

TEXAS INSTRUMENTS

EXPLORER™

TOOLS AND UTILITIES



EXPLORER™ TOOLS AND UTILITIES

MANUAL REVISION HISTORY

Explorer Tools and Utilities (2549831-0001)

Original Issue June 1987

Revision A October 1987

© 1987, Texas Instruments Incorporated. All Rights Reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Texas Instruments Incorporated.

The system-defined windows shown in this manual are examples of the software as this manual goes into production. Later changes in the software may cause the windows on your system to be different from those in the manual.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

Texas Instruments Incorporated
ATTN: Data Systems Group, M/S 2151
P.O. Box 2909
Austin, Texas 78769-2909

PDP is a trademark of Digital Equipment Corporation.

VT100 is a trademark of Digital Equipment Corporation.

TOPS-20 is a trademark of Digital Equipment Corporation.

UNIX is a registered trademark of AT&T.

Symbolics is a trademark of Symbolics, Inc.

LIST OF EFFECTIVE PAGES

Insert latest changed pages and discard superseded pages.

Note: The changes in the text are indicated by a change number at the bottom of the page and a vertical bar in the outer margin of the changed page. A change number at the bottom of the page but no change bar indicates either a deletion or a page layout change.

Explorer™ Tools and Utilities (2549831-0001 *B)

Original Issue June 1987
 Change 1 October 1987
 Change 2 December 1987

© 1986, 1987, Texas Instruments Incorporated. All Rights Reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Texas Instruments.

The system-defined windows shown in this manual are examples of the software as this manual goes into production. Later changes in the software may cause the windows on your system to be different from those in the manual.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

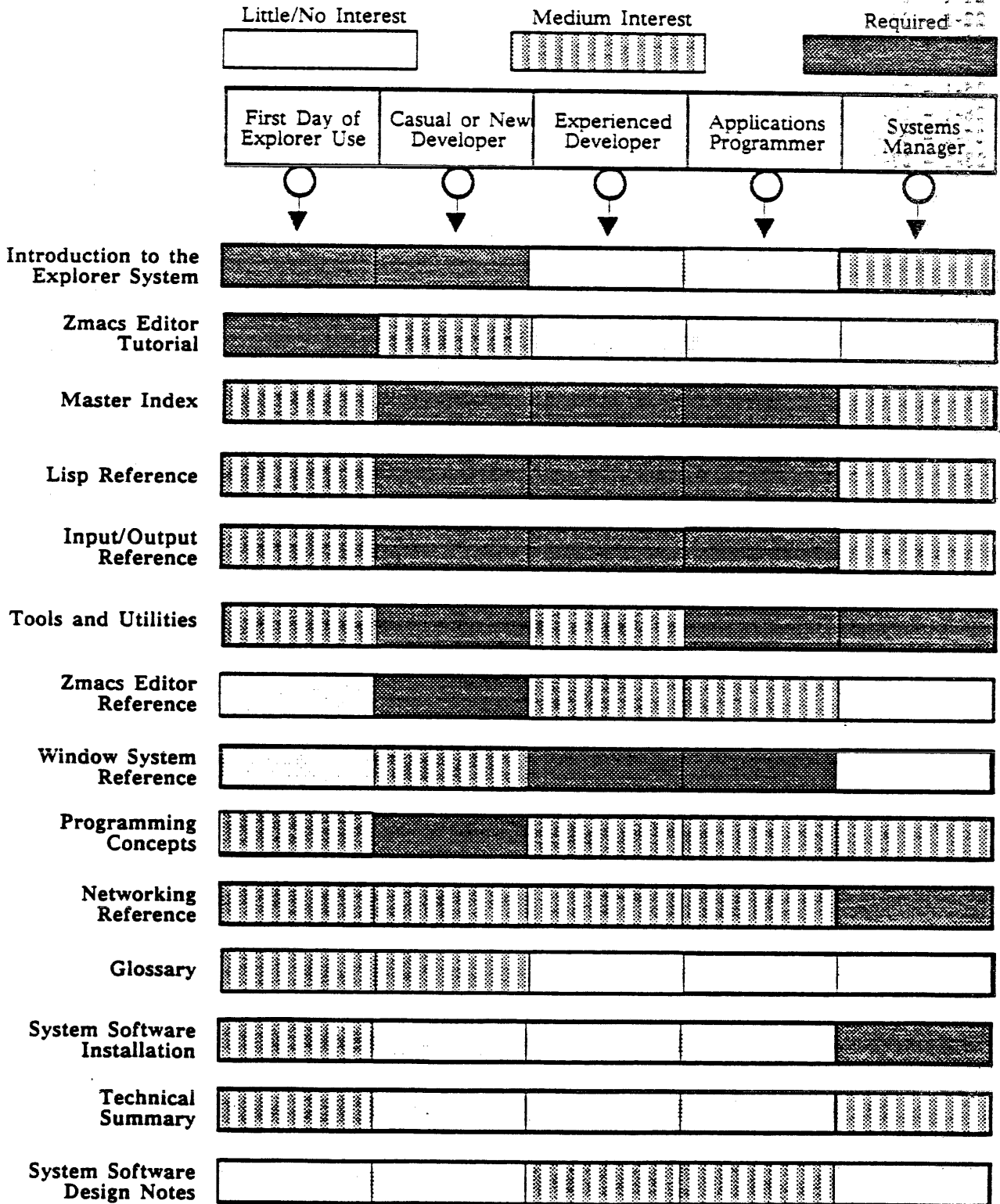
Texas Instruments Incorporated
 ATTN: Data Systems Group, M/S 2151
 P.O. Box 2909
 Austin, Texas 78769-2909

The total pages and change numbers in this publication are as follows:

Page	Change No.	Page	Change No.	Page	Change No.
Front Cover	2	1-1 - 1-2	0	10-1 - 10-2	0
Title Page	0	2-1	0	10-3 - 10-42	2
List of Effective Pages	2	2-2	2	11-1 - 11-15	0
Manuals Frontispieces(6pp)	2	2-3 - 2-5	0	12-1 - 12-31	0
xi	2	3-1 - 3-6	0	13-1 - 13-13	0
xii - xiv	0	4-1	0	14-1 - 14-5	0
xv - xvii	2	5-1 - 5-12	0	15-1 - 15-7	0
xviii - xxii	0	6-1 - 6-26	0	16-1 - 16-8	0
xxiii - xxiv	1	7-1 - 7-38	0	17-1 - 17-18	0
xxv - xxxi	2	8-1 - 8-20	0	18-1 - 18-6	0
xxxii - xxxiii	0	9-1 - 9-14	0	19-1 - 19-3	0

Page	Change No.	Page	Change No.	Page	Change No.
20-1 - 20-2	0	29-1 - 29-3	0	B-2 - B-18	0
21-1 - 21-5	0	30-1 - 30-5	0	B-19	2
22-1 - 22-2	0	31-1 - 31-57	1	B-20	1
23-1 - 23-3	0	32-1 - 32-71	0	B-21 - B-22	0
24-1 - 24-18	0	33-1 - 33-4	0	B-23	2
25-1 - 25-20	0	34-1 - 34-8	2	Index-1 - Index-29	2
26-1 - 26-2	0	35-1 - 35-9	2	Doc. Questionnaire	0
27-1 - 27-45	0	A-1 - A-29	0	Business Reply	0
28-1 - 28-4	0	B-1	2	Back Cover	2

THE EXPLORER™ SYSTEM SOFTWARE MANUALS



THE EXPLORER™ SYSTEM SOFTWARE MANUALS

Mastering the Explorer Environment	Explorer Technical Summary	2243189-0001
	Introduction to the Explorer System	2243190-0001
	Explorer Zmacs Editor Tutorial	2243191-0001
	Explorer Glossary	2243134-0001
	Explorer Networking Reference	2243206-0001
	Explorer Diagnostics	2533554-0001
	Explorer Master Index to Software Manuals	2243198-0001
Explorer System Software Installation Guide	2243205-0001	

Programming With the Explorer	Explorer Programming Concepts	2549830-0001
	Explorer Lisp Reference	2243201-0001
	Explorer Input/Output Reference	2549281-0001
	Explorer Zmacs Editor Reference	2243192-0001
	Explorer Tools and Utilities	2549831-0001
	Explorer Window System Reference	2243200-0001

Explorer Options	Explorer Natural Language Menu System User's Guide	2243202-0001
	Explorer Relational Table Management System User's Guide	2243203-0001
	Explorer Grasper User's Guide	2243135-0001
	Explorer TI Prolog User's Guide	2537248-0001
	Programming in Prolog, by Clocksin and Mellish	2249985-0001
	Explorer Color Graphics User's Guide	2537157-0001
	Explorer TCP/IP User's Guide	2537150-0001
	Explorer LX™ User's Guide	2537225-0001
	Explorer LX System Installation	2537227-0001
	Explorer NFS™ User's Guide	2546890-0001
	Explorer DECnet™ User's Guide	2537223-0001
Personal Consultant™ Plus Explorer	2537259-0001	

System Software Internals	Explorer System Software Design Notes	2243208-0001
	Release Information, Explorer System Software	2549844-0001

Explorer LX and Personal Consultant are trademarks of Texas Instruments Incorporated.

NFS is a trademark of Sun Microsystems, Inc.

DECnet is a trademark of Digital Equipment Corporation.

THE EXPLORER™ SYSTEM HARDWARE MANUALS

System Level Publications	Explorer 7-Slot System Installation	2243140-0001
	Explorer System Field Maintenance	2243141-0001
	Explorer System Field Maintenance Documentation Kit	2243222-0001
	Explorer System Field Maintenance Supplement	2537183-0001
	Explorer System Field Maintenance Supplement Documentation Kit	2549278-0001
	Explorer NuBus™ System Architecture General Description	2537171-0001

System Enclosure Equipment Publications	Explorer 7-Slot System Enclosure General Description	2243143-0001
	Explorer Memory General Description (8-megabytes)	2533592-0001
	Explorer 32-Megabyte Memory General Description	2537185-0001
	Explorer Processor General Description	2243144-0001
	68020-Based Processor General Description	2537240-0001
	Explorer II™ Processor and Auxiliary Processor Options General Description	2537187-0001
	Explorer System Interface General Description	2243145-0001
	Explorer Color System Interface Board General Description	2537189-0001
	Explorer NuBus Peripheral Interface General Description (NUPI board)	2243146-0001

Display Terminal Publications	Explorer Display Unit General Description	2243151-0001
	CRT Data Display Service Manual, Panasonic code number FTD85055057C	2537139-0001
	Explorer Color Console General Description	2537195-0001
	TRINITRON® Graphic Display Monitor GDM-1603 Service Manual, Sony® part number 0-558-986-01	2551107-0001
	Model 924 Video Display Terminal User's Guide	2544365-0001

143-Megabyte Disk/Tape Enclosure Publications	Explorer Mass Storage Enclosure General Description	2243148-0001
	Explorer Winchester Disk Formatter (ADAPTEC) Supplement to Explorer Mass Storage Enclosure General Description	2243149-0001
	Explorer Winchester Disk Drive (Maxtor) Supplement to Explorer Mass Storage Enclosure General Description	2243150-0001
	Explorer Cartridge Tape Drive (Cipher) Supplement to Explorer Mass Storage Enclosure General Description	2243166-0001
	Explorer Cable Interconnect Board (2236120-0001) Supplement to Explorer Mass Storage Enclosure General Description	2243177-0001

143-Megabyte Disk Drive Vendor Publications	XT-1000 Service Manual, 5 1/4-inch Fixed Disk Drive, Maxtor Corporation, part number 20005 (5 1/4-inch Winchester disk drive, 112 megabytes)	2249999-0001
	ACB-5500 Winchester Disk Controller User's Manual, Adaptec, Inc., (formatter for the 5 1/4-inch Winchester disk drive)	2249933-0001
1/4-Inch Tape Drive Vendor Publications	Series 540 Cartridge Tape Drive Product Description, Cipher Data Products, Inc., Bulletin Number 01-311-0284-1K (1/4-inch tape drive)	2249997-0001
	MT01 Tape Controller Technical Manual, Emulex Corporation, part number MT0151001 (formatter for the 1/4-inch tape drive)	2243182-0001
	Viper™ Half-High Intelligent 4 1/4-Inch Streaming Cartridge Tape Drive SCSI Models 2060S and 2125S, Archive Corporation, part number 21136-001	2551106-0001
182-Megabyte Disk/Tape Enclosure MSU II Publications	Mass Storage Unit (MSU II) General Description	2537197-0001
182-Megabyte Disk Drive Vendor Publications	Control Data® WREN™ III Disk Drive OEM Manual, part number 77738216, Magnetic Peripherals, Inc., a Control Data Company	2546867-0001
515-Megabyte Mass Storage Subsystem Publications	SMD/515-Megabyte Mass Storage Subsystem General Description (includes SMD/SCSI controller and 515-megabyte disk drive enclosure)	2537244-0001
515-Megabyte Disk Drive Vendor Publications	515-Megabyte Disk Drive Documentation Master Kit (Volumes 1, 2, and 3), Control Data Corporation	2246129-0002
	Volume 1, General Description, Operation, Installation and Checkout, and Part Data	2246125-0004
	Volume 2, Theory, General Maintenance, Trouble Analysis, Electrical Checks, and Repair Information	2246125-0005
	Volume 3, Diagrams	2246125-0006
1/2-Inch Tape Drive Publications	MT3201 1/2-Inch Tape Drive General Description	2537246-0001

Viper is a trademark of Archive Corporation.

Control Data is a registered trademark and WREN is a trademark of Control Data Corporation.

1/2-Inch Tape Drive Vendor Publications	Cipher CacheTape® Documentation Manual Kit (Volumes 1 and 2 With SCSI Addendum and, Logic Diagram), Cipher Data products	2246130-0001
	1/2-Inch Tape Drive Operation and Maintenance (Volume 1), Cipher Data Products	2246126-0001
	1/2-Inch Tape Drive Theory of Operation (Volume 2), Cipher Data Products	2246126-0002
	SCSI Addendum With Logic Diagram, Cipher Data Products	2246126-0003

Printer Publications	Model 810 Printer Installation and Operation Manual	2311356-9701
	Omni 800™ Electronic Data Terminals Maintenance Manual for Model 810 Printers	0994386-9701
	Model 850 RO Printer User's Manual	2219890-0001
	Model 850 RO Printer Maintenance Manual	2219896-0001
	Model 850 XL Printer User's Manual	2243250-0001
	Model 850 XL Printer Quick Reference Guide	2243249-0001
	Model 855 Printer Operator's Manual	2225911-0001
	Model 855 Printer Technical Reference Manual	2232822-0001
	Model 855 Printer Maintenance Manual	2225914-0001
	Model 860 XL Printer User's Manual	2239401-0001
	Model 860 XL Printer Maintenance Manual	2239427-0001
	Model 860 XI Printer Quick Reference Guide	2239402-0001
	Model 860/859 Printer Technical Reference Manual	2239407-0001
	Model 865 Printer Operator's Manual	2239405-0001
	Model 865 Printer Maintenance Manual	2239428-0001
	Model 880 Printer User's Manual	2222627-0001
	Model 880 Printer Maintenance Manual	2222628-0001
	OmniLaser™ 2015 Page Printer Operator's Manual	2539178-0001
	OmniLaser 2015 Page Printer Technical Reference	2539179-0001
	OmniLaser 2015 Page Printer Maintenance Manual	2539180-0001
	OmniLaser 2108 Page Printer Operator's Manual	2546348-0001
	OmniLaser 2108 Page Printer Technical Reference	2546349-0001
	OmniLaser 2108 Page Printer Maintenance Manual	2546350-0001
OmniLaser 2115 Page Printer Operator's Manual	2546344-0001	
OmniLaser 2115 Page Printer Technical Reference	2546345-0001	
OmniLaser 2115 Page Printer Maintenance Manual	2546346-0001	

Communications Publications	990 Family Communications Systems Field Reference	2276579-9701
	EI990 Ethernet® Interface Installation and Operation	2234392-9701
	Explorer NuBus Ethernet Controller General Description	2243161-0001
	Communications Carrier Board and Options General Description	2537242-0001

CacheTape is a registered trademark of Cipher Data Products, Inc.
Omni 800 and OmniLaser are trademarks of Texas Instruments Incorporated.
Ethernet is a registered trademark of Xerox Corporation.

CONTENTS

Section	Title
1	New User
2	Profile
3	Login Initialization File
4	Bug Reporting
5	Glossary Utility
6	UCL User Interface
7	UCL Programmer Interface
8	Using Suggestions
9	Programming Suggestions
10	Graphics Editor
11	Tree Editor
12	Font Editor
13	Debugger (Error Handler)
14	Window-Based Debugger
15	Inspector
16	Flavor Inspector
17	Peek
18	Trace
19	Stepper
20	Evalhook
21	Advise
22	Breakon
23	MAR
24	Crash Analysis
25	Miscellaneous Debugging Functions
26	Lisp Listener and Break
27	Performance Tools
28	Telnet
29	VT100 Emulator
30	Converse
31	Mail
32	Namespace Utilities
33	Miscellaneous Network Functions
34	Color Map Editor
35	Visidoc
Appendix A	Explorer Fonts
Appendix B	Command Tables

Section	Paragraph	Title	Page
		About This Manual	
		Contents of This Manual	xxix
		Notational Conventions	xxxi
		Keystroke Sequences	xxxi
		Mouse Clicks	xxxii
		Lisp Language Notation	xxxii
1		New User	
2		Profile	
	2.1	Introduction	2-1
	2.2	Requirements	2-1
	2.3	Accessing the Profile Utility	2-1
	2.4	Accessing Variables in the Profile Utility	2-2
	2.5	Commands in the Profile Utility	2-3
	2.6	Typical Variables	2-3
	2.7	Customizing Profile	2-4
3		Login Initialization File	
	3.1	Introduction	3-1
	3.2	Customizations That Can Be Undone	3-2
	3.3	Other Customizations	3-4
	3.3.1	Using Profile	3-4
	3.3.2	Customizing Zmacs	3-4
	3.3.3	Creating Logical Pathnames	3-5
	3.3.4	The with-timeout Macro	3-5
	3.3.5	The sys:load-if Function	3-6
4		Bug Reporting	
5		Glossary Utility	
	5.1	Introduction	5-1
	5.2	Entering the Glossary Utility	5-2
	5.3	Glossary User Mode	5-2
	5.3.1	Glossary Command Menu	5-3
	5.3.2	Keyboard Typein Window	5-4
	5.3.3	Menu of Glossary Entries	5-4
	5.3.4	Text of Selected Glossary Entries	5-4
	5.3.5	Thumb Index	5-4

Section	Paragraph	Title	Page
	5.4	Glossary Expert Mode	5-5
	5.4.1	Define Glossary	5-5
	5.4.2	Delete Glossary	5-6
	5.4.3	Select Glossary	5-6
	5.4.4	Add or Delete Glossary Entry	5-7
	5.4.5	Edit Glossary Entry	5-7
	5.4.6	Write Current Glossary	5-8
	5.4.7	Merge in Glossary	5-8
	5.4.8	(Re)Generate XRefs	5-8
	5.4.9	Turn On XRef Deletion	5-9
	5.4.10	Exit Expert Mode	5-9
	5.5	Using Zmacs to Create a Glossary File	5-10
	5.6	Defining Glossary File Format	5-11
	5.7	Defining a Glossary From the Lisp Listener	5-12

6

UCL User Interface

	6.1	Overview	6-1
	6.2	Basic Command Interpreter Operation	6-2
	6.3	Help Features	6-3
	6.3.1	Help Command	6-4
	6.3.2	Command Type-In Help	6-6
	6.3.3	Command Display	6-7
	6.3.4	Command History	6-8
	6.3.5	Command Name Search	6-9
	6.3.6	Keystroke Search	6-10
	6.3.7	Command Menus	6-10
	6.3.7.1	Mouse Documentation Window	6-10
	6.3.7.2	Icons	6-10
	6.3.8	Completion Commands	6-11
	6.3.9	Mouse Documentation Window Help	6-13
	6.4	Environment Customization Features	6-15
	6.4.1	Command Editor	6-16
	6.4.2	Build Keystroke Macro	6-17
	6.4.3	Build Command Macro	6-18
	6.4.4	Save Commands	6-19
	6.4.5	Load Commands	6-19
	6.4.6	Top Level Configurer	6-19
	6.5	Miscellaneous Features	6-21
	6.5.1	Typed Expressions	6-22
	6.5.1.1	Kinds of Expressions	6-22
	6.5.1.2	Implicit Message Sending (rotl)	6-23
	6.5.1.3	Algorithm Used for Typed Expressions	6-23
	6.5.1.4	User Configuration of Type-In Modes	6-24
	6.5.1.5	Special Expressions	6-24
	6.5.2	Obtaining Arguments	6-24
	6.5.3	Numeric Arguments	6-25
	6.5.4	Redo Command	6-25
	6.5.5	System Menu	6-25
	6.5.6	Error Catcher	6-25
	6.6	Command Summary	6-26

Section	Paragraph	Title	Page
7		UCL Programmer Interface	
	7.1	Introduction	7-1
	7.2	Basic Command Interpreter Operation	7-3
	7.3	defcommand and make-command Macros	7-5
	7.3.1	defcommand Macro	7-5
	7.3.2	:arguments Keyword of defcommand	7-9
	7.3.3	make-command Macro	7-12
	7.4	build-command-table Function	7-13
	7.5	build-menu Function	7-15
	7.6	Hints for Developing a UCL Application	7-19
	7.7	Using and Customizing Command Interpreter Flavors	7-20
	7.7.1	ucl:basic-command-loop Flavor	7-20
	7.7.1.1	Basic Instance Variables	7-21
	7.7.1.2	Instance Variables for Reading Typed Input	7-23
	7.7.1.3	Instance Variables for Printing	7-24
	7.7.1.4	Methods	7-25
	7.7.2	ucl:command-loop-mixin Flavor	7-27
	7.7.3	Other Flavors	7-27
	7.7.3.1	ucl:command-and-lisp-typein-window Flavor	7-28
	7.7.3.2	ucl:typein-mode Flavor	7-28
	7.7.3.3	ucl:selective-features-mixin Flavor	7-31
	7.7.4	Global Variables	7-34
	7.7.4.1	History Variables	7-34
	7.7.4.2	Defaults Variables	7-34
	7.7.4.3	Miscellaneous Variables	7-35
	7.8	Miscellaneous Functions	7-36
8		Using Suggestions	
	8.1	Introduction	8-1
	8.2	How to Access Suggestions	8-2
	8.3	Suggestions Menu Windows	8-2
	8.3.1	Suggestions Window Configurations	8-4
	8.3.1.1	Explorer Landscape Video Display	8-4
	8.3.1.2	Portrait Video Display	8-4
	8.3.2	Panes	8-5
	8.4	Menu Examples	8-8
	8.4.1	Lisp Listener Suggestions	8-9
	8.4.2	Zmacs Suggestions	8-12
	8.4.3	Inspector Suggestions	8-14
	8.4.4	Debugger Suggestions Menus	8-15
	8.5	Menu Tools	8-17
	8.5.1	Back to Initial Suggestions	8-18
	8.5.2	Find Commands	8-18
	8.5.3	Select Suggestions Applications	8-18
	8.5.4	Menu History	8-18
	8.5.5	Suggestions Menu Search	8-18
	8.5.6	Suggestions Menus Off	8-18
	8.5.7	Add Menu to Buffer	8-18
	8.5.8	Turn Off Pop-Up Keystrokes	8-19
	8.5.9	Turn On Pop-Up Keystrokes	8-19
	8.5.10	Reorder Menu Items	8-19
	8.5.11	Find Functions for Menu	8-19

Section	Paragraph	Title	Page
	8.5.12	Remove Current Menu	8-19
	8.5.13	Add a Symbol to a Lisp Expressions Menu	8-19
	8.6	Lisp Expressions	8-19
<hr/>			
9		Programming Suggestions	
	9.1	Introduction	9-1
	9.2	Incorporating Suggestions in an Application	9-1
	9.2.1	Building the Suggestions Menus	9-1
	9.2.2	Initializing Suggestions	9-2
	9.2.3	Example of Building Suggestions	9-3
	9.2.4	Detailed Example of Building Suggestions	9-5
	9.3	sugg:suggestions-build-menu Function	9-8
	9.4	sugg:initialize-suggestions-for-application Macro	9-9
	9.5	Triggering Automatic Menu Changes	9-10
	9.5.1	Using Inline Macros to Change Menus Automatically	9-12
	9.5.2	Advising Commands to Change Menus Automatically	9-14
	9.5.3	Changing Menus Through the Active Command Table	9-15
<hr/>			
10		Graphics Editor	
	10.1	Introduction	10-1
	10.1.1	Worlds, Windows, and Transformations	10-2
	10.1.2	Kinds of Objects	10-2
	10.1.3	Grouping Objects	10-3
	10.1.4	Aids for Drawing Objects	10-3
	10.2	Loading and Entering the Graphics Editor	10-3
	10.2.1	Creating the GWIN and GED Systems	10-3
	10.2.2	Entering the Graphics Editor	10-4
	10.2.2.1	Using the SYSTEM Key	10-4
	10.2.2.2	Using the ged:ged Function	10-4
	10.3	Graphics Editor Window	10-4
	10.3.1	Window Layout	10-4
	10.3.2	Buffers	10-6
	10.3.3	Types of Commands	10-7
	10.3.4	Selecting Commands	10-7
	10.3.4.1	Using the Graphics Editor Command Menu	10-7
	10.3.4.2	Using the Graphics Editor Submenus	10-7
	10.3.4.3	Using the Icon Menu	10-7
	10.3.4.4	Using the Keyboard	10-7
	10.3.5	Mouse and Keyboard Operation	10-7
	10.3.5.1	The Mouse Cursor	10-7
	10.3.5.2	The Mouse Buttons	10-8
	10.4	Objects	10-8
	10.4.1	Characteristics of Objects	10-8
	10.4.1.1	Filled and Unfilled Objects	10-8
	10.4.1.2	Color of Objects	10-9
	10.4.1.3	ALU Value	10-10
	10.4.1.4	Status Variables	10-12
	10.4.2	Drawing an Object	10-12
	10.4.2.1	Choosing a Function	10-12
	10.4.2.2	Selecting and Positioning the Object	10-12

Section	Paragraph	Title	Page
	10.4.3	Illustration Conventions	10-13
	10.4.4	Arcs	10-13
	10.4.4.1	Arc Definition	10-13
	10.4.4.2	Drawing an Arc	10-14
	10.4.5	Circles	10-14
	10.4.5.1	Circle Definition	10-14
	10.4.5.2	Drawing a Circle	10-15
	10.4.6	Lines	10-15
	10.4.6.1	Line Definition	10-15
	10.4.6.2	Drawing a Line	10-15
	10.4.7	Paintings	10-15
	10.4.7.1	Painting Definition	10-15
	10.4.7.2	Drawing a Painting	10-16
	10.4.7.3	Status Variables for Paintings	10-17
	10.4.8	Polylines	10-17
	10.4.8.1	Polyline Definition	10-17
	10.4.8.2	Drawing a Polyline	10-17
	10.4.9	Rectangles	10-18
	10.4.9.1	Rectangle Definition	10-18
	10.4.9.2	Drawing a Rectangle	10-18
	10.4.10	Rulers	10-19
	10.4.10.1	Ruler Definition	10-19
	10.4.10.2	Drawing a Ruler	10-19
	10.4.11	Splines	10-20
	10.4.11.1	Spline Definition	10-20
	10.4.11.2	Drawing a Spline	10-20
	10.4.12	Text	10-21
	10.4.12.1	Text Definition	10-21
	10.4.12.2	Drawing Text	10-21
	10.4.12.3	Editing Text	10-21
	10.4.13	Triangles	10-22
	10.4.13.1	Triangle Definition	10-22
	10.4.13.2	Drawing a Triangle	10-22
	10.5	Functions	10-22
	10.5.1	Selecting Objects	10-22
	10.5.1.1	Selecting Objects Individually	10-22
	10.5.1.2	Selecting Objects by Rubber Banding	10-23
	10.5.2	Copying Objects	10-23
	10.5.2.1	Copy	10-23
	10.5.2.2	Drag-Copy	10-24
	10.5.3	Deleting Objects	10-24
	10.5.4	Editing Parameters	10-24
	10.5.5	Exiting	10-25
	10.5.6	Moving Objects	10-25
	10.5.6.1	Move	10-25
	10.5.6.2	Drag-Move	10-26
	10.5.7	Scaling Objects	10-26
	10.5.8	Status Variables	10-27
	10.5.8.1	Modifying Status Variables	10-28
	10.5.8.2	Saving Status Variables	10-28
	10.5.8.3	Restoring Status Variables	10-29
	10.5.8.4	Reverting Status Variables	10-29
	10.5.9	Undoing Commands	10-29
	10.6	Pictures	10-30

