closer to the frame that contains **break**'s caller. This eliminates irrelevant stack frames when, for example, you press c-SUSPEND to interrupt a program and then decide to enter the Debugger. This behavior is likely to change in a future release.

4.3.4 END and c-END swapped in Converse

When using the Converse facility prior to Release 5.0, pressing END sent a message and exited the Converse window; pressing c-END just sent a message, without leaving Converse. Now the reverse is true by default. Pressing END just sends the message, and pressing c-END sends the message and exits from the window.

The new variable **zwei:*converse-end-exits*** controls the behavior of END and c-END in Converse. If its value is **nil**, END sends and remains in Converse, and c-END sends and exits. If its value is not **nil**, c-END sends and remains in Converse, and END sends and exits. The default value is **nil**.

4.3.5 Changes to Converse Notifications

Converse beeps as soon as a message comes in, and if it is supposed to notify the user, it does so without waiting for the main Converse process to wake up. In pop-up mode, when the main Converse process is busy, an incoming message causes Converse to beep but not to display the message. This is necessary since only one message at a time should pop up. While the pop-up window is exposed, an incoming message causes Converse to beep. When the pop-up window is deexposed it is reexposed immediately with the new message in it.

If the main Converse window is exposed, a new message is never shown via notification or a pop-up window. If the main Converse window is exposed but its process is busy (typically, when it is in the Debugger or in an editor command and waiting for typein), Converse beeps but does not display the message. You can display the message with **print-sends** or by clearing the Converse process. You can usually clear the Converse process by pressing ABORT.

5. Changes to the File System in Release 5.0

5.1 Incompatible Changes to the File System in Release 5.0

5.1.1 New Default LMFS Translation Table for Sys Hosts

The default LMFS translation table for the SYS host has been changed to conform to the new rules for translating logical to physical pathnames. See the section "Changes to Logical Pathname Translations".

Following is the new default translation table:

```
'(("FONTS; TV;" ">sys>tv-fonts>")
  ("FONTS; LGP-1;" ">sys>lgp-1-fonts>")
  ("LMFS-PATCH;" ">sys>lmfs>patch>")
  ("*;" ">sys>*>"))
```

Note that logical directory names are followed by semicolons. Note also that translation tables for file systems other than LMFS might need translations for other logical directories as well. For example, because VMS file names cannot contain hyphens, a translation table for a VMS host might need to include specific translations for logical directories whose names contain hyphens, such as PRESS-FONTS;, L-SYS;, L-UCODE;, and L-FEP;.

Conversion tools exist to convert Release 4 to Release 5 site files. These tools take care of the translations for the SYS host. See the document *Software Installation Guide*. Users defining other logical hosts must convert the translation files on their own. See the function **fs:make-logical-pathname-host**.

5.1.2 LMFS Dumper Supports Accordion Wildcards

The LMFS dumper supports the new accordion wildcards. This is an incompatible change in its behavior.

Old Behavior: Previously, the dumper implicitly (and sometimes gratuitously) recursed over all subdirectories, with the wild name *.*.*, of what you asked it to dump. It did this recursion because there was no way to tell it not to. In addition, this recursion was necessary for file system backup.

New Behavior: The dumper no longer recurses over all subdirectories, regardless of the top-level spec. Now, specify exactly those files you wish to dump using wildcards. Wildcards are powerful enough to say exactly what you mean, subsuming all previously possible cases, and allowing for many new ones. If you want recursion, express it via "accordion" wildcards (**).

The default pathname to be dumped has thus been changed from >*.*.* to >**>*.*.*, which is what the previous one was trying to say. If you give the new dumper >*.*.*, it now dumps exactly that, which is to say files directly in the root directory, and not files in any inferiors.

For example, >**>*.*.* specifies all files in the file system, and >lmach>**>*.*.newest gets all of the newest files in >lmach or any of its subdirectories.

Note that there is no operational change if you use the default of the backup dumper, namely, dump the whole file system. It just looks different on the menu.

The dumper lists directories as it goes down encountering them. It is sufficiently clever to avoid simply expanding the wildcard, thereby listing the whole file system at one time. It has its own implementation of **:wild-inferiors** expansion.

5.2 New Features in the File System in Release 5.0

5.2.1 LMFS Now Supports Directory Links

LMFS now supports directory links. A directory link is a link that acts like a directory and points to a directory or another directory link. Directory links must have the type "directory" and the file version 1 in order to be recognized as such. It is impossible to create directory links and directories with conflicting names. It follows that the target of a directory link must be of type "directory" and version 1, although null string type and null version are accepted to mean this, as well.

The maximum length of link-chasing at any level is 10. Full circular-link checking has also been implemented at the file level; it was not previously present. There is no such thing as multilevel circularity.

Example:

`>abel>foo.bar` is a file.
`>baker>david.directory.1` is a link to `>abel.directory.1`
Opening `>baker>david>foo.bar` in fact opens `>abel>foo.bar`.

Directory links are not marked as such in any way, and are not treated as such unless a directory of that name is being sought. Thus, the setting of link transparencies on a directory link has no effect on its behavior as a directory link.

5.2.2 LMFS Accordion Wildcards

It is now possible to specify wild directories of any arbitrary subtree. The string ** appearing in a LMFS pathname means "any number of intervening levels of directory, including none". This is a form of wildcard, and it creates a wild pathname, which may be used anyplace wild pathnames are used. ** may appear one or many times in the directory component of a filespec, although multiple occurences might be prohibitively expensive. Usually ** is the last directory level; the further away from the last it is, the more expensive it gets.

Note that a directory component of >** means every directory in the file system.

:wild-inferiors represents ** in LMFS directory components. It is no longer

possible to create, from a string, a LMFS pathname with a directory component of **:wild**. Although such pathnames still work, pathnames with directory components of **(:wild-inferiors)** ought to be functionally indistinguishable. Directory components of **:wild** print as though they were **(:wild-inferiors)**.

Pathname	*What it means*
>**>*.lisp	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.

5.2.3 Dumper Restarting and Append-to-tape Default

The LMFS dumper allows you to restart a dump from a given point. The option "restart pathname", when not empty, specifies a pathname that will be compared with the pathname of any file to be dumped or directory to be listed, and processing will be suppressed if the pathname of the object is considered "less" than the restart pathname. This comparison is similar to that done by **fs:pathname-lessp**. All of the dumper's processing is strictly ordered by this predicate.

