symbolics™

Volume 2
System Fundamentals

Volume 2. System Fundamentals

#996020

Design: Schafer/LaCasse
Cover and title page typography: Litho Composition Co.
Text typography: Century Schoolbook and Helvetica produced on a Symbolics 3600 Lisp Machine from Bitstream, Inc., outlines; text master printed on Symbolics LGP-1 Laser Graphics Printer.

The first Lisp Machine System was a product of the efforts of many people at the M.I.T. Artificial Intelligence Laboratory, and of the unique environment there. Portions of earliest versions of many of the documents in this documentation set were written at the AI Lab.

symbolics™

Contents

System Fundamentals

symbolics™

Documentation Map

1 System Index

TOC Table of Contents

INDEX Index

RN Release Notes/ Patch Notes

NEWS Newsletters/ Bug Reports

2 System Fundamentals

NOTA Notation Conventions

LMS Lisp Machine Summary 3600 Edition

3600 Notes on the 3600 for LM-2 Users

INED Using the Input Editor

MISCF Miscellaneous Useful Functions

3 Lisp Language

PRIM Primitive Object Types

EVAL Evaluation

FLOW Flow of Control

ARR Arrays and Strings

FUNC Functions

MAC Macros

DEFS Defstruct

FLAV Objects, Message Passing, and Flavors

COND Conditions

PKG Packages

4 Program Development Tools

TOOLS Program Development Tools and Techniques

HELP Program Development Help Facilities

ZMACS Zmacs Manual

DEBUG Debugger

MAINT Maintaining Large Systems

COMP The Compiler

MISCT Other Tools

5 User Interface Support

WINDOC Using the Window System

WINDEX Window System Program Examples

MENUS Window System Choice Facilities

SCROLL Scroll Windows

MISCUI Miscellaneous Functions

6 Utilities and Applications

ZMAILT Zmail Tutorial and Reference Manual

ZMAILC Zmail Concepts and Techniques

FED Font Editor

HARD Hardcopy System

CONV Converse

FSED FSEdit

MISCU Other Utilities and Applications

7 Networks and I/O

STR Streams

FILE Files

NETIO Networks and Peripherals

PROT Networks and Protocols

8 System Installation, Maintenance, Programming

SIG Software Installation Guide

SITE Site Operations

TAPE Tape

STOR Storage Management

PROC Processes

INIT Initializations

INT Internals

Map to the New Documentation System

The documentation in this eight-volume set includes all previously published Lisp Machine documentation, reorganized by topics and intended use of the information. The most obvious aspects of the reorganization are:

- The *Lisp Machine Manual* has been taken apart, and its various chapters are now scattered throughout the new system.
- Release Notes and Patch Notes through Release 5.0, which had previously been bound separately, have been merged into their relevant sources.

Following is a mapping of old to new documents, listed in alphabetic order by old document title:

Old title	**New title**	**Mnemonic**	**Volume**
Chaosnet	*Networks and Peripherals*	NETIO	7
Chaosnet File Protocol	*Networks and Protocols*	PROT	7
Font Editor	*Font Editor*	FED	6
Front-End Processor	*Networks and Peripherals*	NETIO	7
Introduction to Using the Window System	*Using the Window System*	WINDOC	5
Lisp Machine Choice Facilities	*Window System Choice Facilities*	MENUS	5
Lisp Machine Manual	[See page 4.]		
Lisp Machine Summary 3600 Edition	*Lisp Machine Summary 3600 Edition*	LMS	2
LM-2 Serial I/O	*Networks and Peripherals*	NETIO	7
LM-2 UNIBUS I/O	*Networks and Peripherals*	NETIO	7
Notes on the 3600 for LM-2 Users	*Notes on the 3600 for LM-2 Users*	3600	2
Operating the Lisp Machine	[Discontinued.]		
Program Development Help Facilities	*Program Development Help Facilities*	HELP	4

Old title	New title	Mnemonic	Volume
Program Development Tools and Techniques	*Program Development Tools and Techniques*	TOOLS	4
Release Notes for System 78	[Merged into related documents.]		
Release 4.0 Release Notes	[Merged into related documents.]		
Release 4.1 Patch Notes	[Merged into related documents.]		
Release 4.2 Patch Notes	[Merged into related documents.]		
Release 4.3 Patch Notes	[Merged into related documents.]		
Release 4.4 Patch Notes	[Merged into related documents.]		
Release 4.5 Patch Notes	[Merged into related documents.]		
Scroll Windows	*Scroll Windows*	SCROLL	5
Signalling and Handling Conditions	*Conditions*	COND	3
Software Installation Guide	*Software Installation Guide*	SIG	8
Symbolics File System	*Files*	FILE	7
System 210 Release Notes	[Merged into related documents.]		
Window System Program Examples	*Window System Program Examples*	WINDEX	5
Zmail Concepts and Techniques	*Zmail Concepts and Techniques*	ZMAILC	5
Zmail Tutorial and Reference Manual	*Zmail Tutorial and Reference Manual*	ZMAILT	5
Zmacs Manual	*Zmacs Manual*	ZMACS	4

Lisp Machine Manual

[Has been separated, by chapter, into the following documents:]

Old chapter title	Pages	New document title	Mnemonic	Volume
1. Introduction	1-6	*Notation Conventions*	NOTA	2
2. Primitive Object Types	7-12	*Primitive Object Types*	PRIM	3
3. Evaluation	13-32	*Evaluation*	EVAL	3
4. Flow of Control	33-51	*Flow of Control*	FLOW	3
5. Manipulating List Structure	52-85	*Primitive Object Types*	PRIM	3
6. Symbols	86-91	*Primitive Object Types*	PRIM	3
7. Numbers	92-106	*Primitive Object Types*	PRIM	3
8. Arrays	107-125	*Arrays and Strings*	ARR	3
9. Strings	126-135	*Arrays and Strings*	ARR	3
10. Functions	136-157	*Functions*	FUNC	3
11. Closures	158-162	*Functions*	FUNC	3
12. Stack Groups	163-169	*Internals*	INT	8
13. Locatives	170-171	*Primitive Object Types*	PRIM	3
14. Subprimitives	172-191	*Internals*	INT	8
15. Areas	192-196	*Storage Management*	STOR	8
16. The Compiler	197-207	*The Compiler*	COMP	4
17. Macros	208-232	*Macros*	MAC	3
18. The LOOP Iteration Macro	233-256	*Flow of Control*	FLOW	3

Old chapter title	Pages	New document title	Mnemonic	Volume
19. Defstruct	257-278	*Defstruct*	DEFS	3
20. Objects, Message Passing, and Flavors	279-313	*Objects, Message Passing, and Flavors*	FLAV	3
21. The I/O System				
21.1	314-318	*Streams*	STR	7
21.2	319-331	*Primitive Object Types*	PRIM	3
21.3-21.10	331-375	*Streams*	STR	7
22. Naming of Files	376-391	*Files*	FILE	7
23. Packages	392-405	*Packages*	PKG	3
24. Maintaining Large Systems				
24.1-24.7	406-421	*Maintaining Large Systems*	MAINT	4
24.8	422-427	*Site Operations*	SITE	8
25. Processes	428-439	*Processes*	PROC	8
26. Errors and Debugging				
26.1	440-450	*Conditions*	COND	3
26.2-26.8	450-468	*Debugger*	DEBUG	4
27. How to Read Assembly Language	469-486	*Internals*	INT	8
28. Querying the User	487-489	*Miscellaneous Functions*	MISCUI	5
29. Initializations	490-492	*Initializations*	INIT	8
30. Dates and Times	493-498	*Miscellaneous Functions*	MISCUI	5
31. Miscellaneous Useful Functions				
31.1-31.3	499-504	*Miscellaneous Useful Functions*	MISCF	2
31.4	505	*Storage Management*	STOR	8
31.5-31.7	506-508	*Miscellaneous Useful Functions*	MISCF	2

symbolics™

NOTA Notation Conventions

Cambridge, Massachusetts

Notation Conventions

# 990079

February 1984

This document corresponds to Release 5.0.

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

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

Table of Contents

1. Understanding Notation Conventions

You should understand several notation conventions before reading the documentation set.

The symbol "=>" indicates Lisp evaluation in examples. Thus, when you see "**foo** => **nil**", this means the same thing as "the result of evaluating **foo** is (or would have been) **nil**".

The symbol "==>" indicates macro expansion in examples. Thus, "**(foo bar)** ==> **(aref bar 0)**" means the same thing as "the result of macro-expanding **(foo bar)** is (or would have been) **(aref bar 0)**".

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3 (arg4* **(foo 3)***)* *Function*

The **function-name** function adds together *arg1* and *arg2*, and then multiplies the result by *arg3*. If *arg3* is not provided, the multiplication isn't done. **function-name** then returns a list whose first element is this result and whose second element is *arg4*. Examples:

```
(function-name 3 4) => (7 4)
(function-name 1 2 2 'bar) => (6 bar)
```

Note the use of fonts (typefaces). The name of the function is in boldface in the first line of the description, and the arguments are in italics. Within the text, printed representations of Lisp objects are in the same boldface font, such as **(+ foo 56)**, and argument references are italicized, such as *arg1* and *arg2*. A different, fixed-width font, such as `function-name`, is used for Lisp examples that are set off from the text, as well as to indicate user input.

The word "&optional" in the list of arguments tells you that all of the arguments past this point are optional. The default value can be specified explicitly, as with *arg4*, whose default value is the result of evaluating the form **(foo 3)**. If no default value is specified, it is the symbol **nil**. This syntax is used in lambda-lists in the language. (For more information on lambda-lists: See the section "Functions: Evaluation".) Argument lists may also contain "&rest", which is part of the same syntax.

The descriptions of special forms and macros look like this:

do-three-times *form* *Special Form*

This evaluates *form* three times and returns the result of the third evaluation.

with-foo-bound-to-nil *form...* *Macro*

This evaluates the *forms* with the symbol **foo** bound to **nil**. It expands as follows:

```
(with-foo-bound-to-nil
    form1
    form2 ...) ==>
(let ((foo nil))
    form1
    form2 ...)
```

Since special forms and macros are the mechanism by which the syntax of Lisp is extended, their descriptions must describe both their syntax and their semantics; unlike functions, which follow a simple consistent set of rules, each special form is idiosyncratic. The syntax is displayed on the first line of the description using the following conventions. Italicized words are names of parts of the form that are referred to in the descriptive text. They are not arguments, even though they resemble the italicized words in the first line of a function description. Parentheses ("( )") stand for themselves. Square brackets ("[ ]") indicate that what they enclose is optional. Ellipses ("...") indicate that the subform (italicized word or parenthesized list) that precedes them may be repeated any number of times (possibly no times at all). Curly brackets followed by ellipses ("{ }...") indicate that what they enclose may be repeated any number of times. Thus, the first line of the description of a special form is a "template" for what an instance of that special form would look like, with the surrounding parentheses removed. The syntax of some special forms is sufficiently complicated that it does not fit comfortably into this style; the first line of the description of such a special form contains only the name, and the syntax is given by example in the body of the description.

The semantics of a special form includes not only what it "does for a living", but also which subforms are evaluated and what the returned value is. Usually this is clarified with one or more examples.

A convention used by many special forms is that all of their subforms after the first few are described as "*body...*". This means that the remaining subforms constitute the "body" of this special form; they are Lisp forms that are evaluated one after another in some environment established by the special form.

This imaginary special form exhibits all of the syntactic features:

twiddle-frob *[(frob option...)] {parameter value}...* *Special Form*

This twiddles the parameters of *frob*, which defaults to **default-frob** if not specified. Each *parameter* is the name of one of the adjustable parameters of a frob; each *value* is what value to set that parameter to. Any number of *parameter/value* pairs may be specified. If any *options* are specified, they are keywords that select which safety checks to override while twiddling the parameters. If neither *frob* nor any *options* are specified, the list of them may be omitted and the form may begin directly with the first *parameter* name.

frob and the *values* are evaluated; the *parameters* and *options* are syntactic keywords and are not evaluated. The returned value is the frob whose parameters were adjusted. An error is signalled if any safety checks are violated.

Methods, the message-passing equivalent of ordinary Lisp's functions, are described in this style:

message-name *arg1 arg2* &optional *arg3* (of **flavor-name**) *Method*

This is the documentation of the effect of sending a message named **message-name**, with arguments *arg1*, *arg2*, and *arg3*, to an instance of flavor **flavor-name**.

Descriptions of variables ("special" or "global" variables) look like this:

typical-variable *Variable*

The variable **typical-variable** has a typical value....

Most numbers shown are in octal radix (base eight). Spelled out numbers and numbers followed by a decimal point are in decimal. This is because, by default, Zetalisp types out numbers in base 8; do not be surprised by this. If you wish to change it: See the section "What the Reader Accepts".

All uses of the phrase "Lisp reader", unless further qualified, refer to the part of Lisp that reads characters from I/O streams (the **read** function), and not the person reading this documentation.

Several terms that are used widely in other references on Lisp are not used much in this document set, as they have become largely obsolete and misleading. For the benefit of those who may have seen them before, they are: "S-expression", which means a Lisp object; "Dotted pair", which means a cons; and "Atom", which means, roughly, symbols and numbers and sometimes other things, but not conses. For definitions of the terms "list" and "tree": See the section "Manipulating List Structure".