Section	Paragraph	Title	Page
	10.6.1	Background Pictures	10-30
	10.6.1.1	Reading a Background	10-30
	10.6.1.2	Clearing a Background	10-30
	10.6.2	Changing Buffers	10-30
	10.6.2.1	Selecting Buffers From the List	10-30
	10.6.2.2	Moving to the Next Buffer	10-30
	10.6.2.3	Moving to the Previous Buffer	10-31
	10.6.2.4	Ordering the Buffers	10-31
	10.6.3	Clearing the Foreground	10-31
	10.6.4	Inserting Pictures	10-31
	10.6.5	Killing and Saving Buffers	10-31
	10.6.5.1	Killing a Buffer	10-31
	10.6.5.2	Saving a Picture	10-31
	10.6.5.3	Writing a Picture to a Specific File	10-31
	10.6.5.4	Killing or Saving Buffers	10-32
	10.6.6	Printing Pictures	10-32
	10.6.6.1	Printing From the Screen	10-32
	10.6.6.2	Printing From a File	10-32
	10.6.7	Reading a Picture File	10-32
	10.6.8	Redrawing the Picture	10-33
	10.6.9	Reverting a Buffer	10-33
	10.7	Presentations	10-33
	10.7.1	Graphics Editor Presentations	10-33
	10.7.2	Defining Presentations	10-33
	10.7.3	Restoring Presentation Definitions	10-34
	10.7.4	Viewing Presentations	10-34
	10.7.4.1	Displaying Presentations	10-34
	10.7.4.2	Loading Presentations	10-34
	10.7.5	Killing and Saving Presentation Definitions	10-34
	10.7.5.1	Killing a Presentation Definition	10-34
	10.7.5.2	Saving a Presentation Definition	10-34
	10.7.5.3	Killing or Saving Presentation Definitions	10-35
	10.7.6	Listing and Selecting Presentations	10-35
	10.7.7	Modifying Presentations	10-35
	10.8	Subpictures	10-36
	10.8.1	Using Subpictures	10-36
	10.8.2	Defining Subpictures	10-36
	10.8.3	Exploding Subpictures	10-37
	10.8.4	Inserting Subpictures	10-37
	10.8.5	Saving Subpicture Definitions	10-37
	10.8.6	Restoring Subpicture Definitions	10-37
	10.8.7	Undefining Subpictures	10-38
	10.9	Windowing	10-38
	10.9.1	Panning	10-38
	10.9.2	Zooming	10-39
	10.9.3	Defining a Rewindow Area	10-39
	10.9.3.1	Setting the Area With the Mouse	10-39
	10.9.3.2	Showing the Entire World	10-40
	10.9.3.3	Showing the Default Window	10-40
	10.10	Command Names, Keystrokes, and Menus	10-40

Section	Paragraph	Title	Page
11		Tree Editor	
	11.1	Overview	11-1
	11.1.1	Tree Editor Display	11-2
	11.1.2	Loading the Tree Editor	11-3
	11.1.3	The Sample Interfaces	11-3
	11.1.3.1	String Displayer	11-3
	11.1.3.2	Flavor Displayer	11-4
	11.1.4	Running the Tree Editor	11-4
	11.2	The Accessor File	11-5
	11.2.1	Adding to the List of Tree Flavors	11-5
	11.2.2	Defining the Function of the Mouse Buttons	11-6
	11.2.2.1	Node Types	11-6
	11.2.2.2	Association List for the Node Types	11-6
	11.2.3	Defining the Flavor	11-7
	11.2.4	Building a Displayable Tree	11-7
	11.2.4.1	The :first-node Method	11-7
	11.2.4.2	The :children-from-data Method	11-7
	11.2.4.3	The :print-name Method	11-8
	11.2.4.4	The :font-type Method	11-8
	11.2.4.5	The :highlight-function Method	11-8
	11.2.4.6	The :find-type Method	11-8
	11.2.4.7	The :handle-node Method	11-9
	11.2.4.8	The :get-new-tree Method	11-10
	11.3	Editing Methods	11-10
	11.3.1	The :add-node-before Method	11-10
	11.3.2	The :add-node-after Method	11-11
	11.3.3	The :add-brother-node Method	11-11
	11.3.4	The :delete-subtree Method	11-11
	11.3.5	The :delete-yourself Method	11-11
	11.3.6	The :get-user-data Method	11-11
	11.4	Tree Editor Functions	11-12
	11.4.1	Formatting for the Scroll Window	11-12
	11.4.2	Redrawing the Tree	11-12
	11.4.3	Expanding and Contracting Nodes	11-12
	11.4.4	Panning and Zooming	11-13
	11.4.5	Displaying Error Messages	11-13
	11.4.6	Changing How a Node is Drawn	11-14
	11.5	Tree Editor Variables	11-14
	11.5.1	Local Variables	11-14
	11.5.2	Global Variables	11-14
12		Font Editor	
	12.1	Introduction	12-1
	12.1.1	Properties of Fonts	12-2
	12.1.2	Fonts in the Video Display	12-3
	12.1.3	Families of Fonts	12-3
	12.1.4	Font Directory	12-4
	12.2	Font Editor Window	12-5
	12.2.1	Command Menus	12-5
	12.2.2	Editing Area	12-6
	12.2.2.1	Grid	12-6
	12.2.2.2	Character Box	12-6

Section	Paragraph	Title	Page
	12.2.2.3	Black Plane	12-7
	12.2.2.4	Gray Plane	12-7
	12.2.3	Registers	12-7
	12.2.4	Label Information	12-8
	12.2.5	Lisp Listener Pane	12-8
	12.2.6	Help Features	12-9
	12.3	Invoking the Font Editor	12-12
	12.4	Exiting the Font Editor	12-12
	12.5	Executing Commands	12-12
	12.6	Selecting a Font to Edit	12-12
	12.6.1	Using the Select Command	12-13
	12.6.2	Using the Directory Command	12-13
	12.6.3	Using the Load Command	12-14
	12.6.4	Using the load Function	12-14
	12.7	Selecting a Character to Edit	12-14
	12.7.1	The Font Display	12-15
	12.7.2	Using the Change Variables Command	12-16
	12.7.2.1	Changing the Sample Font	12-16
	12.7.2.2	Changing the Number of Columns	12-16
	12.7.2.3	Changing the Base of the Number Labels	12-16
	12.7.3	Changing the Configuration by Setting Variables	12-16
	12.8	Editing a Character	12-17
	12.8.1	Changing Drawing Mode	12-17
	12.8.2	Choosing Mouse Cursors	12-17
	12.8.3	Drawing	12-18
	12.8.3.1	Using the Mouse to Set Pixels	12-18
	12.8.3.2	Using the Keyboard to Set Pixels	12-18
	12.8.3.3	Drawing Operations	12-18
	12.8.4	Swapping and Merging	12-19
	12.8.5	Moving the Contents of the Editing Area	12-19
	12.8.6	Adjusting the Character Placement	12-19
	12.8.6.1	Moving the Box	12-19
	12.8.6.2	Moving the Character in the Gray Plane	12-19
	12.8.7	Editing Operations	12-20
	12.8.8	Erasing the Contents	12-21
	12.9	Examining a Character	12-22
	12.9.1	Adjusting the Grid Scale	12-22
	12.9.2	Modifying the Sample String	12-22
	12.10	Saving a Character	12-23
	12.10.1	Holding in the Register	12-23
	12.10.2	Saving in the Font	12-23
	12.11	Writing a Font to a File	12-24
	12.12	Creating a Modified Font	12-24
	12.12.1	Using the Font Editing Operations	12-24
	12.12.2	Modifying a Font	12-26
	12.13	Creating a Font From Scratch	12-27
	12.14	Performance Considerations	12-28
	12.14.1	Execution Time	12-28
	12.14.2	Memory Allocation	12-28
	12.15	AST Files	12-30
	12.16	Command Summary	12-31

Section	Paragraph	Title	Page
13		Debugger (Error Handler)	
	13.1	Introduction	13-1
	13.2	Entering the Debugger	13-1
	13.3	How to Use the Debugger	13-3
	13.3.1	Debugger Commands	13-5
	13.3.1.1	Examining Stack Frames	13-6
	13.3.1.2	Examining Arguments, Locals, Functions, and Values	13-7
	13.3.1.3	Examining Special Variables	13-10
	13.3.1.4	Resuming Execution	13-10
	13.3.1.5	Stepping Through Function Calls and Returns	13-10
	13.3.1.6	Transferring to Other Systems	13-11
	13.3.2	Summary of Debugger Commands	13-11
	13.4	Debugging After a Warm Boot	13-11
14		Window-Based Debugger	
	14.1	Introduction	14-1
	14.2	How to Use the Window-Based Debugger	14-1
	14.3	Deexposed Windows and Background Processes	14-4
15		Inspector	
	15.1	Introduction	15-1
	15.2	Lisp Listener Pane	15-3
	15.3	Inspection Panes	15-4
	15.4	History Pane	15-6
	15.5	Command Menu Pane	15-7
16		Flavor Inspector	
	16.1	Overview	16-1
	16.2	Lisp Listener Pane	16-2
	16.3	Inspection Panes	16-4
	16.4	History Pane	16-7
	16.5	Command Menu Pane	16-8
17		Peek	
	17.1	Introduction	17-1
	17.2	Mode and Command Menu Windows	17-3
	17.3	Viewing Window	17-3
	17.3.1	Processes	17-4
	17.3.2	Counters	17-6
	17.3.3	Areas	17-7
	17.3.4	File Status	17-8
	17.3.5	Windows	17-10
	17.3.6	Servers	17-12
	17.3.7	Network	17-14

Section	Paragraph	Title	Page
	17.3.8	Function Histogram	17-16
	17.3.9	Host Status	17-18
18		Trace	
19		Stepper	
20		Evalhook	
21		Advise	
	21.1	Advising a Function	21-1
	21.2	Designing the Advice	21-3
	21.3	:around Advice	21-4
	21.4	Advising One Function Within Another	21-4
22		Breakon	
23		MAR	
24		Crash Analysis	
	24.1	Introduction	24-1
	24.2	Crash Reporting	24-1
	24.3	Preparing NVRAM	24-2
	24.4	Crash Analyzer Functions	24-2
	24.5	Shutdown Record Analysis Format	24-4
	24.6	Hardware Crash Descriptions and Troubleshooting	24-7
	24.6.1	NuBus Crashes	24-8
	24.6.2	Processor Fault Crashes	24-8
	24.6.3	Power Fail Crash	24-9
	24.6.4	Mass Storage Subsystem Crashes	24-9
	24.6.5	Troubleshooting NUPI Device and Controller Error Crashes	24-11
	24.6.6	Troubleshooting NUPI Special Event Crashes	24-14
	24.7	The Force Crash Keychord	24-15
	24.8	Software Crash Descriptions	24-15

Section	Paragraph	Title	Page
25		Miscellaneous Debugging Functions	
	25.1	Introduction	25-1
	25.2	Describe Functions	25-2
	25.3	Apropos Functions	25-7
	25.4	Who-Calls Functions	25-11
	25.5	Property List Functions	25-13
	25.6	Print Functions	25-13
	25.7	Dribble File Functions	25-16
	25.8	Environment Functions and Variables	25-18
26		Lisp Listener and Break	
27		Performance Tools	
	27.1	Overview	27-1
	27.1.1	Metering Overview	27-1
	27.1.2	Timing Macros Overview	27-1
	27.1.3	Function Histogram Overview	27-2
	27.2	Metering	27-2
	27.2.1	Setting Up a Meter Partition	27-5
	27.2.2	Controlling Metered Data	27-6
	27.2.3	Evaluating Forms With Metering	27-7
	27.2.4	Analyzing the Metered Data	27-8
	27.2.4.1	Analyzer	27-9
	27.2.4.2	Use Previous Meter Info?	27-10
	27.2.4.3	Sort Function	27-10
	27.2.4.4	Inclusive?	27-11
	27.2.4.5	Output Type	27-11
	27.2.4.6	Find Callers	27-11
	27.2.4.7	Summarize	27-12
	27.2.4.8	Output File	27-12
	27.2.4.9	Edit Buffer	27-12
	27.2.4.10	Call Tree Inspector	27-12
	27.2.4.11	Call Tree Inspector Examples	27-14
	27.2.5	Resuming Garbage Collection	27-23
	27.2.6	Customized Metering Sessions	27-23
	27.2.6.1	Evaluating Forms	27-23
	27.2.6.2	Examples	27-25
	27.2.6.3	Another Meter Analysis Function	27-28
	27.3	Timing Macros	27-30
	27.4	Function Histogram	27-41
	27.4.1	Peek Function Histogram	27-41
	27.4.2	Function Histogram Functions	27-42

Section	Paragraph	Title	Page
	31.5.3	Mail Functions	31-51
	31.5.4	Mail Variables	31-52
	31.5.4.1	Mail File and Inbox Variables	31-52
	31.5.4.2	Mail Window and Buffer Variables	31-53
	31.5.4.3	Mail Template Buffer Variables	31-55
	31.5.4.4	Reformat Header Variables	31-56
	31.5.4.5	Miscellaneous Variables	31-57

32

Namespace Utilities

	32.1	Overview	32-1
	32.2	Namespace Concepts	32-1
	32.2.1	Objects	32-2
	32.2.2	Aliases	32-3
	32.2.3	Retrieving Objects From a Namespace	32-4
	32.2.4	Types of Namespaces	32-5
	32.2.4.1	Public Namespace	32-5
	32.2.4.2	Personal Namespace	32-6
	32.2.4.3	Symbolics Namespace	32-6
	32.2.4.4	Basic Namespace	32-7
	32.2.5	Namespace Pathnames	32-8
	32.2.5.1	Personal Namespace Pathnames	32-8
	32.2.5.2	Public Namespace Pathnames	32-8
	32.2.6	Default Attributes	32-9
	32.2.6.1	Default Attributes for Personal Namespace	32-9
	32.2.6.2	Default Attributes for Public Namespace	32-10
	32.2.6.3	Default Attributes for Symbolics Namespace	32-11
	32.2.7	Namespace Operations	32-11
	32.2.7.1	General Namespace Operations	32-12
	32.2.7.2	Changing a Namespace	32-12
	32.2.7.3	Finding and Accessing Objects	32-12
	32.3	Namespace Editor (NSE)	32-13
	32.3.1	Accessing the Namespace Editor	32-14
	32.3.2	Configuring the Namespace	32-15
	32.3.3	Basic NSE Operations	32-16
	32.3.4	Symbol Codes	32-18
	32.3.5	General Commands Menu	32-19
	32.3.6	Class Commands	32-20
	32.3.7	Object Commands	32-27
	32.3.8	Attribute Commands	32-29
	32.3.9	Group Attribute Commands	32-32
	32.3.10	Command Summary	32-34
	32.4	NSE Customization	32-36
	32.4.1	Customization Variables	32-36
	32.4.2	Filters	32-37
	32.4.3	Horizontal Formats	32-38
	32.4.4	Expert Editors	32-39
	32.4.4.1	nse:define-nse-expert-editor Macro	32-40
	32.4.4.2	Spec-List Formats	32-44
	32.4.4.3	Expert Editor Variables	32-48
	32.4.4.4	Verification Routine Macros	32-49
	32.4.4.5	Viewing Expert Editors	32-50

Section	Paragraph	Title	Page
	32.5	User Functions	32-51
	32.5.1	Namespace Functions	32-53
	32.5.2	Modification Functions	32-58
	32.5.3	Retrieval Functions	32-62
	32.5.4	Object Manipulation Functions	32-68
	32.5.5	Miscellaneous Functions	32-69
	32.6	Error Messages	32-71
<hr/>			
33		Miscellaneous Network Functions	
	33.1	Introduction	33-1
	33.2	Host Status	33-1
	33.3	Resetting the Network	33-2
	33.4	Eval Serving	33-2
	33.5	Fingering Hosts	33-3
	33.6	Sending and Printing Notifications	33-4
<hr/>			
34		Color Map Editor	
	34.1	Introduction	34-1
	34.2	Loading the Color Map Editor Software	34-2
	34.3	Invoking the Color Map Editor	34-2
	34.4	Editing and Defining Colors	34-3
	34.5	Color Editor Commands	34-5
	34.6	Color Map Commands	34-6
	34.7	Command Summary	34-8
<hr/>			
35		Visidoc	
	35.1	Introduction	35-1
	35.2	Installing the Visidoc Client Software	35-2
	35.3	Invoking Visidoc	35-2
	35.4	Exiting Visidoc	35-2
	35.5	Features of Visidoc	35-3
	35.5.1	Memory Characteristics	35-3
	35.5.2	Display Characteristics	35-3
	35.5.3	Using Visidoc	35-3
	35.6	Making a Host a Visidoc Server	35-5
	35.7	Maintenance of the Visidoc Server Namespace	35-8
	35.7.1	Objects in the :namespace Class	35-8
	35.7.2	Saving Your Changes	35-8
	35.7.3	Maintenance of the Network Namespace :site Option	35-8

Appendix	Paragraph	Title	Page
A		Explorer Fonts	A-1
B		Command Tables	B-1

Index

	Figure	Title	Page
Figures	1-1	Initial Screen of the New User Window	1-2
	2-1	Typical Profile Utility Window	2-2
	5-1	The Glossary Frame	5-3
	6-1	UCL Menu of Help Options	6-5
	6-2	Inspector Help	6-5
	6-3	Command Type-In Help	6-6
	6-4	Command Display	6-7
	6-5	Command History	6-8
	6-6	Command Name Search	6-9
	6-7	Mouse Documentation Window and an Icon Menu	6-11
	6-8	Completion Listing Produced by Pressing SUPER-/	6-13
	6-9	Function Argument List in Mouse Documentation Window	6-14
	6-10	Customization Menu	6-15
	6-11	Command Editor Display and Menu Used for Editing a Command	6-17
	6-12	Build Command Macro	6-18
	6-13	Top Level Configurer Window	6-21
	8-1	Suggestions System Menu Interaction	8-3
	8-2	Suggestions Frame, Explorer Landscape Configuration	8-4
	8-3	Suggestions Frame, Portrait Configuration	8-5
	8-4	Listener Suggestions Basic Menu	8-6
	8-5	Zmacs Suggestions Menu With Keystroke Display	8-7
	8-6	Evaluation of a Symbol Using Lisp Listener Suggestions	8-8
	8-7	Invoking the Display Recent Deletions Menu Using Listener Suggestions	8-9
	8-8	Result of Selecting the Listener List Menus Option	8-10
	8-9	Result of Selecting the List Apropos Completions Option	8-11
	8-10	Result of Selecting the Zmacs List Menus Option	8-12
	8-11	Example Keystroke Commands Listed in Zmacs Suggestions	8-13
	8-12	Inspector Suggestions Menus	8-14
	8-13	Debugger Suggestions Menus	8-16
	8-14	Menu Tools Pop-Up Window	8-17
	8-15	Lisp Expressions Menus	8-20
	11-1	Vertical Window Display	11-2
	11-2	Horizontal Window Display	11-3
	12-1	Font Editor Command Display	12-10
	12-2	SYMBOL-HELP Keyboard Map	12-11

Figure	Title	Page
14-1	Window-Based Debugger	14-2
15-1	Inspector Frame	15-2
16-1	Flavor Inspector Frame	16-2
17-1	Example of the Peek Window	17-2
17-2	Example of the Peek Processes Screen	17-5
17-3	Example of Counters Screen	17-6
17-4	Example of the Areas Screen	17-7
17-5	Example of File Status Screen	17-9
17-6	Example of the Peek Windows Screen	17-11
17-7	Example of Peek Servers Screen	17-13
17-8	Example of Peek Network Screen	17-15
17-9	Example of Peek Function Histogram Screen	17-16
17-10	Example of Peek Host Status Screen	17-18
18-1	Trace Menu	18-1
24-1	Sample Output From report-all-shutdowns	24-4
27-1	Sample Call Tree Inspector Display	27-4
27-2	Meter-Analyze Menu	27-8
27-3	Call Tree Inspector Windows	27-13
27-4	Call Tree Inspector — Initial Screen	27-15
27-5	Call Tree Inspector — Clicking L	27-16
27-6	Call Tree Inspector — Middle Window Items	27-17
27-7	Call Tree Inspector — Clicking L and M	27-18
27-8	Call Tree Inspector — Clicking L2	27-20
27-9	Call Tree Inspector — Clicking L on a * n Function	27-21
27-10	Call Tree Inspector — Clicking L2 on a * n Function	27-22
27-11	Example of Peek Function Histogram Screen	27-42
31-1	Typical Summary Buffer	31-3
31-2	Typical Message Buffer	31-4
31-3	List Mail Buffers Display	31-19
32-1	Namespace Editor Display	32-13
32-2	Horizontally Expanded Class	32-25
34-1	Color Map Editor Display	34-4

Table	Title	Page	
Tables	6-1	Completion Commands	6-12
	6-2	UCL Keystroke Commands	6-26
	10-1	Color Values for Graphic Methods for Monochrome Environments	10-9
	10-2	Named Colors in the Default Color Map	10-10
	10-3	Common ALU Values	10-11
	10-4	Color ALU Operations	10-12
	10-5	Graphics Editor Keystroke Assignments	10-40
	12-1	Some Commonly Used Fonts	12-4
	12-2	Font Editor Keystroke Assignments	12-31

Table	Title	Page
13-1	Special Variable Bindings in the Debugger	13-4
13-2	Summary of Debugger Commands	13-12
14-1	Window-Based Debugger Commands	14-3
15-1	Summary of Inspector Keyboard Commands	15-3
15-2	Inspector Display According to Object Type	15-6
16-1	General Flavor Inspector Commands	16-4
16-2	Flavor Commands	16-6
16-3	Method Commands	16-7
17-1	Peek Modes and Commands	17-3
19-1	Stepper Commands	19-3
24-1	NUPI Device and Controller Error Codes	24-11
24-2	NUPI Special Event Codes	24-14
24-3	Software Crash Descriptions	24-16
28-1	Telnet Commands	28-3
29-1	VT100 Commands	29-3
30-1	Converse Commands	30-2
31-1	Explorer Mail Reader Keystroke Commands	31-38
32-1	Namespace Editor Keystroke Commands	32-34
34-1	Color Map Editor Commands	34-8
B-1	Completion Commands	B-2
B-2	UCL Keystroke Commands	B-3
B-3	Font Editor Keystroke Assignments	B-4
B-4	Graphics Editor Keystroke Assignments	B-5
B-5	Summary of Debugger Commands	B-7
B-6	Window-Based Debugger Commands	B-9
B-7	Summary of Inspector Keyboard Commands	B-11
B-8	General Flavor Inspector Commands	B-13
B-9	Peek Modes and Commands	B-14
B-10	Stepper Commands	B-15
B-11	Telnet Commands	B-16
B-12	VT100 Commands	B-17
B-13	Converse Commands	B-18
B-14	Explorer Mail Reader Keystroke Commands	B-19
B-15	Namespace Editor Keystroke Commands	B-21
B-16	Color Map Editor Commands	B-23

ABOUT THIS MANUAL

Contents of This Manual

The *Explorer Tools and Utilities* manual discusses the tools and utilities available on the Explorer system. The manual includes the following sections and appendixes, as well as an index:

Section 1: New User — Performs helpful operations for a new user, such as setting up a personal directory.

Section 2: Profile — Provides an easy way to change the system environment by changing the values of variables.

Section 3: Login Initialization File — Allows you to customize your environment when you log in.

Section 4: Bug Reporting — Provides an easy-to-use menu for reporting bugs.

Section 5: Glossary Utility — Provides an online glossary of terms relating to the Lisp language and the Explorer system.

Section 6: UCL User Interface — Tells how to use the Universal Command Loop (UCL), which simplifies and standardizes the process of constructing command interfaces to interactive programs.

Section 7: UCL Programmer Interface — Shortens program development time by providing a ready-to-use command interpreter with built-in help features.

Section 8: Using Suggestions — Tells how to use the Suggestions system, which displays menus appropriate for different contexts of an application.

Section 9: Programming Suggestions — Tells how to add Suggestions to your application.

Section 10: Graphics Editor — Allows you to interactively create and modify pictures consisting of graphics objects.

Section 11: Tree Editor — Allows you to display any kind of data organized in a tree structure.

Section 12: Font Editor — Describes how to create and modify fonts.

Section 13: Debugger (Error Handler) — Allows you to examine the environment in which an error condition is signaled in your program.

Section 14: Window-Based Debugger — Provides a window-based alternative to the regular debugger.

Section 15: Inspector — Provides a window-based utility for observing and modifying Lisp objects.

Section 16: Flavor Inspector — Provides a window-based utility for observing and modifying flavors.

Section 17: Peek — Provides a window-based utility that displays a continually updating system status of items such as processes, windows, network protocols, and file system activity.

Section 18: Trace — Allows you to trace certain functions and macros. When a function is traced, certain special actions are taken when the function is called and when it returns.

Section 19: Stepper — Allows you to follow every step of the evaluation of a form and examine what is going on.

Section 20: Evalhook — Allows you to change the way the evaluator works or to write your own evaluator.

Section 21: Advise — Allows you to tell a function to do something in addition to its actual definition.

Section 22: Breakon — Allows you to request that the debugger be entered whenever a certain function is called. When the function is called, you can evaluate lexically scoped variables and see who is calling the function and with what arguments.

Section 23: MAR — Signals a MAR break condition when a particular word or words in memory are referenced. The MAR can be a useful debugging tool if you want to know when a particular memory location (a program variable, for example) is read or written.

Section 24: Crash Analysis — Describes the Explorer crash reporting and analysis utilities.

Section 25: Miscellaneous Debugging Functions — Provides an assortment of helpful debugging functions.

Section 26: Lisp Listener and Break — Discusses the functions and variables available in the Lisp top-level, read-eval-print loop.

Section 27: Performance Tools — Provides details on the metering system, timing macros, and function histogram.

Section 28: Telnet — Allows you to use the Explorer screen as a terminal to another host.

Section 29: VT100 Emulator — Allows you to use the Explorer screen as a VT100 terminal.

Section 30: Converse — Discusses the interactive message editor that displays all messages that you have sent and received.

Section 31: Mail — Discusses the electronic mail system that allows the exchange of messages between users on a computer network.

Section 32: Namespace Utilities — Allow you to create a database for your network configuration as well as create other types of databases, such as databases for your personal use.

Section 33: Miscellaneous Network Functions — Discusses an assortment of helpful functions for network activities.

Section 34: Color Map Editor — Tells you how to edit the colors in the color map on an Explorer color system.

Section 35: Visidoc — Describes the Visual Interactive Documentation Online Manual Viewer (Visidoc), which allows you to view documentation from online reference manuals, such as the *Explorer Window System Reference* and the *Explorer Lisp Reference*.

APPENDIXES

Appendix A: Explorer Fonts — Shows the fonts that are automatically loaded with the Explorer system.

Appendix B: Command Tables — Lists the command tables that are available for utilities such as the font editor, graphics editor, debugger, Stepper, Mail, and so on. You can also find these tables in their respective sections.

Notational Conventions

The following paragraphs describe the notation for keystroke sequences, mouse clicks, and Lisp syntax.

Keystroke Sequences

Many of the commands used with the Explorer system are executed by a combination or sequence of keystrokes. Keys that should be pressed at the same time, or *chorded*, are listed with a hyphen connecting the name of each key. The following table explains the conventions used in this manual to describe keystroke sequences.

Keyboard Sequence	Interpretation
META-CTRL-D	Hold the META and CTRL keys while pressing the D key.
CTRL-X CTRL-F	Hold the CTRL key and press the X key, release the X key, and then press the F key. Alternately, press CTRL-X, release both keys, and press CTRL-F.
META-X Find File RETURN	Hold the META key while pressing the X key, release the keys, type the letters find file and then press the RETURN key.
TERM - SUPER-HELP	Press the TERM key and release it, press the minus key (-) and release it, then press and hold the SUPER key while pressing the HELP key.

Mouse Clicks The mouse has three buttons that enable you to execute operations from the mouse without returning your hand to the keyboard. Pressing and releasing a button is called *clicking*. The following table lists abbreviations used to describe clicking the mouse.

Abbreviation	Action
L	Click the left button (press the left button once and release).
M	Click the middle button (press the middle button once and release).
R	Click the right button (press the right button once and release).
L2, M2, R2	Click the specified button twice quickly. Alternately, you can press and hold the CTRL key while you click the specified button.
LHOLD, MHOLD, RHOLD	Press the specified button and hold it down.

Lisp Language Notation The Lisp language notational convention helps you distinguish Lisp functions and arguments from user-defined symbols. The following table shows the three fonts used in this manual to denote Lisp code.

Typeface	Meaning
boldface	System-defined words and symbols, including names of functions, macros, flavors, methods, variables, keywords, and so on—any word or symbol that appears in the system source code.
<i>italics</i>	Example names or an argument to a function, such as a value or parameter you would fill in. Names in italics can be replaced by any value you choose to substitute. (Italics are also used for emphasis and to introduce new terms.)
<code>monowidth</code>	Examples of program code and output are in a monowidth font. System-defined words shown in an example are also in this font.