The LMFS dumper also does not attempt to read tapes before writing on them unless either:

1. It intends to append to that tape, and is at the beginning of the tape.

2. The new site attribute **:validate-lmfs-dump-tapes**, whose default is **nil**, is set to something other than **nil**. The default disabling of this attribute avoids attempts to read blank tape and potentially unbounded tape reading on some TOPS-20s. However, enabling the attribute offers more protection against accidental overwriting of backup tapes.

In Release 5.0, the default for "append to tape" has been changed to "no" if you are running on the 3600. This is because hardware limitations make it impossible to append to a cartridge tape. Of course, you can dump a 3600 file system to a non-3600 remote noncartridge tape server and append.

6. Changes to Zmacs in Release 5.0

6.1 Incompatible Changes to Zmacs in Release 5.0

6.1.1 Both default pathnames for Source Compare (m-X) now use :newest version

Source Compare (m-X) has been changed. The default pathname for the first argument now uses the **:newest** version instead of the **:oldest**. The default for the second argument continues to use **:newest**. The change was made because **:oldest** is usually not the right thing; also, the asymmetry of the two defaults, as well as the fact that **:oldest** is rarely used, caused unexpected behavior.

6.1.2 Changes to Add Patch Changed Definitions (m-X) and Add Patch Changed Definitions of Buffer (m-X)

Formerly, these commands queried for each definition to be patched if given a numeric argument, but by default they patched everything without asking. This default behavior was incorrect, because the commands patched things that had been patched before, and they made patches for definitions in the patch buffer itself.

The commands no longer look at their numeric arguments. Add Patch Changed Definitions (m-X) queries for each buffer that contains a definition that might need to be patched; answering N skips the rest of that buffer.

Both commands query for each definition to be patched:

Y	patches it
N	skips it
P	patches it and any additional definitions in the same buffer without asking any more questions

If there are more buffers containing definitions to be patched, the commands ask questions again when they get to the next buffer.

A definition that has not been changed since it was patched is not considered a candidate for patching again, even though it has been changed since the file was read in. Note that patching any region of text lying entirely within a definition (with Add Patch (m-X)) counts as patching that definition.

6.1.3 Set Package (m-X) offers to create a package

Set Package (m-X) used to insist that the package must already exist, although related commands (such as Find File (c-X c-F) and Reparse Attribute List (m-X)) offered to create a package for you. Set Package (m-X) has been changed so that it queries you about whether to create a package that doesn't exist.

6.1.4 Change in numeric arguments to Copy File (m-X)

The numeric arguments to Copy File (m-X) have been reorganized.

The numeric argument controls copying of attributes. With no numeric argument, creation date and author are copied, and the mode (binary or character) of copy is determined by the file being copied. To force mode, or to suppress author or creation date copying, supply a numeric argument created by adding the values corresponding to the descriptions below. For example, c-12 m-X Copy File suppresses author and creation date copying.

1	Force copy in 16-bit binary mode
2	Force copy in character (text) mode
4	Suppress copy of author
8	Suppress copy of creation date

6.2 New Features in Zmacs in Release 5.0

6.2.1 New Zmacs command: Resume Patch (m-X)

Resume Patch (m-X) is a new Zmacs command that allows you to go back to a patch that you were not able to finish in the same session in which you started it.

6.2.2 New Zmacs command: Start Private Patch (m-X)

Start Private Patch (m-X) is similar to Start Patch (m-X), but it does not have anything to do with the world of systems, major and minor version numbers, and official patch directories. Instead of prompting for a system, Start Private Patch prompts for a file name. You can use Add Patch (m-X), Finish Patch (m-X), and Abort Patch (m-X). When you finish the patch it is written out to the specified file.

This command allows you to make a private patch file that you can load, try out, and share with other users before you install a change as a numbered patch that all users automatically get.

6.2.3 New Zmacs command: Source Compare Newest Definition (m-X)

Source Compare Newest Definition (m-X) compares the current definition (the definition that surrounds point) with the newest version in the normal source file for this definition, regardless of patch files.

6.2.4 New Buffer-history Mechanism in Zmacs

A new feature keeps the history list of previous buffers independently for each window.

When you create a new window, it takes its history list initially from the global history list. From then on, as you switch from buffer to buffer within that window, the history of those changes is kept on the list for that window. This affects particularly Select Previous Buffer (c-m-L) and the default for Select Buffer (c-X B).

The global history list still exists and is used for sorting in List Buffers (c-X c-B). Its ordering has not changed; it is maintained exactly as it used to be.

With two windows on the screen, you now have single-key access to four buffers instead of three. This can be helpful when you are working on two buffers at a time, each with its own need for switching to refer to another buffer.

6.2.5 New Zwei command: Comment Out Region (c-X c-;)

Comment Out Region (c-X c-;) comments out each of the lines in the region. When the region ends at the beginning of a line, it does not comment out that line. If any part of the line is part of the region, it does comment out that line.

A numeric argument inverts the meaning of the command, taking the comment indicators away from any commented-out lines in the region. When any part of the line is part of the region, it removes commenting from around that line. This assumes that any comment starting in column 1 is fair game. It stops when it encounters a line that does not begin the way a comment would, even if more lines that have been commented out remain in the region. It does keep the remainder of the region in this case, so that you can resume.

This works correctly for the different comment indicators for different major modes.

6.2.6 New Zwei command: Find Files in Tag Table (m-X)

Find Files in Tag Table (m-X) preloads every file in the selected tag table into the editor. Like Tags Search (m-X), it prompts for a tag table if one doesn't exist.

The prompting allows for specification of a system or all buffers as the tag table or for reading a .TAGS file.

6.2.7 New Zwei commands: Lowercase Code in [Region/Buffer] (m-X), Uppercase Code in [Region/Buffer] (m-X)

Four new Zwei commands allow you to change the case of code in a buffer or region:

- Uppercase Code in Buffer (m-X)
- Uppercase Code in Region (m-X)

- Lowercase Code in Buffer (m-X)
- Lowercase Code in Region (m-X)

These commands allow you to change a program that is typed in Electric Shift Lock Mode into one that is typed entirely in lowercase text, and vice versa. Only code is changed; comments, strings, or quoted characters are not affected. The commands work only where the entire file is typed in the same mode.