The characters acute accent (') (also called the single quote character) and semicolon (;) have special meanings when typed to Lisp; they are examples of what are called *macro characters*. Though the mechanism of macro characters is not of immediate interest to the new user, it is important to understand the effect of these two, which are used in the examples.

When the Lisp reader encounters a single quote, it reads in the next Lisp object and encloses it in a **quote** special form. That is, **'foo-symbol** turns into **(quote foo-symbol)**, and **'(cons 'a 'b)** turns into **(quote (cons (quote a) (quote b)))**. The reason for this is that "**quote**" would otherwise have to be typed in very frequently and would look ugly.

The semicolon is used as a commenting character. When the Lisp reader sees one, the remainder of the line is discarded.

The character "/" is used for quoting strange characters so that they are not interpreted in their usual way by the Lisp reader, but rather are treated the way normal alphabetic characters are treated. So, for example, in order to give a "/" to the reader, you must type "//", the first "/" quoting the second one. When a character is preceded by a "/" it is said to be *slashified*. Slashifying also turns off the effects of macro characters such as single quote and semicolon.

The following characters also have special meanings, and may not be used in symbols without slashification. These characters are explained in detail elsewhere: See the section "Printed Representation".

"	Double-quote delimits character strings.
#	Number-sign introduces miscellaneous reader macros.
‘	Backquote is used to construct list structure.
,	Comma is used in conjunction with backquote.
:	Colon is the package prefix.
|	Characters between pairs of vertical-bars are quoted.
⊗	Circle-cross lets you type in characters using their octal codes.

All Lisp code in this document set is written in lowercase. In fact, the reader turns all symbols into uppercase, and consequently everything prints out in uppercase. You may write programs in whichever case you prefer.

Various symbols have the colon (:) character in their names. By convention, all *keyword symbols* in the Lisp Machine system have names starting with a colon. The colon character is not actually part of the print name, but is a package prefix indicating that the symbol belongs to the package with a null name, which means the **keyword** package. (For more information on colons: See the document *Packages*. Until you read that document, just pretend that the colons are part of the names of the symbols.)

The document set describes a number of internal functions and variables, which can be identified by the "**si:**" prefix in their names. The "**si**" stands for "**system-internals**". These functions and variables are documented because they are things you sometimes need to know about. However, they are considered internal to the system and their behavior is not as guaranteed as that of everything else.

Zetalisp is descended from Maclisp, and a good deal of effort was expended to try to allow Maclisp programs to run in Zetalisp. Throughout the documentation, there are notes about differences between the dialects. For the new user, it is important to note that some functions herein exist solely for Maclisp compatibility; they should *not* be used in new programs. Such functions are clearly marked in the text.

The Lisp Machine character set is not the same as the ASCII character set used by most operating systems. For more information: See the section "The Character Set". The important thing to note for now is that the Newline character is the same as Return, and is represented by the number 215 octal. (This number should *not* be built into any programs.) Unlike ASCII, there are no "control characters" in the character set; Control and Meta are merely things that can be typed on the keyboard.

Many of the functions refer to "areas". The *area* feature is of interest only to writers of large systems, and can be safely disregarded by the casual user. For more information: See the document *Storage Management*.

2. Notation Conventions Quick Reference

Modifier keys are designed to be held down while pressing other keys. They do not themselves transmit characters. A combined keystroke like META-X is pronounced "meta x" and written as m-X. This notation means press the META key and, while holding it down, press the X key.

Modifier keys are abbreviated as follows:

Key	*Abbreviation*
CTRL	c-
META	m-
SUPER	s-
HYPER	h-
SHIFT	sh-
SYMBOL	sy-

The keys with white lettering (like X or SELECT) all transmit characters. Combinations of these keys are meant to be pressed in sequence, one after the other. This sequence is written as, for example, SELECT L. This notation means press the SELECT key, release it, and then press the L key.

This document set uses the following notation conventions:

Appearance in document	*Representing*
send, chaos:host-up	Printed representation of Lisp objects in running text.
RETURN, ABORT, c-F	Keyboard keys.
SPACE	Space bar.
`login`	Literal type-in.
`(make-symbol "foo")`	Lisp code examples.
(**function-name** *arg1* <u>*arg2*</u>)	Syntax description of the invocation of **function-name**.
arg1	Argument to the function **function-name**, usually expressed as a word that reflects the type of argument (for example, *string*).
<u>*arg2*</u>	Optional argument; you can leave it out.
Undo, Reply, Start	Command names in Zmacs, Zmail, and the front-end processor (FEP) appear with the initial letter of each word capitalized.
Insert File (m-X)	Extended command names in Zmacs and Zmail. Use m-X to invoke one.
[Map Over]	Menu items.
(L), (R2)	Mouse clicks: L=left, L2=click sh-left, M=middle, M2=click sh-middle, R=right, R2=click sh-right. (You can also double click on a key rather than pressing the SHIFT key while clicking on it.)

The following conventions are used to represent mouse actions:

1. Square brackets delimit a mouse command.

2. Slashes (/) separate the members of a compound mouse command.

3. The standard clicking pattern is as follows:

   - For a single menu item, always click left. For example, the following two commands are exactly the same:

     [Previous]
     [Previous (L)]

     For a compound command, always click right on each menu item except the last, where you click left. For example, the following two compound commands are exactly the same:

     [Map Over / Move / Hardcopy]
     [Map Over (R) / Move (R) / Hardcopy (L)]

4. When a command does not follow the standard clicking order, the notation for the command shows explicitly which button to click. For example:

   [Map Over / Move (M)]
   [Previous (R)]

Index

symbolics™

LMS Lisp Machine Summary 3600 Edition

Release 5.0 Update

Cambridge, Massachusetts

Lisp Machine Summary 3600 Edition: Release 5.0 Update

# 990075

February 1984

This document corresponds to Release 5.0.

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

Table of Contents

Release 5.0 Update

Updated Information for Release 5.0

Introduction

This section summarizes changes to the 3600 in Release 5.0.

New Information on Getting Started

As of FEP Version 16, >configuration.fep files are now called Boot.boot. Files with the type .fep are now reserved for files that the user should never modify.

See the section "Getting Started".

Change to login

The following examples illustrate several options for logging in, where `whit` is your login name. Type all punctuation and parentheses as shown. See the section "Getting Started".

- to log in to the default host machine, using your init file, type `(login 'whit)`
- to log in to the default host machine, without your init file, type `(login 'whit :load-init-file nil)`
- to log in to another host machine "sc3", using your init file, type `(login 'whit :host 'sc3)`

See the function **login**.

Preferred Way to Warm Boot or Halt

The preferred way to halt or begin a warm boot of the 3600 is now **si:halt**, not **si:%halt**.

You can also press `h-c-FUNCTION` instead of `c-FUNCTION` to get to the FEP from Lisp. However, **si:halt** is a better way to stop Lisp than `h-c-FUNCTION` because `h-c-FUNCTION` could interrupt disk I/O operations.

See the section "Warm Booting". See the section "Halting". See the function **si:halt**.

Complementing the Mouse Documentation Line

Press `FUNCTION m-C` to complement the mouse documentation line. Formerly, you pressed `FUNCTION 1 C`.

See the section "Mouse Documentation Line".

Updated Information for Release 5.0, *cont'd.*

Changes to the Keyboard in Release 5.0

Some 3600 keys, the characters they generate, and their functions have changed in Release 5.0. For details: See the section "Hardware Changes".

New SELECT Key Options

SELECT T invokes a new terminal program that replaces the former Telnet and Supdup programs (that were invoked by SELECT T and SELECT S, respectively). See the section "New Terminal Program (SELECT T)".

SELECT X makes the Flavor Examiner available. See the section "Flavor Examiner".

New Terminal Program SELECT T

The new terminal program incorporates the functions of the former Telnet and Supdup programs. It is available on SELECT T. Since 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. (Naming of hosts, setting up host databases, declaring host addresses and supported login services are covered in the new network documentation.) The network system picks the best login service supported by the host and the optimum route to it. The specification of a particular gateway and special contact name or port using ◊ and / is gone. Such control arguments and new higher-level ones (such as a particular protocol to use, rather than the default) are naturally the province of a command processor and will be added to the terminal program when the system includes a command processor. Pressing HELP in response to the initial prompt gives you input editor documentation.

Once connected, commands are given by pressing NETWORK and another single character.

The following commands are available:

A	send an ATTN (in Telnet, a new Telnet "Interrupt Process").
D	Disconnect.
L	Log out of remote host, and break the connection.
Q	Disconnect and deselect this window. (Quit)

Updated Information for Release 5.0, *cont'd.*

M	Toggle MORE processing.

More complicated commands are entered with the extended command, `NETWORK X`. This command would use a command processor; in the interim (Release 5.0), this command uses a choose variable values window, one of whose variables allows you to control overstrike processing (formerly available as `NETWORK O`).

`NETWORK X` provides the capability to 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. There is also a facility of logging host output to a file (wallpaper).

It is no longer possible to type `NETWORK` to the `Connect to host:` prompt to change things before connecting. Again, this deficiency will be made up by providing full command processing.

Index of Function Keys: Lisp Machine Summary 3600 Edition

Introduction

This is a quick reference guide to the 3600's function keys. It supersedes a similar guide in the former document, *Operating the Lisp Machine*.

For more information on keys: See the section "Hardware Changes".

ABORT

When this is read by a program, the program aborts what it is doing and returns to its "command loop". Lisp Listeners, for example, respond to ABORT by throwing back to the read-eval-print loop (top level or **break**). Note that ABORT takes effect when it is read, not when it is pressed; it will not stop a running program.

c-ABORT

Aborts the operation currently being performed by the process you are typing at, immediately (not when it is read). For instance, this will force a Lisp Listener to abandon the present computation and return to its read-eval-print loop.

m-ABORT

When this is read by a program, the program aborts what it is doing and returns through all levels of commands to its "top level". Lisp Listeners, for example, throw completely out of their computation, including any **break** levels, then start a new read-eval-print loop.

c-m-ABORT

A combination of c-ABORT and m-ABORT, this immediately throws out of all levels of computation and restarts the process you type at it.

BACKSPACE

Moves the cursor back so that you can superpose two characters, should you really want to.

CLEAR-INPUT

Usually flushes the input expression you are typing.

COMPLETE

Completes partially typed commands.

Index of Function Keys: Lisp Machine Summary 3600 Edition, *cont'd.*

END

Marks the end of input to many programs. Single-line input may be ended with RETURN, but END will terminate multiple-line input where RETURN is useful for separating lines. The END key does not apply when typing in Lisp expressions, which are self-delimiting. END terminates input you have edited: See the document *Using the Input Editor*.

ESCAPE

Displays the input editor history. c-ESCAPE displays the global kill history. Sends Escape/Altmode (octal 033) in the Terminal program.

FUNCTION Key: Display and Hardcopy Commands

This key is a prefix for a family of commands relating to the display, which you may type at any time, no matter what program you are running. The FUNCTION commands that control screen display and hardcopying are:

RUBOUT	Does nothing; press this key to cancel FUNCTION if you typed the latter by accident.
CLEAR-INPUT	Discards typeahead.
REFRESH	Clears and redisplays all windows.
A	Arrests the process shown in the status line. FUNCTION - A resumes the process.
B	Buries the currently selected window, if any -- that is, it moves it underneath all other windows. This usually brings up some other window, which is automatically selected.
C	Complements the entire screen. An argument of 1 means white-on-black; an argument of 0 means black-on-white.
c-C	Complements the selected window, with the same argument as FUNCTION C.
m-C	Complements the mouse documentation line, with the same argument as FUNCTION C.
F	Shows users logged in on the associated machine. With numeric arguments, it shows users logged in on various machines.
H	Shows status of network hosts. With an argument, it prompts for hosts.

[illegible] toggles

Index of Function Keys: Lisp Machine Summary 3600 Edition, *cont'd.*

M	Controls global MORE processing. No argument means toggle, 0 means turn off, 1 means turn on.
c-M	Controls MORE processing for the selected window. The arguments are the same as for FUNCTION M.
O	Selects another exposed window.
Q	Hardcopies the entire screen.
c-Q	Hardcopies the selected window.
m-Q	Hardcopies the entire screen, minus the status and mouse documentation lines.

FUNCTION Key: Selection and Notification Commands

The FUNCTION commands that control window selection and notification are:

S Selects the most recently selected window. With an argument *n* (default is 2), it selects the *n*th previously selected window and rotates the top *n* windows. An argument of 1 rotates through all windows (a negative argument rotates in the other direction); 0 selects a window that requires attention (for example, to report an error).

T Controls the selected window's input and output notification characteristics. If an attempt is made to output to a window when it is not exposed, one of three things can happen: the program can simply wait until the window is exposed, it can send a notification that it wants to type out and then wait, or it can quietly type out "in the background"; when the window is next exposed the output will become visible. Similarly, if an attempt is made to read input from a window that is not selected (and has no typed-ahead input in it), the program can either wait for the window to become selected, or send a notification that it wants input and then wait.

The FUNCTION T command controls these characteristics based on its numeric argument, as follows:

no argument	If output notification is off, turns input and output notification on. Otherwise turns input and output notification off. This essentially toggles the current state.
0	Turns input and output notification off.