For example, this sentence contains the word **setf** in boldface because **setf** is defined by the system.

Section	Paragraph	Title	Page
28		Telnet	
	28.1	Introduction	28-1
	28.2	Entering a Telnet Window	28-2
	28.3	Telnet Commands	28-3
	28.4	Telnet Server	28-4
29		VT100™ Emulator	
30		Converse	
	30.1	Introduction	30-1
	30.2	Zmacs Editor Commands With Converse	30-2
	30.3	Converse Functions	30-3
	30.4	User Options With Converse	30-4
31		Mail	
	31.1	Introduction	31-1
	31.2	Mail Reader — Getting Started	31-1
	31.2.1	Entering and Exiting the Mail Reader	31-2
	31.2.2	Getting Help	31-5
	31.2.3	Mail Displays	31-5
	31.2.4	Executing Mail Commands	31-7
	31.2.5	Basic Mail Commands	31-8
	31.3	Mail Reader Commands	31-11
	31.3.1	Mail Files and Inboxes	31-11
	31.3.1.1	Mail File Formats	31-12
	31.3.1.2	Inbox Locations	31-12
	31.3.1.3	Mail File and Inbox Commands	31-13
	31.3.2	Mail Buffers and Windows	31-16
	31.3.3	Mail Messages	31-22
	31.3.3.1	Selecting and Viewing Messages	31-24
	31.3.3.2	Sending Mail	31-27
	31.3.3.3	Deleting and Expunging Messages	31-31
	31.3.3.4	Editing Messages	31-33
	31.3.3.5	Printing Messages	31-33
	31.3.3.6	Message Keywords	31-34
	31.3.3.7	Miscellaneous Mail Commands	31-34
	31.3.3.8	Command Summary	31-38
	31.4	Mailer	31-40
	31.4.1	Mail Addresses	31-40
	31.4.1.1	Mailing Lists	31-40
	31.4.1.2	User Mail Forwarding	31-42
	31.4.2	Address Routing	31-43
	31.4.2.1	Options Affecting Address Routing	31-43
	31.4.2.2	Address-Routing Algorithm	31-45
	31.4.3	Mail Daemon	31-46
	31.5	Customizing the Mail System	31-46
	31.5.1	Defining Mail Filters	31-48
	31.5.2	Defining Mail Template Buffers	31-48

Some function and method names are very long—for example, `get-unicode-version-of-band`. Within the text, long function names may be split over two lines because of typographical constraints. When you code the function name `get-unicode-version-of-band`, however, you should not split it or include any spaces within it.

Within manual text, each example of actual Lisp code is shown in the monowidth font. For instance:

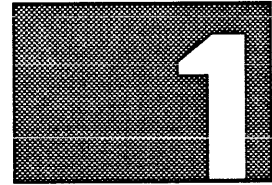
```
(setf x 1 y 2) => 2  
(+ x y) => 3
```

The form `(setf x 1 y 2)` sets the variables `x` and `y` to integer values; then the form `(+ x y)` adds them together.

In this example of Lisp code with its explanation, `setf` appears in the monowidth font because it is part of a specific example.

For more detailed information about Lisp syntax descriptions, see the *Explorer Lisp Reference*.

NEW USER



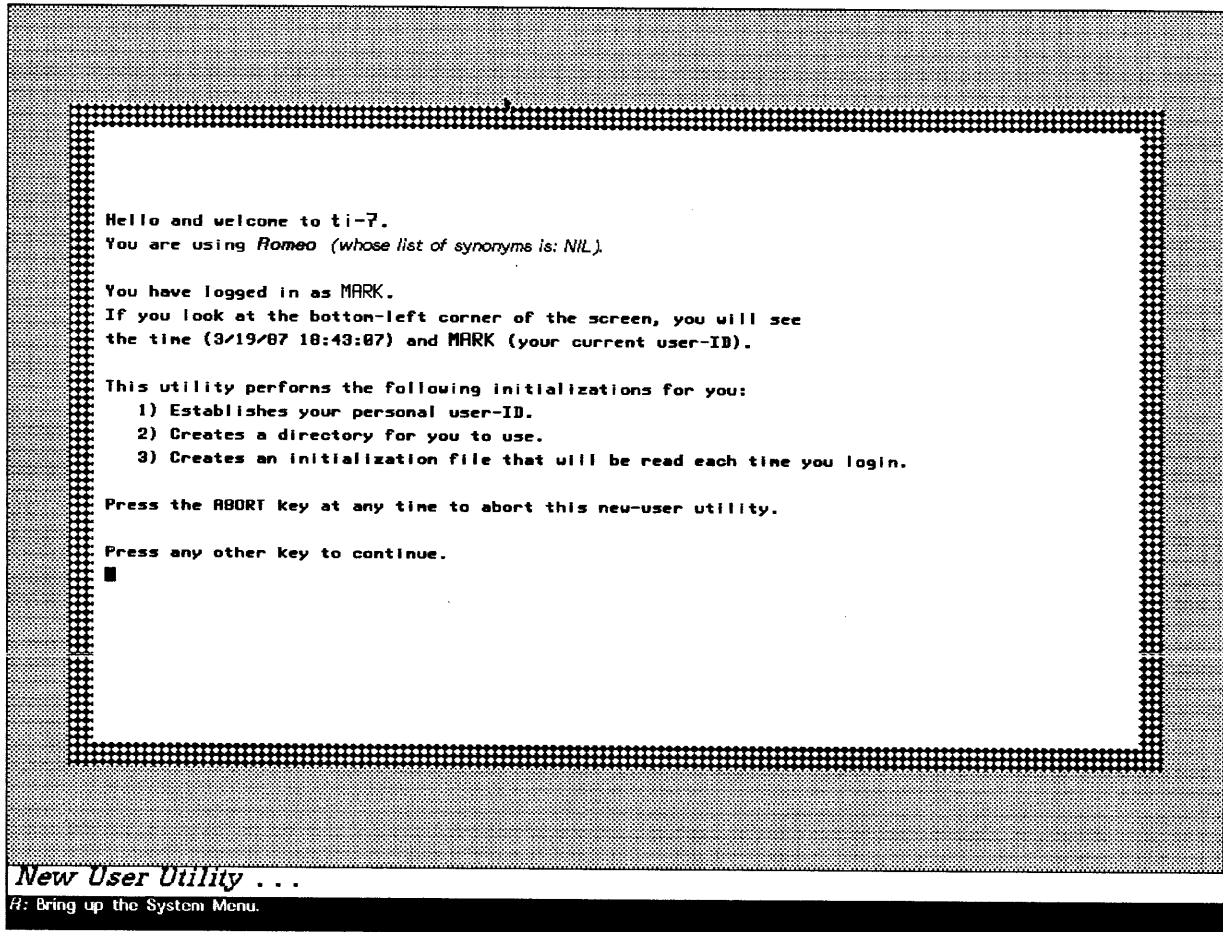
The New User utility does the following for you:

- Identifies you to other users on the network
- Lists the name and synonyms of the machine you are using
- Creates a user directory in your name
- Creates an initialization file in your user directory
- Describes how to log out this session and how to log in for your next session on this machine

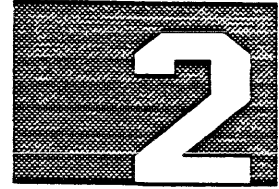
Because the New User utility creates a new directory each time you invoke it, you should execute this utility only once for each login directory you want to create. (You may want a different environment for working on different programs, for example.) The system you are using must include a local file system where the utility can create a directory.

You can execute the New User utility by selecting the New User item from the System menu or by issuing the `(new-user)` function to the Lisp Listener. Answer the prompts to complete the utility. The initial screen that appears is similar to that shown in Figure 1-1.

Figure 1-1 Initial Screen of the New User Window



PROFILE



Introduction

2.1 The Profile utility provides a simple way to change the system environment by changing the values of variables. You can change your environment for the current session; in addition, you can create an initialization file that sets up the same environment each time you log in and returns the environment to its default condition when you log out.

Requirements

2.2 Before you can use all the features of the Profile utility, you must fulfill certain requirements:

- To use the Profile utility, you must be logged in.
- To create an initialization file, you must have an existing *login* directory, which is a directory with the same name as your login name.

Accessing the Profile Utility

2.3 To access the Profile utility, you select the Profile option in the System menu or by issuing the `(profile)` function to the Lisp Listener. The Profile utility displays a window that consists of three parts: a choose-variable-values menu, a menu of actions or commands, and a menu of types of variables. When you first invoke the Profile utility, the menu displayed at the top is the Important Variables menu, as indicated by the item Important Variables displayed in reverse video. A typical window displayed by the Profile utility is shown in Figure 2-1. From the Profile utility window, you can change the values of variables or execute commands.

Figure 2-1 Typical Profile Utility Window

```

#PRINT-BASE# : ..... Binary Octal Decimal Hexidecimal
#PRINT-LENGTH# : ..... NIL
#PRINT-LEVEL# : ..... NIL
#PRINT-STRUCTURE# : ..... T [yes] NL [no]
#READ-BASE# : ..... Binary Octal Decimal Hexidecimal
PROFILE::CLOCK-FORMAT : ..... 24 Hour Clock 12 Hour Clock
EH:#ENTER-WINDOW-DEBUGGER# : ..... NL [never use it] T [ask] :ALWAYS [always use it]
EH:#USE-OLD-DEBUGGER# : ..... T [yes] NL [no]
PROFILE::KEYCLICK-STATE : ..... On Off
PROFILE::LOAD-PATCHES-AT-LOGIN : ..... T [yes] NL [no]
PROFILE::NUMBER-OF-ROUSE-DOCUMENTATION-LINES : 2
PROFILE::P-LISP-MODE : ..... COMMON-LISP ZETALISP
PROFILE::SAVE-BUFFERS-AT-LOGOUT-ACTION : ..... Via Menu Via Prompts Don't Save Buffers
PROFILE::SCREEN-REVERSE-VIDEO-FUNCTION : ..... Black on White White on Black
PROFILE::SUGGESTIONS-MENUS-ON? : ..... T [yes] NL [no]
TU::#FLAVOR-INSPECTOR-CONFIGURATIONS : ..... THREE-PANES ONE-PANE TWO-HORIZONTAL-PANES TWO-VERTICAL-PANES
TU::#INSPECTOR-CONFIGURATIONS : ..... THREE-PANES ONE-PANE TWO-HORIZONTAL-PANES TWO-VERTICAL-PANES
TU::MORE-PROCESSING-GLOBAL-ENABLE : ..... T [yes] NL [no]
ZWEI::#DEFAULT-MAJOR-MODE# : ..... COMMON-LISP ZETALISP TEXT MACSYMA MIDAS TEX ZTOP
ZWEI::#INITIAL-MINOR-MODE# : ..... NIL
ZWEI::#REGION-MARKING-MODE# : ..... UNDERLINE REVERSE-VIDEO
    
```

Actions

Store Options	Restore System Defaults	Restore User Defaults	Exit
---------------	-------------------------	-----------------------	------

Variables currently displayed:

UCL Variables	Error Handling Variables	Network Variables	GC Variables
Compiler Variables	Zmacs Variables	Mouse Variables	Input Variables
File System Variables	Evaluation Variables	Common Lisp Globals	Display Variables
			Color System Variables

Profile Pane

L: move to an item and select it, H: move to an item and exit it.
 Press the HELP key for command descriptions.

Accessing Variables 2.4 The variables changed by the Profile utility are grouped according to the area they affect, such as Input Variables, Display Variables, Zmacs Variables, and so on. The Important Variables grouping includes variables from several other areas that are frequently changed.

To edit a group of variables, you select the group name in the bottom menu. When you select a different area of variables to change, the previously selected item is displayed in normal video, the newly selected item is displayed in reverse video, and the displayed variables are replaced by a menu of variables from the new area.

You can edit the values of the variables listed in the display just as you would edit the values in a choose-variables-values menu. A description of the variable appears in the mouse documentation window.

**Commands in
the Profile Utility**

2.5 In addition to changing the values of variables, you can select one of the actions listed in the Actions menu.

Command Name	Description
Store Options	Creates or updates a special initialization file in the directory with the same name as your login name. This initialization file specifies your login environment to be the same as the current environment. When you log out, the system returns most variables to their default values.
Restore System Defaults	Restores the original system default values to all variables.
Restore User Defaults	Restores the default values listed in the initialization file for the login directory.
Exit	Ends the Profile utility and returns to the window from which you called it. You should select the Exit item to quit the utility.

Typical Variables

2.6 The following variables from the Input Variables menu are examples of the kinds of items that can be changed by using the Profile utility:

Label	Description
Continuous Repeat Delay	Sets the amount of delay in 60ths of a second between each repeated appearance of a character while you hold that key.
Initial Repeat Delay	Sets the amount of delay in 60ths of a second between the time you press a key and the time the key begins to repeat. Setting this value to zero disables the repeating features of keys.
Mouse Handedness	Specifies whether the mouse is a right-handed or a left-handed mouse.
Keyclick State	Enables or disables the audible key click.
Read Base	Specifies the base or radix of numbers you type. By default, the system assumes that you type decimal numbers rather than hexadecimal, octal, or binary numbers.

Customizing Profile 2.7 You can use the `profile:define-profile-variable` macro and the `profile:profile-setq` function to customize Profile. The `profile:define-profile-variable` defines a Profile variable. The `profile:profile-setq` function initializes a Profile variable. You can use `profile:profile-setq` forms in your login initialization file.

`profile:define-profile-variable` *variable variable-classes* Macro
 &key :cvv-type :declare-special-p :documentation :get-value
 :set-effect :name :variable-init :long-time-to-set-p :form-for-init-file

The `profile:define-profile-variable` macro defines a Profile variable. You can use this macro to make any bound variable known to the Profile utility. After the first two arguments, all arguments are optional keyword arguments. All arguments should be unquoted (they are not evaluated).

Arguments: *variable* — A symbol (bound variable) that is being defined as a Profile variable.

variable-classes — A list of keywords designating the Profile classes that contain the *variable*. This list of keywords tells Profile in which menu or menus this variable should be displayed. The following are recognized keywords:

- :important — Variables that are most often modified by users.
- :ucl — Variables that affect the Universal Command Loop (UCL).
- :network — Variables that affect the way the network works.
- :file — Variables that affect the file system.
- :mouse — Variables that affect the way the mouse works.
- :compile — Variables that affect the way the compiler works.
- :display — Variables that affect the display.
- :common — Common Lisp global variables.
- :eval — Variables that affect the evaluation of code.
- :input — Variables that affect the way the mouse works.
- :zmacs — Variables that affect the way Zmacs works.
- :mail — Variables that affect the Mail system.
- :gc — Variables that affect the way garbage collection (GC) works.
- :error — Variables that affect error handling.

:cvv-type — A choose-variable-values (CVV) keyword or a list whose first element is a CVV type keyword. Any valid CVV menu item type works. This keyword controls how the variable is displayed and how it is updated. Refer to the *Explorer Window System Reference* for details.

:declare-special-p — Declares a Profile variable as special. This is only done with variables unique to the Profile utility.

:documentation — The documentation string for the variable. This keyword defaults to the string that is already present.

:get-value — The form to evaluate to obtain the value of this variable.

:set-effect — The form to evaluate after each time the value of this variable is changed within the Profile utility.

- :name** — The name of the variable used for display purposes. If unspecified, the print name of the variable is used.
- :variable-init** — The form to evaluate to initialize this variable upon entering the Profile utility.
- :long-time-to-set-p** — This keyword should be non-nil if the variable takes more than a few seconds to set and execute the proper **:set-effect**. The default is nil.
- :form-for-init-file** — A one-argument lambda expression or function name that returns a form appropriate for setting the Profile variable to the correct value. The Profile variable is the argument to the lambda expression.

Example:

```
(profile:define-profile-variable user:my-variable (:important)
  :cvv-type :string-or-nil
  :documentation "Some variable I want in Profile."
  :variable-init "Initial value"
  :form-for-init-file (lambda (var) `(profile:profile-setq ,var
    ,user:my-variable)))
```

profile:profile-setq *variable value* Function

Initializes a *variable* recognized by the Profile utility to *value*.

Also note that **profile:profile-setq** forms appear in your login initialization file after you have chosen the Store Options menu item from the Profile utility. These **profile:profile-setq** forms initialize the variables to your customized values as well as perform some other side effects for the Profile utility. The **profile:profile-setq** function is similar to the **login-setq** function because it places the variable on the **logout-list**. (Refer to Section 3, Login Initialization File.)

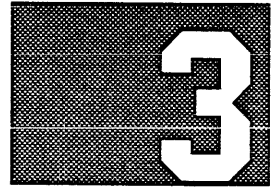
Arguments:

- variable* — The name of the variable that is recognized by the Profile utility.
- value* — The initial value of the Profile variable. This value can be a form to be evaluated, a previously defined and bound variable, a symbol, a keyword, or a string.

Example:

```
(profile-setq zwei:*region-marking-mode* `(:reverse-video))
```


LOGIN INITIALIZATION FILE



Introduction

3.1 The login initialization file, commonly referred to as the *login-init file*, allows you to customize your environment when you log in. The login-init file is loaded when you log in, evaluating any Lisp forms that are in the file. Although you can put any Lisp form in your login-init file, you should mainly use one of the functions discussed in paragraph 3.2, Customizations That Can Be Undone. These functions will undo the customizations when you log out.

The Other Customizations paragraph (3.3) discusses other useful operations typically performed in a login-init file. The topics covered are using the Profile utility, using the Zmacs customization utilities, creating logical pathnames, and prompting yourself on whether to load certain files when you log in.

First, you need to create a login-init file in your login directory on the local machine. (The *login directory* is the directory with the same name as your login name.) If you have used the New User or Profile utility, you have already created this login-init file. You can edit this file as you would any other file.

The name of this file must be LOGIN-INIT. For example, if your login name is JONES, your login-init file is LM: JONES; LOGIN-INIT.LISP. You can also use a compiled version of the file.

To load your login-init file when you log in, you use the **login** function:

login *user-name* &optional *host inhibit-init-file-p* Function

The **login** function provides access to the file system and to the Mail utility.

Arguments: *user-name* — Provides a logical address for electronic mail or messages and specifies your default directory. The *user-name* argument can be either a symbol, such as `name` or a string such as `"name"`. If you use a symbol, the letters are changed to all lowercase or all uppercase letters, depending on certain system variables. If you use a string, the case of the letters remains the same.

host — Specifies which host's file system to use. The specified *host* becomes your default file server; all references to files that do not specify another host use files in that system. The default value is the local machine (lm). In particular, this is where your login-init file comes from. If you specify *host* as `t`, then **login** assumes that *host* equals `lm` and *inhibit-init-file-p* equals `true`.

inhibit-init-file-p — Specifies whether to use a login-init file. If *inhibit-init-file-p* is `nil`, the system attempts to find a file called LOGIN-INIT.LISP or PROFILE-LISPM.INIT saved under a directory of the same name as *user-name* on *host*. If the system cannot find a login-init file, it simply returns the Lisp Listener prompt. The default value of *inhibit-init-file-p* is `nil`, which means to use a login-init file if one exists.

Examples: If your login name is JONES, the following list shows the various options you have when logging in:

```
(login 'jones)
  Attempts to load host:JONES;LOGIN-INIT.LISP, where host is the
  value of sys:associated-machine.
```

```
(login 'jones 'lm) OR
(login 'jones t)
  Either of these forms attempts to load LM:JONES;LOGIN-INIT.
```

```
(login 'jones 'bart)
  Attempts to load BART:JONES;LOGIN-INIT.
```

```
(login 'jones 'lm t)
  Does not load any login-init file because the value of the inhibit-init-file-p
  argument is t.
```

Customizations That Can Be Undone

3.2 The functions described in this paragraph allow login-init file customizations that are undone at logout. All of these functions push forms onto the **logout-list** variable. When you log out, these forms are evaluated to perform the undo. Note that you can also push your own forms onto this **logout-list** variable to have additional operations performed when you log out.

login-forms &body *forms*

Macro

Executes *forms*, arranging to undo them at logout. The forms that are supported are **setq**, **psetq**, **setf**, **psetf**, **defun**, **macro**, **deff**, **defsubst**, **advise**, or any form that has an **:undo-function** property with its value being a function that will undo that form. These undo functions are pushed onto the **logout-list** variable so that the effects of these forms are undone at logout. If a form does not have an **:undo-function** property, a warning is issued indicating that the effects of this form are permanent.

The following **login-forms** example shows how you can set variables; define functions, macros, and **deff**s; turn on **advise**; and so on for your own session. All of these effects are returned to their original form when you log out.

```
(login-forms
  (setq foo 'bar)           ;foo has value 'bar only while logged in
  (defun my-fun () t)      ;my-fun available only while logged in
  (macro my-mac (form)    ;my-mac macro available only while logged in
    `(cons ,(cadr form)))
  (advise my-fun)         ;function is advised only while logged in
  (deff foo 'bar)         ;function foo will act like function bar
```

login-setq "e &rest *variables-and-values*

Special Form

Like **setq** except that the changes are undone at logout. Sets the value of one or more variables, which are specified as *variable value variable1 value1 ... variableN valueN*. The *variable* arguments are not evaluated, and the *value* arguments are evaluated. Forms are pushed onto the **logout-list** variable that will undo these changes at logout.

The following `login-setq` example customizes Zmacs. The example changes the display of a marked region from underlining to reverse video. It checks for unbalanced parentheses when saving a Common Lisp mode or Zetalisp mode buffer. When you log out, these changes will be undone. Note that this particular Zmacs customization is typically done by using the Profile utility.

```
(login-setq zwei:*region-marking-mode* :reverse-video
           zwei:*check-unbalanced-parentheses-when-saving* t)
```

login-eval *form*

Function

Arranges to undo the effects of *form* at logout. The value produced by *form* is assumed to be another form that will undo it. That value is pushed onto the `logout-list` variable so that the effects of *form* are undone at logout.

The following `login-eval` example loads the file `LM:FONT;MY-MOUSE` when logging in and loads the file `LM:FONT;DEFAULT-MOUSE` when logging out.

```
(login-eval
 (progn (load "lm;font;my-mouse") ;do this first
        '(load "lm;font;default-mouse"))) ;return this form for login-eval
```

login-fdefine *function-spec definition*

Function

Like `fdefine` except that the changes are undone at logout. Changes the *definition* of a *function-spec*. For details on the `fdefine` function, refer to the *Explorer Lisp Reference*.

In the following example, the function definition of `foo` is replaced with the lambda definition, which is exactly what `fdefine` would do. However, when you log out, `login-fdefine` restores the original function definition of `foo`.

```
(login-fdefine 'foo #'(lambda (x) (list (cadr x) (caddr x))))
```

In the next example, the definition of the function `push` is changed to the definition of the function `my-push` (whatever that is) until you log out.

```
(login-fdefine 'push 'my-push)
```

logout-list

Variable

List of forms to evaluate at logout, to undo the effects of the init file. Note that you can also push your own forms onto this `logout-list` variable to have additional operations performed when you log out.

Other Customizations

3.3 The following discussion describes other useful operations typically performed in a login-init file. The topics covered are as follows:

- Using the Profile utility
- Customizing Zmacs
- Creating logical pathnames
- Using the `with-timeout` macro
- Using the `sys:load-if` function

Using Profile

3.3.1 When you make customizations in the Profile utility, you can optionally save them into your login-init file by using the Store Options command. You can also set the Profile variables yourself in your login-init file if you do not want the Profile utility to do this for you. These customizations are undone when you log out. (Refer to Section 2, Profile.)

The following example shows some of the Profile options you can save in your login-init file by using the Store Options command. You can find documentation in the Profile utility on these variables and many more.

```
(PROGN
  :PROFILE-OPTIONS
  (PROFILE::PROFILE-SETQ ZWEI::*MAIL-SUMMARY-MODE* `(:FILTERED))
  (PROFILE::PROFILE-SETQ FS:USER-PERSONAL-NAME-FIRST-NAME-FIRST
    "Rosemary")
  (PROFILE::PROFILE-SETQ ZWEI::*DEFAULT-BASE* `10)
  (PROFILE::PROFILE-SETQ ZWEI::*REGION-MARKING-MODE* `(:REVERSE-VIDEO))
  (PROFILE::PROFILE-SETQ ZWEI::*INITIAL-MINOR-MODES*
    `(ZWEI::ANY-BRACKET-MODE ZWEI::ELECTRIC-FONT-LOCK-MODE))
  (PROFILE::PROFILE-SETQ W::*SUGGESTIONS-MENUS-ON?* `T))
```

Customizing Zmacs

3.3.2 Zmacs provides many customization features. The following briefly lists some of the customizations you can perform in your login-init file. Refer to the *Explorer Zmacs Editor Reference* manual for details.

- You can perform many Zmacs customizations by using the Profile utility.
- Zmacs provides utilities to allow you to create your own Zmacs commands and to bind keystrokes to commands. Typically, you define and load these commands from your login-init file:
 - `zwei:defcom` macro — Creates a Zmacs command that you can invoke with META-CTRL-X and the name of the command.
 - `zwei:set-comtab` function — Assigns a keystroke to a Zmacs command.
 - `zwei:set-comtab-return-undo` function — Same as `zwei:set-comtab` except that it returns a form to execute to undo this command. You can use `zwei:set-comtab-return-undo` with `login-eval`.

Creating Logical Pathnames

3.3.3 You can simplify your access to long, complicated, or frequently-used pathnames by creating logical pathname translations. A logical pathname allows you to define your own pathname that maps to a real pathname. Refer to the *Explorer Input/Output Reference* for details on creating logical pathnames. Also, you can permanently create these logical pathnames in your namespace. Refer to Section 32, Namespace Utilities, in this *Explorer Tools and Utilities* manual and to the *Explorer Networking Reference*.

The following example allows you to access the pathname ADOLF-HITLER-HOST:SYSTEM.UTILITIES.GAMES;FOO.BAR with the much simpler pathname A:GAMES;FOO.BAR:

```
(fs:add-logical-pathname-host "A" "ADOLF-HITLER-HOST"
  ;; A is a logical host for ADOLF-HITLER-HOST

  (('("GAMES" "SYSTEM.UTILITIES.GAMES;")
    ;; Defines mapping for A:GAMES;

    ("EDIT" "SYSTEM.APPLICATIONS.EDITOR;"))))
  ;; Defines mapping for A:EDIT;
```

NOTE: Currently, there is no way to undo a logical host when you log out. You can redefine a logical host and its translations, but you cannot remove it (in a standard way) once you create it, without cold booting.

The with-timeout Macro

3.3.4 Many people use the **with-timeout** macro to specify a time limit on a prompt to load files when they log in. If they do not respond to the prompt within the time limit, the files are loaded.

with-timeout (*duration* . *timeout-forms*) &body *body* Macro

Executes *body* with a timeout set for *duration* sixtieths of a second from time of entry. If the timeout elapses while *body* is still in progress, the *timeout-forms* are executed and their values returned. Whatever is left of *body* is not done, except for its **unwind-protects**. If *body* returns, its values are returned and the timeout is cancelled. The timeout is also cancelled if *body* throws out of the **with-timeout**.

The following **with-timeout** example prompts to load some files for you. If you do not respond within about a minute, it assumes yes.

```
(when (with-timeout ((* 60 60) t) ;60 seconds in 60ths of a second.
  ;; Prompts you with the following string and you have to answer
  ;; either yes or no.
  (y-or-n-p "Load XNS file transfer code?"))
  ;; If yes, it does the following; if no, it exits.
  (load "lm:xns;exp-defsystem")
  (make-system 'xns :noconfirm))
```

NOTE: In this case, the changes made in the environment cannot be undone without cold booting.

The sys:load-if Function 3.3.5 The `sys:load-if` function loads a file only if it has been changed since it was last loaded or if it has not been loaded previously. The `sys:load-if` function is similar to the `load` function, which unconditionally loads a file.

`sys:load-if` *pathname* &optional &key :package :verbose Function
:set-default-pathname :if-does-not-exist :print

Loads this *pathname* if it needs to be loaded (that is, if *pathname* has been changed since last loaded or it has not been loaded at all). Refer to the `load` function in the *Explorer Lisp Reference* for documentation on the remaining arguments.

The following examples illustrates the use of the `sys:load-if` function:

```
(sys:load-if "lm:dir;foo.lisp") ;Loads the file the first time
```

```
(sys:load-if "lm:dir;foo.lisp") ;Does nothing since the file is already loaded
```

Also refer to the Maintaining Large Systems section in the *Explorer Lisp Reference* for documentation on the following similar functions: `sys:compile-if`, `sys:compile-load-if`, `sys:dep-compile-if`, and `sys:dep-compile-load-if`.

BUG REPORTING

4

You can use one of the following procedures to submit a bug report:

- Enter the function `(bug)` in a Lisp Listener.
- Enter the META-X Bug command in Zmacs.
- Press CTRL-M from the debugger or window-based debugger. A backtrace of n frames is provided (the default is 5). If you supply a numeric argument such as CTRL-1 CTRL-5 with CTRL-M, a backtrace of 15 frames is provided.