Like other Buffer commands, Uppercase Code in Buffer (m-X) and Lowercase Code in Buffer (m-X) query for a buffer name (the default is the current buffer) before operating on that buffer.

Like other Region commands, Uppercase Code in Region (m-X) and Lowercase Code in Region (m-X) operate on the region if there is one; otherwise they operate on the current definition.

The Uppercase commands put into uppercase all code in the associated area. They have the same effect as retyping that code using Electric Shift Lock Mode.

The Lowercase commands put into lowercase all code in the associated area. They have the same effect as retyping that code without using Electric Shift Lock Mode.

6.2.8 New canonical file type: :mss

The canonical file type **:mss** has been added for Scribe manuscript files. The editor treats these files like files of canonical type **:text**.

6.3 Improvements to Zmacs in Release 5.0

6.3.1 Default File Name Changed for Commands in Dired Buffer

In Dired, if you execute an editor command that accepts a file name as an argument, the default is the file name that appears on the line of the Dired buffer containing point.

This change makes it easier to operate on the file selected in Dired. For example, moving point to some line in Dired and doing Source Compare (m-X) causes the Source Compare command (m-X) to default to that file name.

6.3.2 Major-mode-setting Commands Now Query About Updating File Attribute List

Major-mode-setting commands such as Lisp Mode (m-X) now give the standard query about updating the file attribute list: "Set it for the file and attribute list, too?" (This is like the query for Set Package (m-X) and other attribute-setting commands.)

Programmers who call the **zwei:com-*mode*-mode** function directly (to get their own newly created buffer into the correct mode, for example) should bind **zwei:*set-attribute-updates-list*** to **nil** around the call to suppress the query.

6.3.3 Change in Zmacs command Modified Two Windows (c-X 4)

When two windows are being displayed, invoking the Zmacs command Modified Two Windows by typing c-X 4 J r is equivalent to invoking the Jump to Saved Position (c-X J r) command in the other window. When only one window is displayed, Zmacs goes into two-window mode, with the current buffer in one of the windows, and then jumps to the saved position in the other window.

6.3.4 Internal changes to macros zwei:defmajor and zwei:defminor

The Major and Minor mode system in the editor has been reimplemented using Flavors. For simple applications, the changes that were made are compatible with Release 4.

7. Changes to Zmail in Release 5.0

7.1 Incompatible Changes to Zmail in Release 5.0

7.1.1 Zmail Init File Pathnames Standardized

The names of Zmail init files have been standardized, and init files are now of canonical type **:lisp** for source files and **:bin** and **:qbin** for compiled files. You must change the name of your init file for Release 5.0.

For hosts that support long file names, the standard Zmail init file name is ZMAIL-INIT. Hosts that do not support long file names have conventions peculiar to each system.

Following are the names of Zmail init source files on some hosts:

Host system	*File name*
LMFS/TOPS-20	ZMAIL-INIT.LISP
UNIX	zmail-init.l
VMS	ZMAILINI.LSP
ITS	If user has own directory: ZMAIL >. If user does not have own directory: *USER* ZMAIL.

7.1.2 Babyl files with summary-window-format other than t or nil need to be edited

Any Babyl files with a Babyl file option of **summary-window-format** other than **t** or **nil** must be edited. Previously, Babyl file options were written in the **user** package, which was also where keywords resided. Now that the **keyword** package is separate from **user**, other keyword values (such as **:subject** and **:calendar**) require a colon prefix.

7.1.3 Ramifications of Host Colon Change for Babyl Files

The line MAIL: *HOST*:<*USER*>MESSAGE.TXT;1, which older versions of Zmail might have put in a Babyl file, doesn't work anymore, because it is parsed with *HOST*: interpreted as a device.

You can edit this using Zmail Babyl file options.

The essence of the new scheme is that Babyl files pointing to hosts other than that on which they reside can reside only on Lisp Machines or ITS's, for those are the only systems that natively support *HOST*: as designating hosts.

7.2 New Features in Zmail in Release 5.0

7.2.1 Sorting by Conversations Available

Sorting by conversations is available from the [Sort] command. Use [Sort (R)] to get a menu, then click on [Conversations]. As a result, messages that reference one another within the current sequence are grouped together.

7.2.2 New [map Over] Menu Item: [reply]

[Reply] is a new item in the menu that results from [Map Over (R)]. Clicking on this item replies to all messages in the sequence. A line identifying each message is inserted into the reply's In-reply-to field. When composing the reply, yanking (using c-m-Y) yanks all messages into the reply.

7.2.3 New [map Over] Menu Item: [select Conversation]

[Select Conversation] is a new item in the menu that results from [Map Over (R)]. You can also perform this operation by Select All Conversations By References (m-X); this is implemented by **zwei:com-zmail-select-all-conversations-by-references**.

[Map Over / Select Conversation] (or its extended command counterpart) selects messages that a message in the sequence refers to, or that refer to a message in the sequence, recursively. It is equivalent to appending together all sequences that result from Select Conversation By References (m-X) for each message in the current sequence. An argument gives a menu of universes to search. The universe defaults to loaded files.

7.3 Improvements to Zmail in Release 5.0

7.3.1 Previously undocumented commands: Delete Conversation By References (m-X), Append Conversation By References (m-X)

Delete Conversation By References (m-X) deletes messages that this message refers to, or that refer to this message, recursively. With an argument, it provides a menu of universes to search. The universe defaults to loaded files.

Append Conversation By References (m-X) appends messages that this message refers to, or that refer to this message, recursively. With an argument, it provides a menu of universes to search. The universe defaults to loaded files.

7.3.2 Rfc822 Domain Addressing Supported

Zmail now understands and, when appropriate, generates RFC822 domain addressing, used by the Arpanet.

Zmail now recognizes Resent (-to, -by, -date, -comments) header fields in addition to Redistributed (-to, -by, -date, -comments) fields.