Index of Function Keys: Lisp Machine Summary 3600 Edition, *cont'd.*

	1	Turns input and output notification on.
	2	Turns output notification on, and input notification off.
	3	Turns output notification off, and input notification on.
	4	Allows output to proceed in the background, and turns input notification on.
	5	Allows output to proceed in the background, and turns input notification off.
W	Controls the status line. With no argument, the status line is redisplayed. The numeric arguments control what process the status line watches. The options are:	
	0	Gives a menu of all processes, and freezes the status line on the process you select. When the status line is frozen on a process, the name of that process appears where your user ID normally would (next to the date and time), and the status line does not change to another process when you select a new window.
	1	The status line watches whatever process is talking to the keyboard, and changes processes when you select a new window. This is the default initial state.
	2	Changes the status line so that it displays the name of the process instead of the name of the user. This also freezes the status line on that process; normally the status line switches to display a different process whenever the window system tells it to.
		Use this if you see an unexpected state in the status line. It will help you find out what process is in that state; you may find that you are not talking to the process you think you should be.

Index of Function Keys: Lisp Machine Summary 3600 Edition, *cont'd.*

3	Rotates the status line among all processes.
4	Rotates the status line in the other direction.

FUNCTION Key: Recovering From Stuck States

The following FUNCTION commands should all be used with caution.

ESCAPE	Helps you recover from stuck states such as "Output Hold" and "Sheet Lock".
c-A	Arrests all processes except the one shown in the status line and critical system processes, such as the keyboard and mouse processes. FUNCTION - c-A resumes all processes arrested by this command.
SUSPEND	Gets to the cold-load stream.
c-T	Deexposes temporary windows. This is useful if the system seems to be hung because there is a temporary window on top of the window that is trying to type out.
c-CLEAR-INPUT	Clears window system locks. This is a last resort, although not as drastic as warm booting. Use this when none of the windows will talk to you, when you cannot get a System menu, and so on.

HELP

Usually gets you some online documentation or programmed assistance.

LINE

The function of this key varies considerably. It is used as a command by the Debugger, and sends a line feed character in the Terminal program.

NETWORK

This key is used to get the attention of the Terminal program. As such it functions as a command prefix. You must be connected to a host via the Terminal program before you can use this key.

Index of Function Keys: Lisp Machine Summary 3600 Edition, *cont'd.*

PAGE	In Zmacs (in searches and after c-Q) this key inserts a page separator character, which displays as "page" in a box.
REFRESH	Usually erases and refreshes the selected window. In Zmacs (in searches and after c-Q) this key inserts a page separator character, which displays as "page" in a box.
RESUME	Continues from the **break** function and the Debugger. In the Terminal program this sends a backspace character.
RETURN	"Carriage return" or end of line. Exact significance may vary.
RUBOUT	Usually erases the last character typed.
SELECT	This key is a prefix for a family of commands, generally used to select a window of a specified type, such as a Lisp Listener or Zmail. For more information, press SELECT HELP at any window.
SUSPEND	Usually forces the process you are typing at into a **break** read-eval-print loop, so that you can see what the process is doing, or stop it temporarily. The effect occurs when the character is read, not immediately. Press RESUME to continue the interrupted computation (this applies to the three modified forms of the SUSPEND key as well).
c-SUSPEND	This is like SUSPEND, but takes effect immediately rather than when it is read.
m-SUSPEND	Forces the process you type it at into the Debugger when it is read. It should type out ">>BREAK" and the Debugger prompt "→". You can poke around in the process, then press RESUME or c-C to continue.

Index of Function Keys: Lisp Machine Summary 3600 Edition, *cont'd.*

`c-m-SUSPEND`

Forces the process you type it at into the Debugger, whether or not it is running.

`SYMBOL`

Acts as a modifier key to produce special characters. Pressing `sym-HELP` produces a display of special function and special character keys.

`TAB`

This key is only sometimes defined. Its exact function depends on context, but in general it is used to move the cursor right to an appropriate point.

Keys Not Currently Used

The following keys currently have no function:
`SCROLL`
`MODE-LOCK`
`REPEAT`

The following keys are reserved for use by the user (for example, to put custom editor commands or keyboard macros on):
`CIRCLE`
`SQUARE`
`TRIANGLE`

The following key is reserved for functions local to the console:
`LOCAL`

For more information: See the section "Keys Not Used by Standard Software".

Index

symbolics™

3600 Notes on the 3600 for LM-2 Users

Cambridge, Massachusetts

Notes on the 3600 for LM-2 Users

# 990105

February 1984

This document corresponds to Release 5.0.

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

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

Table of Contents

List of Tables

1. Introduction to the 3600

Internally, your Symbolics 3600 is very unlike a Symbolics LM-2; the hardware is completely new, the instruction set is different, and most of the operating system and Lisp system software have been developed recently. However, the 3600 reimplements the same user and software environment. If you are familiar with the operation and programming of the LM-2, using the 3600 is easy, because in almost all these respects it is just like an LM-2.

This document presents some of the differences between the two systems. Some of these differences are fundamental. Others are present because your 3600 is a new machine and some aspects of development are unfinished; differences of this latter sort will disappear over time as you receive new software updates (and possibly hardware updates in a few cases).

1.1 Performance

The 3600 architecture is designed to optimize different aspects of performance than the LM-2 architecture. Because much less of the 3600 system is written in microcode, system performance depends more on measurement and tuning than on the extensive use of microcode.

The 3600 system has not yet received much tuning and performance optimization, except in the low-level architectural design. Therefore:

- Programs whose performance is limited by the architectural limitations of the LM-2, such as the slow function call or the slow fixnum arithmetic, run faster on the 3600.
- Programs whose performance is limited by higher-level features of the system might run at the same speed on the 3600 as on the LM-2; in some cases they might run substantially slower. For example, line drawing, implemented in microcode on the LM-2, is written in Lisp on the 3600. Therefore, the QIX program, available on the **hacks:demo** menu, runs more slowly on the 3600, as it spends most of its time drawing lines.

Some areas of known lack of tuning include storage allocation, compiler optimizations, the compiler itself, paging algorithms, the network, the primitives for string operations, graphics primitives, the process scheduler, and real-time tracking of the mouse.

2. Hardware Changes

2.1 The 3600 Keyboard

The most obvious difference you will notice between machines is the keyboard. The 3600 keyboard has fewer keys; however, many of the keys removed were not actually used on the LM-2. In Release 5.0, system software performs a mapping of 3600 keys into equivalent LM-2 keys.

Substantial changes have been made to the arrangement of the 3600 keyboard in Release 5; for example, some keys have been deleted, others perform new or different functions, some keytops have been replaced.

2.1.1 LM-2 Keys That Do Not Exist on the 3600

The following keys do not exist on the 3600. In Release 5.0, these keys are not used by the software, so they are not missed on the 3600.

Some of these keys are explained in greater detail below.

The four Roman numeral keys
The four "hand" keys
ALT-LOCK
CALL
MACRO
QUOTE
STOP-OUTPUT
HOLD-OUTPUT
STATUS
DELETE
ALTMODE
GREEK

Key	*LM-2*	*3600*
CALL	Use TERMINAL BREAK to get to the the cold-load stream.	Use FUNCTION SUSPEND to get to cold-load stream.
HOLD-OUTPUT	Use TERMINAL ALTMODE to request assistance when the window system is stuck.	Use FUNCTION ESCAPE to request assistance when the window system is stuck.
STATUS	STATUS displays the input editor history. c-STATUS displays the	ESCAPE displays the input editor history. c-ESCAPE displays the

	global kill history.	global kill history.
ALTMODE	Escape (labelled ALTMODE) is a synonym for the Complete character when both End and Complete are possible, and is a synonym for End otherwise (editor Search commands).	Use COMPLETE for command completion.
	Network remote-login software uses Altmode as 33.	Network remote-login software uses Escape as 33.
	The "specify new user name" response to the password query is on the ALTMODE key.	The "specify new user name" response to the password query is on the ESCAPE key.
MACRO	Used by keyboard-macro streams as a command prefix. This is different from c-X, because it works even in a noncommand context (for example, in the middle of an editor search string) and is different from FUNCTION, because it is associated with a particular stream, not asynchronous.	
	The Macro Escape character is still the MACRO key.	c-m-FUNCTION replaces MACRO in the editor and Zmail.
QUOTE	Not used significantly by the current software.	Not used significantly by the current software.
DELETE	Not used significantly by the standard software. DELETE is the Complete character.	

2.1.2 Renamed Keys

The following keys have been renamed for clarity:

LM-2	*3600*
TERMINAL	FUNCTION
SYSTEM	SELECT
BREAK	SUSPEND
TOP	SYMBOL
OVERSTRIKE	BACKSPACE
CLEAR-SCREEN	REFRESH

2.1.3 Keys Not Used by Standard Software

Key	*Description*
SCROLL	The SCROLL key is not currently used by the software. In a future release it might be used by the system as a command for scrolling text and other objects within windows.
CIRCLE	The CIRCLE key is reserved for user application programs.
SQUARE	The SQUARE key is reserved for user application programs.
TRIANGLE	The TRIANGLE key is reserved for user application programs.
LOCAL	The LOCAL key is not currently used by the software.
MODE-LOCK and REPEAT	The MODE-LOCK and REPEAT keys are not currently used by the software and generate no characters when pressed. You can sense whether they are depressed by calling **tv:key-state**. See the function **tv:key-state**.

2.1.4 No Audible Tone Yet

The audio output feature does not currently exist in the 3600, so the keyboard is not capable of making a *beep*.

2.1.5 Major Changes in the Keyboard

Table 1, page 6, summarizes the changes in the keyboard from Release 4 to Release 5 and the differences between the LM-2 and 3600 keyboards for each release.

Note that the character codes in the table are in octal.

2.1.5.1 Some Notes on the Table

ASCII code 33, which was formerly called Altmode, is now called the Lozenge character. The ESCAPE key (ALTMODE key on the LM-2) sends a different code (237) than it used to in Release 4. When accessing a host via SELECT T (SYSTEM T on the

Table 1. Comparison of the LM-2 and 3600 Keyboards in Release 4 and Release 5

Character Set and Keyboard Changes									
	Release 4				Release 5				
Character Code	LM-2 Primary Name	LM-2 Keycap Legend	3600 Keycap Legend	Other Names	LM-2 Primary Name	LM-2 Keycap Legend	3600 Primary Name	3600 Keycap Legend	Other Names
33	Altmode	ALTMODE	ESCAPE	Diamond Alt-mode Alt	Lozenge	TOP-ALTMODE	Lozenge	SYMBOL-ESCAPE	Diamond
201	BREAK	BREAK	SUSPEND	BRK	BREAK	BREAK	BREAK	SUSPEND	BRK
203	CALL	CALL	SQUARE	-	*Obsolete*				
205	MACRO	MACRO	-	BACKNEXT BACK-NEXT	MACRO	MACRO	-	-	BACKNEXT BACK-NEXT
210	Overstrike	OVERSTRIKE	BACKSPACE	BS Over-strike Backspace	Overstrike	OVERSTRIKE	Back-space	BACKSPACE	Backspace Over-strike BS
213	Delete	DELETE	-	VT Vertical-Tab	Clear-screen	CLEAR-SCREEN	Refresh	REFRESH	-
214	Page	CLEAR-SCREEN	PAGE	Form FF Form-Feed Formfeed	Page	TOP- CLEAR-SCREEN	Page	PAGE	Form FF Form-Feed Formfeed
216	QUOTE	QUOTE	-	-	*Obsolete*				
217	HOLD-OUTPUT	HOLD-OUTPUT	TRIANGLE	-	*Obsolete*				
220	STOP-OUTPUT	STOP-OUTPUT	-	-	*Obsolete*				
223	STATUS	STATUS	-	-	*Obsolete*				
225	Roman-I	I	-	Roman-1 Roman-One	Roman-I	I	Square	SQUARE	Roman-1 Roman-One
226	Roman-II	II	-	Roman-2 Roman-Two	Roman-II	II	Circle	CIRCLE	Roman-II Roman-Two
227	Roman-III	III	-	Roman-3 Roman-Three	Roman-III	III	Triangle	TRIANGLE	Roman-III Roman-Three
230	Roman-IV	IV	-	Roman-4	*Obsolete*				
231	Hand-Up	HAND-UP	-	-	*Obsolete*				
233	Hand-Left	HAND-LEFT	-	Left-Hand	*Obsolete*				
234	Hand-Right	HAND-RIGHT	-	Right-Hand	*Obsolete*				
232	Hand-Down	*Thumbdown*	-	Down-Hand	Hand-Down	*Thumbdown*	Scroll	SCROLL	Down-Hand
237	-	-	-	-	Altmode	ALTMODE	Escape	ESCAPE	Alt-mode Alt Esc
240	-	-	-	-	Complete	DELETE	Complete	COMPLETE	-
241	-	-	-	-	Top-Help	TOP-HELP	Symbol-Help	SYMBOL-HELP	-

LM-2), ESCAPE, COMPLETE, and ALTMODE (LM-2 only) all send the Lozenge character (33), which corresponds to the ASCII Escape character (Altmode on the LM-2). In the editor, Lozenge is generally treated as a printing character; END and ESCAPE (ALTMODE) are used interchangeably to terminate searches and the like; COMPLETE is not treated specially in searches.

Obsolete characters are being phased out of the user interface. LOCAL is used only by the FEP, the console, or both.

The character names Delete and Vt are invalid in Release 5. Code 213 now maps to Refresh (Clear-screen on the LM-2).