Each one of these commands creates a preformatted mail message with information about the current environment. You can then describe the problem or design request by using the available Zmacs and Mail commands. You may find it useful to move around to other windows or buffers to collect information for the bug report after it has been started. Please include a detailed description of the problem and information leading up to the problem. Examples showing how to reproduce the error are valuable.

For more information on how to submit a bug report, refer to the file `SYS: HELP; HOW-TO-REPORT-BUGS.TEXT`.

If you have access to the bug mailbox via the network, press the END key to electronically mail this information to the mailbox. If you do not have access to the bug mailbox, create a hardcopy of the bug report by using the META-X Print Buffer command and send the report via postal mail to the address shown at the end of the bug template.

If you attempt to fill out a bug report when in the cold-load stream (that is, the window system is not available), the bug information is obtained from various prompts for keyboard input and is mailed to the address specified.

GLOSSARY UTILITY



Introduction

5.1 Both the Lisp language and the Explorer system employ very specific terminology to effectively communicate the many concepts innate to this type of hardware and software. To help you understand this terminology, the Explorer system provides a Glossary utility.

The Glossary utility accesses an expandable list of terms that are cross referenced so that you can trace your way through several related definitions to understand any given subject.

This section discusses the following topics:

- Entering the Glossary utility — Describes how to enter the Glossary utility.
- Glossary User mode — Tells how to use the Glossary utility.
- Glossary Expert mode — Discusses how to create and modify your own glossaries.
- Using Zmacs to create a glossary file — Describes how to create a glossary file using Zmacs.
- Defining glossary file format — Tells how to define your own glossary file format.
- Defining a glossary from the Lisp Listener — Tells how to use the **define-glossary** function to load your own glossary file (instead of the standard glossary file) when the Glossary utility is invoked. You can enter this function from the Lisp Listener or from your login initialization file. Your glossary file of terms must already exist before you can use this function.

For a printed copy of the terms available in the standard glossary file, refer to the *Explorer Glossary*.

Entering the Glossary Utility

5.2 To enter the Glossary utility, perform one of the following steps:

- Display the System menu and select the Glossary item with the mouse
- Press SYSTEM Z on the keyboard
- Enter the `gloss:glossary` function

`gloss:glossary` &optional *entry-name-string*

Function

The optional argument *entry-name-string* is the name of a glossary entry that you want to look up (not the name of a glossary). If you specify *entry-name-string*, you enter the Glossary utility in the current glossary. (The currently selected glossary is the one displayed by the utility, and the one with which you interact.) If an entry exists in the current glossary for the value of *entry-name-string*, then that definition is displayed.

Any of these steps places you in the User mode of the Glossary utility. The default mode is the User mode; however, if you leave the Glossary and then reenter it by pressing SYSTEM Z, the utility returns you to the glossary mode you left. If you left while in the Expert mode, then you reenter in the Expert mode.

The standard glossary file, complete with cross-references, is defaulted into the system. If you want to specify another glossary as the currently selected glossary, you can modify your initialization file so that another glossary file is loaded when the Glossary utility is invoked. This subject is discussed in more detail later in this section. See paragraph 5.7, Defining a Glossary From the Lisp Listener.

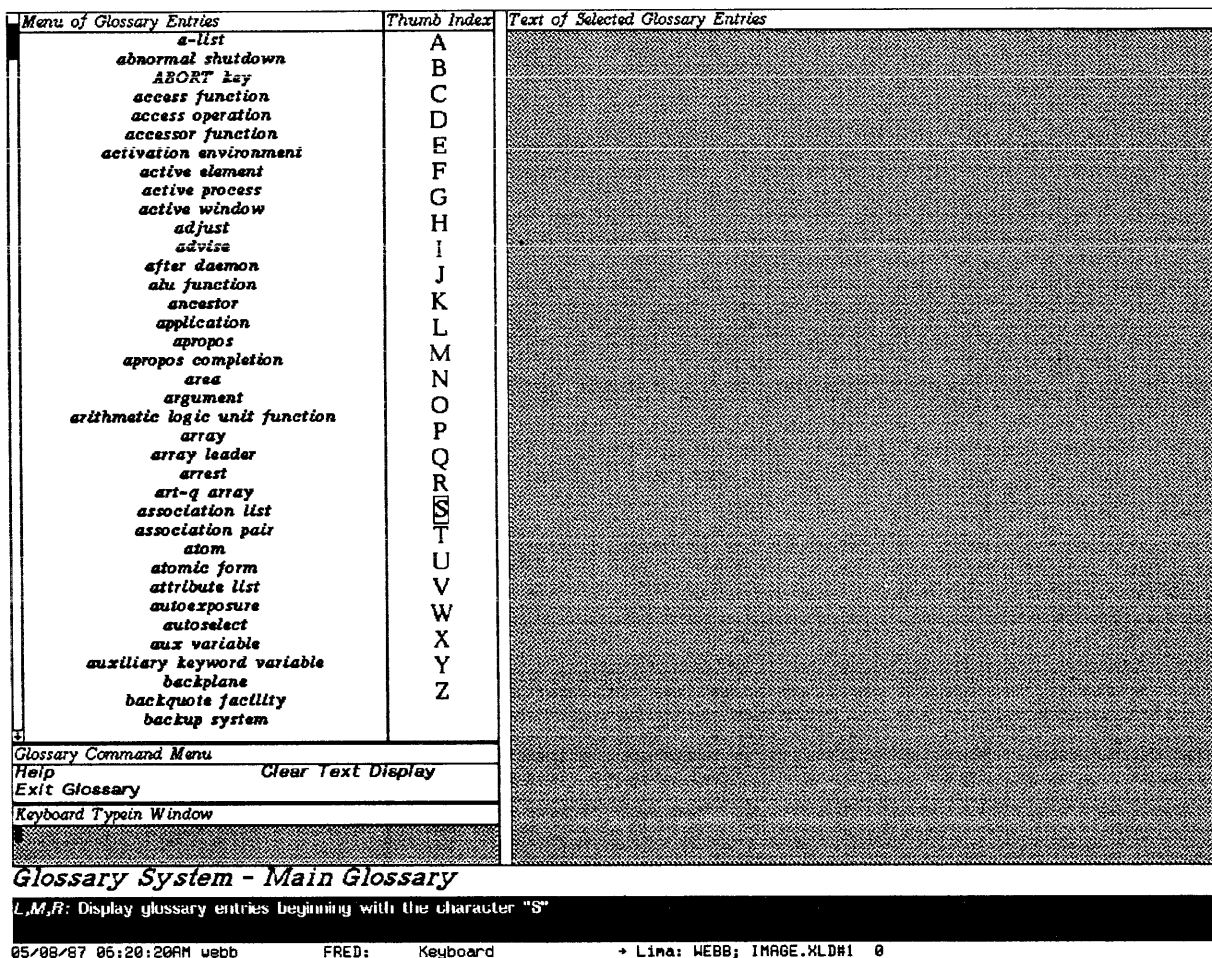
Glossary User Mode

5.3 On entering the User mode, the Explorer system formats a display to facilitate interaction between you and the Glossary utility. The formatted display you see is the glossary frame. Figure 5-1 shows the glossary frame as it appears when you first enter the utility.

In the User mode, the major components of the glossary frame are as follows:

- Glossary Command Menu
- Keyboard Typein Window
- Menu of Glossary Entries
- Text of Selected Glossary Entries
- Thumb Index

Figure 5-1 The Glossary Frame



Glossary Command Menu 5.3.1 The Glossary Command Menu contains a minimum of three mouse-selectable entries: Help, Clear Text Display, and Exit Glossary.

When you click on Help, the utility displays a help menu listing each pane of the glossary frame. When you choose an entry from this submenu, the Explorer system displays information about how to use that pane.

When you click on Clear Text Display, the utility clears the entire pane showing the Text of Selected Glossary Entries.

When you click on Exit Glossary, the utility returns you to where you were prior to entering the Glossary utility (Lisp Listener, Zmacs, Inspector, and so on).

If more than one glossary is loaded into the system, a fourth command, **Select Glossary**, appears. This command allows you to designate which of the loaded glossaries is to be the currently selected glossary. With the mouse, click on the **Select Glossary** command. When you do so, a pop-up window appears with a mouse-sensitive list of all defined glossaries. With the mouse, choose which glossary you want the utility to display. When you click the mouse, the pop-up window disappears, and whichever glossary you specified becomes the currently selected glossary. If the file containing terms and definitions must first be loaded, the Explorer system does so at this time. For more information, see paragraph 5.4.3, **Select Glossary**.

When the glossary has no predefined terms, for instance, when you are creating a glossary interactively, the **Menu of Glossary Entries** is empty. The **Text of the Selected Glossary Entries** pane retains any definitions you investigated from the previously selected glossary; otherwise, it is empty.

**Keyboard
Typein Window**

5.3.2 The **Keyboard Typein Window** allows you to locate any glossary entry by typing the entry on the keyboard. When you finish typing the entry, press **RETURN**, and if such an entry exists, its definition appears in the pane labeled **Text of Selected Glossary Entries**. In addition, this entry and all entries following it alphabetically appear in the pane labeled **Menu of Glossary Entries**. If no such entry exists, a message appears in the **Keyboard Typein Window** to inform you of this fact; however, the **Menu of Selected Glossary Entries** displays those entries closest alphabetically to what you wanted to see.

**Menu of
Glossary Entries**

5.3.3 The **Menu of Glossary Entries** displays the available terms in alphabetical order. Each displayed entry has a definition in the glossary. The menu begins with entries starting with the letter *a* and lists as many entries as fit in the pane. To see the other available entries, you can use the mouse for scrolling, as described in the *Introduction to the Explorer System*, or use the **Thumb Index**, which is described later in this section.

**Text of Selected
Glossary Entries**

5.3.4 The glossary frame contains a pane titled **Text of Selected Glossary Entries**. This pane displays the definitions of terms selected from the **Menu of Glossary Entries** or typed into the **Keyboard Typein Window**. Any displayed definition may also have other terms in it that are defined in the glossary. These terms appear in boldface and can be selected by clicking on them with the mouse.

Each new definition is added to the window above the currently displayed definition. The oldest terms begin to scroll off the screen after the pane is filled and new definitions are added; however, you can see the text for any previously selected entry by using the cursor keys to scroll forward and backward. (The mouse can be used for scrolling also; see the *Introduction to the Explorer System*.) To completely clear the pane of all text, select the **Clear Text Display** command in the **Glossary Command Menu**.

Thumb Index

5.3.5 The **Thumb Index** displays the characters of the alphabet. When you select a character with the mouse, the utility displays entries beginning with this character in the **Menu of Glossary Entries**. This allows you to move around quickly in the **Menu of Glossary Entries**.

Glossary Expert Mode

5.4 While in the Glossary User mode, you have a handy reference for Explorer system and Lisp terms. However, the Glossary utility can do much more in the Expert mode. To enter the Expert mode, press META-CTRL-T. When you do so, the Glossary Command Menu expands, displaying these additional commands:

- Define Glossary
- Delete Glossary
- Select Glossary
- Add Glossary Entry
- Delete Glossary Entry
- Undelete Glossary Entry
- Edit Glossary Entry
- Write Current Glossary
- Merge in Glossary
- (Re)Generate XRefs
- Turn on XRef Deletion
- Exit Expert Mode

Define Glossary

5.4.1 The Define Glossary command makes a glossary known to the system. You can have more than one glossary on your system. For example, you may have glossaries for individual applications. To create a new glossary, you must first define it. When you select the Define Glossary command with the mouse, the Keyboard Typein Window prompts you as follows:

Name of new glossary:

Glossary names are strings. They need not represent a specific file or directory. After typing a string into the Keyboard Typein Window, press RETURN, and the following prompt appears in the Keyboard Typein Window:

Filename of glossary entries:

The filename can reside under any directory on the system; however, you might want to keep a glossary-related directory and file descriptive pathname such as the following:

```
LM: GLOSSARY; DEFINITIONS.TEXT
```

In this example, the directory component is GLOSSARY, the specific file-name is DEFINITIONS, and the file type is TEXT. You can create other file formats as described in paragraph 5.6, Defining Glossary File Format.

If you already have a file that contains your new glossary entries, enter this filename and press RETURN.

At this point, the Glossary utility creates the necessary overhead information for you to select the newly defined glossary. See paragraph 5.4.3, Select Glossary.

NOTE: If you are creating a new glossary and plan to add terms and definitions interactively with the Glossary utility, you must select the newly defined glossary before going any further.

You can load user-defined glossaries either before a **disk-save** operation or at run time when the glossary is first referenced.

An alternate means to define a glossary exists. Because this method is intended for use with initialization files, see paragraph 5.7, Defining a Glossary From the Lisp Listener.

Delete Glossary 5.4.2 To delete a glossary, select the Delete Glossary command from the Glossary Command Menu. When you do so, a pop-up window appears with a mouse-sensitive list of all defined glossaries. With the mouse, choose which glossary you want to delete. When you click the mouse, the pop-up window disappears, and the utility then removes the glossary from the list of glossaries that the Glossary utility knows to exist. It does not actually delete the file containing the glossary terms.

If you choose to delete the currently selected glossary, you are prompted to specify which defined glossary is to replace it. This glossary then becomes the currently selected glossary upon deletion of the original.

Select Glossary 5.4.3 To access a glossary other than the main glossary, use the mouse to click on Select Glossary in the Glossary Command Menu. When you do so, a pop-up window appears with a mouse-sensitive list of all defined glossaries. With the mouse, choose which glossary you want the utility to display. When you click the mouse, the pop-up window disappears, and the utility loads the glossary file containing terms and definitions into the glossary buffer.

Notice the run bars near the bottom center of your screen in the status line. Depending on the length of your input file, loading can take from a few seconds to a few minutes.

When the glossary has no predefined terms (for instance, when you are creating a glossary interactively), the Menu of Glossary Entries is empty. The Text of Selected Glossary Entries pane retains any definitions you investigated from the previously selected glossary; otherwise, it is empty.

**Add or Delete
Glossary Entry**

5.4.4 You can add entries to the currently selected glossary by selecting Add Glossary Entry from the Glossary Command Menu. The following prompt then appears in the Keyboard Typein Window:

Entry Name:

As you enter the term, it appears in the Keyboard Typein Window. Press RETURN to complete entering the term. In response, the Keyboard Typein Window displays the following prompt:

Now type in text for "x":

In this prompt, "x" is the term you are ready to define. As you enter the term's definition, it appears in the Text of Selected Glossary Entries pane. You signal the end of the definition by pressing END. You can now access the new glossary entry with any of the methods described in the User mode portion of this section.

When you select the Delete Glossary Entry command from the Glossary Command Menu, the utility displays the following prompt in the Keyboard Typein Window:

Click on the name of the entry to be deleted.

Click on any term in the Menu of Glossary Entries or in the Text of Selected Glossary Entries. The utility deletes the indicated entry.

Should you delete an entry by accident, you can simply undelete it by selecting the Undelete Glossary Entry command from the Glossary Command Menu.

NOTE: Remember that you must resave the currently selected glossary before exiting your session on the Explorer system. Otherwise, the entries you add are not permanently added to the currently selected glossary, and the entries you delete are not actually deleted from the glossary file. For more information on saving the glossary, see paragraph 5.4.6, Write Current Glossary.

Edit Glossary Entry

5.4.5 When you select this command from the Glossary Command Menu, the utility displays the following prompt in the Keyboard Typein Window:

Click on the name of the entry to edit.

Click on any term in the Menu of Glossary Entries or in the Text of Selected Glossary Entries. The utility places the indicated entry at the top of the Text of Selected Glossary Entries pane. At this point you can edit the entry just as you would in Zmacs. When satisfied with your changes, press END.

Remember that you must resave the currently selected glossary before exiting your session on the Explorer system. Otherwise, the entries you modified are

not permanently added to the glossary file. For more information on saving the glossary, see paragraph 5.4.6, Write Current Glossary.

**Write
Current Glossary**

5.4.6 To preserve newly created glossaries or entries to existing glossaries, you must write the glossary in question before exiting the Explorer session. To do so, be sure that the glossary you are going to write is the selected glossary; then click on the Write Current Glossary command in the Glossary Command Menu. At this point, the Keyboard Typein Window displays the following prompt:

Name of file to write: (default is xxxxxxxx)

In this case, xxxxxxxx is the default value (the name of the file provided when the glossary was created). You can either accept the default by pressing RETURN, or enter another filename. The filename can reside under any directory on the system; however, you might want to keep a glossary-related directory and file descriptive pathname such as the following:

LM: GLOSSARY; DEFINITIONS.TEXT

In this example, the directory component is GLOSSARY, the specific filename is DEFINITIONS, and the file type is TEXT.

When you finish entering the filename, press RETURN. A pop-up menu then appears, requesting that you specify the file's format. The file can have one of several formats, including text, binary, or a format that you define yourself. For more information on file formats, see paragraph 5.6, Defining Glossary File Format.

When you select a format with the mouse, the utility saves the contents of the currently selected glossary to the specified file. You can then end the Explorer session without losing any modifications that you made to this glossary.

Merge in Glossary

5.4.7 You can merge the definitions and terms of any existing glossary with those of the currently selected glossary. To do so, click on the Merge in Glossary command from the Glossary Command Menu. A pop-up window prompts you to click on the name of the glossary you want to merge with the current glossary. You can abort the merge operation by moving the mouse away from the pop-up window or by pressing the ABORT key.

Merging two glossaries does not automatically combine the cross-references for both. You must regenerate all cross-references via the (Re)Generate XRefs command from the Glossary Command Menu.

When you merge two glossaries, both are placed in the destination glossary. The other original disappears from the glossary menus. (That is, the Glossary utility no longer knows about it. The actual file containing the glossary terms is not touched.)

(Re)Generate XRefs 5.4.8 After merging a glossary with the current glossary or adding or deleting glossary entries, you must then regenerate all cross-references for the currently selected glossary. To do so, use the mouse to select the (Re)Generate XRefs command from the Glossary Command Menu. When you click on the command, the Glossary utility regenerates all the cross-references. Definitions that you view after performing this command display all other entries for the newly merged glossary in boldface.

Remember that you must resave the currently selected glossary before exiting your session on the Explorer system. Otherwise, those cross-references you generated disappear when you end this session. For more information on saving the glossary, see paragraph 5.4.6, Write Current Glossary.

Turn On XRef Deletion 5.4.9 If, for any reason, you want to delete cross-references for a glossary entry, you can do so with the Turn On XRef Deletion command from the Glossary Command Menu. Notice that when you select this command with the mouse, the displayed command changes to Turn Off XRef Deletion.

Once you have turned on XRef Deletion, select any displayed cross-reference with the mouse, click right, and all cross-references for this term are deleted. The entry is no longer displayed in boldface for any definitions you will view in the future.

NOTE: When you finish deleting cross-references, be sure you return to the Glossary Command Menu and select the Turn Off XRef Deletion command. This action prevents accidental deletion of cross-references by clicking right while XRef Deletion is in effect. (Clicking left *selects* a cross-reference, but you may accidentally click right.)

Remember that you must resave the currently selected glossary before exiting your session on the Explorer system. Otherwise, the cross-references you deleted are not deleted from the glossary file. For more information on saving the glossary, see paragraph 5.4.6, Write Current Glossary.

Exit Expert Mode 5.4.10 By selecting this command from the Glossary Command Menu, you can return to the Glossary User mode.

Using Zmacs to Create a Glossary File

5.5 As an alternative to creating a glossary file interactively, you can create a glossary file in text-file format by using Zmacs. One drawback to creating the glossary file via Zmacs is the likelihood of introducing errors; a situation not likely to occur if you create your glossary file interactively.

To create the glossary file via Zmacs, first find the value of the **gloss:*entry-name-left-margin*** variable (the default is 12). This is the value you must use for the beginning column to list all terms to be defined in your glossary file.

Next, find the value of the **gloss:*entry-text-left-margin*** variable (the default is 16). This is the value you must use for the beginning column of all the definitions in your glossary file. Also, note that each definition must be on a separate line from the term itself.

Here are some tips when using Zmacs. Refer to the *Explorer Zmacs Editor Reference* for details.

- Use the Zmacs CTRL-= command to find out what column you are in.
- Do not use tab characters when spacing out to the column. Make sure they are spaces. To remove tabs, mark your text as a region and enter the Zmacs META-X Untabify command.
- Make sure there are no spaces or tabs after the name of the term.

Once you complete the file and save it, you can then define your glossary according to the instructions described for Define Glossary. When the `Filename of glossary entries` prompt appears, enter your glossary filename.

Defining Glossary File Format

5.6 You can use the following function to define your own glossary file format if you want a format not already provided. You may want your glossary file to be readable by some other system or application.

define-glossary-file-format *name filetype entry-write-method glossary-read-method &optional priority documentation-string* Function

Arguments: **name** — This argument simply indicates what you want to call the file format you are defining. It is normally a keyword with some mnemonic quality relating to the newly created format. Some of the predefined glossary formats include:

:text — The **:text** keyword describes the text file format used. In order for the utility to read glossaries properly, the glossary entry names must begin in column **gloss:*entry-name-left-margin***, and all the definitions for the glossary entries must be indented to the right of the value specified for **gloss:*entry-text-left-margin***.

:binary — This keyword describes a binary format.

filetype — This argument is the type of file to use for this format, such as **:text** or **:xld**.

entry-write-method — This keyword is the message name that is sent to a glossary-entry object to write itself to a file (for example, **:write-cgloss-self**). The method takes one argument, specifying the stream to write to.

glossary-read-method — This keyword is the message name that is sent to a glossary object to load the glossary from a file. The method accepts one optional argument: the namestring or pathname object to load.

priority — The *priority* argument is an integer that specifies the priority of loading this type of glossary format relative to the other formats. The priority is based on how fast the file can be loaded and how much information (such as cross-references or no cross-references) the format contains. A high value for *priority* means that the format with this priority takes precedence over a format with a lower priority value. The following are the current values of each format's *priority*:

■ **:binary** — 8

■ **:text** — 2

For example, suppose your glossary is saved in two formats: MY-GLOSS.TEXT and MY-GLOSS.XLD. If you load MY-GLOSS without specifying the file type, *priority* tells the system which file type you meant (as a default). Otherwise, you can specify the file type in the pathname to load the file with the format you want.

documentation-string — The documentation string is a string of text in which you can briefly describe the newly defined glossary, that is, its purpose, intended audience, and so on.

Defining a Glossary From the Lisp Listener

5.7 You can use the `define-glossary` function to define a glossary from the Lisp Listener or from your login initialization file. Your glossary file of terms must already exist before you can use this function. This function gives you the option of loading your own glossary file instead of the standard Explorer glossary file when you invoke the Glossary utility.

`define-glossary` *glossary-name file-to-load* Function
 &key `:load-immediately-p` `:make-current-glossary-p`

Arguments: *glossary-name* — This argument is a string, just as with the other methods of defining glossaries. Again, the string need not represent a specific file or directory, merely something easily recognizable by a user.

file-to-load — The *file-to-load* argument must be the namestring or pathname object of a file that contains the terms and definitions for the glossary you are defining.

The optional keywords of this function are as follows:

`:load-immediately-p` — This keyword determines whether the file containing the terms and definitions for the newly defined glossary should be loaded immediately (non-`nil`) or loaded when the glossary is first referenced by the Glossary utility (`nil`). The default is `nil`.

`:make-current-glossary-p` — This keyword determines whether the newly defined glossary is to be the currently selected glossary (non-`nil` if so, `nil` if not). The default is `nil`.

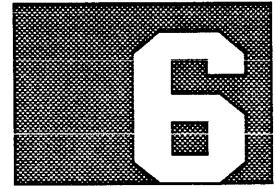
The following list describes what happens when you use these optional keywords:

- If you accept the default of `nil` for `:load-immediately-p` and `:make-current-glossary-p`, your glossary is listed by the Select Glossary command. The standard Explorer glossary file is loaded when you invoke the Glossary utility, and it is selected.
- If you specify non-`nil` for `:load-immediately-p` and `:make-current-glossary-p`, your glossary is the currently selected glossary when you invoke the Glossary utility. The standard Explorer glossary file is not loaded unless you select it with the Select Glossary command.
- If you specify `:load-immediately-p` non-`nil` and `:make-current-glossary-p` `nil`, your glossary is loaded immediately, and it is listed by the Select Glossary command. The standard Explorer glossary file is the currently selected glossary.
- You cannot specify `:load-immediately-p` `nil` and `:make-current-glossary-p` non-`nil` because the file must be loaded in order to be the currently selected glossary.

Example: Assume the following example code is in your login initialization file. When you log in, the glossary named `Druid` is defined to the system and is immediately loaded. When you invoke the Glossary utility, it is the currently selected glossary. The standard Explorer glossary file is not loaded unless you select it with the Select Glossary command.

```
(define-glossary "Druid" "lm: glossary; trees.text"
  :load-immediately-p t :make-current-glossary-p t)
```

UCL USER INTERFACE



Overview

6.1 The Universal Command Loop (UCL) simplifies and standardizes the process of constructing command interfaces to interactive programs. It provides the flavors and functions that construct a command interpreter, online help features, and environment customization features.

This section describes how to use the features that UCL provides for any application. The next section, Section 7, describes how to incorporate UCL into your application.

This section discusses the following topics:

Basic command interpreter operation — Provides an overview of how the UCL command interpreter works.

Help features — Discusses the interpreter's help mechanisms that are invoked from one universal command bound to the HELP key. These help commands (and the environment customization commands) are known collectively as the *universal* commands because they are available in any UCL application.

By pressing the HELP key, you can easily access an application's custom online documentation, browse through documentation for commands, and perform searches on keystrokes and command names. Because the UCL can process various kinds of typed expressions, the HELP key also provides documentation on the syntax for each kind of expression when you select the Command Type-In Help option.

Environment customization features — Allows you to modify and enhance the standard features provided by the application. For example, if a command is assigned to a keystroke that you feel is difficult to type or if a command is used so frequently that you want to simplify the keystroke, then you can reassign the keystroke. If a command does not have a keystroke assigned to it, you can assign one. You can build command macros to execute sequences of commands in order to use the application more efficiently. You can save this customized working environment to a file and reload it for subsequent sessions.

Miscellaneous features — Includes implicit message sending to a flavor instance, acceptance of typed commands and arguments, recognition of numeric arguments to keystroke commands, and user-friendly error catching that avoids invoking the debugger for insignificant errors.

**Basic Command
Interpreter
Operation**

6.2 The UCL command interpreter is helpful to users having different levels of expertise. UCL commands can be executed by the following kinds of user input:

- Keystroke sequences — Keystroke sequence input allows the expert user to work efficiently because using keystroke sequences is faster than using the other methods of input.
- Mouse button clicks — Input from mouse clicks is especially useful for applications that manipulate graphics objects with the mouse. For instance, a mouse click can execute a command that performs an operation on the graphics object the mouse cursor is near.
- Command menus — Command menus are designed to meet the needs of the novice. Although typing a keystroke command is slightly faster than using a menu, the command menus allow the novice to be productive while learning new commands. The menus also help an experienced user locate infrequently used commands.
- Typed command names — Typed command name input is useful in applications that contain large command sets. This capability allows you to execute a command by typing the command name instead of attempting to remember the keystroke assigned to a command.

Help Features

6.3 The following paragraphs describe each of the help mechanisms provided by the UCL.

- Help command — Displays a pop-up menu that contains help options for the Explorer system in general and for the current program. You invoke the Help command by pressing the HELP key. If your application is set up to handle the Help option and the Tutorial option, they are also included in the menu.
- Command type-in help — Provides help on what kind of typed expressions are processed by the application.
- Command display — Displays the currently active UCL commands and allows you to execute them.
- Command history — Shows previously executed significant UCL commands in a command display window and allows you to execute them.
- Command name search — Searches for UCL commands whose names contain a string that you specify and allows you to execute them.
- Keystroke search — Displays UCL commands assigned to a keystroke and allows you to execute them.
- Command menus — Discusses the help that UCL command menus provide: mouse documentation window help and icon representation.
- Completion commands — Discusses the commands that help you complete typed input.
- Mouse documentation window help — Talks about the help that the mouse documentation window provides on symbols, command names, function names, and some of the completion commands.

Most of these features are automatically included by the UCL in an application; they are among the collection of *universal* commands that you can execute on demand. Two help mechanisms, the UCL command menus and mouse documentation window help, provide passive help to you and are optionally included in the application.

Help Command 6.3.1 The universal command Help, assigned to the HELP key, displays a pop-up menu of help options (see Figure 6-1). This menu is divided into two parts:

- General help options that describe basic features of the Explorer system.
- Local help options that describe features of the current UCL application and environment. Some of these local help options are universal and work the same in all applications.