8. Changes to the FEP in Release 5.0

This chapter describes changes to the FEP software that are concurrent with Release 5.0. FEP software is distributed in its own versions, which are separate from Lisp software releases. Release 5.0 requires FEP version 17 or higher. This chapter describes changes to FEP versions 14 through 17 for which Lisp support exists in Release 5.0. It also describes one change for FEP version 18 for which Lisp support does not exist in Release 5.0. Unless otherwise indicated, each change in any FEP version applies to later versions as well.

8.1 FEP Version 14: New Features

8.1.1 FEP Supports Hdlc Serial I/O

FEP version 14 contains support for synchronous serial I/O using HDLC-like bitstuffing protocols. HDLC serial I/O is available only in Release 5.0 and later releases. See the section "Hdlc Serial I/O on the 3600".

8.2 FEP Version 15: Incompatible Changes

8.2.1 h-c-*upper-left* stops execution of Lisp

As of FEP version 15, if you cannot obtain a Lisp Listener window or if no Lisp Listener is responding to keyboard input, you must press h-c-*upper-left* instead of c-*upper-left* to get to the FEP from Lisp. In earlier FEP versions, either c-*upper-left* or h-c-*upper-left* gets to the FEP from Lisp. (*upper-left* is the key in the upper left corner of the keyboard. It corresponds to LOCAL on old keyboards and FUNCTION on new keyboards.)

However, calling **si:halt** is a better way to stop Lisp than pressing h-c-*upper-left* because h-c-*upper-left* could interrupt disk I/O operations. This can render directories unusable, making all the files in the directories inaccessible until Symbolics personnel recover them.

See the section "Summary of Boot and Halt Operations".

8.2.2 si:halt replaces sys:%halt

On the 3600, **si:halt**, not **sys:%halt**, is now the preferred way to stop execution of Lisp.

si:halt *Function*

On the 3600, **si:halt** stops execution of Lisp and gives control to the FEP. This function stops Lisp without the danger of interrupting disk I/O operations. The function **sys:%halt** should no longer be used.

Interrupting a disk write can cause a fatal ECC error later, because the contents of the disk block are incomplete. This can render directories and other files inaccessible. This is a particular problem when halting the machine while using LMFS.

8.2.3 >Configuration.fep Files Are Now Called >Boot.boot

As of FEP version 15, you should rename >Configuration.fep files to >Boot.boot. The file type .fep is now reserved for files that the user should never modify.

8.2.4 New Defaults for FEP Commands

In FEP version 15, the default file name for each of the FEP commands Boot, Show File, Load Microcode, Load World, and Load Sync-Program is the last file name typed to that command.

The initial default for both the Boot command and the Show File command is now >Boot.boot, not >Configuration.fep. Show File uses and sets the same default string that Boot does. This lets you use Show File to look for an appropriate configuration file and then simply type Boot to cold boot using that file.

The Boot command and Show File command share the default, so that

```
Show File >magic.boot
```

causes Boot's default now to be >magic.boot and vice versa.

When the machine is powered up or the FEP is reset, the defaults are initialized as follows:

Boot/Show File	>Boot.boot
Load Microcode	>Microcode1.mic
Load World	>World1.load
Load Sync-Program	>Sync.sync

8.2.5 Disk Format Command Asks Different Question

When you type the Disk Format command, you are asked a series of questions. One of the questions used to be "*To* cylinder", which expected an *exclusive* upper bound as the answer. As of FEP version 15, Disk Format asks "*Through* cylinder", expecting an *inclusive* upper bound as the answer.

8.3 FEP Version 15: New Features

8.3.1 Loading Sync Programs

As of FEP version 15, files of type .sync contain sync programs for the monitor. A new FEP command, Load Sync-Program *file-name*, loads the specified file (of type .sync) into the sync program memory of the I/O board and causes the screen to clear. This is used for machines with monitors that require different sync programs than the one that is preprogrammed into the FEP.

8.4 FEP Version 15: Improvements

8.4.1 Show Configuration Command Displays More Information

Show Configuration has changed considerably in FEP version 15. It still displays the hardware configuration, but in considerably more detail, telling you part numbers, serial numbers, revisions, manufacture dates and other information peculiar to various parts of the machine (for example, the Ethernet address on the I/O paddle card).

Here is an example of what Show Configuration displays:

```
NanoFEP (P.N. 170018) S.N. 311, manufactured on 83-8-11
  Machine serial number 0.
  Manufactured as rev 1, functions as rev 1, ECO level 0
Datapath (P.N. 170032) S.N. 1192, manufactured on 83-9-9
  Manufactured as rev 3, functions as rev 3, ECO level 0
Sequencer (P.N. 170042) S.N. 186, manufactured on 83-4-1
  Manufactured as rev 4, functions as rev 4, ECO level 0
Memory Control (P.N. 170052) S.N. 1200, manufactured on 83-9-16
  Manufactured as rev 5, functions as rev 5, ECO level 0
Front End (P.N. 170062) S.N. 2040, manufactured on 83-8-19
  Manufactured as rev 5, functions as rev 5, ECO level 0
512K Memory (P.N. 170002) S.N. 515, manufactured on 83-9-20
  Manufactured as rev 2, functions as rev 2, ECO level 0
  LBUS slot 0 (octal base address 0)
512K Memory (P.N. 170002) S.N. 589, manufactured on 83-10-14
  Manufactured as rev 2, functions as rev 2, ECO level 0
  LBUS slot 1 (octal base address 2000000)
IO (P.N. 170082) S.N. 176, manufactured on 83-3-25
  Manufactured as rev 2, functions as rev 2, ECO level 0
  LBUS slot 8 (octal base address 20000000)
FEP Paddle Card (P.N. 170066) S.N. 287, manufactured on 83-7-16
  Manufactured as rev 1, functions as rev 1, ECO level 0
IO Paddle Card (P.N. 170086) S.N. 358, manufactured on 83-9-20
  Ethernet address: 08-00-05-01-30-0B
  Manufactured as rev 1, functions as rev 1, ECO level 0
```

8.4.2 Memory Board Not Needed in Lbus Slot 0

As of FEP version 15 and Release 5.0, it is possible to run Lisp and use the FEP commands Disk Format and Disk Restore without a memory board in LBUS slot 0. You must be running microcode version 238 or higher. This change allows for a 32-bit color display.

8.5 FEP Version 16: New Features

8.5.1 New FEP Commands: Add Disk-type and Clear Disk-types

FEP version 16 contains the new commands Add Disk-type and Clear Disk-types.