The LM-2 does not have separate keys for Page and Refresh. CLEAR-SCREEN generates the Refresh character, while TOP CLEAR-SCREEN generates the Page character. m-CLEAR-SCREEN inserts a Page character, as it did in Release 4 for different reasons. c-Q CLEAR-SCREEN inserted a Page character in Release 4, but in Release 5 inserts a Refresh character. Also, in the search commands you now have to press TOP CLEAR-SCREEN in order to search for a Page character. sh-CLEAR-SCREEN works as well here.

When the name of a key differs between machines, each machine outputs characters using its own key name but will accept either name on input. For example, typing `(format t "~:c" #o210)` to a 3600 produces `Back-Space`, whereas typing it to an LM-2 produces `Overstrike`. Both machines, however, accept `#\Back-Space` and `#\overstrike` as input. Various "historical" names are also accepted on input, for compatibility with previous systems, Maclisp, and so forth. However, we do not recommend that you use these names.

2.1.5.2 Summary of Changes to Character Names

Changes to 3600 (except where noted) since Release 4.

These character names have new character codes (given in octal):

Character	*Code Changed*	
	From	*To*
Square	203	225
Triangle	217	227
Escape	33	237
Esc	204	237
Clear-Screen[1]	214	213
Altmode	33	237
Alt-mode	33	237
Alt	33	237

[1]LM-2 only.

These characters have been deleted:

Delete
Vertical-Tab
Vt
Local

These are new character names:

Lozenge
Circle
Scroll
Complete
Symbol-Help

These character codes were unused in Release 4:

Code	*New character*
237	Escape (Altmode on the LM-2)
240	Complete (mislabelled Delete on the LM-2)
241	Symbol-Help (Top-Help on the LM-2)

2.1.6 Supdup Key Mappings

Table 2. Supdup Key Mappings on the LM-2 and 3600

Supdup character	*LM-2 key*	*3600 key*
Top-B (BRK)	BREAK, QUOTE, NETWORK	SUSPEND, NETWORK
Top-C (CLR)	CLEAR-INPUT	CLEAR-INPUT
Call	CALL, ABORT	ABORT
Top-A (ESC)	TERMINAL	FUNCTION
Backnext	MACRO, STOP-OUTPUT	SCROLL
Top-H (HELP)	HELP	HELP
Rubout	RUBOUT	RUBOUT
Backspace	BACKSPACE	BACKSPACE
Tab	TAB	TAB
LF	LINE	LINE
VT	*not typeable*	*not typeable*
FF	CLEAR-SCREEN	PAGE, REFRESH
CR	RETURN	RETURN
Control-S	HOLD-OUTPUT	c-S
Control-H	RESUME	RESUME
Control-◊	END	END
Top-1	I	SQUARE
Top-2	II	CIRCLE
Top-3	III	TRIANGLE
Top-4	IV	*not typeable*
Top-u	HAND-UP	*not typeable*
Top-d	HAND-DOWN	*not typeable*
Top-l	HAND-LEFT	*not typeable*
Top-r	HAND-RIGHT	*not typeable*
◊	ALTMODE	ESCAPE, COMPLETE
none	STATUS, SYSTEM	SELECT

2.1.7 Symbol Characters on the LM-2 and 3600

Table 3. Comparison of the Symbol Characters

Char.	*Name*	*Release 5 LM-2 key*	*Release 4 3600 key*	*Release 5 3600 key*[1]
·	Center-Dot	FRONT-’[2]	sy-sh-’	sy-’
↓	Down-Arrow	TOP-H	sy-H	
α	Alpha	GREEK-a	sy-sh-A	
β	Beta	GREEK-b	sy-sh-B	
∧	And-sign	TOP-Q	sy-Q	
¬	Not-sign	FRONT-‘	sy--	
ε	Epsilon	GREEK-e	sy-sh-E	
π	Pi	GREEK-p	sy-sh-P	
λ	Lambda	GREEK-l	sy-sh-L	
γ	Gamma	GREEK-g	sy-sh-G	
δ	Delta	GREEK-d	sy-sh-D	
↑	Up-Arrow	TOP-G	sy-G	
±	Plus-Minus	TOP-:[3]	sy-:	
⊕	Circle-Plus	FRONT-HAND-DOWN	sy-=/+	sy-+
∞	Infinity	TOP-I	sy-I	
∂	Partial-Delta	TOP-P	sy-P	
⊂	Left-Horseshoe	TOP-T	sy-T	
⊃	Right-Horseshoe	TOP-Y	sy-Y	
∩	Up-Horseshoe	TOP-E	sy-E	
∪	Down-Horseshoe	TOP-R	sy-R	
∀	Universal-Quantifier	TOP-U	sy-U	
∃	Existential-Quantifier	TOP-O	sy-O	
⊗	Circle-X	FRONT-HAND-LEFT	sy-8/*	sy-*
↔	Double-Arrow	TOP-L	sy-L	
←	Left-Arrow	TOP-J	sy-J	
→	Right-Arrow	TOP-K	sy-K	
≠	Not-Equals	TOP-C	sy-C	sy-=
◊	Lozenge	TOP-ALTMODE	COMPLETE	sy-ESCAPE
≤	Less-Or-Equal	TOP-N	sy-N	sy-,
≥	Greater-Or-Equal	TOP-M	sy-M	sy-.
≡	Equivalence	TOP-B	sy-B	sy-~
∨	Or-sign	TOP-W	sy-W	
∫	Integral	FRONT-/	*not typeable*	sy-/

[1] Blank lines indicate no change from Release 4 to Release 5. [2] FRONT is the GREEK key. [3] Use the : key next to the numeric 1 key.

2.2 Data Format

LM-2	*3600*
Fixnums are 24 bits long.	Fixnums are 32 bits long.
Word addresses are 24 bits long.	Word addresses are 28 bits long.

On the 3600 the high-order 4 bits of fixnums and flonums overlap the type-code field. Programs that use subprimitives (for example, **%p-data-type**) to access the type code field must take account of this.

For a discussion of flonums: See the section "Floating-point Numbers".

2.3 Peripherals

Peripheral devices currently supported by the 3600 are:

- The console, including video screen, keyboard, and mouse — but not audio output
- The network (10-Mbit Ethernet)
- Disk drives
- The cartridge tape drive
- RS-232-compatible serial I/O devices

Tape I/O is supported via the remote tape protocol over the network. Printed output is supported via the network.

3. Software Changes

3.1 New Compiler

3.1.1 Introduction

The 3600 has a new instruction set and a completely new Lisp compiler to convert Lisp programs into this instruction set. The new compiler works the same way as the LM-2 compiler; you can use all the same commands to compile functions and files. When you invoke these commands on the 3600, the new compiler is used.

3.1.2 Incompatible Changes

3.1.2.1 Compiled Code File Types

The new compiler writes out a different kind of compiled file than the LM-2 compiler. The results of the 3600 compiler are written to a .bin file. For example, compiling the file test.lisp on an LM-2 creates test.qbin, whereas compiling it on the 3600 produces test.bin. This use of a different type field in the file pathname allows a program to be used on both the LM-2 and the 3600, even though neither machine can read the other machine's compiled files.

The canonical type for bin files is **:bin**. The following table summarizes the file types for compiled files on various hosts.

Host type	*File type for compiled files*
3600	.bin
UNIX	.bn (.bin is also accepted)
All other	.BIN

See the section "Canonical Types in Pathnames". This section contains a discussion of canonical types.

3.1.2.2 Changed Functions

The functions listed in column 2 of the following table replace old functions. The new functions work on both the 3600 and the LM-2.

Old function	*Replacement*
fasload	**si:load-binary-file**
qc-file-load	**compiler:compile-file-load**
qc-file	**compiler:compile-file**

If you used the **si:unfasl** tool on the LM-2, the corresponding tool for the 3600 bin files is called **si:unbin-file**. The output format from **si:unbin-file** is similar to that of **si:unfasl** but is improved to include disassembled code for any compiled functions in the bin file.

3.1.2.3 Assembly Language for the 3600

The **disassemble** function is still available on the 3600, and the Inspector and Debugger still display disassembled code. The instruction set is different; however, the assembly language for the 3600 is very similar to that of the LM-2, and should not be hard to understand if you have mastered the reading of LM-2 assembly language. The most important difference is the way function calling works.

The LM-2 uses an "inverted" calling sequence, as follows:

1. The CALL instruction opens a stack frame.
2. Argument values are pushed on the stack.
3. The last argument value is moved to destination D-LAST.
4. D-LAST starts the function call.

The 3600 uses a more conventional calling sequence in that the concept of destination D-LAST does not exist.

1. The arguments are all pushed on the stack.
2. The CALL instruction is then executed.
3. The CALL instruction (unlike the LM-2's CALL instruction) executes the procedure call.

3.1.3 New Features

3.1.3.1 Conditional Code

In some cases it will be necessary to conditionalize pieces of programs so that one version runs on the LM-2 and another runs on the 3600.

To facilitate this, the list returned by **(status features)** on the 3600 contains the Lisp object **3600** (as a fixnum, 3600 decimal), whereas on the LM-2 it does not. To conditionalize a piece of a program so that it runs on both the LM-2 and the 3600, use the #+ conditional expressions.

Example: Suppose a function **solarize-screen** that on the LM-2 expects coordinate pairs of the form *(x,y)* was changed to expect them in *(y, x)* order on the 3600. One way to write machine-dependent code is to conditionalize it, as follows:

```
#+cadr (solarize-screen arg1 arg2) ;the LM-2 version
#+3600 (solarize-screen arg2 arg1) ;the 3600 version
```

For information on sharp-sign (#) abbreviations: See the section "Sharp-sign Abbreviations".

3.1.4 Internals

The optimizer and style-checker features of the 3600 compiler work differently in the LM-2 compiler. You might notice some of the differences, but none requires changes in user procedures. The most important difference is that when an optimizer for a function (not for a special form) is run, the argument forms it sees have already been optimized.

3.2 Floating-point Numbers

3.2.1 Floating Point on the LM-2

The LM-2 supports two kinds of floating-point numbers: *flonums* and *small-flonums*.

The advantage of an LM-2 flonum is that it has sufficient precision and range to be considered a single-precision floating-point number in the usually accepted sense — 11 bits of exponent and 32 bits of mantissa. Its disadvantage is that it must be implemented as a pointer to a two-word block in memory, which impairs its speed and requires a special garbage-collection mechanism.

The advantage of an LM-2 small-flonum is that it requires no block of memory, because it is implemented as an immediate datum. The disadvantage is its limited range and precision — 7 bits of exponent and 18 bits of mantissa.

Both representations have the disadvantage of being nonstandard.

3.2.2 Floating Point on the 3600

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
(minusp -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. See the section "Numbers".

The only floating-point data type currently provided on the 3600 is the flonum data type, which conforms exactly to IEEE single-precision format, including 8 bits of exponent and 23 bits of fraction. This representation is standard and well-documented in the literature. A 3600 flonum has the advantages of both the LM-2 flonum and small-flonum:

- It has sufficient range and precision to be considered a single-precision floating-point number.
- It is implemented as an immediate datum and so has no storage overhead.

This last advantage is made possible because of the larger word size of the 3600.

The concept of small-flonums does not exist on the 3600. When the 3600 Lisp reader encounters the syntax that signifies a small-flonum, for example, 3.4s6, it reads it as an ordinary flonum to avoid the need to conditionalize such constants. The lack of a small-flonum is an incompatible difference between the LM-2 and the 3600 and might require you to use #+cadr conditionals in any program that uses small-flonums. See the section "Conditional Code". See the section "Sharp-sign Abbreviations".

The trigonometric and other mathematical functions have been subjected to numerical analysis with respect to the new word length and other changes for 3600 systems.

3.2.3 typep on the LM-2 and the 3600

If you are using **typep** to check for floating-point numbers, the **:float** type will work on both machines. On the LM-2, it recognizes both flonums and small-flonums. On the 3600, it recognizes both single- and double-precision floating-point numbers.

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 a union of these two types.) The 3600 does not recognize **:flonum** and **:small-flonum** as known arguments to **typep**.

3.3 self on the LM-2 and the 3600

The Lisp variable **self** has a special meaning to the Flavor System: Its value is the instance of the innermost method currently executing. Flavors are implemented differently on the 3600 than on the LM-2, with one important visible difference.

- On the LM-2, **self** is implemented as a special variable and is lambda-bound by methods.
- On the 3600, **self** is implemented as a lexical variable and is passed implicitly to methods, **defun-method** functions, and other parts of the Flavor System.

This means that **self** is dynamically scoped on the LM-2 but lexically scoped on the 3600.

Any programs having functions with free references to the variable **self** must pass the value of **self** as an argument or bind some special variable to the value, or the equivalent.

You should usually (but not always) write such functions using **defun-method**; this mechanism takes care of passing the value of **self**. If your program has functions with free references to the variable **self** and you neglect to change them before running them on the 3600, the compiler produces a warning and **self** is unbound at run time.

3.4 Arrays

3.4.1 Change in Array Referencing

The 3600 does not support **store** and function-style array referencing. Historically **store** existed only for the sake of certain large programs written in Maclisp, most of which have been converted to use modern-style array referencing. **store** has some fundamental semantic problems; to provide a fully compatible implementation would require the saving of state in processes, which would slow down all processes.

Use one of these two methods for altering programs that use **store**.