If your application is set up to handle the Application Help option and the Tutorial option, they are included in the menu. These options do whatever your program specifies. The following describes typical uses of these options:

- Application Help option — Prints a message describing the application. The name of this help option is specific to the current UCL application (for example, Lisp Listener Help). Figure 6-2 shows the Inspector Help option screen.
- Tutorial option — Executes a tutorial that explains the current UCL application oriented toward the novice.

The following describes the general help options:

- Explorer Overview — Displays a simple overview of the Explorer system and its capabilities.
- System Menu — Displays the System menu. (Press the right mouse button also displays this menu.)
- System Application — Displays a list of all the system applications installed on the Explorer system. (Pressing SYSTEM HELP also displays this list.)
- TERM Key Help — Displays all the TERM key features. (Pressing TERM HELP also displays these features.)

Most of the other help options described in the following paragraphs appear in this pop-up Help menu. The menu also contains the title of a submenu that provides the environment customization features described in paragraph 6.4,

The help options that appear in the Help menu also have associated keystrokes. When you position the mouse cursor over an option in the menu, the mouse documentation line displays its keystroke. The keystrokes are also listed in the Command Display window (described later). Also refer to paragraph 6.6, Command Summary.

Figure 6-1

UCL Menu of Help Options

```

General Help
Explorer Overview
System Menu
System Applications
Term Key Help

Local Help
Lisp Listener Help
Command Name Search
Command Display
Command Type-in Help
Command History
Keystroke Search
Customization Menu

```

Figure 6-2 Inspector Help

```

INSPECTOR HELP
-----
*** OPTIONAL THIRD INSPECTION PANE ***
Displays previously inspected item.
-----
*** OPTIONAL SECOND INSPECTION PANE ***
Displays previously inspected item.
-----
*** MAIN INSPECTION PANE ***
This pane displays the structure of the most recently inspected item.
Specify objects to inspect by either:
    * Entering them into the Inspection Pane
    * Clicking Mouse-Left on the mouse-sensitive elements of previously inspected items

Clicking Mouse-Right on items in this pane tries to inspect the item's function definition.
When locked, the inspected item in that pane will be frozen until unlocked. Only two of the
3 panes may be locked.
-----
COMMAND MENU
Click Mouse-Left to select a command.
*** HISTORY PANE ***
This pane displays a list of the objects that have been inspected.
To bring an object back into the Main Inspection Pane, click Mouse-Left on that object in this pane.

To remove an item from the History Pane, position the mouse-cursor to the left of the item until the cursor becomes a right-pointing arrow (this is the item's "line area"). Now click Mouse-Middle.
-----
*** INTERACTION PANE ***
Enter items to inspect in this pane.

This pane can also be used for command name typein and for Lisp typein. For Lisp typein, use the Mode command.

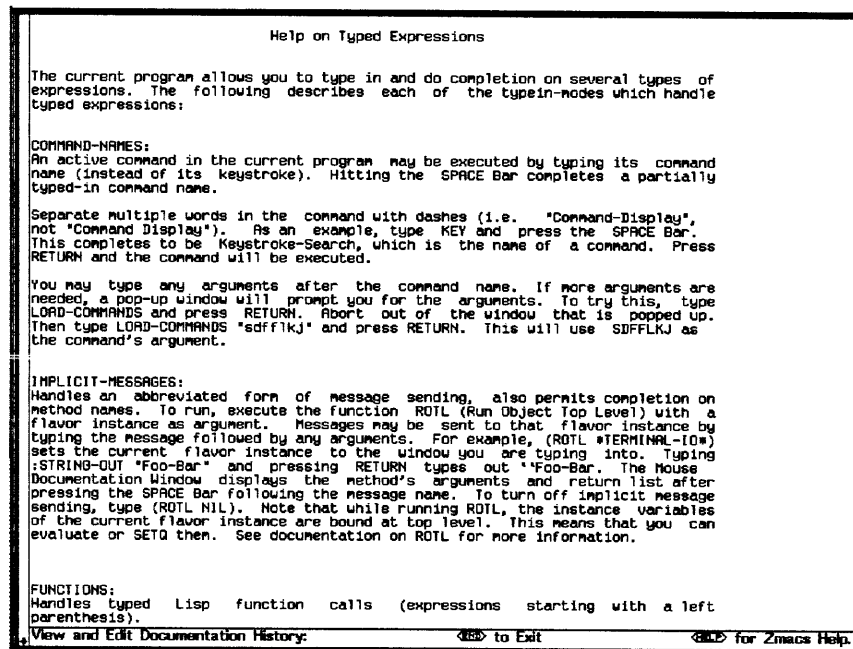
The last three inspected objects are stored in the top three Inspection Panes.
-----
Press the space bar to remove this message.
-----
Documentation for Inspector
R: Bring up the System Menu

```

Command Type-In Help

6.3.2 Command Type-In Help is a help menu option that tells you what kind of typed expressions are processed by the application. For example, in a UCL application set up to interpret typed command names and Prolog expressions, this option displays documentation on the format of command names and Prolog expressions. In an application that accepts the default expression processing, this option displays documentation on typed command names and Lisp forms. (Paragraph 6.5.1 discusses the processing of typed expressions.) Figure 6-3 shows an example of Command Type-In Help.

Figure 6-3 Command Type-In Help



Command Display 6.3.3 The universal command Command Display shows in a scroll window the currently active UCL commands. Each line of the scroll window contains the following information for a command:

- Name
- Keystroke
- Short description

Figure 6-4 shows a sample command display. A command can be executed from the display by boxing it with the mouse cursor and clicking left. Full documentation for the command is displayed by clicking middle.

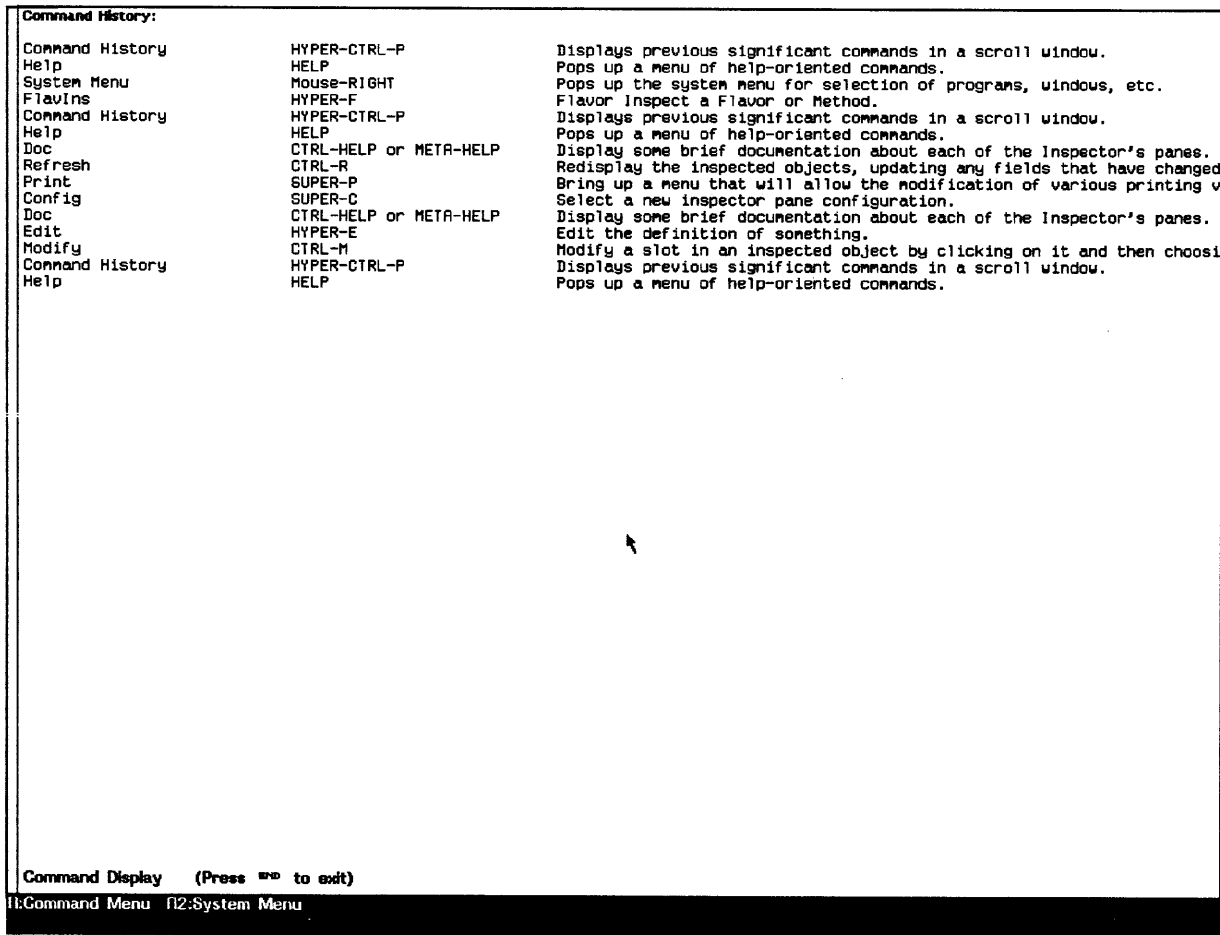
Figure 6-4 Command Display

UNIVERSAL COMMANDS:		
Command Name	Assigned Keystroke	Description
Build Command Macro	HYPER-CTRL-C	Creates a command which queues other commands for execution.
Build Keystroke Macro	HYPER-CTRL-M	Creates a command which forces a sequence of keys into the IO buffer.
Command Display	HYPER-CTRL-HELP	Displays the currently active commands in a scroll window.
Command Editor	HYPER-CTRL-STATUS	Allows editing of this program's commands in a scroll window.
Command History	HYPER-CTRL-P	Displays previous significant commands in a scroll window.
Command Name Search	HYPER-CTRL-N	Lists commands whose names contain a given substring.
Configure Type-In Modes	HYPER-CTRL-(Allows modification of the modes used in interpreting and completing type-in.
Help	HELP	Pops up a menu of help-oriented commands.
Keystroke Search	HYPER-CTRL-K	Lists commands assigned to a given keystroke or keystroke sequence.
Load Commands	HYPER-CTRL-L	Load UCL commands saved earlier.
Numeric Argument	HYPER-SUPER-META-CTRL-9 or	Passes a numeric argument to next command(s).
Redo Command	HYPER-CTRL-R	Repeats previous command using same arguments.
Save Commands	HYPER-CTRL-S	Save all of the user's tailored commands out to disk.
System Menu	Mouse-RIGHT	Pops up the system menu for selection of programs, windows, etc.
Top Level Configurer	HYPER-CTRL-T	Modify the attributes of Command & Lisp type-in.
INPUT EDITOR COMMANDS:		
Command Name	Assigned Keystroke	Description
Apropos Complete	SUPER-ESCAPE	Completes input as a substring, similar to the APROPOS function.
Backward Character	CTRL-B or ←	Moves the cursor backward one character.
Backward Parentheses	META-CTRL-B or META-CTRL--	Moves the cursor backward one set of parentheses.
Backward Word	META-B or META-←	Moves the cursor backward one word.
Basic Help	HELP	Suggests various ways of getting help.
Beginning Of Buffer	META-← or HYPER-↑	Moves the cursor to the beginning of the current input.
Beginning Of Line	CTRL-A or SUPER-→	Moves the cursor to the beginning of the current line.
Clear Input	CLEAR-INPUT	Clears the current input.
Complete	ESCAPE	Completes using Recognition style of completion.
Delete Character	CTRL-D	Deletes the character under the cursor.
Delete Parentheses	META-CTRL-K	Deletes the Lisp form to the right of the cursor.
Delete Word	META-D	Deletes the word to the right of the cursor.
Display Internal State	META-CTRL-HELP	Displays the internal state of the Input Editor.
End Of Buffer	META-→ or HYPER-↓	Moves cursor to the end of current input.
End Of Line	CTRL-E or SUPER-→	Moves cursor to the end of current line.
Exchange Words	META-T	Exchanges words on either side of cursor.
Forward Character	CTRL-F or →	Moves the cursor forward one character.
Forward Parentheses	META-CTRL-F or META-CTRL--→	Moves the cursor forward one set of parentheses.
Forward Word	META-F or META-→	Moves the cursor forward one word.
Kill Line	CTRL-K	Kills input right of cursor on the current line.
Kill Region	CTRL-W	Kills a region of input marked by user.
List Apropos Completions	SUPER-↖	Lists possible Apropos-style completions on a symbol left of cursor.
List Commands	CTRL-HELP	Displays information on Input Editor commands in a scroll window.
List Completions	CTRL-↖	Lists possible ESCAPE (Recognition) style completions on a symbol left of cursor.
List Input Ring	META-STATUS	Displays the input ring.
List Kill Ring	META-CTRL-STATUS	Displays the Emacs kill ring.
List Spelling Completions	HYPER-↖	Lists possible Spelling corrections on a symbol left of cursor.
Mark Beginning	CTRL-←	Marks kill region as input from cursor to beginning of buffer.
Mark End	CTRL-→	Marks kill region as input from cursor to end of buffer.
Menu Pop Up Input Ring	STATUS	Displays the input ring in a pop-up menu for input selection.
Menu Pop Up Kill Ring	CTRL-STATUS	Displays the Emacs kill ring in a pop-up menu for input selection.
Command Display	(Press ESC to exit)	

R:Command Menu R2:System Menu

Command History 6.3.4 The universal command Command History shows previously executed significant UCL commands in a scroll window (Figure 6-5). Only significant commands are shown; those commands marked as insignificant by the programmer are not displayed. The most recent command appears at the top of the display, with the commands displayed in reverse sequence of execution. You can view command documentation and execute a command by selecting it with the mouse.

Figure 6-5 Command History



Command Name Search

6.3.5 The universal command Command Name Search is derived from the Zmacs Apropos command. It reads a search string and displays all of the commands whose names contain the search string (see Figure 6-6). The search string can contain special search operators (described in the online documentation when you press CTRL-H HELP) to constrain the search, such as the following:

- Not next character
- Match any whitespace character
- Match any character

The UCL commands that meet the search criteria are then listed and displayed in a scroll window. You can view the command documentation and execute a command by selecting it with the mouse.

Figure 6-6 Command Name Search

```

Results of command name search:

Build Command Macro      HYPER-CTRL-C           Creates a command which queues other commands for execution.
Command Display          HYPER-CTRL-HELP       Displays the currently active commands in a scroll window.
Command Editor           HYPER-CTRL-STATUS     Allows editing of this program's commands in a scroll window.
Command History          HYPER-CTRL-P           Displays previous significant commands in a scroll window.
Command Name Search      HYPER-CTRL-N           Lists commands whose names contain a given substring.
Load Commands            HYPER-CTRL-L           Load UCL commands saved earlier.
Redo Command             HYPER-CTRL-R           Repeats previous command using same arguments.
Save Commands            HYPER-CTRL-S           Save all of the user's tailored commands out to disk.
List Commands            CTRL-HELP              Displays information on Input Editor commands in a scroll window.

Command Display (Press ESC to exit)
R: Bring up the System Menu.

```

Keystroke Search 6.3.6 The universal command Keystroke Search displays UCL commands assigned to a keystroke or keystroke sequence that you input. The commands are displayed in a command display (scroll) window. You can view the command documentation and execute a command by selecting it with the mouse.

Command Menus 6.3.7 UCL command menus help the novice user learn application commands and provide an efficient means of inputting the commands. Two features that the command menus provide are documentation in the mouse documentation window and icon representation, as described in the following paragraphs.

Note that these features of UCL command menus depend on how the UCL application is written, using the tools described in Section 7, UCL Programmer Interface.

*Mouse
Documentation
Window*

6.3.7.1 UCL command menus display commands using their assigned names (by default), but, unlike many application command menus, UCL also displays documentation for the command. The command name usually does not provide enough information for a novice user to determine what a command does. Thus, the user is forced either to blindly execute the command or search for the command documentation. To solve this problem, UCL command menus provide informative documentation in the mouse documentation window on each command in the menu. When the mouse cursor boxes a command in the menu, the mouse documentation window displays a short description of the command.

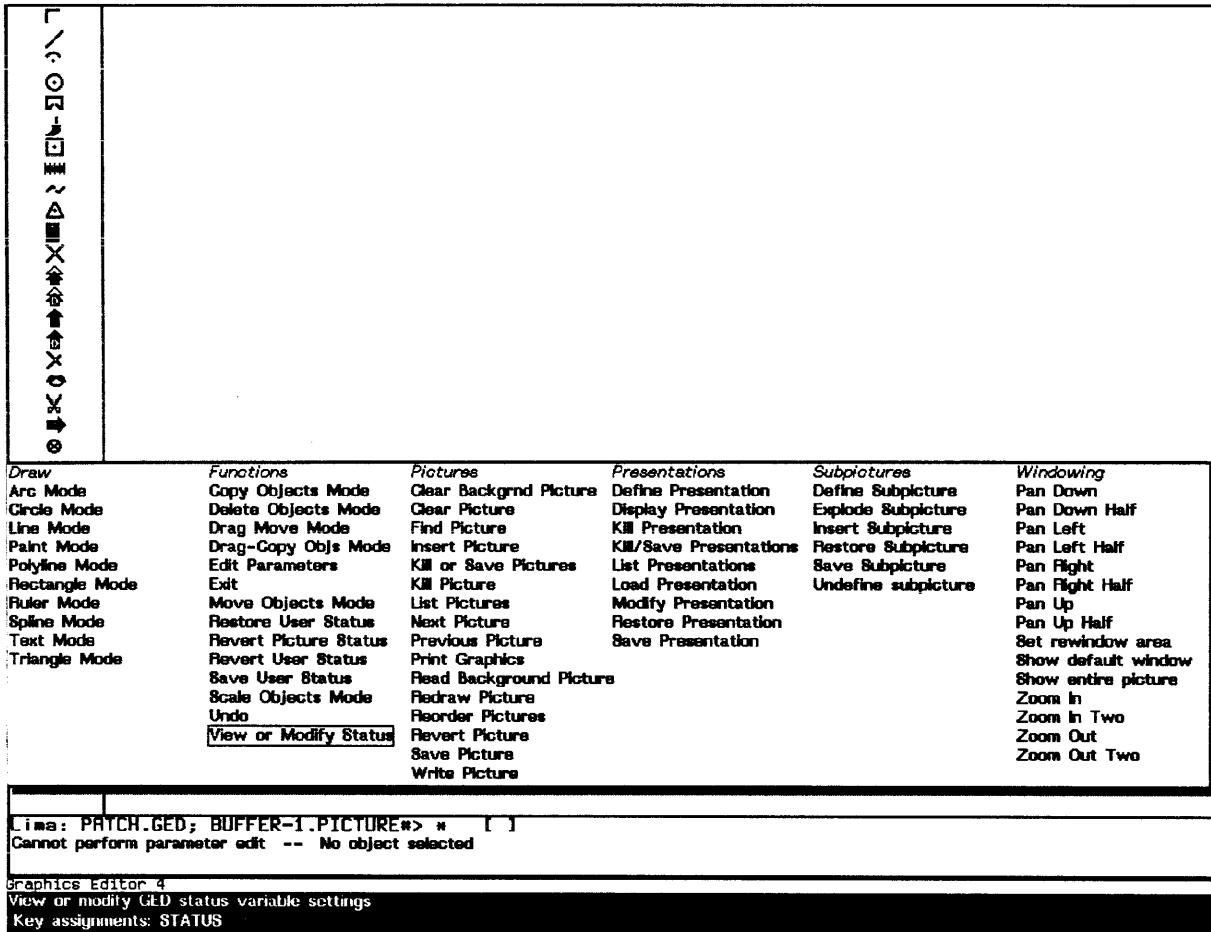
The mouse documentation window also displays the keystrokes (if any) that execute the command. This helps novice users learn a more efficient way to input the command. Figure 6-7 shows an example of the documentation in the mouse documentation window for a command on the graphics editor command menu. This figure also shows an icon menu, which is described in the next paragraph. The menu item lists for these menus are built using the UCL.

Icons

6.3.7.2 Instead of (or in addition to) displaying command names, UCL command menus can represent commands as icons. An icon is simply a character or shape that represents an action. Several fonts provide common icons, such as arrows, scissors, and paint brushes. You can use the font editor to construct new icons appropriate for your application. (The font editor is described in Section 12.)

Icon menus can help novice or infrequent users understand an application through graphic representation of the command action. Icon menus also help expert users by taking up less screen space than an ordinary text menu. An example of an icon menu is shown in Figure 6-7.

Figure 6-7 Mouse Documentation Window and an Icon Menu



Completion Commands

6.3.8 The UCL gives an application the ability to read part of an expression that you are typing and then, upon your command, to complete the rest of the expression for you automatically. The UCL completion commands are keystrokes that invoke different styles of completion on various kinds of typed expressions. (The different kinds of typed expressions recognized by the UCL are discussed in paragraph 6.5.1.) Completion commands are actually implemented by another system component known as the input editor (see the *Introduction to the Explorer System*).

By default, the UCL can perform completion on such expressions as Lisp function names and command names, but completion is also possible on other types of expressions (even application-specific types).

Table 6-1 describes each completion command and the keystroke that invokes it.

Table 6-1

Completion Commands	
Command Key	Completion Description
ESCAPE	<p><i>Recognition completion</i> treats the typed input as the initial substring and completes the word when possible. If several words start with the substring, the word is completed as far as possible.</p> <p>For instance, in an application that has commands named <code>print-results</code> and <code>print-query</code>, the typed character <code>p</code> followed by pressing the ESCAPE key causes the word to complete as <code>print-.</code> Whereas, if <code>print-query</code> is not in the application, the same typed input using the ESCAPE key completes as the full command name <code>print-results</code>.</p>
SUPER-ESCAPE	<p><i>Apropos completion</i> treats the typed input as a substring included anywhere in the completing word and attempts to complete it.</p> <p>For instance, <code>(output-to-string</code> followed by pressing SUPER-ESCAPE completes as follows, which is the only Lisp function that contains this string:</p> <pre>(with-output-to-string</pre>
HYPER-ESCAPE	<p><i>Spelling-corrected completion</i> attempts to complete a word by treating the input string as slightly misspelled with missing, extra, or misplaced characters.</p>
Space bar	<p><i>Auto-completion</i> is recognition completion on typed command names. This allows you to type part of an application command name and then press the space bar to complete the name.</p>
CTRL-/	<p>Displays a mouse-sensitive menu of possible recognition completions.</p>
SUPER-/	<p>Displays a mouse-sensitive menu of possible apropos completions. Refer to Figure 6-8.</p>
HYPER-/	<p>Displays a mouse-sensitive menu of possible spelling-corrected completions.</p>

Figure 6-8 Completion Listing Produced by Pressing SUPER-/

<pre>> (array Select one of the following Functions: #ARRAY ADJUST-ARRAY ADJUST-ARRAY-SIZE ADJUSTABLE-ARRAY-P APPEND-TO-ARRAY ARRAY ARRAY-#-DIMS ARRAY-ACTIVE-LENGTH ARRAY-BITS-PER-ELEMENT ARRAY-DIMENSION ARRAY-DIMENSION-N ARRAY-DIMENSIONS ARRAY-DISPLACED-P ARRAY-ELEMENT-SIZE ARRAY-ELEMENT-TYPE ARRAY-ELEMENTS-PER-Q ARRAY-GROW ARRAY-HAS-FILL-POINTER-P ARRAY-HAS-LEADER-P ARRAY-IN-BOUNDS-P ARRAY-INDEXED-P ARRAY-INDIRECT-P ARRAY-INITIALIZE ARRAY-LEADER ARRAY-LEADER-LENGTH ARRAY-LENGTH ARRAY-POP ARRAY-PUSH ARRAY-PUSH-EXTEND ARRAY-RANK ARRAY-ROW-MAJOR-INDEX ARRAY-TOTAL-SIZE ARRAY-TYPE ARRAY-TYPES ARRAYCALL ARRAYDIMS ARRAYP COPY-ARRAY-CONTENTS COPY-ARRAY-CONTENTS-AND-LEADER COPY-ARRAY-PORION DISPLACED-ARRAY-P FILLARRAY GET-LOCATIVE-POINTER-INTO-ARRAY LIST-ARRAY-LEADER LISTARRAY MAKE-ARRAY MAKE-ARRAY-INTO-NAMED-STRUCTURE MAKE-PIXEL-ARRAY NUMBER-INTO-ARRAY PIXEL-ARRAY-HEIGHT PIXEL-ARRAY-WIDTH RETURN-ARRAY SORT-GROUPED-ARRAY-GROUP-KEY STORE-ARRAY-LEADER</pre>	<pre>Rubout Last Char Delete Character Retrieve Last Input Retrieve Last Kill</pre>
---	---

Lisp Listener 1
Returns the last used element of ARRAY, and decrements the fill pointer.
For an ART-Q-LIST array, the cdr codes are updated so that the overlaid list

Mouse Documentation Window Help

6.3.9 As you are typing expressions, the UCL uses the mouse documentation window to display various kinds of helpful information, such as command arguments or alternative completions.

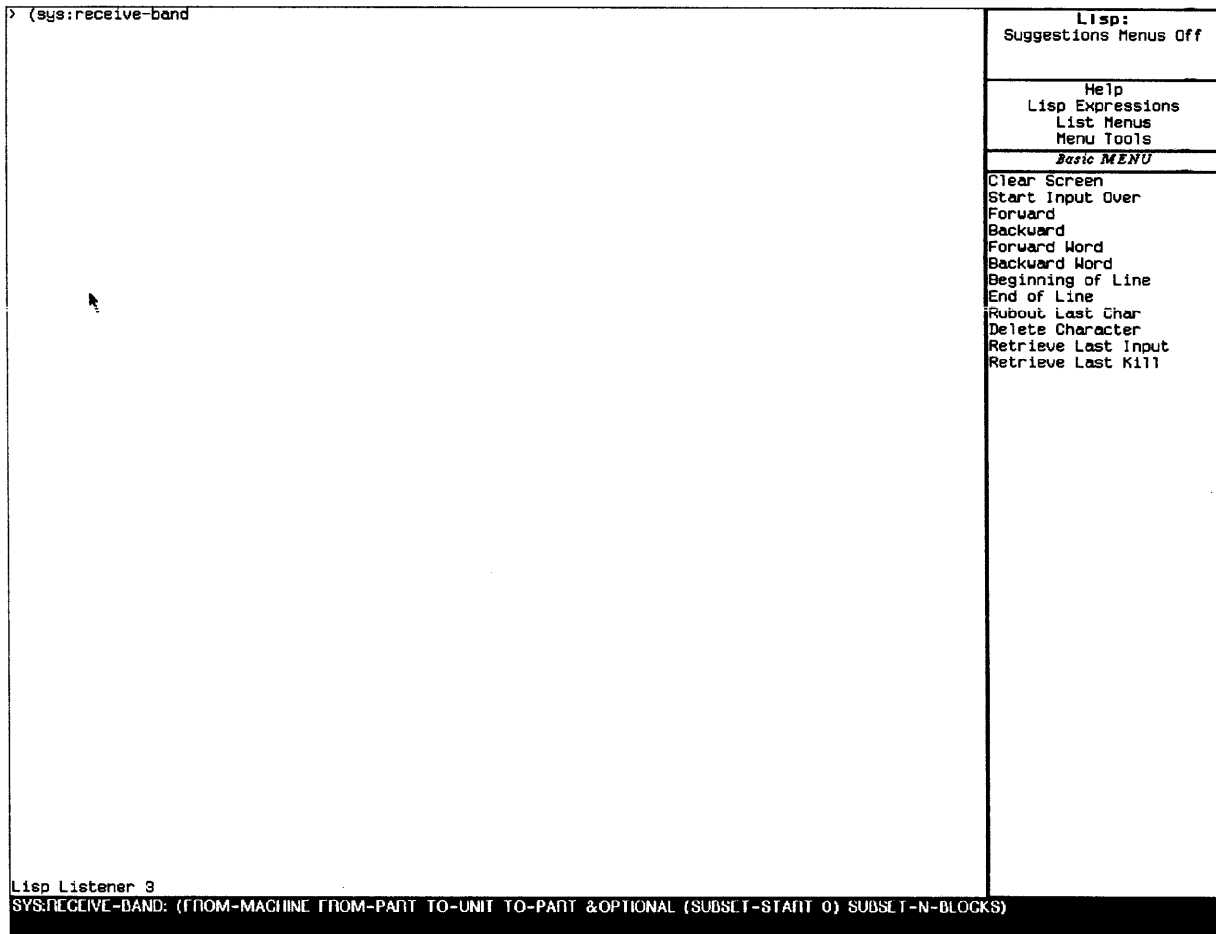
When you press the space bar after typing the first symbol of the input line, the UCL attempts to tell you something about the symbol:

- If the symbol is a function name preceded by a left parenthesis, the function's argument list is displayed. (Figure 6-9 shows an example of the function documentation in the mouse documentation window.)
- If the symbol is an implicit message (see paragraph 6.5.1, Typed Expressions), the message argument list is displayed.
- If the symbol is an application command name, the command's short description and required arguments (if any) are displayed in the mouse documentation window.
- If the symbol is a special expression that the application processes, the application can easily designate what help is provided in the mouse documentation window.