Add Disk-type lets you declare an arbitrary disk type to the FEP. You can declare up to four disk types before you have to give the Clear Disk-types command. Add Disk-Type is needed only to format and restore disks. It is not needed for normal operation of any validly formatted disk with a FEP file system.

Add Disk-type has the following arguments, for which it prompts with the argument names in parentheses:

name	The textual name by which this disk type is known
cylinders	The number of cylinders supported by the drive
heads	The number of heads on the drive
sectors	The number of sectors
gap1	The length of "gap1"
gap2	The length of "gap2"
gap3	The length of "gap3"
fast	0 for slower disks, 1 for faster disks

These numbers require careful computation and involve some restrictions of the 3600 hardware. The calculation should be done by Symbolics personnel.

Example:

```
Add Disk-type (name) M2284 (cylinders) 823 (heads) 10
    (sectors) 16 (gap1) 27 (gap2) 31 (gap3) 52 (fast) 0
```

Clear Disk-types clears all disk types declared with the Add Disk-type command.

8.6 FEP Version 16: Improvements

8.6.1 Unplugging Lemo Cables Should Not Halt the FEP

As of FEP Version 16, you should not have to reset the FEP after unplugging the lemo cables from either end. If you find that you must still reset the FEP, please let us know.

Sometimes what appears to be a FEP problem when unplugging lemo cables might be a console problem. You might have to power cycle the console rather than resetting the FEP.

8.6.2 Continue Command Sends an All-keys-up Character to Lisp

As of FEP version 16, the Continue command now sends an all-keys-up character to Lisp. Thus, if you press h-c-*upper-left* and then type Continue, Lisp gets the all-keys-up code and knows that HYPER and CONTROL are up.

The Return-keyboard-to-lisp command also sends an all-keys-up code.

8.6.3 More Information Available on Causes of Crashes

In addition to the information it displays in earlier versions, the Show Status command in FEP version 16 prints a section called "3600 program counters". This section can be useful in diagnosing why your machine crashed. It contains the following information:

Macro PC — The address of the current instruction of compiled Lisp code. This is prefaced with either (Odd) or (Even) since there are two instructions per word.

Next micro PC (NPC) — Usually CPC+1, calculated for pipelining. This item is not displayed in FEP version 17 and later versions.

Current micro PC (CPC) — The address of the current microinstruction.

Old PCs (OPC) — The addresses of the 16 most recently executed microinstructions. OPC+0 was executed most recently, OPC+17 least recently.

Use the function **dbg:decode-micro-pc** to decode the microcode PCs printed by the FEP command Show Status.

dbg:decode-micro-pc *pc* &optional (*name* **sys:%microcode-version**) (*version* **(sys:microcode-version-number sys:%microcode-version**)) *Function*

(3600 only) **dbg:decode-micro-pc** is useful for investigating why a machine crashed. It decodes the octal microinstruction addresses printed by the FEP

command Show Status. To use this function you should first write down the Show Status output. You can then either warm boot the machine using the Start command or call **dbg:decode-micro-pc** on another machine.

pc is an address in the microcode, taken from the CPC or OPC information printed by the Show Status command. Show Status prints these numbers in octal; if your default radix is decimal, precede *pc* by **#o**. Normally the number in the Show Status output with the arrow (→) pointing to it is the relevant number, but it can sometimes be useful to try decoding all of the numbers to get additional clues.

name and *version* are optional; they specify the version of the microcode that was running at the time of the crash. You can omit these arguments if you call **dbg:decode-micro-pc** while using the machine that crashed and while running the same microcode version as at the time of the crash. You can also omit these arguments if you call this function from another machine that has a software *and* hardware configuration that is *identical* to that of the machine that crashed. To find the microcode version name and number that a machine is running, use **(print-herald :verbose t)** or take the name and version number of the microcode file in the machine's boot file (normally fep0:>Boot.boot). Microcode version numbers are decimal; include a period at the end of the number if your default radix is octal.

Example:

```
(dbg:decode-micro-pc #o44552 "tmc5-mic" 253.)
```

dbg:decode-micro-pc prints information that depends on the microinstruction:

Microinstruction	*Information printed*
Halt instruction	The reason it halts the machine. An example is "error in the error handler". These reasons are constant strings in the microcode source program and do not represent any dynamic analysis of the state of the machine.
Signaller of a Lisp error	The internal form of the error message. This is not the same form of error message you would ever see otherwise; normally Lisp software translates these messages into conditions and signals them, and the conditions define more readable error messages. This is useful mainly in decoding OPCs earlier than the one with the arrow, when the machine halted because of "error in the error handler".

Handler for a macroinstruction in compiled Lisp code
: The name of that macroinstruction. A halt here might be caused by running a world together with an incompatible microcode, such as a microcode from an earlier release, that does not implement an instruction used by that world.

If all else fails, the function offers to load the microcode symbol table (from the sys:l-ucode; directory) and then prints the symbolic name of the microinstruction. Loading the microcode symbol table takes a few minutes. Microinstruction symbolic names can sometimes be clues to help in figuring out what the machine was doing at the time it crashed.

Two types of symbolic names exist: those with and without parentheses.

If the name includes parentheses, it is a list of the name of a microcode routine and the path through that routine to reach the microinstruction in question. Beware of a pitfall! These names are not unique; the same microinstruction can be reached by multiple paths from different microcode routines. For example, a microinstruction named (FTN-AR-1 3) might also be part of the microcode for the CAR instruction; you cannot assume too much from the name if it contains parentheses. It is only a clue.

If a symbolic name is just a symbol and has no parentheses, it is unique and names the first microinstruction of a microcode routine.

Beware of assuming too much. If the reason Lisp stopped itself is not "microcode halted", the information that **dbg:decode-micro-pc** prints is not likely to be helpful, though it might be useful to people who understand the hardware.

To decode the macrocode PC printed by the FEP command Show Status, warm boot or go to another machine running identical software and call the function **%find-structure-header** on the number printed by the FEP. This is an octal number; use **#o** if necessary. It should return a compiled-function object, which is the function that was executing at the time. To find the exact place in the function that was executing, note the different between the number printed by the FEP and the address in the printed representation of the compiled-function object. You can use **%pointer-difference** to compute this difference. Multiply this by 2, and add 1 if the FEP said the PC was odd (not even). The result is the instruction number of the current instruction; disassemble the compiled function to see it.