- Although time-consuming, the preferred method is to convert to modern-style array reference.
    - Store arrays in variables rather than as function definitions and use **aref** and **aset**. You will have to change every occurrence of **store** to **aset** or **setf**.

      Example: Change `(store (x 3 4) 14)` to either `(aset 14 x 3 4)` or `(setf (aref x 3 4) 14)`.

  An alternative method supported for compatibility relies on function-style array reference.

    - Change all occurrences of **store** to **setf**.

- Use the new special form **array-used-as-function**, which currently exists only on the 3600. Place **array-used-as-function** as a top-level form near the beginning of your file to indicate that you are referencing an array.

  Example: **(array-used-as-function x)** declares that `(x 3 4)` is really an array reference, not a function call.

  Note: If you do not include the **array-used-as-function** declaration, you can still call an array as a function; however, this method is slow, and **setf** will not recognize the array as a valid place to store values.

3.4.2 New Primitives Replace apply and lexpr-funcall on Arrays

Programs that use **apply** or **lexpr-funcall** on arrays of a run-time-varying number of dimensions should now use one of the following new primitives.

These three take an array and a list of subscripts.

sys:%lexpr-aref
sys:%lexpr-aset
sys:%lexpr-aloc

These three take an array and a single subscript, and access the array as if it were one-dimensional.

sys:%1d-aref
sys:%1d-aset
sys:%1d-aloc

3.4.3 Array Types on the 3600

The 3600 does not recognize the following LM-2 array types. The remaining array types exist with the same names and characteristics.

art-error
art-32b
art-stack-group-head
art-special-pdl
art-half-fix
art-reg-pdl
art-float
art-fps-float

Note: **art-float** is not needed on the 3600 because **art-q** arrays can store floats without any storage overhead.

The new array type for the 3600, **art-boolean**, is an array whose elements can take on the values **t** and **nil**. It uses only one bit of storage per element.

3.4.4 Subscript Bounds Checking

The 3600 in some cases does subscript bounds checking more carefully for multidimensional arrays and hence might uncover undetected bugs in programs that run acceptably on the LM-2.

3.5 Fonts

The internal representation of fonts for use on bit-mapped displays is completely different on the 3600.

On the LM-2 **%draw-char** does not work for all characters, and its caller must handle wide characters by looking at the font. See the section "Format of Fonts".

If you are using the **%draw-char** character-drawing subprimitive and are properly handling the case of wide characters on the LM-2, your code must be different for the 3600, which has no concept of wide characters. **%draw-char** works for all sizes of characters on the 3600.

You should probably do one of the following:

- Send the **:draw-char** message to a window.
- Consult the system code for the **:draw-char** message for an example of code that works on both machines.

You receive .bfd files (which can be read by either machine from the sys:fonts;tv; directory) for all the normal TV fonts that are not loaded into the machine already. If you are explicitly loading a .bfd file to get some font, you are probably doing something wrong, since the system automatically loads fonts as needed. However, you can use the function **fed:read-font-from-bfd-file** on both the LM-2 and the 3600 to load a font from a file you specify. (The function **fed:find-and-load-font,** which also exists on both machines, looks for the specified font in the same sys:fonts;tv; directory the system automatically searches.)

3.6 Subprimitives

3.6.1 General Information

This section discusses issues related to subprimitives and contains material that is likely to change in future releases. As the information is mainly of benefit to Symbolics system programmers, most readers can safely skip over this section.

Additional information can be found in the system definition files:

sys: l-sys; sysdef lisp	Data structure definitions
sys: l-sys; sysdfl lisp	Communication areas, escape routines
sys: l-sys; opdef lisp	Instruction set definition

Most LM-2 subprimitives exist also on the 3600. Many subprimitives that are used only for their side effect return different values on the 3600. A few look like functions but are really macros. They do not evaluate their arguments in left-to-right order. Some of the LM-2's subprimitives exist on the 3600 as macros that are defined in terms of even more primitive subprimitives; this is true of the offset subprimitives, for instance:

%p-contents-offset
%p-ldb-offset
%p-dpb-offset

Many of the internal storage formats are different on the 3600, so many of the symbols that name data types, internal fields, and internal field values are different also. Where the 3600 is compatible with the LM-2, in almost all cases the same name was used to avoid unnecessary incompatibility.

Several new subprimitives provide interfaces with specific pieces of 3600 hardware; these subprimitives are not documented here. In general, each is used by only one program and is simply an interface between the Lisp portion of that program and its microcode kernel.

The following subsections list the differences between subprimitives used on the LM-2, and their use on the 3600.

3.6.2 Data Types

The following data types do not exist on the 3600. This mostly reflects internal changes in storage organization. The only real user-visible difference in functionality is that small-flonums, entities, and microcode-entry functions no longer exist.

dtp-array-header
dtp-entity
dtp-free
dtp-header
dtp-instance-header
dtp-instance-variable-pointer
dtp-select-method
dtp-small-flonum
dtp-stack-closure
dtp-stack-group
dtp-symbol-header
dtp-trap
dtp-u-entry

The following data type names have been changed on the 3600, because the storage representation associated with them has been changed.

Old Name	*New Name*
dtp-fef-pointer	**dtp-compiled-function**
dtp-array-pointer	**dtp-array**

The following data types are new and apply only to the 3600.

dtp-element-forward
dtp-even-pc
dtp-float
dtp-header-i
dtp-header-p
dtp-monitor-forward
dtp-nil
dtp-odd-pc

The following data types are the same on the LM-2 and the 3600.

dtp-body-forward (obsolete; being phased out of the 3600)
dtp-closure
dtp-extended-number (almost the same)
dtp-external-value-cell-pointer
dtp-fix
dtp-gc-forward
dtp-header-forward
dtp-instance
dtp-list
dtp-locative
dtp-null
dtp-one-q-forward
dtp-symbol

Note that **nil** has a data type of **dtp-nil**, rather than **dtp-symbol**, and does not have a pointer field of zero. **symbolp** of **nil** remains true, and the address field points to the same storage representation as all other symbols.

The **q-data-types** variable has been replaced by **sys:*data-types***. The **q-data-types** "function" has been replaced by **si:data-types**.

3.6.3 Byte Specifiers

The global byte specifiers **%%q-flag-bit**, **%%q-high-half**, and **%%q-low-half** do not exist on the 3600. The LM-2's flag bit does not exist, and the halfwords are not intrinsically interesting.

The new byte specifiers **%%q-fixnum** and **%%q-high-type** reflect the fact that the number of bits in a fixnum does not equal the number of bits in a pointer.

Substitute **%%q-fixnum** for some applications of **%%q-pointer**, **%pointer**, **%p-pointer**, and **%p-store-pointer**.

The byte specifiers, field values, and accessor macros for the internal data structures are generally different. A few of the names are the same. For details, see the file sys: l-sys; sysdef lisp.

The fields in a fixnum that represents a character remain the same (both in name and numerically).

3.6.4 Symbols Specific to the LM-2

The following symbols (and prefixes of a whole family of symbols) do not exist on the 3600. They are specific to the LM-2 architecture, and their use is an indication of machine-dependent code.

llpfrm
QZ...
%initially-disable-trapping
%q-flag-bit
adi-
array-
fef-
sg-
%arg-desc- (%% remain)
%array-
%chaos-
%disk-
%ether-
%fef-
%fefh-
%fefhi-
%lp-
%meter-
%pht-
%sys-com-
%unibus-
%%adi-
%%area-
%%array-
%%chaos-
%%disk-
%%fef-
%%fefh-
%%fefhi-
%%lp-
%%meter-
%%m-esubs-
%%m-flags-
%%pht1-
%%pht2-
%%sg-
%%us-

The naming convention for machine-dependent source files that are maintained in parallel versions is Q*xxx* for the LM-2 and L*xxx* for the 3600. Thus, sys:sys;qcons contains LM-2 storage-allocation routines, and sys:sys;lcons contains like-named routines that are specific to the 3600.

3.6.5 New Subprimitives: sys:%fixnum and sys:%flonum

The new subprimitives **sys:%fixnum** and **sys:%flonum** set the data type field to convert a flonum to a fixnum or a fixnum to a flonum. These new subprimitives are not the functions **fix** and **float** but provide direct access to the internal bit representation of single-precision floating-point numbers.

3.6.6 Analyzing Structures Remains the Same

%find-structure-header and **%find-structure-leader** remain the same. However, the set of data types for which there is a difference between these two functions is machine-dependent.

On the LM-2, an array pointer might or might not contain the address of the first word of storage. On the 3600, a compiled function pointer does *not* contain the address of the first word of storage.

%structure-boxed-size does not exist on the 3600; **%structure-total-size** suffices, as all structures are made up of boxed elements.

3.6.7 Subprimitives Not Existing on the 3600

The following subprimitives do not exist on the 3600:

%p-mask-field
%p-deposit-field
%p-mask-field-offset
%p-deposit-field-offset

The LM-2 allocation subprimitives **%allocate-and-initialize** and **%allocate-and-initialize-array** do not exist on the 3600.

Except for **sys:%halt**, LM-2 I/O subprimitives do not exist on the 3600. See the section "I/O Device Subprimitives".

See the section "The Paging System". The following paging subprimitives, which are described there, do not exist.

si:wire-page
si:unwire-page
sys:%change-page-status
sys:%compute-page-hash
sys:%create-physical-page
sys:%delete-physical-page
sys:%disk-restore
sys:%disk-save

See the section "Closure Subprimitives". The subprimitives for closures described there do not exist.

See the section "Microcode Variables". See the section "Meters". The variables and meters described in these sections are specific to the LM-2 architecture and do not exist on the 3600.

3.6.8 Locking Subprimitive

store-conditional (formerly called **%store-conditional**) locks out microtasks but cannot lock out the FEP or external-DMA devices. Protocols for communicating with such devices must use locking methods that do not depend on atomic read-modify-write, such as those based on cells that are only written by one party and only read by the other party.

3.6.9 New Subprimitives

3.6.9.1 Subprimitives Existing on the LM-2 and 3600

The following subprimitives exist on both the LM-2 and the 3600. They were added to facilitate writing machine-independent low-level code.

Subprimitive	*Description*
sys:%pointer-lessp	Compares two addresses.
sys:%p-store-cdr-type-and-pointer	More general **%p-store-tag-and-pointer**.
sys:%instance-flavor	Gets the flavor structure of an instance.
sys:%change-list-to-cons	Changes a two-element cdr-coded list to a dotted pair by altering cdr codes.
sys:%pointerp	Returns **t** when its argument has an address.
sys:%pointer-type-p	Returns **t** when its argument is a data type code that has an associated address.

3.6.9.2 Subprimitives Existing Only on the 3600

The following subprimitives exist only on the 3600.

Subprimitive	*Description*
sys:%block-store-cdr-and-contents	Takes these arguments: • An address • A number of words • A cdr code • An object • An increment to the object (should be zero if the object is not a fixnum) The specified contiguous region of memory is efficiently filled with the object and the cdr code. The addresses to be initialized must not be mapped into A memory. If the increment is nonzero, it must not be used to increment a pointer across GC-space boundaries, or the GC tags will be set incorrectly.

sys:%block-store-tag-and-pointer	Similar to **sys:%block-store-cdr-and-contents**, except that the word to be stored is assembled from a tag field and a pointer field, allowing construction of invisible pointers.
sys:%p-structure-offset	Captures the inherent primitive underlying **%p-ldb-offset** and the like. It does **follow-structure-forwarding** on its first argument, then **%make-pointer-offset dtp-locative** of that and its second argument.
sys:%p-store-cdr-and-contents	Stores a cdr code and an object into a memory location, without reading the previous contents of that location. Use this subprimitive to store fixnums, as **%p-store-tag-and-pointer** cannot reasonably be used to do so.
sys:%unsynchronized-device-read	Reads registers on the rev. 2 I/O board. It allows data that are not properly synchronized to the Lbus clock to be read without causing a parity error.

3.6.10 Storage Layout Changes

See the section "Storage Layout Definitions". The variables described there are largely compatible, with exceptions due to the removal of the flag bit, the removal of cdr-error, and the fact that the number of bits in a pointer (28) and the number of bits in a fixnum (32) are not equal.

3.6.11 Function-calling Subprimitives

Except for **%push** and **%pop**, the subprimitives for calling with a run-time-variable number of arguments, without consing a list, have been replaced by the **%start-function-call** and **%finish-function-call** special forms. This change applies to both the LM-2 and the 3600. **%assure-pdl-room** does not currently exist on the 3600. See the section "Function-calling Subprimitives".

%start-function-call and **%finish-function-call** each take the same four subforms. Different subsets of the subforms are ignored, depending on whether the machine is an LM-2 or a 3600. The subforms are:

function	A form evaluated to yield the function to be called.
destination	The disposition of its results. Not evaluated. It takes these values:

Value	*Meaning*

nil	Call for effect.
t	Receive one value on the stack.
return	Return all values from the function in which it is being used.

There is no provision for receiving multiple values.

n-arguments	A form evaluated to yield the number of times **%push** has to be done.
lexpr	True if the last **%push** is a list of arguments rather than a single argument; false in the normal case. Not evaluated.

Follow these steps:

1. Do a **%start-function-call.**
2. Do a **%push** on each argument.
3. Do a **%finish-function-call.**

The order of evaluation of the subforms is not guaranteed, and you must make certain to pass the same subform values to the **%start** and the **%finish**. Generally it is best to use variables and not do computations in these subforms.