When you execute the ESCAPE, HYPER-ESCAPE, or SUPER-ESCAPE completion commands, the mouse documentation window displays the possible completions if there are several. If there are no completions, an error message is displayed.

NOTE: All of these uses of the mouse documentation window are optional, depending on how the UCL is written and on your default UCL options (refer to paragraph 6.4.6, Top Level Configurer).

Figure 6-9 Function Argument List in Mouse Documentation Window



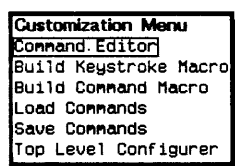
**Environment
Customization
Features****6.4** UCL provides the following environment customization mechanisms:

- **Command Editor** — Allows you to modify commands.
- **Build Keystroke Macro** — Creates a command that executes a series of commands that you specify by their keystrokes.
- **Build Command Macro** — Creates a command that executes a series of commands that you specify by their names.
- **Save Commands** — Saves all of your changes so you can reload them in a later session.
- **Load Commands** — Loads the customizations that you saved with Save Commands.
- **Top Level Configurer** — Sets default UCL options and tailors how typed expressions are processed. You can also save these changes and reload them in a later session.

These customization mechanisms are automatically added to an application by the UCL; they are among the *universal* commands. All are accessible from the Help command menu that is invoked by pressing the HELP key (described in paragraph 6.3.1). When you select the Customization Menu option from the Help command menu, the pop-up menu shown in Figure 6-10 is displayed. Additional menus are shown, as appropriate, for any item you choose that creates a new element. The following paragraphs describe each of the options on the Customization menu.

Note that these options also have associated keystrokes. When you position the mouse cursor over an option in the menu, the mouse documentation line displays its keystroke. The keystrokes are also listed in the Command Display window.

Figure 6-10

Customization Menu

Command Editor

6.4.1 The Command Editor command displays all of the commands of the current application in a scroll window similar to the command display (see Figure 6-11). Clicking left on a command (the entire line, including the command name, keystroke, and description should be boxed) pops up a window that allows you to modify any of the command's data, including the following;

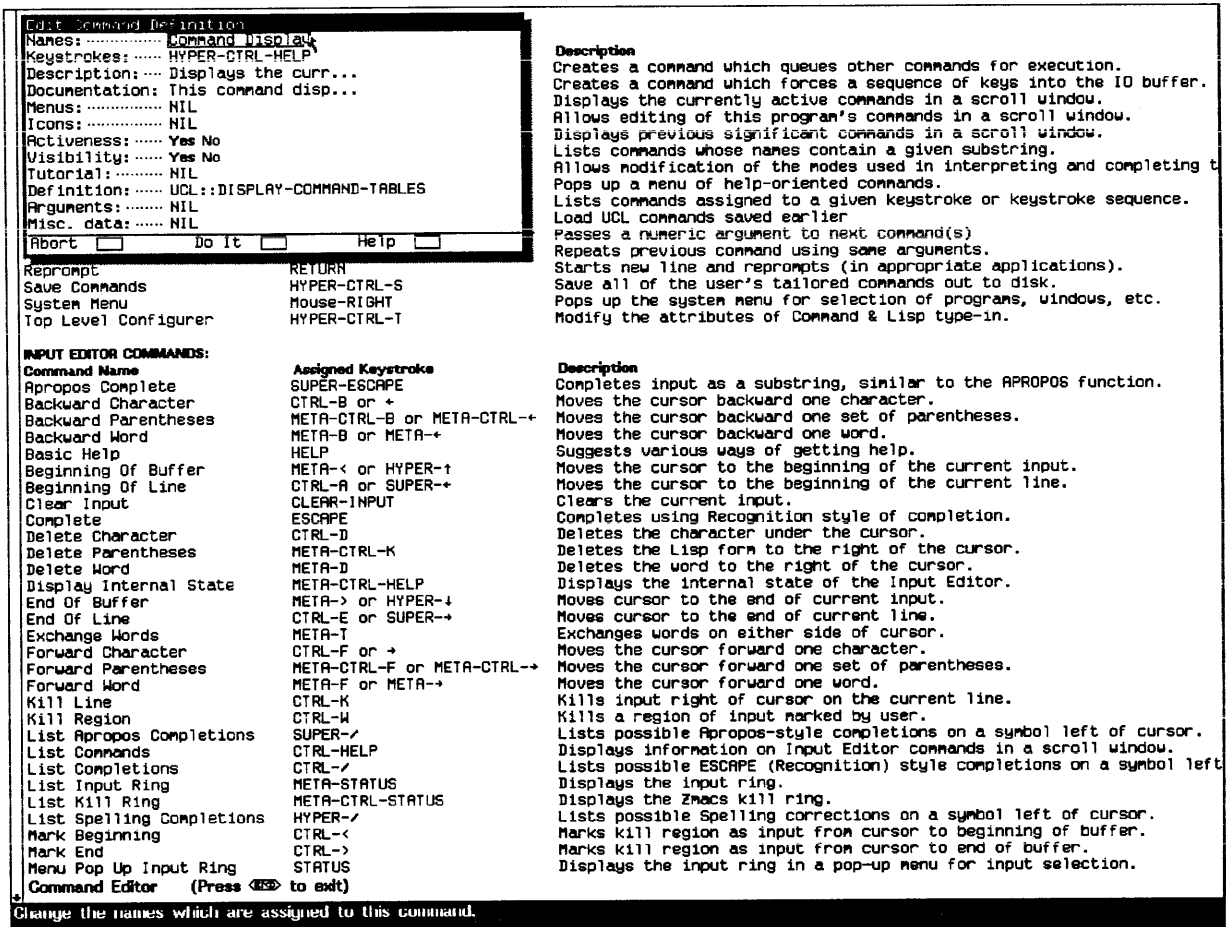
- Command names
- Keystrokes
- Description
- Documentation
- Menus
- Function or method to execute

If you want to modify only the command name, keystroke, or description, you can box only that part of the line where the command is listed.

The command editor allows you to control the command environment. You can bind keystrokes to the key or mouse button that is most convenient to use, change command names to whatever is easier to type or remember, and reconfigure command menus to display the desired commands.

For sophisticated users, the command editor has an option for finding and editing the source code of a command. You click right on the entire line containing the command name and select Edit Command Source. Another option allows you to define and add new commands to the application. You click right on the *command table* item, such as UNIVERSAL COMMANDS, and select Add Command.

Figure 6-11 Command Editor Display and Menu Used for Editing a Command



Build Keystroke Macro

6.4.2 The Build Keystroke Macro command allows you to create a macro command that forces a series of keystrokes into the selected input/output (I/O) buffer. You can supply in the choose-variable-values window a name and description for the macro and a keystroke or keystroke sequence to invoke the macro. When you click on Do It, the next keystrokes you press are collected for the macro. When you press HYPER-CTRL-M, collecting stops.

Keystroke macros give you a fast way to input often-used keystroke sequences. The macros can be designed to execute sequences of keystroke commands, automate the typing of expressions, or do both. For instance, a Lisp Listener keystroke macro can be built and assigned to a single keystroke to invoke the following:

```
(print-herald)
(print-disk-label)
(hostat)
CTRL-4
CTRL-C
```

This results in calls to the functions **print-herald**, **print-disk-label**, and **hostat**, followed by two input editor commands (CTRL-4 CTRL-C) that copy previously typed expressions into the I/O buffer. All of this is accomplished by simply pressing the macro's assigned keystroke.

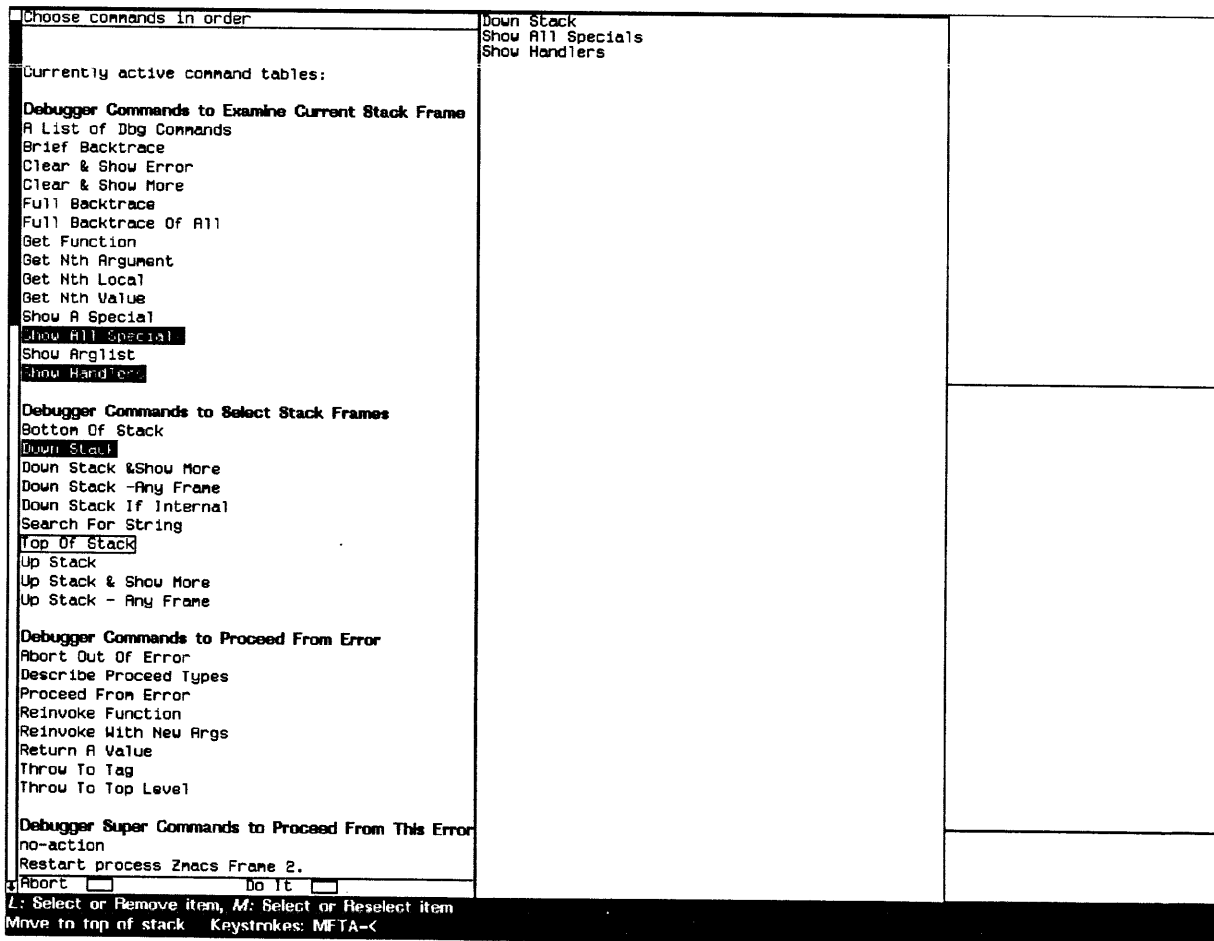
Build Command Macro

6.4.3 The Build Command Macro command allows you to create a macro command that directly executes a series of other commands. The Build Command Macro displays a choose-variable-values window that allows you to enter the macro name, its description, and the keystrokes that will invoke it. When you click on Do It, a menu appears that lists the commands in the currently active command tables. You select the commands in the order you want them executed and click on Do It when finished.

Command macros are provided for situations in which keystroke macros are not sufficient for executing the desired command sequence. For instance, applications with large command sets do not always assign keystrokes to every command. Command macros can be built for these applications because they do not depend on assigned keystrokes to execute.

Figure 6-12 shows this option executed from the window-based debugger. At the left is a menu of window-based debugger commands with the selected ones highlighted. The next column to the right shows the order of the selected commands. When executed, this macro command moves down the stack and displays information about the stack frame.

Figure 6-12 Build Command Macro



Save Commands 6.4.4 The Save Commands command saves all your command modifications, keystroke and command macros, and added commands in a file that you specify, normally in your login directory. This command not only saves the modifications made to the current application but saves modifications made to *any* UCL application; that is, it saves your entire command environment in one file that can be loaded at a later time.

Load Commands 6.4.5 The Load Commands command loads your command environment customizations that were previously saved using Save Commands described in the preceding paragraph. Load Commands reinstalls macro commands, key bindings, and any other modifications that you made. You can invoke this command after logging in or can include a load form in your LOGIN-INIT file to load the command environment file that was saved by the Save Commands command.

Top Level Configurer 6.4.6 The Top Level Configurer command allows you to set default UCL options and to tailor how typed expressions are processed in the UCL command interpreter to suit your preference and skill level. For instance, if you reset the Prompt String option, it is reset for the Lisp Listener, break, and any other UCL applications that use the default prompt. The Top Level Configurer window is shown in Figure 6-13.

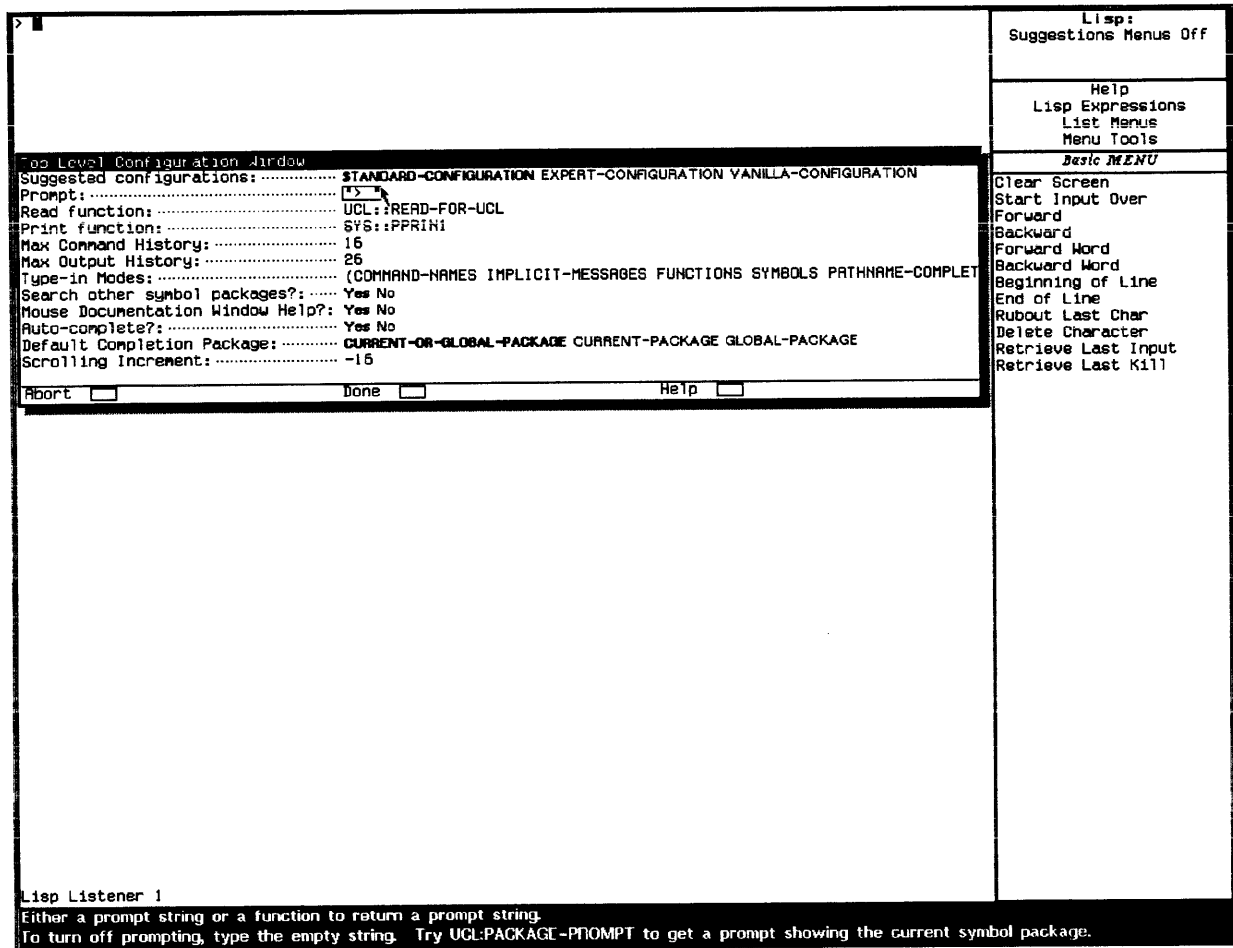
The Top Level Configurer command provides an easy-to-use window interface. When you finish setting the configuration, a prompt is shown that allows you to save the configuration into your LOGIN-INIT file for use in future sessions.

Initially, the options are set to a configuration (the standard configuration) that is most helpful for novice users. Expert users can configure the options for improved efficiency by using the expert configuration. The vanilla configuration turns off all features (that can be turned off). The following are descriptions of the options that can be configured:

- *Prompt* — A string or a function that returns a prompt string. The default prompt is the greater-than sign (>).
- *Read Function* — The read function called for reading typed expressions. The default is `ucl:read-for-ucl`, which can read both multi-line function calls (enclosed in parentheses) and multiple symbols on one line (for implicit message sending and implicit parentheses function calling). You might prefer `read-for-top-level` if you merely want to type standard function calls and single symbols since `read-for-top-level` terminates reading of symbols when you press the space bar.
- *Print Function* — The function called for printing results of evaluated Lisp forms. The default pretty prints the results. Some users prefer a more efficient print function.
- *Max Command History* — The maximum number of commands to store and display using the Command History command (described in paragraph 6.3.4).
- *Max Output History* — The maximum number of Lisp forms to store and to display using the HYPER-STATUS command. (HYPER-STATUS displays the previous output values in a pop-up menu for input selection.)

- *Type-In Modes* — The hierarchy of modes used for interpreting and for completing partially typed expressions. You can reorder, activate, and deactivate modes as preferred. For instance, you can deactivate implicit message sending mode if it is never used and can activate implicit parenthesis function call mode when it would be convenient. (See paragraph 6.5.1 for more information about type-in modes.)
- *Search other symbol packages?* — Specifies whether the UCL should search other packages for unrecognized Lisp symbols.
- *Mouse Documentation Window Help?* — Specifies whether the mouse documentation window should display help on typed expressions and completion, as described in paragraph 6.3.9. Expert users can disable the help for greater efficiency.
- *Auto-complete?* — Specifies whether auto-completion should be attempted when the space bar is pressed after typing the first symbol, as described in paragraph 6.3.8, Completion Commands.
- *Default Completion Package* — Specifies the package to search when you complete on a Lisp symbol that has no package prefix. The choices are current package only, global package only, or both the current and the global packages.
- *Scrolling Increment* — Controls the scrolling in Lisp Listeners. Scrolling can be made faster (less smooth), slower (more smooth), or can be turned off. When scrolling is turned off, output that exceeds the window space wraps to the top of the window.

Figure 6-13 Top Level Configurer Window



Miscellaneous Features

6.5 The following miscellaneous UCL features that enhance applications are described in subsequent paragraphs:

- Interpreting various kinds of typed expressions
- Sending messages implicitly to a designated flavor instance
- Reading command arguments
- Reading numeric arguments to commands
- Redoing the previous command
- Displaying the System menu
- Catching errors

Typed Expressions

6.5.1 The UCL has a flexible mechanism for interpreting and performing completion on different kinds of typed expressions. The following paragraphs describe the expression types that are processed, the algorithm used, user configuration of the feature, and suggestions for the processing of special types of expressions.

*Kinds
of Expressions*

6.5.1.1 Six kinds of expressions are processed, allowing you to type and complete on application command names, various types of Lisp expressions, and pathnames. The following discussion describes each kind of expression:

- **command-names** — Command names in the current application can be typed to execute the corresponding command. If a command accepts arguments from the user, you can type the arguments next to the command name (see paragraph 6.5.2, Obtaining Arguments). Auto-completion may be available for command names, but generally it is not used because of efficiency.
- **functions** — A normal Lisp function call can be either typed or partially typed and completed. A function call is entered by typing it enclosed in a matching set of parentheses. The call is evaluated and the returned values are printed.
- **symbols** — Lisp symbols can be typed and completed. The value of the symbol is printed.
- **implicit-paren-functions** — Function calls can be made without typing the outer set of parentheses. By default, this type of expression entry is deactivated because ambiguities occur between Lisp function names and symbols.
- **pathname-completion** — This allows you to use completion on a pathname that is partially typed when typing a pathname as an argument to a command, function call, or implicit message.
- **implicit-messages** — Refer to paragraph 6.5.1.2 for details.

NOTE: By default, all of the expression types are active except for implicit-paren-functions. You can make implicit-paren-functions active by adding it to the type-in modes list in the Top Level Configurer. Refer to paragraph 6.4.6.

*Implicit Message
Sending (rotl)*

6.5.1.2 An implicit-messages expression is used to send a message to a designated flavor instance. This type of expression is useful when you are working with flavors. It both reduces the amount of typing you need to do and helps you locate a flavor's method names through the use of completion. The following list describes how to make use of this type of expression:

- `implicit-messages` must be in the type-in modes list that you can see in the Top Level Configurer. The default value of this list contains `implicit-messages`.
- Call the function `rotl` (Run Object Top Level) with a flavor instance as the argument.
- While implicit message sending is active, function calls, symbol evaluations, and implicit messages are made with the instance variables of the designated flavor instance bound.
- Messages can then be sent to the instance by typing the message followed by any arguments. The returned values are printed. Auto-completion is available on these message names. In Lisp Listeners, a message's argument list is displayed in the mouse documentation window, just as function argument lists are displayed (see paragraph 6.3.9).
- `(rotl nil)` turns off implicit message sending.

*Algorithm Used for
Typed Expressions*

6.5.1.3 Although several kinds of expressions are processed, you do not have to specify the kind of expression; the UCL determines what kind of expression it is.

Each type of expression has a defined type-in mode that handles completion and processing. The type-in modes are stored in a list in the order that determines their precedence when ambiguities occur during completion and processing of expressions. When you type an expression or invoke completion on it, the list is searched sequentially; each type-in mode is given the chance to claim responsibility for the expression.

The modes use the syntax of the expression to help determine whether to claim it. For example, the mode that handles normal function calls claims expressions that are enclosed in parentheses. The mode that handles symbol evaluation claims expressions consisting of one symbol. The `command-names`, `implicit-messages`, and `implicit-paren-functions` modes all claim expressions starting with a symbol followed by zero or more forms.

Ambiguities occur occasionally. For instance, suppose an application has a command name *Load* and the Lisp symbol `load` is set to some value. If you type *Load*, the list of modes is searched. Because the `command-names` mode occurs on the list before the symbol mode, the *Load* command in the application is executed.

User Configuration of Type-In Modes

6.5.1.4 Each kind of typed expression is processed by an object known as a *type-in mode*. The universal command `Configure Type-In Modes` (`HYPER-CTRL-()`) can be used to modify the hierarchy of the modes that handle completion and processing of typed expressions. It also can be used to activate or deactivate modes. For instance, if you like `implicit-paren-functions` mode, you can activate it. If you want to have completion on Lisp symbols but find that the presence of application commands interferes with completion, you can move the `symbols` mode above the `command-names` mode in the precedence list. The `Configure Type-In Modes` command provides a quick means of modifying the modes; the universal `Top Level Configurer` command provides a similar mechanism and allows you to store a mode precedence list in your `LOGIN-INIT` file.

Special Expressions

6.5.1.5 The UCL processing of typed expressions can be easily customized for an application. Applications can specify a list of type-in modes to process, instead of accepting the modes described previously. For example, an application intended for a user who knows nothing about Lisp can specify a list that contains no Lisp processing modes.

It is relatively simple to create special-purpose type-in modes. The following are three reasons why you would want to do so:

- To build Lisp type-in modes for user convenience, such as a function call mode that automatically quotes unbound symbols.
- To construct interpreters for languages such as Prolog or Scheme.
- To process typed expressions that are not commands but have some meaning within an application. For instance, an expert system could have a type-in mode that recognizes typed rule names and prints information about rules.

Special type-in modes can perform all of the tasks that the other modes do, including mouse documentation window help and completion. Implementation of special type-in modes is discussed in Section 7, `UCL Programmer Interface`.

Obtaining Arguments

6.5.2 The UCL provides a feature for defining commands that obtain arguments from the user. If you input a command by typing its name (instead of selecting it from a menu or pressing its keystroke), you can type the arguments to the right of the name. The mouse documentation window displays any arguments the command accepts when you type a space after the command name. (The mouse documentation window display is discussed in paragraph 6.3.9.)

If you input a command by using a menu, keystroke, or mouse button or by typing the command name followed by an incorrect number of arguments, a `choose-variable-values` window pops up that requests the required arguments. Default values, prompt strings, and data types can be provided for each argument obtained.

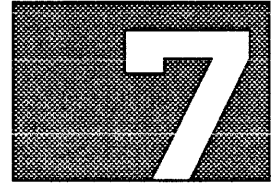
-
- Numeric Arguments** 6.5.3 The UCL allows you to supply numeric arguments to keystroke commands by typing an integer with a modifier key before the keystroke (or keystroke sequence) of the command. By default, a numeric argument causes the command to be repeated the specified number of times. However, application commands can be defined to handle the argument differently, instead of executing repeatedly. For example, a hypothetical *Delete Rule* command could be defined to delete the *n*th rule when a numeric argument is supplied.
- A numeric argument is input by chording any combination of the HYPER, SUPER, META, and CTRL keys with a number key. This provides an initial, one-digit value. Further digits can be added in the same manner. Additionally, CTRL-U, the universal argument, can be pressed to multiply the current numeric argument value by four. Once the desired value is reached, you then type the keystroke sequence for the command.
- An additional feature allows you to use a numeric argument to perform a series of keystroke commands. Once a numeric argument is input, you can press CTRL-(followed by any number of command keystrokes, terminated by pressing CTRL-). The sequence of commands is then repeated the desired number of times. This feature saves you from having to construct a keystroke macro for an operation that is to be used only once.
-
- Redo Command** 6.5.4 The universal command Redo Command (HYPER-CTRL-R) repeats the previously executed command, reusing any arguments it previously calculated or obtained from you (see paragraph 6.5.2, Obtaining Arguments). This command permits you to reexecute a command without having to retype its command name and arguments.
-
- System Menu** 6.5.5 The universal command System Menu displays the System menu. It is invoked by clicking right, unless an application command claims that button. This command is provided to support the system convention of assigning the right mouse button to the System menu, unless you have reassigned the right mouse button for use in the given application.
-
- Error Catcher** 6.5.6 The UCL provides a feature that shields a naive user from the debugger. Inexperienced users have trouble recovering from errors, especially when the debugger has filled the application windows with internal information the user does not understand. When an error occurs in a UCL application, a window pops up, displays the error message, and asks the user whether to enter the debugger or to return to the top level of the command interpreter. Often the error condition is local to particular commands, and possibly the user can continue with the application by choosing not to enter the debugger. You can remove the error catcher feature in applications for which it is not appropriate. For example, the Lisp Listeners do not use the error catcher because programmers usually want to enter the debugger when an error occurs.

Command Summary 6.6 Table 6-2 lists the UCL keystroke commands.

Table 6-2

UCL Keystroke Commands	
Command Name	Keystroke
Help	HELP
Application Help	Select from Help menu
Tutorial	Select from Help menu
Explorer Overview	Select from Help menu
System Menu	Mouse right
System Application	SYSTEM HELP
TERM Key Help	TERM HELP
Command Type-In Help	Select from Help menu
Command Display	HYPER-CTRL-HELP
Command History	HYPER-CTRL-P
Command Name Search	HYPER-CTRL-N
Keystroke Search	HYPER-CTRL-K
Command Editor	HYPER-CTRL-STATUS
Build Keystroke Macro	HYPER-CTRL-M
Build Command Macro	HYPER-CTRL-C
Save Commands	HYPER-CTRL-S
Load Commands	HYPER-CTRL-L
Top Level Configurer	HYPER-CTRL-T
Redo	HYPER-CTRL-R

UCL PROGRAMMER INTERFACE



Introduction

7.1 The Universal Command Loop (UCL) is a tool that can shorten program development time by providing a ready-to-use command interpreter with built-in help features. This section discusses the programmer interface to UCL. (Section 6 discusses the user interface to the UCL.)