Example:

```
Fep>Show status
...
3600 program counters:
  Macro PC/ (Odd)1244531
  ...
Fep>Start
...
(%find-structure-header #o1244531)
#<DTP-COMPILED-FUNCTION EQUAL 1244530>
(%pointer-difference #o1244531 *)
1
(1+ (* * 2))
3
(disassemble ***)
  0 ENTRY: 2 REQUIRED, 0 OPTIONAL
  1 PUSH-LOCAL FP|0              ;A
  2 PUSH-LOCAL FP|1              ;B
  3 BUILTIN EQL STACK
  ...
```

Instruction 3 (EQL) is the one that halted.

8.7 FEP Version 17: Improvements

8.7.1 Show Status Command Displays More Useful Information

As of FEP version 17, the Show Status command prints more useful information.

The "Sequencer status line", which was confusing and difficult to interpret, has been split into two lines, "Sequencer error status" and "Sequencer miscellaneous status". The former contains only conditions that can stop the machine; it no longer contains reported errors that are not in fact errors. The miscellaneous status line contains the sequencer status bits that may be of occasional interest but are not reasons the machine stopped. For the names of the status bits printed in these two lines: See the section "FEP Show Status Command Output".

The confusing and uninformative "MC status" line has been removed.

8.8 FEP Version 18: Improvements

8.8.1 h-c-*upper-left* waits for Lisp to stop itself

With FEP version 18 and a future release, pressing h-c-*upper-left* does not immediately stop Lisp. Instead, the FEP asks Lisp to stop itself cleanly. If Lisp does stop itself, the FEP prints the message "Lisp stopped itself." If Lisp does not stop itself after about three seconds, the FEP prints, "Waiting for Lisp to stop

itself..." If after another three seconds Lisp does not stop itself, the FEP forcibly stops Lisp and prints, "Halting execution of Lisp."

The Lisp support for this change is not installed in Release 5.0. This means that if you have FEP version 18 and Release 5.0, Lisp never stops itself after you press h-c-*upper-left*. The FEP waits three seconds, prints "Waiting for Lisp to stop itself...", waits another three seconds, then forcibly stops Lisp and prints, "Halting execution of Lisp."

The purpose of this change is to reduce the chance of halting the 3600 during a disk write, which might cause ECC errors. Although the Lisp support for the change is not in Release 5.0, the FEP's delay of about six seconds before halting Lisp reduces (but does not eliminate) the risk of stopping Lisp at a dangerous time. The preferred way of stopping Lisp is still to call **si:halt**; use h-c-*upper-left* only if no Lisp Listener is responding.

9. Release 5.0: Notes and Clarifications

9.1 Clarifications and Corrections for Release 5.0

9.1.1 What happens when you cold boot

Because of the new network system, the procedure the Lisp Machine follows when cold booting has changed. While this does not normally require a different response from you, the process states in the status line are visibly different. This section briefly describes what the machine is doing. For background information: See the section "Network Database". See the section "The Lisp Machine Generic Network System".

The Lisp Machine first needs to determine the current time. Previously, the site file contained a list of hosts on the local network that were time servers. That list no longer exists. Instead, the machine issues a Chaosnet broadcast request (BRD) for the time. While the machine is waiting for the network, the status line displays a process state of "BRD Wait". Any host on the local network can answer this request. This means that in normal operation, if your site has more than one Lisp Machine, it is not necessary to enter the time by hand. However, if no host responds after a short time, the machine asks you to enter the time manually.

Special world loads for distribution no longer exist. A world load produced at one site can be loaded onto a machine at another site and automatically reconfigures itself for the new site. To do this, the machine must find out when booted whether the site has changed. It does this by asking the other machines on the local network, using a Chaosnet broadcast request. The status line again displays a process state of "BRD Wait". If the site has changed, or if no one answers this "Who am I?" query, the machine might ask you whether the site has changed. For information on what to do in this case: See the document *Software Installation Guide*.

Information in the network namespace database is cached in a world load and timestamped. However, namespace information might have changed since the world load was made. To find out about these changes, the machine connects to the namespace server machine. The status line now displays a process state of "Connect *host*". The machine tells the server the timestamp it previously knew for the local namespace. If the world load was saved recently, nothing has changed and the server responds immediately that the timestamp is still current. If some objects in the namespace have changed, the server provides the new timestamp for the local namespace and the names of the changed objects.

Your machine marks as valid any object that the server does *not* say has changed. The old information on the objects that did change still has the old timestamp. When the system needs to access that information, it discovers that the timestamp

no longer matches the timestamp of the local namespace. It then asks the namespace server for the information associated with the current timestamp. While the machine checks the names of old objects, the status line displays a process state of "Net In". For information on the protocol used for this incremental update: See the section "Network Database".

Your machine gets most of the lists of hosts and other network namespace objects that were formerly stored explicitly in the site file by doing a pattern-matching lookup using **net:find-objects-from-property-list**. This information includes, for example, the set of hosts at the local site that can provide store and forward mail service, the set of hardcopy printers at the local site, and the hosts that can spool files to a specific local hardcopy printer. The lists of objects resulting from these lookups are cached in a world load. If the machine does not have a cached list matching the result of a given property list query, or if the incremental update while booting reveals that the cached list is old, the system connects to the namespace server to get the latest list when it is needed.

To avoid the inconvenience of this connection the first time you need to send mail or hardcopy a file, the machine maintains a list of frequently asked questions. After the machine gets the incremental update while booting, a background process called Get Common Property Lists starts. It connects to the namespace server and caches the latest version of these commonly used sets of objects. If the world load is relatively new, this process runs for only a few seconds. The process runs at a lower priority than the Lisp Listener process to avoid interfering with your work.

Booting takes longer with an old world load than a new one because many aspects of the network database have changed. If you have an old world load and find that it takes a long time to boot or that it pages for a long time after booting (while running the Get Common Property Lists process), you might disk-save the world to make a new world load. The new world load caches information with the new namespace timestamp and therefore does not take so long to boot.