Also, you must not allocate or deallocate any local variables between the **%start** and the **%finish,** because on the 3600 they will get in the way of the **%push** subprimitives. Thus, the following will not work:

```
(%start-function-call ...)
(dolist (x l) (%push x))
(%finish-function-call ...)
```

Instead write:

```
(let ((x l))
  (%start-function-call ...)
  (do () ((null x)) (%push (pop x)))
  (%finish-function-call ...))
```

4. Current Incompatibilities

This chapter contains information covered in various other documents.

4.1 Differences Between the LM-2 and 3600

This section details differences between the LM-2 and the 3600.

4.1.1 Package Differences

The sets of symbols in the **system** and **global** packages are different.

4.1.2 Numeric Argument Descriptors

Numeric argument descriptors are similar but different. (These are the "magic" numbers returned by **args-info** and **%args-info**.) The numerical values as well as the field sizes have been changed. See the section "How Programs Examine Functions".

The following fields remain the same in name and meaning, but not in size:

%%arg-desc-interpreted
%%arg-desc-max-args
%%arg-desc-min-args

The following fields exist only on the LM-2. Also, the single-% symbols (bit-masks) — except for **%arg-desc-interpreted** — exist only on the LM-2.

%%arg-desc-evaled-rest
%%arg-desc-fef-bind-hair
%%arg-desc-fef-quote-hair
%%arg-desc-quoted-rest

The following fields exist only on the 3600:

%%arg-desc-rest-arg	Replaces the LM-2 scheme with 2 rest-arg bits.
%%arg-desc-quoted	Replaces the LM-2 scheme with 2 quoting bits.

If both the above bits are set and **%%arg-desc-max-args** is 0, the function is a **fexpr**. Otherwise, **eval** must check the debug information to get the argument-quoting pattern, because some arguments are evaluated and some are quoted.

4.1.3 Missing Functions

The following is a list of LM-2-specific and Maclisp-compatible functions not found in the 3600's **global** package.

Old function	*New function*
***plus**	**+**
***dif**	**-**
***times**	*****
***quo**	**/**
fasload	**load** or **si:load-binary-file**
include	
fixnum	
flonum	
notype	
ap-3	**aloc**
ar-3	**aref**
as-3	**aset**
disk-restore	FEP Load World command
fasd-update-file	
fasl-append	
font-next-plane	
font-rasters-per-word	
font-words-per-char	
get-list-pointer-into-array	
get-locative-pointer-into-array	
number-gc-on	
print-error-mode	
qc-file	**compiler:compile-file**
qc-file-load	**compiler:compile-file-load**
read-meter	
return-next-value	
set-current-band	**si:set-current-world-load**
set-current-microload	
set-error-mode	
set-mar	
set-memory-size	
sg-return-unsafe	
small-float	
small-floatp	
write-meter	
xstore	

4.1.4 Nonlocal Exits: Differences Between the LM-2 and 3600

***unwind-stack** and **catch-all** are not supported.

4.1.5 Bit and Byte Manipulations

%logldb, **%logdpb**, **lsh**, and **rot** operate on 32-bit words rather than 24-bit words.

Bytes can be any size that can be expressed in a byte specifier (up to 63 bits currently); any program that uses bytes larger than 23 bits will not run on the LM-2.

4.1.6 Stack Groups

Stack groups are conceptually the same, but: See the section "Stack Group States". The stack group states documented there are not compatible for the 3600.

The following list shows the permissible options for **make-stack-group**. The options are compatible for the LM-2 and the 3600 and, except for **:allow-unknown-keywords**, are documented as follows: See the section "Stack Group Functions".

:allow-unknown-keywords
:regular-pdl-area
:regular-pdl-size
:safe
:special-pdl-area
:special-pdl-size
:sg-area

:allow-unknown-keywords permits you to specify any keyword at all, without causing an error.

4.1.7 Areas

The permissible options for **make-area** differ for the 3600 in the following ways. See the section "Area Functions and Variables".

- These options remain the same for the LM-2 and the 3600.

  :region-size
  :representation
  :room
  :size
  :name
  :gc
  :read-only

  The **:pdl** option is accepted but is ignored on the 3600.

- **:swap-recommendations** is an undocumented option that is accepted on both the LM-2 and the 3600. **:swap-recommendations** sets the number of extra pages to be read in from disk after a page from this area is brought in due to demand paging.

- The LM-2 option **sys:%%region-map-bits**, whose name is not a keyword symbol, is not supported on the 3600.

The 3600 has many fewer areas than the LM-2, and the names of some areas have been changed. The following table shows the names of these areas and their new 3600 names. See the section "Interesting Areas".

Old Name	*New Name*
sys:p-n-string	**pname-area**
sys:nr-sym	**symbol-area**
macro-compiled-program	**compiled-function-area**
sys:init-list-area	**constants-area**
sys:fasl-constants-area	**constants-area**

The area tables (for example, **area-name** and **sys:region-length**) still exist and have the same names. However, in the 3600 they are not areas in their own right but are simply arrays. The area **sys:wired-control-tables** includes the wired communication areas and all the permanently wired programs and data structures that come from the boot image, but does not include the dynamically allocated wired tables: PHTC, PHT, MMPT, and SMPT.

The following LM-2 areas do not exist in any directly corresponding form on the 3600.

area-swap-recommendations
fasl-table-area
fasl-temp-area
gc-table-area
obt-tails
physical-page-data
system-communication-area

4.1.8 Maintaining a Patchable System on the LM-2 and 3600

Maintaining a patchable system to run on both the LM-2 and the 3600 requires care, because each machine has its own set of compiled files (including compiled patch files). Since the system must be compiled twice (once with each compiler), the **:no-increment-patch** option to **make-system** should be used the second time it is compiled, so that the two machines get the same system major version number. Take care not to edit source files between the two compilations in order to ensure that the two machines are running equivalent systems.

User-maintained systems do not normally contain machine-dependent patches. However, if they do: See the section "Making Patches". Then take these steps:

1. Do one of the following to add conditional expressions:
   - Manually edit #+3600 and #+cadr conditionals into the patch files before you use Finish Patch (m-X).
   - Use Add Patch (m-X) to extract conditionals from conditionalized source files.
2. Use Finish Patch (m-X) to install the patch file.
3. Use **si:compile-uncompiled-patches** on the type of machine on which you did not make the patch. Specify a system name as the argument. The function allows you to view or edit the source of the patch file and then to run the compiler over any patch files that have not been compiled.

   Example: If you make patch 21.4 to the Eschatology system on the LM-2, running (**si:compile-uncompiled-patches** "Eschatology") on a 3600 that has that system loaded will compile patch 21.4 for the 3600.

4.1.9 Loading and Saving Disk Partitions

All the functions for manipulating the disk label, updating software, and installing new software are different on the 3600. See the section "Front-end Processor".

4.1.10 Hash Tables

Hash tables are implemented slightly differently on the 3600. For example, the order in which **maphash** traverses a hash table can be different.

Note: The order in which **maphash** traverses the table is not defined as part of the function; therefore, your programs should *not* depend on the order in which **maphash** finds the elements.

4.2 Features Not Existing on the 3600

This section lists features that do not work yet on the 3600 but does not include differences due to normal system evolution. These latter differences are covered in the appropriate documents.

4.2.1 Predicates

The functions **small-floatp** and **entityp** have been removed, and **typep** has been changed accordingly.

4.2.2 Numeric Type Change

The function **small-float** has been removed.

4.2.3 Machine-dependent Numeric Functions

The following machine-dependent functions do not exist on the 3600.

- Functions for manipulating 24-bit numbers. (See the section "24-bit Numbers".)
- Functions for doing double-precision arithmetic with 24-bit numbers. (See the section "Double-precision Arithmetic".)

4.2.4 entity

The **entity** function has been removed.

4.2.5 Compiler Does Not Support Various Switches

The 3600 compiler does not support the following two variables, which are documented elsewhere: See the section "Compiler Declarations".

allow-variables-in-function-position-switch
run-in-maclisp-switch

4.2.6 New Function for Putting Data in Compiled Code Files

For both the LM-2 and the 3600, **sys:dump-forms-to-file** replaces the following functions:

compiler:fasd-symbol-value
compiler:fasd-font
compiler:fasd-file-symbols-properties

See the section "Putting Data in Compiled Code Files".

4.2.7 MAR

The MAR feature and the variable-monitoring feature do not have equivalents on the 3600. See the section "The MAR". See the section "Variable Monitoring".

4.2.8 set-memory-size

set-memory-size does not currently exist on the 3600.

4.2.9 Garbage Collector

The world-load compressor is not needed on the 3600. To make the world-load file smaller, type `(gc-immediately)` or `(si:full-gc)` before doing a **disk-save**.

The difference between **gc-immediately** and **si:full-gc** is that the latter garbage-collects areas that are not normally collected, and so takes longer.

Index

symbolics™

INED Using the Input Editor

Cambridge, Massachusetts

Using the Input Editor
# 990106

February 1984

This document corresponds to Release 5.0.

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

Table of Contents

1. Introduction

A *history* remembers commands and pieces of text, placing them in a history list. Additions to the history are placed at the top of the list, so that history elements are stored in reverse chronological order — the newer elements at the top of the history, the older elements toward the bottom.

Yanking commands pull in the elements of a history. *Top-level commands* start a yanking sequence. Other commands perform all subsequent yanks in the same sequence. A yanking sequence ends when you type new text, execute a form or command, or start another yanking sequence.

The system has different histories for different contexts. One of these is always the *current history*.

2. Summary of the Major Changes

The Release 5.0 yank system is based on a new concept of a history, generalized from such concepts as the kill ring and the input editor history. The following summary lists the major user-visible changes that have been made to histories and to the yanking commands since Release 4.

- The yank system supports two top-level yanking commands — `c-Y` and `c-m-Y`, whereas earlier releases used `c-Y`, `c-C`, and `c-X ALTMODE`.
- `m-Y` performs all subsequent yanks in the same sequence, whereas earlier releases used `m-C` and and `c-m-Y` for this purpose.
- The yanking commands have been made more consistent throughout the histories.

  Example: Pressing `m-Y` after a top-level yanking command retrieves immediately previous text in both the input editor and Zmacs. Formerly, you used `m-C` in the input editor and `m-Y` in Zmacs.
- The first element yanked by a top-level command given without an argument is called the *origin* of the history. Release 4 did not use the concept of an origin.
- The origin changes as you use `m-Y` to cycle through a history. As a result, the origin is not necessarily the newest element of the history.
- `c-Y` and `c-m-Y` given without an argument retrieve the element at the origin, which is the *last* element yanked by `m-Y` in the previous yanking sequence. Thus, the last element yanked in a previous sequence becomes the first element yanked in the current sequence. Formerly, a top-level command given after a `m-Y` behaved inconsistently.
- By default, when you display a history, element #1 is always the most recently added element. If the origin is not the newest element (element #1), it is indicated with a pointer. (Optionally, you can set the variable **zwei:*history-rotate-if-numeric-arg*** so that the origin is labelled #1 when you display the history. See the section "Customization Variables".)

  Example: `(+ 210 32)` and `(* 17 6)` are both newer than the origin, `(load-patches ':noselective)`.

```
    Lisp Listener 1 Input history:
       1: (+ 210 32)
       2: (* 17 6)
    -> 3: (load-patches ':noselective)
       4: (print-system-modifications)
       5: (print-disk-label)
       6: (login "sr")
```

- By default, arguments to `c-Y` and `c-m-Y` are measured relative to the newest element. To yank a particular element, give its number as an argument to `c-Y` or `c-m-Y`. (Optionally, you can set the variable **zwei:*history-rotate-if-numeric-arg*** so that arguments to the yanking commands are measured from the origin, whether or not it is the newest element in the history. See the section "Customization Variables".

Example: `c-m-4 c-m-Y` yanks element #4, `(print-system-modifications)`", from the history displayed in the previous example.

- Adding text to a history moves the origin to the newly added element.

  Example: Suppose you run the function **print-herald** and then redisplay the history list shown in the previous example.

```
Lisp Listener 1 Input history:
  1: (print-herald)
  2: (+ 210 32)
  3: (* 17 6)
  4: (load-patches ':noselective)
  5: (print-system-modifications)
  6: (print-disk-label)
  7: (login "sr")
```

- By default, a history remembers everything that has been typed to it since the last cold boot. Formerly, some histories were of fixed length; for example, the kill history saved only eight elements.
- All editors now use a single kill history. This allows you to move text easily from one window to another, for example, from the editor to a Lisp Listener. See the section "Input Editor".
- Displayed history elements are now mouse-sensitive, except in the input editor. If all the elements of a history are not displayed because the history is too long, then the screen displays a mouse-sensitive line:
  n `more elements in history`. Clicking left displays the rest of the history.
- Activating yanked forms in the input editor works differently. To reexecute a yanked form, just press `END` anywhere within or at the end of the form. (You can, of course, edit the input first if you wish.) Formerly, a form was yanked without its terminating character (a close parenthesis, a Return, or a Space); to have the form evaluated, you had to explicitly type the terminating character at the end of the form.

3. Types of Histories

Release 5.0 uses the following histories:

Type	*Description*
Input	History containing text typed at the input editor; a separate history exists for each window.
Kill	History of text deleted or saved in any window; a global history.
Replace	History of arguments to Query Replace (m-X) and related commands.
Buffer	History of editor buffers visited in this window.
Pathname	History of file names that have been typed.
Command	History of editor commands that use the minibuffer, and their arguments. Commands that do not use the minibuffer, such as m-RUBOUT, are not recorded in the history.
Definition	History of names of definitions that have been typed.