The UCL provides the following macros, functions, and flavors for use in constructing an application command interpreter:

- **defcommand** and **make-command** macros — The **defcommand** and **make-command** macros create a command object, an instance of the **ucl:command** flavor. A command instance is defined by the code that the command executes, as well as other data, such as command names, command keystrokes, and command documentation.
- **build-command-table** function — The **build-command-table** function collects a group of commands defined by **defcommand** or **make-command** into a command table. A command table, which is an instance of the **ucl:command-table** flavor, is used to determine which application commands are active. Most applications require only one command table. Some applications may use several command tables to represent different application contexts.
- **build-menu** function — The **build-menu** function is a powerful tool for defining command menus. This function generates a command menu item list that is used by other UCL functions to automatically construct command menus.
- **ucl:basic-command-loop** and **ucl:command-loop-mixin** flavors — A UCL application command interpreter (or command loop) is an instance of the **ucl:basic-command-loop** flavor. Alternatively, your application command interpreter can be an instance of your own flavor, which uses the **ucl:command-loop-mixin**. A command loop performs the basic function of reading user input, recognizing command names and keystrokes, and executing the appropriate command object. Many other advanced features for displaying help information and controlling user input are built into the **ucl:basic-command-loop** and **ucl:command-loop-mixin** flavors.

Because UCL commands and command interpreters are implemented with flavors, you can customize the basic UCL features for your application by using method daemons, wrappers, and redefinitions. For example, although the basic UCL command interpreter responds to unrecognized user input by beeping, you can modify a method of **ucl:basic-command-loop** to respond instead by printing an error message.

Here is an example of a simple UCL application. It contains five commands that do the following: beep, make the sound of a doorbell, make the sound of a ricochet, pop up a menu of sounds, and exit.

First, a flavor is created that contains the command-loop flavor and some suitable application window flavor. Only one command table is defined, and it is always active (that is, its commands are executable).

```
;; Define an application command-loop window flavor.

(defflavor sound-demo ()
  (ucl:basic-command-loop
   w:window                               ; Your program must specify the
                                           ; window mixins you need.
  )
  ;; The following keywords are instance variables that your program
  ;; must initialize. The active-command-tables instance variable
  ;; specifies a list of command tables of active commands. The
  ;; all-command-tables instance variable is used
  ;; by UCL to access all the commands in your application for various
  ;; help and customization features, such as Command Name Search.
  (:default-init-plist
   :active-command-tables '(sound-commands)
   :all-command-tables   '(sound-commands)
  )
  )

;; Create the application command-loop object.

(defparameter sound-demo-loop (make-instance 'sound-demo))

;; Define five application commands.

(defcommand beep-command ()
  ^(:description "A beep sound"
   :names      ("Beep")
   :keys       (#\control-b)
  )
  (beep)
  )

(defcommand doorbell-command ()
  ^(:description "A doorbell sound"
   :names      ("Doorbell")
   :keys       (#\control-d)
  )
  (beep :doorbell)
  )

(defcommand ricochet-command ()
  ^(:description "A ricochet sound"
   :names      ("Ricochet")
   :keys       (#\control-r)
  )
  (beep :shoop)
  )

(defcommand display-sounds-menu ()
  ^(:description "Pop up a menu of sounds"
   :names      ("Sounds")
   :keys       (#\mouse-r-1)
  )
  (ucl:pop-up-command-menu 'sound-command-menu)
  )
```

```

(defcommand quit ()
  (:description "End the sounds demo"
   :names      ("Quit")
   :keys       (#\end)
  )
  (send sound-demo-loop :quit)
)

;; Build an application command table.

(build-command-table 'sound-commands 'sound-demo
  (display-sounds-menu
   beep-command
   doorbell-command
   ricochet-command
   quit
  )
)

;; Create the item list used by the command menu.

(build-menu 'sound-command-menu 'sound-demo
  :item-list-order
  (beep-command
   doorbell-command
   ricochet-command
   quit
  )
)

;; Select the application command-loop window and begin processing
;; user input. You should try to:
;;   - Invoke a command by typing the command name. (Note that a typein window
;;     appears after you start typing.)
;;   - Invoke a command by pressing the command keystrokes.
;;   - Invoke a command from the pop-up command menu.
;;   - Press HYPHER-CTRL-HELP to see the Command Display.
;;   - Use some of the other UCL universal commands.

(w:window-call (sound-demo-loop :deactivate)
  (send sound-demo-loop :command-loop)
)

```

NOTE: The file `SYS:UCL; STARTER-KIT.LISP` contains examples of many of the UCL features described in this section.

Basic Command Interpreter Operation

7.2 On every iteration of its command loop, the basic UCL command interpreter reads standard Lisp machine input: a keystroke, mouse button click, menu blip, or typed expression (which can be a command name, Lisp form, or whatever the application permits). The interpreter tries to convert the input into an executable command. If the conversion is successful, the command is executed. If not, the interpreter notifies the user by beeping or printing an error message, whichever is appropriate.

The following list describes this process:

1. Read one object that is input from the user. The object is read from the stream that is the value of the `*standard-input*` variable.

2. Try to translate the object into a command to execute. The object can be one of the following:

a. Menu blip. First, execute the blip. (Refer to the *Explorer Window System Reference* for information about blips.)

If the execution returns an instance of a UCL-defined command (the usual case for UCL command menus) and the command is in one of the active command tables, go to Step 3.

Otherwise, if the execution returns an inactive command, go to Step 4.

Otherwise, return to Step 1. (The execution is assumed to have caused some desirable side effect.)

b. Mouse button or keystroke. Mouse clicks are processed like keystrokes. The list of active command tables is searched for a match to the button or keystroke input.

If one of the command tables contains a `ucl:command` that claims (matches) the key or that directs the fetching of more keystrokes (as would be the case for the Zmacs commands that use a CTRL-X prefix), then go to Step 3.

Otherwise, if the key has control bits (such as CTRL, META, SUPER, HYPER, or a mouse blip), assume that the user typed an incorrect keystroke and go to Step 4.

Otherwise, assume the input is a typed expression. Read the expression. (At this point, UCL completion commands can be used to complete a partial command expression; refer to paragraph 6.3.8, Completion Commands.)

- If the list of active command tables contains a UCL-defined command with that name, go to Step 3.
- Otherwise, if the expression is a valid Lisp form, evaluate the form, print the results, and return to Step 1. This alternative is easily customized by the programmer; if you do not want Lisp processing in your application, you can disable it.
- Otherwise, go to Step 4.

3. Execute the command. The input is translated into a command and executed by sending the command an `:execute` message, which in turn calls the function or method defined for the command. Go to Step 1.

4. Handle unknown input. Beep on bad keystrokes, mouse clicks, and menu blips; print an error message for mistyped command names. This behavior is easily customized by the programmer. Go to Step 1.

In addition to the command-loop processing outlined previously, the basic UCL command interpreter automatically handles a number of special input conditions:

- Input editor commands — Before any command expression has been typed, the interpreter recognizes and executes any input editor

command. For example, a user can press CTRL-C to recall the previous typed expression or CLEAR SCREEN to clear the type-in window.

- Preempted input — A partially typed command expression can be preempted by another command (for example, by clicking on a command in a command menu). In this case, the preempting command is executed, and the preceding typed expression input is saved. After execution of the preempting command, the preceding partially typed command expression is redisplayed. The user can then either finish typing the expression or preempt it again.

defcommand and make-command Macros

7.3 The **defcommand** and **make-command** macros create a command object (typically, an instance of the `ucl:command` flavor) by using various data that define the name, keystrokes, and code for the command.

If you use **defcommand**, you can choose from two different ways of saving the code executed by the command: either as a function or as a method of the application command-loop flavor. You should define a command as a method if the command code makes direct references to the instance variables of the application command-loop flavor. Otherwise, you should probably specify that the command code is to be saved as a function.

The **make-command** macro is essentially the same as **defcommand**. The difference is that, for a command created by **make-command**, the command code is given by a function/method defined elsewhere by a **defun/defmethod** form. This means that **make-command** is particularly useful for integrating an existing application with the UCL.

defcommand Macro

7.3.1 The following describes the **defcommand** macro.

defcommand *defname lambda-list keyword-values . body* Macro

Defines a command and returns a command object instance. The **defcommand** macro expands into two forms that do the following:

1. Define a function or method to execute the command.
2. Store all of the command data in a `ucl:command` flavor instance.

The first expanded form is either a **defun** or a **defmethod** using the *defname*, *lambda-list*, and *body* arguments. The particular expansion depends on *defname*:

- If *defname* is a symbol, the expansion is as follows:

```
(defun defname lambda-list . body)
```

- If *defname* is a list, the first element is the name of the application command-loop flavor and the second is a method name. The expansion is as follows:

```
(defmethod defname lambda-list . body)
```

A UCL command interpreter executes a command defined as a function by calling the defined function. A command defined as a method is executed by sending a message to the application command-loop object.

The second expanded form of `defcommand` creates a `ucl:command` flavor instance that holds the data supplied by *keyword-values*. The *keyword-values* argument is a form that `defcommand` evaluates. The evaluation is done to permit the convenient use of the backquote syntax. For example, the following `defcommand` calls a function to supply the `:documentation` value:

```
(defcommand (foo-application :quit) ()
  `(:names "exit"
     :keys #\super-end
     :documentation ,(fetch-documentation :quit))
  (send self :bury))
```

- Arguments:*
- defname* — Either a symbol or a method specification list.
 - lambda-list* — The lambda list of the function or method defined.
 - keyword-values* — A form (usually a quoted list) that is evaluated to produce a list of all the command data, including command names, keystrokes, short description, long description, and so forth, that document the command and specify how the command is to input. The `ucl:get-command` function can be used to access the `ucl:command` instance that stores this data.

All keywords allowed in *keyword-values* are optional; default values are provided when a keyword is omitted. The following keywords are available:

Keyword and Default	Description
:active? Default: <code>t</code>	A flag indicating whether this command is executable through normal input by the user. If the flag is <code>nil</code> , the command causes the console to beep instead of executing the defined function or method. This keyword allows you to deactivate a command until it is debugged or implemented.
:active-in-display? Default: <code>t</code>	A flag indicating whether this command is executable when the user selects it for execution from the UCL Command Display window (see paragraph 6.3.3, Command Display). The default is <code>t</code> because most commands can be executed from the display. If the flag is <code>nil</code> , the user cannot execute this command from the Command Display window. For example, a command that uses the location of the mouse in its computation must be input by a mouse click and thus is not executable from the display.
:arguments Default: <code>nil</code>	A list describing the arguments applied to the function or method to execute this command. Because this keyword has several options and its details are complex, it is described in paragraph 7.3.2, <code>:arguments</code> Keyword of <code>defcommand</code> .
:command-flavor Default: <code>ucl:command</code>	The name of a flavor to use for instantiating the command. This option provides a way of using your own command flavors. If you construct your own flavor, be sure to mix in <code>ucl:command</code> .
:description Default: The first line of :documentation	A one-line text string that describes this command. The string is used in help features such as the Command Display and the mouse documentation window for UCL command menus.

Keyword and Default	Description
:documentation Default: " <i>command-name</i> command"	A form whose value is a string that contains the full documentation of the command. The string can contain several pages of text. This string is used in UCL help features such as the Command Display command.
:icons Default: nil Default <i>font-spec</i> : fonts:mouse	A list of icon characters that can be used as the printed representation of this command in UCL command menus (see the subsequent :menus option). Entries in this list are of the form <i>n</i> or (<i>n</i> :font <i>font-spec</i>), where <i>n</i> is an integer character code.
:keys Default: none	A list of keystroke sequence entries. Each entry is a list of one or more characters representing a keystroke sequence that the user can press to execute this command. For example, the following list specifies that the user can press either the END key or CTRL-X CTRL-Z to execute this command. <pre>:keys ((#\end) (#\ctrl-x #\ctrl-z))</pre> Entries can contain mouse button characters, such as #\mouse-r, because the UCL treats mouse clicks as keystrokes. In addition, the following abbreviations are accepted for this option: <pre>:keys #\f is treated as :keys ((#\f))</pre> <pre>:keys (#\f #\g) is treated as :keys ((#\f) (#\g))</pre>
:menus Default: nil	A list of menu item descriptors. This option can be used to construct command menus in a bottom-up fashion. Each menu item descriptor in the list defines how the command is displayed in a command menu. Each menu item descriptor is one of the following: <ul style="list-style-type: none"> ■ Menu symbol — A symbol that is subsequently used as the first argument in a call to build-menu (see paragraph 7.5, build-menu Function). ■ List of menu symbol and keyword-value pairs — A list consisting of a menu symbol followed by keyword-value pairs. Any menu item keyword, such as :font or :documentation, is allowed (refer to the Choice Facilities section in the <i>Explorer Window System Reference</i>). Usually, the :type menu item keyword is not given here because the preferred item type for command menus is generated by default. In addition, the following two keywords are also accepted: <ul style="list-style-type: none"> ■ :column <i>column-heading</i> — The <i>column-heading</i> is a string that identifies the menu column under which the command is displayed. Use of this option establishes the menu given by the menu symbol as a multicolumn menu. Any other command definition that places a command in this menu must also specify the column option. Otherwise, an error is signaled.

Keyword and Default	Description
	<ul style="list-style-type: none"> ▪ :print-form <i>keyword-list</i> — The value is a list of keyword-value pairs that specify which command name or icon to use in the menu item (see the :names and :icons options for defcommand). The available keywords for :print-form are as follows: <ul style="list-style-type: none"> ▪ :use-icon — If non-nil, a command icon is displayed in the menu item. The default value is nil, which means that a command name is displayed. ▪ :index — An integer index that selects one of the command's list of names (or icons). The default value is 0, which selects the first name or icon.

The following examples demonstrate the use of the **:menus** option:

```
(defcommand command-in-menu1 ()
  ^(:menus (menu1)))

(defcommand command-in-menu2 ()
  ^(:menus ((menu2 :font fonts:hl12b))))

(defcommand command-in-two-menus ()
  ^(:menus ((menu1 :print-form (:use-icon t :index 1))
            {menu2 :font fonts:hl12b :column "Column A"})
    :names ("First Name" "Second Name")
    :icons ((1 :font fonts:mouse) (2 :font fonts:mouse))
  ))
```

:names
Default: '(*defname*)

A list of command names for this command. The first entry of this list is used as the default command name in command menus and in help features such as the Command Display command. Each entry of this list is either a string or a list containing a string followed by the **:typein-name?** keyword and its value.

The **:typein-name?** keyword controls whether the command can be executed by typing the associated command name string. If the value for **:typein-name?** is **nil**, then the string can appear in command menus and help displays but cannot be typed to execute the command. An application can use this keyword for command name abbreviations used only for command menus; in this case, it may be more efficient and less confusing to prevent the user from typing the abbreviated name. By default, the value of **:typein-name?** is **t**, which means that the command name string can be typed to execute the command.

The following example shows the use of the **:names** option:

```
(defcommand a-command ()
  ^(:names
    ("First Name" ; can be typed
     ("Second Name" ; can be typed
      ("Yet Another Name" :typein-name? nil)))) ; cannot be typed
```


Keyword and Default	Description
:property-list Default: nil	A property list for storing command information that does not fit anywhere else. It is intended for the programmer's use. This property list is provided by <code>sys:property-list-mixin</code> , which gives the associated methods <code>:get</code> , <code>:putprop</code> , and <code>:remprop</code> to the <code>ucl:command</code> instance. Information stored in this property list must include only objects such as lists, strings, atoms, and fixnums, not structures, arrays, or flavor instances. This limitation is required because the data fields of commands are saved in source form when the user saves the command environment.
:tutorial Default: nil	A form whose value is a string that contains a brief tutorial on the use of this command. Users can access this command tutorial from the UCL Command Display.
:visible? Default: t	A flag indicating whether the command is to be shown in UCL command menus and the Command Display. It allows your application to have commands that are active but not visible to the user.

:arguments Keyword
of `defcommand`

7.3.2 The UCL gives the programmer many different ways to specify exactly how a command obtains its arguments. Basically, the programmer can decide which of the command's argument values are *coded in* and which are given by the user interactively. The two different types of arguments are thus referred to as *program-supplied* and *user-supplied*. Furthermore, the UCL provides the user two different methods of entering a value for a user-supplied argument. A user-supplied argument can be typed after typing the command name (see paragraph 6.5.2, Obtaining Arguments). Otherwise, executing a UCL command automatically pops up a `choose-variable-values` window that allows the user to input a value for any user-supplied argument not yet typed.

The `:arguments` keyword specifies, for each argument of the command's function or method, how to determine the argument's value when the command is executed. The `:arguments` option must be given unless the function or method has no arguments. Its value is a list that contains an entry for each argument to the command, in calling-sequence order. Entries in the `:arguments` list can be any of the following:

- *argument form* — A *form* that is evaluated each time the command executes, binding the value of the form to the command argument. Such a form is used for all program-supplied arguments.
- `:label keyword-value` — The `:label` keyword indicates that the following entry in the `:arguments` list is a string. This string is the label that appears at the top of a `choose-variable-values` window used to input user-supplied arguments.
- `:user-supplied` keyword — The `:user-supplied` keyword indicates that subsequent entries are to be interpreted as user-supplied argument descriptors. See the subsequent description.
- `:program-supplied` keyword — The `:program-supplied` keyword turns off user-supplied arguments and indicates that subsequent entries are to be interpreted as argument forms, as described previously.

- *user-supplied argument descriptor* — A user-supplied argument descriptor defines a choose-variable-values item that determines how an argument is presented and how its value is to be input from a choose-variable-values window. A user-supplied argument descriptor can be either of the following:
 - String — A string defines a nonselectable choose-variable-values item. This type of item is useful as a heading that separates other items.
 - List of keyword-value pairs — This kind of descriptor is used for each user-supplied command argument. The keywords available include a subset of those used for a general choose-variable-values item, plus others unique to `:arguments`, as follows:
 - `:label string` — The value is a string that is displayed as the name of the argument in the choose-variable-values window. This keyword is required.
 - `:type type-specifier` — The value is any valid choose-variable-values item type specifier, such as `:number`, `:string`, `:pathname`, and so on. (Refer to the Choice Facilities section in the *Explorer Window System Reference*.) The default item type is `:sexp`.
 - `:default argument-form` — The value is a form that is evaluated and displayed as the default argument value. This value is assigned to the argument if it is not modified interactively by the user. The default value for `:default` is `nil`.

Note that the `:arguments` list can freely alternate between user-supplied and program-supplied argument entries by using the `:user-supplied` and `:program-supplied` keywords. Arguments up to the first `:user-supplied` keyword are assumed to be program-supplied.

The following example demonstrates the use of the `:arguments` keyword:

```
(defcommand a-command (arg1 arg2 arg3 arg4 arg5 arg6)
  `(:arguments
    (3 ; A program-supplied form
      (list 'a 'list) ; A program-supplied form
      :label "Enter three arguments:" ; A label for CVV
      :user-supplied ; Here begin user-supplied arguments
      "" ; Just a nonselectable string
      (:label "A number"
        :type (:documentation "A user-supplied number" :number)
        :default 42) ; A user-supplied number
      "" ; Just a nonselectable string
      (:label "A sexp"
        :default (list 'a 'sexp)) ; A user-supplied sexp
      :program-supplied ; Here resume program-supplied arguments
      ^argument-5 ; A program-supplied form
      :user-supplied ; Here resume user-supplied arguments
      "" ; Just a nonselectable string
      (:label "A pathname"
        :type (:documentation "A user-supplied pathname":pathname)
        :default "lm:-") ; A user-supplied pathname
    ))
  (format t
    "-%Argument 1: -a-%Argument 2: -a-%Argument 3: -a"
    arg1 arg2 arg3)
  (format t
    "-%Argument 4: -a-%Argument 5: -a-%Argument 6: -a"
    arg4 arg5 arg6)
)
```

Assuming a-command belongs to an active command table, typing a-command and pressing RETURN causes the following choose-variable-values window to appear:

```

Enter three arguments:
A number: ... 42
A sexp: ..... (A SEXP)
A pathname: lm:~
Abort  Do It  Help 

```

If the user immediately clicks on the Do It margin choice, the result of the command is as follows:

```

Argument 1: 3
Argument 2: (A LIST)
Argument 3: 42
Argument 4: (A SEXP)
Argument 5: ARGUMENT-5
Argument 6: lm:~;

```

If the user types in a-command 5, clicks on the pathname argument and enters SYS:UCL;COMMAND.LISP, then clicks on Do It, the result of the command is as follows:

```

Argument 1: 3
Argument 2: (A LIST)
Argument 3: 5
Argument 4: (A SEXP)
Argument 5: ARGUMENT-5
Argument 6: SYS:UCL;COMMAND.LISP

```

A few other features that the UCL offers for inputting command arguments are worth mentioning here:

- **Aborting a command** — The choose-variable-values window for user-supplied arguments presents an Abort margin choice. Selecting Abort terminates command input immediately without executing the command.
- **Displaying command documentation** — The choose-variable-values window for user-supplied arguments presents a Help margin choice. Selecting Help displays command documentation in a view documentation window. This window also contains any other previously displayed documentation.
- **Numeric arguments with commands** — By default, numeric arguments instruct the UCL command interpreter to repeat the subsequent command. (Refer to paragraph 6.5.3, Numeric Arguments.) You can override this behavior by including the symbol **ucl:numeric-argument** among the program-supplied arguments of the command. This binds the value of the numeric argument to one of the command's arguments. The command function or method is then free to interpret the numeric argument in its own fashion. For example, this technique is used by the input editor's Yank Kill History command (CTRL-Y) to allow CTRL-5 CTRL-Y to yank the fifth kill entry.

make-command Macro 7.3.3 The following describes the **make-command** macro.

make-command *defname keyword-values* Macro

Defines a command and returns a command object instance.

The **make-command** macro is essentially the same as **defcommand**. The difference is that, for a command created by **make-command**, the command code is given by a function/method defined elsewhere by a **defun/defmethod** form. This means that **make-command** is particularly useful for integrating an existing application with the UCL. The **make-command** macro also permits several commands to share the same code (perhaps with different arguments).

Arguments: *defname* — The same as the *defname* argument for **defcommand** (refer to paragraph 7.3.1, **defcommand** Macro).

keyword-values — Any of the **defcommand** *keyword-value* pairs are allowed. An additional keyword, **:definition**, is also available.

:definition — The name of the function or method that implements the command code. The value is either a symbol that is a function name or a method specification list of the form (*application-command-loop-flavor-name method-name*). If you omit this option, the function or method given by *defname* is used. The default is *defname*.

The following example illustrates the use of **make-command**:

```
(make-command hello
  (:documentation "Prints the string \"Hello!\"")
  :definition      print
  :keys            #\meta-control-h
  :arguments       ("Hello!")
))

(make-command ciao
  (:documentation "Prints the string \"Ciao!\"")
  :definition      print
  :keys            #\meta-control-c
  :arguments       ("Ciao!")
))

(make-command (ucl:basic-command-loop :quit)
  (:keys           #\meta-control-q
  ))
```

build-command-table 7.4

The following describes the **build-command-table** function.
Function

build-command-table *command-table-name* *flavor* *defnames* &key (:init-options nil) Function

Creates a command table that contains the given list of commands. The *command-table-name* symbol is bound to a command table instance. By default, an instance of the **ucl:command-table** flavor is created, although you can use initialization options to create a different flavor of command table.

Most applications require only one command table. Some applications use several command tables to represent different application contexts. If you have an application with a large number of commands, you may find that organizing the commands as a set of several command tables makes the command structure more comprehensible to the user. Note that the Command Display groups commands by command table and allows the user to request command table documentation.

Arguments: *command-table-name* — A symbol that is bound to the command table instance. This symbol is used by the UCL command interpreter to refer to the command table (see **ucl:active-command-tables** and **ucl:all-command-tables** in paragraph 7.7.1, **ucl:basic-command-loop** Flavor).

flavor — The name of your application command-loop flavor. Because this argument is used only to complete abbreviated method names (see the description of *defnames*), the *flavor* can be nil if *defnames* does not contain any abbreviated method names.

defnames — A list that represents the commands in the command table. Each entry in this list is one of the following types:

- A function name or logical command name that appears as the first argument in a **defcommand** or a **make-command**.
- A method specification of the form (:method *flavor method-name*), where (*flavor method-name*) appears as the first argument in a **defcommand** or a **make-command**.
- A method name, where (*flavor method-name*) appears as the first argument in a **defcommand** or a **make-command**. This type of entry is simply an abbreviated form of the previous type of entry.

:init-options — A list of *keyword-value* pairs used to initialize various attributes of the command table instance. This list can contain any of the following keywords.

Keyword	Description
:name <i>string</i>	The name displayed for the command table in the Command Display window. The default value is "Command Table".
:documentation <i>string</i>	A string containing complete documentation of the command table. A user can view this string by clicking on the command table name in the Command Display. This string typically consists of a general overview of the command table and its purpose. The default value is nil.

Keyword	Description
<code>:visible?</code> <i>boolean</i>	Used to make the command table invisible. The commands of the command table are displayed in the Command Display, unless the value of this keyword is <code>nil</code> . Note that this keyword has no effect on whether the command table is active or not. The default value is <code>t</code> .
<code>:editable?</code> <i>boolean</i>	Used to protect the command table from user editing. The commands of the command table are displayed in the Command Editor, where they can be modified by a user interactively, unless the value of this keyword is <code>nil</code> . The default value is <code>t</code> .
<code>:command-table-flavor</code> <i>flavor</i>	Used to customize the flavor of the command table instance created. The value is the name of an application flavor, which should mix in <code>ucl:command-table</code> . The default value is <code>ucl:command-table</code> .
<code>:name-table</code> <i>array</i> <code>:key-table</code> <i>array</i> <code>:commands</code> <i>array</i> <code>:name-lookup-fun</code> <i>function</i> <code>:key-lookup-fun</code> <i>function</i> <code>:command-lookup-fun</code> <i>function</i> <code>:table-sorts</code> <i>sort-functions</i>	These options can be given to initialize the command table's instance variables that are used to look up command names and keystrokes. For most applications, there is no reason to change the default values of these instance variables. However, they are available if you want to implement special search hash tables and search functions. For more information, see the source code for the <code>ucl:command-table</code> flavor.

The following example illustrates the use of the `build-command-table` function. Refer to the sample application example given at the beginning of this chapter.

```
;; Define two sound-demo methods.

(defmethod (sound-demo :print-active-command-tables) ()
  (format t "~%The active command tables for -a are -a"
    self ucl:active-command-tables)
  )

(defmethod (sound-demo :help) ()
  (eval (send self :basic-help))
  )

;; Define commands for the previous methods.

(make-command (sound-demo :print-active-command-tables)
  `(:documentation "Prints a list of the active sound demo command
    tables"
    :keys          #\control-t
  ))

(make-command (sound-demo :help)
  `(:documentation "Displays help for the sound demo"
    :keys          #\control-h
  ))
```

```
;; Build a command table for the sound demo.

(build-command-table 'sound-commands 'sound-demo
  '(display-sounds-menu
    beep-command
    doorbell-command
    ricochet-command
    (:method sound-demo :help)
    :print-active-command-tables
    quit
  )
  :init-options
  '((:name "Sound Demo Commands"
    :documentation "This is a table of commands used to demonstrate
the various sounds that can be produced using
the BEEP function."
  )))
```

build-menu Function

7.5 The following describes the `build-menu` function.

`build-menu` *menu-name* *flavor* &key (:sort-items nil) Function
 (:item-list-order nil) (:default-item-options nil)
 (:item-form ucl:general-parse-menu-item) (:sort-columns nil)
 (:column-list-order nil) (:temporary? nil) (:superior-menus nil)
 (:documentation nil)

Constructs a list of menu items for a command menu. Refer to the Choice Facilities section of the *Explorer Window System Reference* for a complete discussion of menu items. Entries in the list given by the `:item-list-order` keyword argument describe how each command is displayed as a menu item. The `build-menu` function uses this and other keyword arguments to construct a menu item list and to bind it as the value of the *menu-name* symbol.

The *menu-name* symbol is used by the UCL command interpreter and other UCL functions whenever the menu item list that this symbol contains is needed to define and present the command menu. For example, the `ucl:pop-up-command-menu` function (refer to paragraph 7.8, Miscellaneous Functions) creates and displays a pop-up command menu by using a menu item list built by `build-menu`. The UCL command interpreter looks for such *menu-name* symbols in its `ucl:menu-panes` instance variable (refer to paragraph 7.7.1.1, Basic Instance Variables) when updating its command menus to reflect changes in the command environment (for example, changes made interactively by the user from the Command Editor).

The `build-menu` function is closely related to the `:menus` option of `defcommand` and `make-command`. The `:menus` option in a command definition adds the command as an item in a command menu; the final menu item list includes any such items, together with those given by the `build-menu` `:item-list-order` (if any). Most of the `build-menu` keyword options apply to all entries in the menu item list, to those given by command definitions as well as those given by `:item-list-order`.

You may prefer to develop your application command menus using the various choice facilities of the window system. However, there are some advantages in using `build-menu`. You must use `build-menu` if command menus are to be updated dynamically by the UCL. Many of the details of specifying a command menu item (such as providing useful mouse documentation) can be done by default with `build-menu`. Furthermore, using `build-menu` defaults provides a consistent interface style among your UCL applications and system utilities such as the Inspector.