A world load that comes from another site takes longer to boot for the same reason: It has no cached information about the local namespace. This is why the installation procedure for new world loads includes booting and disk-saving them.

9.1.2 sort predicate should return nil for equal elements

The predicate for **sort** should return **nil** if its arguments are equal. For example, to sort in the opposite direction from <, use >, not $\geq$. This is because the quicksort algorithm used to sort arrays and cdr-coded lists becomes very much slower when the predicate returns non-**nil** for equal elements while sorting many of them.

See the function **sort**.

9.1.3 store not supported on the 3600

store is not supported on the 3600. **store** existed only for some large programs written in Maclisp, most of which now use more recent array referencing. **store** would require the saving of state in processes, which would slow down all processes. Use **aref** and **aset** (or **setf**) instead.

See the section "Change in Array Referencing".

9.1.4 Using copy-array-portion on the same array

It is safe to use **copy-array-portion** to copy from and to the same array only when the from-index is not less than the to-index, or when the "from" and "to" portions of the array do not overlap.

See the function **copy-array-portion**.

9.1.5 bitblt width from the destination array

The *width* argument to **bitblt** is in units of elements of the destination array. This is important if the source and destination arrays are of different types.

See the function **bitblt**.

9.1.6 Inspecting hash arrays of eq hash tables not permitted

You cannot inspect the hash array of an **eq** hash table. Trying to do so is likely to signal an error.

9.1.7 Known problem: char-upcase and char-downcase undefined for modified characters

char-upcase and **char-downcase** preserve font information. Because font information and control characters use the same character bits, the results of **char-upcase** and **char-downcase** are undefined for characters with modifier bits (CONTROL, META, SUPER, and HYPER).

See the function **char-upcase**. See the function **char-downcase**.

9.1.8 How to use the sys:function-parent declaration

The *Release 4.0 Release Notes* documented the use of the **sys:function-parent** declaration when a definition results from the macro expansion of a source definition. Following is a more comprehensive explanation:

A *definition* is a Lisp expression that appears in a source program file and has a name by which a user would like to refer to it. Definitions come in a variety of types. The main point of definition types is that two definitions with the same name and different types can exist simultaneously, but two definitions with the same

name and the same type redefine each other when evaluated. Some examples of definition type symbols and special forms that define such definitions are:

Type symbol	*Type name in English*	*Special form names*
defun	function	**defun, defmacro, defmethod**
defvar	variable	**defvar, defconst, defconstant**
defflavor	flavor	**defflavor**
defstruct	structure	**defstruct**

Things to note: More than one special form can define a given kind of definition. The name of the most representative special form is typically chosen as the type symbol. This symbol typically has a **si:definition-type-name** property of a string that acts as a prettier form of the name for people to read.
record-source-file-name and related functions take a name and a type symbol as arguments. The editor understands certain definition-making special forms, and knows how to parse them to get out the name and the type. This mechanism has not yet been made user-extensible. Currently the editor assumes that any top-level form it does not know about that starts with "(def" must be defining a function (a definition of type **defun**) and assumes that the cadr of that form is the name of the function. Heuristics appropriate for **defun** are applied to this name if it is a list. In general, a definition whose name is not a symbol and whose type is not **defun** does not work properly. This will be fixed in a future release.

The declaration **sys:function-parent** is of interest to users. The function with the same name is probably not of interest to users; it is part of the mechanism by which the Zmacs command Edit Definition (M-.) figures out what file to look in.

Example:

We have functions called "frobulators" that are stored on the property list of symbols and require some special bindings wrapped around their bodies. Frobulator definitions are not considered function definitions, because the name of the frobulator does not become defined as a Lisp function. Indeed, we could have a frobulator named **list** and Lisp's **list** function would continue to work. Instead we make a new definition type.

```
(defmacro define-frobulator (name arg-list &body body)
  `(progn 'compile
          (add-to-list-of-known-frobulators ',name)
          (record-source-file-name ',name 'define-frobulator)
          (defun (:property ,name frobulator) (self ,@arg-list)
            (declare (sys:function-parent ,name define-frobulator))
            (let (,(make-frobulator-bindings name arg-list))
              ,@body))))

(defprop define-frobulator "Frobulator" si:definition-type-name)
```

Here we would tell the editor how to parse **define-frobulator** if its parser were

user-extensible. Because it is not, we rely on its heuristics to make M-. work adequately for frobulators.

Next we define a frobulator. This is not an interesting definition, for we do not actually know what the word "frobulate" means. We could always recast this example as a symbolic differentiator: We would define the + frobulator to return a list of + and the frobulations of the arguments, the * frobulator to return sums of products of factors and derivatives of factors, and so forth.

```
(define-frobulator list ()
  (frobulate-any-number-of-args self))
```

In **define-frobulator**, we call **record-source-file-name** so that when a file containing frobulator definitions is loaded, we will know what file those definitions came from. Inside the function that is generated, we include a function-parent declaration because no definition of that function is apparent in any source file. The system will take care of doing
(record-source-file-name '(:property list frobulator) 'defun), as it always does when a function definition is loaded. Suppose an error occurs in a frobulator function — in the **list** example above, we might try to call **frobulate-any-number-of-args**, which is not defined — and we use the Debugger C-E command to edit the source. This will be trying to edit **(:property list frobulator)**, the function in which we were executing. The definition that defines this function does not have that name; rather, it is named **list** and has type **define-frobulator**. The **sys:function-parent** declaration enables the editor to know that fact.

If your definition-making special form and your definition type symbol do not have the same name, you should define the special form's **zwei:definition-function-spec** property to be the definition type symbol. This helps the editor parse such special forms.

For another example, more complicated but real, use **mexp** or the Zmacs command Macro Expand Expression (C-sh-M) to look at the macro expansion of:

```
(defstruct (foo :conc-name) one two)
```

The macro **sys:defsubst-with-parent** that it calls is just **defsubst** with a **sys:function-parent** declaration inside. It exists only because of a bug in an old implementation of **defsubst** that made doing it the straightforward way not work.