Except for the input histories, which are per-window, only a single instance of each of these histories exists, shared among all editors, including Zmacs, Zmail, and Dired.

4. Changes to the Yanking Commands

Cmd.	*Release 5.0*	*Release 4*
c-Y	Yanks from the global kill history.	Yanked from the editor kill ring.
	Is not context-sensitive. See c-m-Y.	Was context-sensitive only in the minibuffer; yanked either the last text typed in that context or the displayed default.
c-m-Y	Is a top-level command.	Was not a top-level command, with one exception: it restarted the minibuffer command. See the section "The Command History".
	Is context-sensitive; yanks from the appropriate history for the context.	
	Replaces the old functioning of c-Y in the minibuffer.	
m-Y	Replaces the text yanked by c-Y, c-m-Y, or m-Y with the previous element of the same history.	Replaced text yanked by c-Y or m-Y with the previous element of the kill ring.
	Replaces the old functioning of c-m-Y in the minibuffer.	
	Skips adjacent duplicate elements; yanks immediately preceding element.	Not applicable; identical pieces of text were not placed on the input editor history.
	Yields an error if it does not immediately follow c-Y or c-m-Y, except for a special case in the minibuffer. See the section "The Command History".	Yielded an error if it did not immediately follow c-Y.
	Accepts a negative argument in the minibuffer.	Did not accept a negative argument in the minibuffer.

m-0 m-y	Displays the current history after a top-level yank has been done.	Deleted the last text yanked but did not replace it with the previous element on the history. Use c-W to get this functioning.
c-C	Obsolete. Same as c-m-Y in the input editor.	Was a top-level command. Yanked from the input editor history.
m-C	Obsolete. Same as m-Y in the input editor.	Replaced text yanked by c-C with the previous element in the input editor history.
c-X ALTMODE	Obsolete. Same as c-m-Y in Zwei.	Yanked from the editor command history.

5. Numeric Arguments

1. A numeric argument of 0 to any yank command displays a list of the history and the numeric argument required to get each element of the history.

   Example: The input history invoked in a Lisp Listener by `c-m-0 c-m-Y`:

   ```
   Lisp Listener 1 Input history:
     1: (+ 210 32)
     2: (* 17 6)
     3: (load-patches ':noselective)
     4: (print-system-modifications)
     5: (print-disk-label)
     6: (login "sr")
   ```

   The history is displayed in reverse chronological order — the newest element first, for example, `(+ 210 32)`; the oldest last, for example, `(login "sr")`.

2. By default, a positive argument to `c-Y` and `c-m-Y` specifies how far from the newest element into *past* history is the element to be yanked. The numbers in the history display can be used as numeric arguments. (Optionally, you can set the variable **zwei:*history-rotate-if-numeric-arg*** so that arguments to the yanking commands are measured relative to the origin. See the section "Customization Variables".)

   Example: `c-m-1 c-m-Y` yanks element #1, `(+ 210 32)`, from the history displayed in step 1.

   Example: `c-m-2 c-m-Y` yanks element #2, `(* 17 6)`, from the history displayed in step 1.

3. A positive or negative argument to `m-Y` is measured relative to the last element yanked, not the newest element.

   Example: Pressing `c-m-2 c-m-Y` yanks `(* 17 6)`; then pressing `m-4 m-Y` yanks `(login "sr")`, not element #4. Displaying the history at this point looks this:

   ```
   Lisp Listener 1 Input history:
     1: (+ 210 32)
     2: (* 17 6)
     3: (load-patches ':noselective)
     4: (print-system-modifications)
     5: (print-disk-label)
   -> 6: (login "sr")
   ```

   Element #6, marked by a pointer, is the origin. (Note: The origin is not the most recent element because `m-Y` has changed the origin.)

4. A top-level command given without an argument retrieves the element at the origin, which is the last element yanked in the previous yanking sequence, not necessarily the newest element of the history.

   Example: `c-m-Y` yanks `(login "sr")` from the history displayed in step 3.

5. A numeric argument of `c-U` not followed by any digits is the same as no numeric argument with one exception: Point is placed before the text yanked and mark is placed after — the reverse of the ordinary placement.

6. The Displayed Default

When a command that reads an argument in the minibuffer displays a default, it puts the default onto the history temporarily. After reading and defaulting your input, it puts the argument onto the history instead. Thus `c-m-Y` always yanks the displayed default and `c-m-2 c-m-Y` yanks the last thing typed in that context. If no default is displayed, `c-m-Y` yanks the last thing typed in that context.

The displayed default is usually not the same as the most recent item in the history; often it is computed according to some heuristic based on past history and the exact command being given. It is pushed onto the top of the history in order to allow you to easily yank and edit it. This is useful when the heuristic comes close but does not provide exactly what you want.

7. The Command History

The command history replaces c-X ALTMODE and the Release 4 definition of c-m-Y. This history remembers all editor commands that use the minibuffer in any way. In normal editor context, when you are not in the minibuffer, c-m-Y yanks from the command history. Yanking from this history does not insert the command into the text being edited; instead it reexecutes the command, giving you a chance to edit each argument as the minibuffer for that argument appears. Immediately after a c-m-Y, when you are being presented with the first minibuffer, m-Y switches to an earlier command on the history list.

m-Y is a special case in the minibuffer when a command uses more than one minibuffer. If m-Y is typed as the first command to a minibuffer other than the first or is typed when a minibuffer other than the first is empty, the editor no longer causes an error. Instead it starts the current command over again with its first minibuffer. This functioning replaces the former behavior of the c-m-Y command.

Example: When m-. cannot find any definitions and prompts you for a file name, pressing m-Y lets you cycle backward and edit the definition name.

The same function can be achieved by pressing ABORT followed by c-m-Y, but m-Y is convenient and quicker.

Note this present anomaly: In Zmail reply mode, the c-m-Y command yanks the message being replied to. For this reason, c-m-Y is not available for accessing history there; you must use the EMACS-compatibility key, c-X ESCAPE (c-X ALTMODE on the LM-2).

Keep in mind that all (Zwei-based) editors use a single command history. An attempt to reexecute a command in a context where it does not work now gives an explanatory error message. Example: Trying to reexecute a Zmail command in Zmacs or trying to reexecute a Dired command while not in a Dired buffer gives an error.

8. Input Editor

In the input editor c-m-Y yanks from the history of previous inputs. For convenience and compatibility with Release 4, this command is also bound to c-C; for the same reason, m-C is synonymous with m-Y.

Because the input editor's kill history is now the same as the Zwei kill history, c-SPACE, c-W, m-W, c-<, c->, c-Y, and related commands can be used in the input editor to move text back and forth between the two editors. (Press c-HELP for a summary of commands.) Unlike Zwei, however, the input editor does not underline the region.

You can still use most Zwei editing commands on yanked forms, but reexecuting a yanked form is simpler: just press END anywhere within or at the end of the form. Formerly, a form was yanked without its terminating character; to have the form evaluated, you had to explicitly type the terminating character at the end of the form.

ESCAPE (STATUS on the LM-2) displays the history of previous inputs.

- With no numeric argument ESCAPE is equivalent to c-m-0 c-m-Y, displaying the default input history.

  A numeric argument controls the length of the input history to be displayed. An argument of 0 displays the entire history.

- With no numeric argument c-ESCAPE (c-STATUS on the LM-2) is equivalent to c-0 c-Y, displaying the default kill history.

  A numeric argument controls the length of the kill history to be displayed. An argument of 0 displays the entire history.

9. Customization Variables

To change the behavior of the yank system, use **login-forms** and **setq-globally** to set the following Lisp internal variables in your init file.

Alternatively, you can set them with Set Variable (m-X); when Set Variable prompts you for a variable name, supply the name given in each of the following descriptions.

zwei:*history-menu-length* *Variable*

The maximum number of history elements displayed. Default is 20.

History Menu Length is the name to use with Set Variable (m-X).

zwei:*history-yank-wraparound* *Variable*

Determines what happens after m-Y runs off the end of a history or m- - m-Y runs off the beginning of a history. Default is **t**.

Value	*Meaning*
t	m-y wraps around to the other end of the history. For example, after m-Y yanks the oldest element in the history, it returns to the top of the history and yanks the newest element.
nil	m-Y does not wrap around to the other end of the history. Instead, the 3600 flashes (the LM-2 beeps).

History Yank Wraparound is the name to use with Set Variable (m-X).

zwei:*history-rotate-if-numeric-arg* *Variable*

Determines what happens when c-Y or c-m-Y is given after m-Y. Default is **nil**.

Value	*Meaning*
t	A numeric argument to c-Y or c-m-Y is measured from the origin, not the newest element in the history. The origin is always element #1. All other elements are numbered relative to the origin. Elements that are newer than the origin are assigned negative numbers, in ascending order with their distance from the origin.
nil	A numeric argument to c-Y or c-m-Y is measured from the the newest history element, not the origin. However, c-Y or c-m-Y given *without* an argument yanks the element at the origin; thus, the origin has meaning only when you use a top-level command without an argument. When you display a history, its elements are numbered from 1 on and the origin is indicated with a pointer.

History Rotate If Numeric Arg is the name to use with Set Variable (m-X).

See the document *Streams*. This document contains more information on programming the input editor.

Index

symbolics™

MISCF Miscellaneous Useful Functions

Cambridge, Massachusetts

Miscellaneous Useful Functions

# 990102

February 1984

This document corresponds to Release 5.0.

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

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

Table of Contents

1. Poking Around in the Lisp World

This document describes a number of functions that do not logically fit in anywhere else. Most of these functions are not normally used in programs, but are "commands", that is, things that you type directly at Lisp.

who-calls *symbol* &optional *pkg* (*do-inferiors* **t**) (*do-superiors* **t**) *Function*

symbol must be a symbol or a list of symbols. **who-calls** tries to find all of the functions in the Lisp world that call *symbol* as a function, use *symbol* as a variable, or use *symbol* as a constant. (It won't find things that use constants that contain *symbol*, such as a list one of whose elements is *symbol*; it will only find it if *symbol* itself is used as a constant.) It tries to find all of the functions by searching the function cells and properties of all the symbols in a certain set of packages. The set always includes the package *pkg*. If *do-inferiors* is true, the set also includes all packages that use *pkg*. If *do-superiors* is true, the set also includes all packages that *pkg* uses. *pkg* defaults to the **global** package, and so normally all packages are checked.

If **who-calls** encounters an interpreted function definition, it simply tells you if *symbol* appears anywhere in the interpreted code. **who-calls** is smarter about compiled code, since it has been nicely predigested by the compiler.

If *symbol* is a list of symbols, **who-calls** does them all simultaneously, which is faster than doing them one at a time.

The editor has a command, List Callers (m-X), that is similar to **who-calls**.

The symbol **unbound-function** is treated specially by **who-calls**. **(who-calls 'unbound-function)** will search the compiled code for any calls through a symbol that is not currently defined as a function. This is useful for finding errors such as functions you misspelled the names of or forgot to write.

who-calls prints one line of information for each caller it finds. It also returns a list of the names of all the callers.

who-uses *symbol* &optional *pkg* (*do-inferiors* **t**) (*do-superiors* **t**) *Function*

who-uses is an obsolete name for **who-calls**.

See the function **who-calls**.

what-files-call *symbol-or-symbols* &optional *pkg* (*do-inferiors* **t**) (*do-superiors* **t**) *Function*

Similar to **who-calls** but returns a list of the pathnames of all the files that contain functions that **who-calls** would have printed out. This is useful if you need to recompile and/or edit all of those files.

apropos *apropos-substring* &optional *pkg* (*do-packages-used-by* **t**) *do-packages-used* *Function*

(**apropos** *apropros-substring*) tries to find all symbols whose print-names contain *apropos-substring* as a substring. Whenever it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so, and prints the names of the arguments (if any) to the function. It checks all symbols in a certain set of packages. The set always includes *pkg*. If *do-packages-used-by* is true, the set also includes all packages that use *pkg*. If *do-packages-used* is true, the set also includes all packages that *pkg* uses. *pkg* defaults to the **global** package, so normally all packages are searched. **apropos** returns a list of all the symbols it finds.

where-is *pname* *Function*

Find all symbols named *pname* and print on **standard-output** a description of each symbol. The symbol's home package and name are printed. If the symbol is present in a different package than its home package (that is, it has been imported), that fact is printed. A list of the packages from which the symbol is accessible is printed, in alphabetical order. **where-is** searches all packages that exist, except for invisible packages.

If *pname* is a string it is converted to uppercase, since most symbols' names use uppercase letters. If *pname* is a symbol, its exact name is used.

where-is returns a list of the symbols it found.

The **find-all-symbols** function is the primitive that does what **where-is** does without printing anything.

describe *x* *Function*

describe tries to tell you all of the interesting information about any object *x* (except for array contents). **describe** knows about arrays, symbols, all types of numbers, packages, stack groups, closures, instances, structures, compiled functions, and locatives, and prints out the attributes of each in human-readable form. Sometimes it will describe something that it finds inside something else; such recursive descriptions are indented appropriately. For instance, **describe** of a symbol will tell you about the symbol's value, its definition, and each of its properties. **describe** of a floating-point number will show you its internal representation in a way that is useful for tracking down roundoff errors and the like.