9.1.9 Use record-source-file-name instead of (remprop *symbol* ':source-file-name)

When redefining functions, some users try to avoid redefinition warnings and queries by using the form **(remprop** *symbol* **':source-file-name)**. The preferred way to do this is to use the form **(record-source-file-name** *function-spec* **'defun t)**. The former method causes the system to forget both the original definition and other definitions for the same symbol (as a variable, flavor, structure, and so forth). **record-source-file-name** lets the system know that the function is defined in two places, and it avoids redefinition warnings and queries.

Of course, if you are redefining something other than a function, use the appropriate definition type symbol instead of **defun** as the second argument to **record-source-file-name**. For example, if you are redefining a flavor, use **defflavor** as the second argument.

9.1.10 Use cdr with locatives returned by locf

When using **locf** to return a locative, you should use **cdr** rather than **car** to access the contents of the cell to which the locative points. This is because **(locf (cdr** *list*)) returns the list itself instead of a locative.

Example:

```
(car (locf (cdr '(a b)))) => a          ;wrong
(cdr (locf (cdr '(a b)))) => (b)        ;right
```

9.1.11 rplaca can be used with stack lists

The *Release 4.0 Release Notes*, section 2.2.30, page 43, stated that **rplaca** could not be used on stack lists. This is incorrect; you can use **rplaca** with stack lists, but not **rplacd**.

See the special form **with-stack-list**. See the special form **with-stack-list***.

9.1.12 FUNCTION 2 W displays current process name in status line

Typing FUNCTION 2 W changes the status line so that it displays the name of the process instead of displaying the name of the user. This also freezes the status line on that process; normally the status line switches to displaying a different process whenever the window selection mechanism tells it to.

If you see an unexpected state in the status line, you can use FUNCTION 2 W to find out what process is in that state (it might be that you are not talking to the process you think you should be.)

FUNCTION 1 W returns the status line to normal.

9.1.13 Known problem with si:gc-reclaim-immediately

The 3600 has a known bug in the garbage collector stimulated by turning on **si:gc-reclaim-immediately**. The typical manifestation is the following error message:

```
>>Trap: The first argument given to SYS:INTERNAL-=, NIL, was not a number.
While in the function SI:SCAVENGE-REGION ← SI:%GC-SCAVENGE ← SI:SCAVENGE-ALL
```

This bug will be fixed in a future release. In the meantime, avoid setting **si:gc-reclaim-immediately** to anything other than **nil**. You might be able to set **si:gc-reclaim-immediately** to **t** if you also set **inhibit-idle-scavenging-flag** to **t**. You must set **inhibit-idle-scavenging-flag** *after* calling **gc-on** and must reset it after warm booting.

9.1.14 tv:set-default-font not supported

The undocumented function **tv:set-default-font** is not supported and should not be used. The purpose of this function is to change the default font used by the system from **fonts:cptfont** to something else. However, this change requires changes in window geometry, and many programs are not prepared to handle the new geometry. Calling **tv:set-default-font** is likely to break these programs. This problem will be addressed in a future release.

9.1.15 Avoid Errors in the Mouse Process

It is not a good idea to perform lengthy or error-prone calculations in the mouse process. An error in the mouse process might make it necessary to reset that process. If you have a complex calculation to perform in a **:mouse-click** method, use **process-run-function** to spin off a new process for the calculation, or send a blip to the window's process and perform the calculation there.

9.1.16 nil not a valid menu item

nil is not a valid menu item. If an item list contains **nil**, it is removed from the list. A menu with no choices at all appears as a small blank menu.

You might be tempted to specify **nil** as a menu item in a form like **(tv:menu-choose '(t nil))**. But this causes a menu with only one choice, **t**, to appear. Instead, you should specify a form like **(tv:menu-choose '(("Yes" . t) ("No" . nil)))**. This presents a menu with two choices: "Yes", which returns **t**, and "No", which returns **nil**. You could also specify **(tv:menu-choose '(("T" . t) ("NIL" . nil)))**. See the document *Window System Choice Facilities*.

10. Release 5.0: Operations and Site Management

10.1 Notes on Operations in Release 5.0

10.1.1 Backup Tape Reliability

To ensure the reliability of backup tapes, it is important to use cartridge tapes of adequate density and to compare the contents of backup tapes with the contents of the original files.

Cartridge tape used on the 3600 should be certified at a minimum of 10000 BPI. Some customers have been using lower density tapes and producing unreliable backups. You can order tapes through your local Symbolics sales office.

It is important to keep the tape heads clean when reading and writing tapes. The heads should be cleaned after every two hours of new cartridge tape use and after every 8 hours of old tape use. "Old tape" is tape that has been written or read more than two hours. Inmac offers a cleaning kit called the "Clean Cycle Kit, Quarter Inch Cartridge Head Cleaner", P/N 7148. The Inmac telephone number is (408) 727-1970.

After writing a backup, press SELECT F, click left on [Reload/Retrieve], and click left on the "Compare" operation in the choose-variable-values window. The "Compare" operation provides a bit-by-bit comparison of the backup tape to the original file. If it reveals discrepancies, you should take a consolidated dump. A consolidated dump dumps all files saved after a specified date. To take a consolidated dump: Click left on [Complete Dump], click left on the "Consolidated" dump type in the choose-variable-values window, and specify the date in the "Consolidate from" field to be the date of your last successful dump.

See the section "Reloading and Retrieving".

10.1.2 Site Configuration for Dialnet

This is the basic information on how to configure your site for Dialnet. It allows the generic network system access to the Vadic autodialer.

Configuring a site:

You should define a network named "DIAL" of type DIAL. This network represents the international dial network. Addresses on this network are the telephone numbers, relative to your site, of the host in question. This means that they should contain any prefixes or area codes that are necessary to dial that number. Add addresses on this network to any hosts you wish to access via telephone.

You must also define what services each host supports. The basic system supports only the LOGIN service. Add the service triple "LOGIN DIAL TTY-LOGIN" to any hosts which support login over dialup lines.

To allow a 3600 or LM-2 to use an autodialing modem with the general network system you must describe its connection with a PERIPHERAL option. For each autodial modem add a peripheral entry of type MODEM which specifies the modem model and the serial port to which it is connected. For example:

```
MODEM MODEL VA3451 UNIT 3
```

Index

B

C

E E E

F **F** **F**

G G G

H

J J J

K K K

L

M M M

N N N

P P P

Q

R

S

S

S

T

U U U

V V V

W W W