If *x* is a named-structure, **describe** handles it specially. To understand this, you should read the section on named structures. See the document *Defstruct*. First it gets the named-structure symbol, and sees whether its function knows about the **:describe** operation. If the operation is known, it applies the function to two arguments: the symbol **:describe**, and the named-structure itself. Otherwise, it looks on the named-structure symbol for information that might have been left by **defstruct**; this information would tell it what the symbolic names for the entries in the structure are, and **describe** knows how to use the names to print out what each field's name and contents is.

describe describes an instance by sending it the **:describe** message. The default method prints the names and values of the instance variables.

describe always returns its argument, in case you want to do something else to it.

inspect &optional *object* *Function*

A window-oriented version of **describe**. See the window system documentation for details, or try it.

disassemble *function* *Function*

function is either a compiled function, or a symbol or function spec whose definition is a compiled function. **disassemble** prints out a human-readable version of the macroinstructions in *function*. See the document *Internals*. That document contains an explanation of the macrocode instruction set.

The **grindef** function may be used to display the definition of a noncompiled function. See the document *Streams*.

set-memory-size *n-words* *Function*

set-memory-size tells the virtual memory system to use only *n-words* words of main memory for paging. Of course, *n-words* may not exceed the amount of main memory on the machine. (LM-2 only)

2. Utility Functions

zwei:save-all-files *Function*

This function is useful in emergencies in which you have modified material in Zmacs buffers that needs to be saved, but the editor is partially broken. This function does what the editor's Save All Files (`m-X`) command does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

zwei:zmail-save-all-files is similar, but saves mail files from Zmail.

print-sends &optional (*stream* **standard-output**) *Function*

Prints out all messages you have received (but not messages you have sent), in forward chronological order, to *stream*. Converse is more useful for looking at your messages, but this function predates Converse and is retained for compatibility.

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

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.

si:print-disk-error-log *Function*

Prints information about the half dozen most recent disk errors (since the last cold boot). (LM-2 only)

si:print-login-history &optional (*history* **si:login-history**) *Function*

Prints one line for each time the **login** function has been called in this world load. Each line contains the name of the user that logged in, the name of the machine on which the world load was running at that time, and the date and time. If you cold boot, log in, and then call **si:print-login-history**, the last line refers to your own login and all previous lines refer to logins that were done before running **disk-save**.

This information is useful to determine how many times a world load has been disk-saved, on what machines it was disk-saved, and who disk-saved it.

The first couple of lines do not contain any date or time, because they were made during the initial construction of the world load before it found out the current time. Names of users at other sites that are not in the local site's namespace search list are qualified with the site's namespace name and a vertical bar. The user SCRC|LISP-MACHINE is the dummy user used by **si:login-to-sys-host** at SCRC, the site where new world loads are created.

hostat &rest *hosts* *Function*

Interrogates the specified hosts, or all known hosts if none are specified, with the **STATUS** protocol and prints the results in columns as a table.

uptime &rest *hosts* *Function*

Queries the specified *hosts*, asking them for their "uptime"; each host responds by saying how long it has been up and running. **uptime** prints out the results. If **uptime** reports that a host is "not responding", either the host is not responding to the network, or it does not support the UPTIME protocol.

The **uptime** function is a variant of **hostat**.

3. The Lisp Top Level

These functions constitute the Lisp top level and its associated functions.

si:lisp-top-level *Function*
This is the first function called in the initial Lisp environment. It calls **lisp-reinitialize**, clears the screen, and calls **si:lisp-top-level1**.

lisp-reinitialize &optional (*called-by-user* **t**) *Function*
This function does a wide variety of things, such as resetting the values of various global constants and initializing the error system.

si:lisp-top-level1 *terminal-io* *Function*
This is the actual top-level loop. It reads a form from **standard-input**, evaluates it, prints the result (with slashification) to **standard-output**, and repeats indefinitely. If several values are returned by the form, all of them will be printed. Also the values of *****, **+**, **-**, **//**, **++**, ******, **+++**, and ******* are maintained.

break *tag* [*conditional-form*] *Special Form*
break is used to enter a breakpoint loop, which is similar to a Lisp top-level loop. (**break** *tag*) will always enter the loop;
(**break** *tag conditional-form*) will evaluate *conditional-form* and only enter the break loop if it returns non-**nil**. If the break loop is entered, **break** prints out

```
;Breakpoint tag; Resume to continue, Abort to quit.
```

and then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top-level loop is that when reading a form, **break** checks for the following special cases: If the ABORT key is pressed, control is returned to the previous break or Debugger, or to top level if there is none. If the RESUME key is pressed, **break** returns **nil**. If the list (**return** *form*) is typed, **break** evaluates *form* and returns the result.

Inside the **break** loop, the streams **standard-output**, **standard-input**, and **query-io** are bound to be synonymous to **terminal-io**; **terminal-io** itself is not rebound. Several other internal system variables are bound, and you can add your own symbols to be bound by pushing elements onto the value of the variable **sys:*break-bindings***. (See the variable **sys:*break-bindings***.)

If *tag* is omitted, it defaults to **nil**.

There are two easy ways to write a breakpoint into your program: **(break)** gets a read-eval-print loop, and **(dbg)** gets the Debugger. (These are the programmatic equivalents of the SUSPEND and m-SUSPEND keys on the keyboard.)

prinl *Variable*

The value of this variable is normally **nil**. If it is non-**nil**, then the read-eval-print loop will use its value instead of the definition of **prinl** to print the values returned by functions. This hook lets you control how things are printed by all read-eval-print loops--the Lisp top level, the **break** function, and any utility programs that include a read-eval-print loop. It does not affect output from programs that call the **prinl** function or any of its relatives such as **print** and **format**; if you want to do that, you will need more information on customizing the printer. See the document *Primitive Object Types*. If you set **prinl** to a new function, remember that the read-eval-print loop expects the function to print the value but not to output a Return character or any other delimiters.

- *Variable*

While a form is being evaluated by a read-eval-print loop, - is bound to the form itself.

+ *Variable*

While a form is being evaluated by a read-eval-print loop, + is bound to the previous form that was read by the loop.

***** *Variable*

While a form is being evaluated by a read-eval-print loop, * is bound to the result printed the last time through the loop. If several values were printed (because of a multiple-value return), * is bound to the first value. If no result was printed, * is not changed.

// *Variable*

While a form is being evaluated by a read-eval-print loop, // is bound to a list of the results printed the last time through the loop.

++ *Variable*

++ holds the previous value of +, that is, the form evaluated two interactions ago.

+++ *Variable*

+++ holds the previous value of ++.

****** *Variable*

** holds the previous value of *, that is, the result of the form evaluated two interactions ago.

******* *Variable*

*** holds the previous value of **.

sys:*break-bindings* *Variable*

When **break** is called, it binds some special variables under control of the list that is the value of **sys:*break-bindings***. Each element of the list is a list of two elements: a variable and a form that is evaluated to produce the value to bind it to. The bindings happen sequentially. Users may **push** things on this list (adding to the front of it), but should not replace the list wholesale since several of the variable bindings on this list are essential to the operation of **break**.

lisp-crash-list *Variable*

The value of **lisp-crash-list** is a list of forms. **lisp-reinitialize** sequentially evaluates these forms, and then sets **lisp-crash-list** to **nil**.

In most cases, the *initialization* facility should be used rather than **lisp-crash-list**. See the document *Initializations*.

4. Logging in

Logging in tells the Lisp Machine who you are, so that other users can see who is logged in, you can receive messages, and your init file can be run. An init file is a Lisp program that gets loaded when you log in; you can use it to set up a personalized environment.

When you log out, it should be possible to undo any personalizations you have made so that they do not affect the next user of the machine. Therefore, anything done by an init file should be undoable. Thus, for every form in the init file, you should add to the list that is the value of **logout-list** a Lisp form to undo its effects. The functions **login-forms** and **login-setq**, described below, help make this easy.

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.

logout *Function*

First, **logout** evaluates the forms on **logout-list**. Then it sets **user-id** to an empty string and **logout-list** to **nil**. Then it runs the **:logout** initialization list and returns **t**. See the document *Initializations*.

user-id *Variable*

The value of **user-id** is either the name of the logged in user, as a string, or else an empty string if there is no user logged in. It appears in the status line.

site-name *Variable*

The value is a keyword, the name of the site at which this machine is located. See the section "Site Objects".

site-name can be used to conditionalize programs. For example:

```
(when (eq site-name :acme)
  (load "apricot:>smith>cerebrum-server"))
```

logout-list *Variable*

The value of **logout-list** is a list of forms that are evaluated when a user logs out.

login-forms *body ...* *Special Form*

login-forms is a special form for wrapping around a set of forms in your init file. It evaluates the forms and arranges for them to be undone when you log out. It is intended to replace **login-setq** and **login-eval**.

login-forms always evaluates the forms, even when it does not know how to undo them. For forms that it cannot undo, it prints a warning message.

In the following example, **login-forms** arranges for **foo** either to become unbound or to get its old value and for **bar** either to become undefined or to get its old function definition. It would warn you about **quux** being impossible to undo.

```
(login-forms
  (setq foo 3)
  (defun bar (x y) (+ x y))
  (quux 3))
```

You can create functions to undo forms that **login-forms** does not recognize. To undo a given form, you put a property on the symbol that is the car of the form to undo. For example, to create a function to undo **quux**:

```
(defun (:property quux :undo-function) (form)
  `(undo-quux ,(cadr form)))
```

The value returned by an undo function is a form to be evaluated at logout time.

setq-globally *{variable value}...* *Special Form*

setq-globally should be used with **login-forms**, rather than **setq**, for anything that might be bound while evaluating the **login-forms**.

setq-globally works like **setq** but sets the global values, bypassing any special-variable bindings. **login-forms** knows how to undo this. **setq-globally** is the recommended way to set things in one's init file; for instance, setting **base** with plain **setq** does not work if the init file has a Base attribute in its -*- line, because that causes **base** to be bound during the loading of the file.

An example:

```
(login-forms
  (setq-globally base 10.
                 ibase 10.
                 zwei:*converse-beep-count* 4))
```

login-setq *{variable value}...* *Special Form*

login-setq is like **setq** except that it puts a **setq** form on **logout-list** to set the variables to their previous values. It is now obsolete; use **login-forms** instead.

login-eval *x* *Function*

login-eval is used for functions that are "meant to be called" from init files, such as **zwei:set-comtab-return-undo**, which conveniently return a form to undo what they did. **login-eval** adds the result of the form *x* to the **logout-list**. It is now obsolete; use **login-forms** instead.

5. Dribble Files

dribble-start *filename* &optional *editor-p* *Function*

dribble-start opens *filename* as a "dribble file" (also known as a "wallpaper file"). It rebinds **standard-input** and **standard-output** so that all of the terminal interaction is directed to the file as well as the terminal. If *editor-p* is non-**nil**, then instead of opening *filename* on the file computer, **dribble-start** dribbles into a Zmacs buffer whose name is *filename*, creating it if it does not exist.

dribble-end *Function*

This closes the file opened by **dribble-start** and resets the I/O streams.

6. status and sstatus

The **status** and **sstatus** special forms exist for compatibility with Maclisp. Programs that wish to run in both Maclisp and Zetalisp can use **status** to determine which of these they are running in. Also, **(sstatus feature ...)** can be used as it is in Maclisp.

status *Special Form*

(status features) returns a list of symbols indicating features of the Lisp environment. The complete list of all symbols that may appear on this list, and their meanings, is given in the Maclisp manual. The default list for the Lisp Machine is:

```
(:DEFSTORAGE :LOOP :DEFSTRUCT :LISPM :SYMBOLICS 3600 :CHAOS :SORT
 :FASLOAD :STRING :NEWIO :ROMAN :TRACE :GRINDEF :GRIND)
```

The value of this list will be kept up to date as features are added or removed from the Lisp Machine system. Most important is the symbol **:lispm**; this indicates that the program is executing on the Lisp Machine. The order of this list should not be depended on, and may not be the same as shown above.

The following symbols in the features list can be used to distinguish different Lisp implementations, using the #+ and #- reader syntax.

- Three symbols indicate which Lisp Machine hardware is running:

:lispm	Any kind of Lisp Machine, as opposed to Maclisp
:cadr	An LM-2 or an M.I.T. CADR
:3600	A 3600

- One symbol indicates which kind of Lisp Machine software is running:

:symbolics	Symbolics software

See the section "Conditional Code". See the section "Sharp-sign Abbreviations".

(status feature *symbol***)** returns **t** if *symbol* is on the **(status features)** list, otherwise **nil**.

(status nofeature *symbol***)** returns **t** if *symbol* is not on the **(status features)** list, otherwise **nil**.

(status userid) returns the name of the logged-in user.

(status tabsize) returns the number of spaces per tab stop (always 8). Note that this can actually be changed on a per-window basis, however the **status** function always returns the default value of 8.

(status opsys) returns the name of the operating system, always the symbol **:lispm**.

(status site) returns the name of the local machine, for example, **"MIT-LISPM-6"**. Note that this is not the same as the value of **site-name**.

(status status) returns a list of all **status** operations.

(status sstatus) returns a list of all **sstatus** operations.

sstatus *Special Form*

(sstatus feature *symbol***)** adds *symbol* to the list of features.

(sstatus nofeature *symbol***)** removes *symbol* from the list of features.

Index