- Arguments:** *menu-name* — A symbol that is to be bound to a command menu item list. The UCL command interpreter also uses the property list of this symbol to record information needed to update the command menu during execution.
- flavor* — The name of the application command-loop flavor. Because this is needed only to complete abbreviated method names in the `:item-list-order` argument (described next), it can be `nil` if no abbreviated method names are used.
- Keywords:** `:item-list-order` — An ordered list that represents the commands and special menu items displayed in the command menu. Each entry in this list is one of the following types:
- **Command** — A command entry has the same form as entries in the *defnames* argument for `build-command-table` (refer to paragraph 7.4, `build-command-table` Function). That is, it can be a `defcommand/make-command defname`, a method specification list, or an abbreviated method name.
 - **Command and keywords** — An entry can consist of a list containing a command entry, as described previously, followed by a sequence of *keyword-value* pairs. The keywords can be any of those allowed for the `:menus` option of `defcommand` (refer to paragraph 7.3.1, `defcommand` Macro).
 - **String** — A string entry is displayed as a nonselectable menu item. This type of item is useful as a heading or to add blank space between other items. A nonselectable string entry can also be a list of the form (*string keyword arg ...*), where the keywords are either of the following:
 - `:font font-spec`
 - `:documentation string`
 - **Special menu item** — A special menu item entry is a list of the form (*string keyword arg keyword arg ...*). You can use this kind of entry to specify any of the usual menu items described in the Choice Facilities section of the *Explorer Window System Reference*.
- `:column-list-order` — A list that defines the order of columns in a multi-column command menu. The command menu is multicolumn if at least one entry in the menu item list specifies a `:column` keyword (refer to the `:menus` option of `defcommand` in paragraph 7.3.1, `defcommand` Macro). Each entry in the `:column-list-order` list is either a column name string or a list containing a column name string followed by keyword-value pairs. The keywords can be `:font` or any of the other keywords allowed in a column specification list (refer to the Choice Facilities section of the *Explorer Window System Reference*). The default value of `:column-list-order` is `nil`.
- `:default-item-options` — A list of menu item keyword-value pairs that are used as defaults for all items in the command menu. The keywords can be any of those allowed in the `:menus` options of `defcommand`. For any menu item, these default options are overridden by the options specified by the entry in the menu item list. The default value is `nil`.
- `:documentation` — A string used as the mouse documentation for the command menu's entry in a superior menu. See the subsequent description of `:superior-menus`. The default value is `nil`.

:item-form — A symbol that is the function used to transform an **:item-list-order** list entry into a menu item understood by a **w:menu** instance (refer to the Choice Facilities section of the *Explorer Window System Reference*). The default function, **ucl:general-parse-menu-item**, is preferred for most applications because it automatically creates the item's mouse documentation string containing the command description and keystrokes. If you decide to provide an alternative function, model its interface and behavior on that of **ucl:general-parse-menu-item**.

:sort-columns — Used to control the order of columns in a multicolumn command menu. See **:column-list-order** described previously. The value of **:sort-columns** determines how columns are reordered after the menu is modified during execution. The value is a form that can be one of the following:

- **nil** — This value (the default) indicates that no sorting is done and that the order of columns depends on the **:column-list-order** (if given), the order of entries in **:item-list-order**, and the order in which commands are defined.
- **t** — The columns are sorted in alphabetical order.
- Function or lambda expression — A sort function that accepts an association list argument and returns a sorted association list. The car of each element of the association list argument is a column name string.

:sort-items — Used to control the order of items in the command menu. The value is a form that can be one of the following:

- **nil** — This value (the default) indicates that no sorting is done and that the order of items depends on the order of entries in **:item-list-order** and the order in which commands are defined.
- **t** — The items are sorted in alphabetical order.
- Function or lambda expression — A sort function that accepts a menu item list argument and returns a sorted menu item list.

:superior-menus — A list of menu name symbols. This option causes an item that represents the whole command menu to be constructed and inserted in the item list of each menu in the **:superior-menus** list. The default value is **nil**. (This keyword is similar to the *superior-menu* argument of the **sugg:suggestions-build-menu** function; refer to paragraph 9.3.)

:temporary? — If the value is non-**nil**, the command menu is considered to be temporary and is never updated as a result of changes to the command environment. A temporary command menu is typically used only as an argument to the **ucl:pop-up-command-menu** function. The default value is **nil**.

The following examples illustrate the use of the **build-menu** function. Refer to the sample UCL application used in previous examples.

:: Build a single-column command menu with several groups of commands.

```
(build-menu `sound-command-menu `sound-demo
: item-list-order
  `(("Make a Sound" :font fonts:tr12b) ; Nonselectable string entry

    beep-command ; Command entries
    doorbell-command
    ricochet-command

    "" ; Nonselectable string entry
    ("Sound Demo Info" :font fonts:tr12b) ; Nonselectable string entry

    :help ; Command entries
    (:method sound-demo :print-active-command-tables)

    "" ; Nonselectable string entry
    (quit :font fonts:tr12b) ; Command and keyword entry
  )

: default-item-options
  `(:font fonts:h112)

: temporary? t
)
```

:: Build a multicolumn version of the same command menu.

```
(build-menu `sound-command-menu `sound-demo
: item-list-order
  `((:help :column "Sound Demo Info") ; Command and keyword
    ; entries
    (:print-active-command-tables
      :column "Sound Demo Info")

    (quit :font fonts:tr12b :column "")

    (doorbell-command :column "Make a Sound")
    (beep-command :column "Make a Sound")
    (ricochet-command :column "Make a Sound")
  )

: column-list-order ; Sort columns left-to-right.
  `(("Make a Sound" :font fonts:tr12b )
    ("Sound Demo Info" :font fonts:tr12b)
    ""
  )

: sort-items t ; Sort items in each column.

: default-item-options
  `(:font fonts:h112)
)
```

Hints for Developing a UCL Application

7.6 The following list provides hints for developing a UCL application.

- Initializing the command interpreter — By default, a UCL command interpreter has no active command tables when created. Before the command interpreter can be truly effective, you must initialize two of its instance variables: `ucl:active-command-tables` and `ucl:all-command-tables`. Refer to paragraph 7.7.1.1, Basic Instance Variables, for a complete discussion of these instance variables of the `ucl:basic-command-loop` flavor. Typically, these instance variable are initialized when the command interpreter is first instantiated (refer to the sample application shown at the beginning of this chapter).
- Starting and stopping the command interpreter — Use the `:command-loop` and `:quit` methods of the `ucl:basic-command-loop` flavor to enter and exit the UCL command loop. A command interpreter flavor that mixes in `ucl:command-loop-mixin` has slightly different ways of beginning and exiting. Refer to paragraph 7.7.2, `ucl:command-loop-mixin` Flavor.
- When command definitions change — Typically, during the development of a UCL application, many changes are made to the `defcommand` and `make-command` calls that define application commands. It is not unusual, for example, to spend a lot of time fine-tuning the `:names` or `:keys` in command definitions. A common programmer mistake is to forget to update application command tables and command menus after a command definition has been changed and reevaluated. The new command definition is not reflected in the application until all command tables and command menus that contain the command are redefined by reevaluating the appropriate `build-command-table` and `build-menu` calls. The correct sequence is first to reevaluate the changed `defcommand` and `make-command` calls, and then to reevaluate the relevant `build-command-table` and `build-menu` calls.

Using and Customizing Command Interpreter Flavors

7.7 An instance of the UCL command interpreter flavors is an object that operates on the data structures defined by `defcommand`, `make-command`, `build-command-table`, and `build-menu` in order to read and execute application commands. The UCL command interpreter is available in two flavors:

- `ucl:basic-command-loop`
- `ucl:command-loop-mixin`

Typically, a UCL application combines one of these flavors with a window flavor in order to create a command interpreter with a window stream for input/output.

NOTE: These UCL flavors do not specify a window, so your program must specify the window mixins you need.

The `ucl:basic-command-loop` flavor is the base flavor. It provides your application with a `:command-loop` method that starts the command interpreter. The second flavor, `ucl:command-loop-mixin`, is built upon the base flavor. In addition to the `:command-loop` method, `ucl:command-loop-mixin` provides a process (`w:process-mixin`), which is set to run the `:command-loop` method when your application window is selected. If you want a process, use `ucl:command-loop-mixin`; most applications that have a window use this mixin. Utilities such as `break` and the debugger, which do not have their own processes, use `ucl:basic-command-loop`.

The paragraphs that follow discuss all of the UCL features for creating and customizing your application command interpreter and its flavors. The following topics are included:

- The UCL command-loop flavors, with their instance variables and methods
- Other useful flavors for type-in windows and typed expression processors
- Global UCL variables for command history and interpreter defaults

`ucl:basic-command-loop` Flavor

7.7.1 This paragraph describes the `ucl:basic-command-loop` flavor and its instance variables and methods.

`ucl:basic-command-loop`

Flavor

Provides the `:command-loop` method. You should use this flavor for applications that do not require a process. This flavor provides a set of instance variables that allow you to control and monitor the operation of the command interpreter. These instance variables are all declared *special*. You can examine or modify the variables in any part of your code during execution of the interpreter. All of these instance variables are inittable, and your application flavor can supply the inittable values in its `:default-init-plist`.

Basic Instance Variables 7.7.1.1 This paragraph describes the instance variables that are usually important to all applications.

ucl:active-command-tables Instance Variable of **ucl:basic-command-loop**

A list of command table symbols. The value of each symbol in this list is a command table instance built by **build-command-table**. Commands in these tables are *active* and can be executed by keystrokes, mouse clicks, command menus, and typed command names. You *must* initialize this list before the command loop starts; if this list is *nil*, none of the commands are executable. Send **:set-active-command-tables** to your application command loop to change the list of active command tables for the current command context.

ucl:all-command-tables Instance Variable of **ucl:basic-command-loop**

A list of all command table symbols used in your application. The UCL uses this variable to access all the commands in your application for the various help and customization features. These features include the Command Editor, Command Name Search, and Keystroke Search universal commands. You *must* initialize this variable but do not modify it during execution of your program.

ucl:basic-help Instance Variable of **ucl:basic-command-loop**

A form that displays help information for new users of your application. If this variable is *non-nil*, the UCL Help command menu contains an item that represents basic help for your application. When a user selects this item, the help form is evaluated. Note that if this variable is *non-nil*, your application command-loop flavor must provide a **:name** method. The UCL Help command uses the string returned by this **:name** method to construct the application's help menu item.

ucl:tutorial Instance Variable of **ucl:basic-command-loop**

A form that is evaluated to execute a tutorial on your application. If you supply a form, the UCL Help command menu includes a Tutorial option. When the user selects the option, the system evaluates the form and runs the tutorial.

ucl:menu-panes Instance Variable of **ucl:basic-command-loop**

A list that specifies the application constraint-frame panes that are used for command menus or that need to be updated when the command environment changes. This variable is irrelevant if your application does not use constraint frames. Each entry in the list is one of the following types:

- Pane name — A symbol that appears in the **:panes** initialization option of your application constraint-frame flavor. Refer to the Frames section in the *Explorer Window System Reference*.
- Pane instance — A window flavor instance for a pane.
- Menu instance — A menu flavor instance. Refer to the Choice Facilities section in the *Explorer Window System Reference*.
- Menu pane specification list — A list of the form (*pane-specification menu-symbol*), where *pane-specification* is an entry of one of the preceding types and *menu-symbol* is a symbol that has been used as the first argument in a call to **build-menu**.

Any menu instance or pane mentioned in this list must be of a flavor that has mixed in either `w:dynamic-item-list-mixin` or `w:dynamic-multicolumn-mixin`. This allows the pane to be updated during application execution.

The `ucl:menu-panes` variable provides a way for the UCL command interpreter to update a pane when a user modifies the command environment. For example, if a user changes a command name, then all command menus containing the command can be updated to exhibit the new name. This variable is also an easy way to get a command menu initially displayed in a pane.

ucl:input-mechanism Instance Variable of `ucl:basic-command-loop`

A flag indicating how a command was input. This variable can be useful in application commands that can be invoked by several different methods. Possible values are the following:

- `ucl:key-or-button` — Command was input by a keystroke, keystroke sequence, or mouse button that the user pressed.
- `ucl:typein` — Command was input by a typed command name.
- `ucl:menu` — Command was selected from a menu.
- `ucl:queue` — Command was obtained from the `ucl:command-execution-queue` instance variable.

ucl:command-entry Instance Variable of `ucl:basic-command-loop`

The result of translating the user's input into a command to execute. This instance variable is set to a command instance whenever the input translates to an active command, to a `ucl:typein-mode` instance when the user types a special expression (such as a Lisp form), or to `nil` if the input is unrecognizable (such as a mistyped keystroke).

ucl:kbd-input Instance Variable of `ucl:basic-command-loop`

Holds the first object read on each iteration of the command loop. This instance variable holds either a keystroke character or a blip (a list). In the case of typed command names, it holds only the first keystroke typed. A typical use for `ucl:kbd-input` is an application that has commands invoked by pressing mouse buttons. These commands can reference `ucl:kbd-input` to determine the coordinates at which the user clicked the mouse (mouse blips hold this information) and can carry out their operation based on the coordinates.

ucl:blip-alist Instance Variable of `ucl:basic-command-loop`

An association list that controls the processing of blips that are input at the top level of the interpreter. The association list consists of entries of the form (*blip-type method-name*), where *blip-type* is the car of the blip and *method-name* is a command interpreter method to call to process the blip. (The blip is stored in instance variable `ucl:kbd-input` documented previously.)

The default value is the following list:

```
((:menu :handle-menu-input)
 (:mouse-button :handle-mouse-input)
 (:direct-command-entry :handle-direct-command-input))
```

If you need to process another type of blip, add it to `ucl:blip-alist`. The default list is necessary to process UCL menu and mouse input.

ucl:command-history Instance Variable of **ucl:basic-command-loop**

A circular list of previously executed commands. The car holds the most recent command. Each entry in this list is either a **ucl:command** instance or a list whose car is a **ucl:command** instance and whose cdr contains the arguments used in executing the command function or method. These entries correspond directly to the **:arguments** field of **ucl:command**.

ucl:max-command-history Instance Variable of **ucl:basic-command-loop**

An integer indicating the number of entries maintained in **ucl:command-history**. If **ucl:max-command-history** is **nil**, the *user-customizable* global variable **ucl:*default-max-command-history*** is used instead.

ucl:command-execution-queue Instance Variable of **ucl:basic-command-loop**

A list of commands to be executed before more input is read. The car is the next **ucl:command** to execute. Command macros use this variable to queue a sequence of commands for execution. You can use it in your applications to execute a batch of commands.

ucl:numeric-argument Instance Variable of **ucl:basic-command-loop**

An integer specifying a numeric argument that the user provided before inputting the current command. This instance variable is used internally to repeat commands and to execute a command that processes its numeric argument in some special way. (Refer to paragraph 7.3.2, **:arguments** Keyword of **defcommand**.)

*Instance Variables
for Reading
Typed Input*

7.7.1.2 This paragraph describes the instance variables that control the reading of Lisp expressions and other typed input.

ucl:typein-modes Instance Variable of **ucl:basic-command-loop**

A list of type-in modes to use for input editor completion and interpretation of typed expressions. When this instance variable is **nil** (the default), the user-customizable global variable **ucl:*default-typein-modes*** is used instead. The default list permits interpretation and completion of typed command names and Lisp forms (refer to paragraph 6.5.1, Typed Expressions).

ucl:typein-handler Instance Variable of **ucl:basic-command-loop**

A method to handle reading of expressions typed by the user. The following values can be used:

- **:handle-pop-up-typein-and-typeout** — Reads the expression using a pop-up window. If a command name is typed, the command executes and the window is buried. If any other type of expression (such as a Lisp form) is typed, the value is displayed in the window and another typed expression is read. This is the default type-in handler for the UCL because it interferes least with application windows.
- **:handle-pop-up-typein-input** — Works like **:handle-pop-up-typein-and-typeout** but does not remain exposed after a Lisp form is typed. The value of the expression is printed on the stream value of the variable ***standard-output***.
- **:handle-typein-input** — Reads the expression from ***standard-input***. This handler is recommended for applications that have room for a type-in/typeout pane.

- **nil** — Deactivates the reading of typed expressions altogether. For small applications that display all available commands in a command menu, this method is acceptable because the commands are always available to the user without the user typing them.

Only **:handle-typein-input** causes the interpreter to print a prompt after each expression read (for example, in the debugger or Lisp Listener). Other type-in handlers are responsible for printing their own prompt. If you need to modify how **:handle-typein-input** works but want the interpreter to prompt after typed expressions, redefine it instead of using your own method.

ucl:read-function Instance Variable of **ucl:basic-command-loop**

The function to use for reading typed expressions (command names and Lisp forms or whatever is specified in **ucl:typein-modes**). When this instance variable is **nil** (the default), the user-customizable global variable **ucl:*default-read-function*** is used instead.

ucl:prompt Instance Variable of **ucl:basic-command-loop**

A form whose value is a string that is used to prompt the user for typed expressions. When this instance variable is **nil** (the default), the user-customizable global variable **ucl:*default-prompt*** is used instead.

ucl:read-type Instance Variable of **ucl:basic-command-loop**

Whatever **ucl:read-function** returns as a second value. This second value typically indicates something about the typed expression. For example, the default read function sets **ucl:read-type** to **:atom**, **:cons**, or **:implicit-list**. The **ucl:typein-modes** instance variable uses this flag to determine how to interpret the typed expression. If you design your own special read function and type-in modes to read and process expressions of various syntax, **ucl:read-type** might be useful.

Instance Variables for Printing 7.7.1.3 This paragraph describes the instance variables that control the printing done by the command interpreter.

ucl:print-results? Instance Variable of **ucl:basic-command-loop**

A function that takes no arguments and returns a Boolean value indicating whether to print the results of an executed command or type-in mode (such as a typed Lisp function call). The results of a command are the values returned by its function or method. You can set this instance to **nil** to inhibit printing.

The default value of this variable is **ucl:abnormal-command**. This is a predicate that allows the printing of the results of typed expressions, without printing the values returned from executed commands. For most applications, command-function or method returned values are of no interest.

ucl:print-function Instance Variable of **ucl:basic-command-loop**

A function to output results of each executed command. If the value is **nil** (the default), the user-customizable global variable **ucl:*default-print-function*** (described in paragraph 7.7.4.2, Default Variables) is used instead.

ucl:inhibit-results-print? Instance Variable of **ucl:basic-command-loop**

Lets you disable printing the results of a command when you set this variable to **t**. UCL resets it to **nil** before the next command is executed so that printing of the results is enabled again. For some applications, it is useful to print the values of only certain commands.

ucl:output-history Instance Variable of **ucl:basic-command-loop**

A circular list of the values returned by typed Lisp forms. The list's **car** is the most recent output.

ucl:max-output-history Instance Variable of **ucl:basic-command-loop**

An integer that specifies the number of output values to retain in the output history. If the value is **nil** (the default), the user-customizable global variable **ucl:*default-max-output-history*** is used instead.

Methods 7.7.1.4 The UCL command interpreter is implemented as a set of methods of **ucl:basic-command-loop** to provide you with an easy way to make customizations for your applications. The flavor system allows you to define daemons and wrappers to modify command-loop methods and to redefine methods for your flavor. To design command-loop customizations, you should examine the source code for the **ucl:basic-command-loop** flavor and make similar adjustments. The file **SYS:UCL;STARTER-KIT.LISP** contains examples of this type of customization.

The command loop starts when the **:command-loop** method is sent to your application. Other methods are called as input is read and processed. The following descriptions explain the task performed by each of these methods.

:command-loop Method of **ucl:basic-command-loop**

Drives the command interpreter. It locally binds certain variables, such as ***terminal-io*** and *****.

:initialize Method of **ucl:basic-command-loop**

The first method that **:command-loop** calls before starting to interpret commands. It does various chores that cannot be done in an **:after :init** method, such as calling **:designate-io-streams**.

:designate-io-streams Method of **ucl:basic-command-loop**

Sets ***standard-input*** and ***standard-output*** to **self**. The various command-loop methods read input from ***standard-input*** and print results to ***standard-output***. Your application flavor can redefine this method to set these variables or the other I/O stream variables ***terminal-io***, ***error-output***, and ***debug-io*** to appropriate window panes of your constraint frame.

:loop Method of **ucl:basic-command-loop**

Executes a loop that obtains input, translates the input into a command, and executes the command. It performs error catching (as described in paragraph 6.5.6, Error Catcher) and sets up a catch routine to handle abort operations. Applications can redefine this method to specify an alternate looping behavior. For example, the Lisp Listener flavor redefines it to remove the error-catching feature.

This method is also useful for creating a temporary or subordinate command loop within your application top-level loop. For example, a command might execute by switching the active command tables and calling `:loop` to process the new command context until an exiting command is input. (This is similar to the Incremental Search command in Zmacs.) The file `SYS:UCL;STARTER-KIT.LISP` shows an example of this method.

:fetch-and-execute Method of `ucl:basic-command-loop`

Obtains (fetches) and executes one command. The method is called inside the loop performed by the previously described `:loop` method. You can also use it to cause one command to obtain and execute another command. The file `SYS:UCL;STARTER-KIT.LISP` shows an example of this method.

:fetch-input Method of `ucl:basic-command-loop`

Reads the first object from `*standard-input*` and stores it in the `ucl:kbd-input` instance variable. You can tailor this method to process or test the object for a particular condition or to read the object in a special way. For example, an application can use `:any-tyi-no-hang` instead of `:any-tyi` so that a background computation can occur until the user types input.

:handle-menu-input Method of `ucl:basic-command-loop`

Processes menu blips. If execution of the menu blip returns a `ucl:command` instance, the command is executed.

:handle-mouse-input Method of `ucl:basic-command-loop`

Processes mouse blips. The mouse button signal is extracted from the blip and treated as a keystroke by calling the `:handle-key-input` method.

:handle-key-input Method of `ucl:basic-command-loop`

Processes a typed keystroke. The active command tables are searched for the keystroke. If a command matching the keystroke is found, it is executed. If a matching command is not found, if the keystroke contains no control bits, and if the instance variable `ucl:typein-handler` is not `nil`, then the method stored in `ucl:typein-handler` is called to read and process a typed expression. Otherwise, the `:handle-unknown-input` method is called.

During the keystroke search, other keystrokes are obtained as needed from the user if the active command tables contain multiple keystroke sequences. In any event, the list of obtained keystrokes is bound specially to the variable `ucl:key-sequence` so that applications can tailor the method `:handle-unknown-input` to display the mistyped keystrokes. The file `SYS:UCL;STARTER-KIT.LISP` shows an example of this method.

:handle-typein-input Method of `ucl:basic-command-loop`

:handle-pop-up-typein-input Method of `ucl:basic-command-loop`

:handle-pop-up-typein-and-typeout Method of `ucl:basic-command-loop`

These methods are available for use as the value of the instance variable `ucl:typein-handler`. They handle the reading and processing of typed expressions (including typed command names).

:execute-command Method of `ucl:basic-command-loop`

Sent by the various input handler methods to execute the input command. It causes the command to execute itself and perform other chores, such as collecting the returned values in the variable `/` and updating the output history.

:handle-unknown-input Method of **ucl:basic-command-loop**

Handles unrecognized keystrokes, mouse clicks, menu commands, and typed expressions. In the first three cases, it beeps; in the last case, it prints an error message on the stream value of **standard-input**. Your application can redefine this method to print an explanation instead of beeping.

:quit Method of **ucl:basic-command-loop**

Uses a **throw* routine to cause the command interpreter to exit the **:loop** method.

ucl:command-loop-mixin Flavor 7.7.2 This paragraph describes the **ucl:command-loop-mixin** flavor.

ucl:command-loop-mixin Flavor

A variant of the **ucl:basic-command-loop** flavor intended for an application that runs in its own process. Because it incorporates **ucl:basic-command-loop** in its flavor definition, all documentation on the instance variables and methods of **ucl:basic-command-loop** also applies to **ucl:command-loop-mixin**. This flavor provides the following additional initialization option and method.

:process-options list Initialization Option of **ucl:command-loop-mixin**

Specifies a list of keywords and arguments to use in creating the application process. The available keywords are those provided by the **make-process** function, described in the *Explorer Lisp Reference*. The keywords allow you to specify values such as the process priority and stack group size. For example, if your commands perform a large amount of computation and you find that the stack is overflowing, include the following in the **:default-init-plist**:

```
:process-options '(:regular-pdl-size 60000)
```

:quit Method of **ucl:command-loop-mixin**

Deselects and then buries the application window. It does not exit the command interpreter for **ucl:command-loop-mixin** because doing so would leave the process in a stopped state.

Other Flavors 7.7.3 The UCL also provides several other flavors for creating such useful objects as type-in windows and typed expression processors (type-in modes). This paragraph describes these additional flavors, their instance variables, and their methods:

- **ucl:command-and-lisp-typein-window** — Provides a helpful window for reading typed expressions
- **ucl:typein-mode** — Used to build custom processors of typed expressions
- **ucl:selective-features-mixin** — Allows you to turn off UCL features that are not appropriate for your application

ucl:command-and-lisp-typein-window Flavor 7.7.3.1 The following describes the **ucl:command-and-lisp-typein-window** flavor:

ucl:command-and-lisp-typein-window Flavor

The recommended command type-in window for the UCL. It provides help features in the mouse documentation window and auto-scrolling. Examples of programs that use this type of window are the Lisp Listener and the Relational Table Management System (RTMS) user interface input pane.

If your application flavor uses a simple window without multiple panes, simply mix **command-and-lisp-typein-window** into your application flavor. If the application window is defined as a constraint frame, do the following:

1. Include the **command-and-lisp-typein-window** flavor as one pane in the constraint frame.
2. Define a **:designate-io-streams** method for your application flavor that sets ***terminal-io*** to the pane.
3. Initialize your application **ucl:typein-handler** instance variable to the symbol **:handle-typein-input**. This symbol causes typed expressions to be read from ***terminal-io***.

Refer to the file **SYS:UCL; STARTER-KIT.LISP** for an example that uses **ucl:command-and-lisp-typein-window**.

ucl:typein-mode Flavor 7.7.3.2 You use the **ucl:typein-mode** flavor to define the processors of expressions entered from the keyboard.

The UCL interprets typed expressions by looking at a list of default **ucl:typein-mode** instances. The various modes handle typed command names, Lisp function calls, Lisp symbol evaluations, implicit messages, and completion on pathnames (refer to paragraph 6.5.1, Typed Expressions). The instance variable **ucl:typein-modes** allows you to specify a list of modes that your application can accept, instead of the default modes.

The following are suggested uses of customized type-in modes:

- A type-in mode can be easily designed to process Lisp-like languages. (An example section in the UCL starter kit shows you how to define a mode to complete on and evaluate Scheme expressions.) You can design a type-in mode to process other languages, such as a query language for a database, a semantic network manipulator, or whatever.
- You can allow processing of other kinds of typed input besides command names and Lisp. For an application that requires frequent editing, you could design a type-in mode that lets the user type a buffer name to move to that buffer. For a production rule system, you could let the user type and use completion on rule names to examine the rules.

ucl:typein-mode

Flavor

A simple mixin used to define processors of typed expressions in the UCL command interpreter.

Required Methods: Your type-in mode flavor must define the following methods:

- **:handle-typein-p** *expression type* — The *type* argument is either **:atom**, **:cons**, or **:implicit-list**. An implicit list is an atom followed by one or more forms all typed on the same line. The method should return two values:
 - The value **self** if it wants to handle *expression*; otherwise, **nil**.
 - An error string if *expression* is probably a mistyped expression of the kind handled by your type-in mode; otherwise, **nil**.
- **:execute** *application* — The *application* argument is the currently running UCL command-loop instance. This method evaluates the current expression found in the global variable **-**.

The UCL interpreter processes typed expressions by sending **:handle-typein-p** to all type-in modes until a mode returns a non-**nil** first value and a **nil** second value; then the mode is told to execute the expression by sending it an **:execute** message. If all modes return a **nil** first value, the first non-**nil** error message returned by a mode is displayed for the user. If all modes return **nil** error messages, the message **** Unrecognized Input** is printed.

Optional Methods: Also, your type-in mode flavor can handle the following optional methods:

- **:complete-p** *syntax* — This form returns non-**nil** if completion can be performed on symbols having the syntax specified by *syntax*. The value returned is a short string describing the type of expression processed by this mode; it is used in the input editor List Completions command. The *syntax* argument can be any of the following:
 - **:first-function** — The first symbol in the expression, which is preceded by an opening parenthesis (().
 - **:first-atom** — The first symbol in the expression, which is not preceded by an opening parenthesis.
 - **:function** — Any symbol inside the expression (except the first) that is preceded by an opening parenthesis.
 - **:atom** — Any symbol inside the expression (except the first atom) that is not preceded by an opening parenthesis.

When the user enters an input editor completion command, **:complete-p** is sent to each mode instance of the current mode list until one returns non-**nil**. The **:complete** method for that instance is then sent to return the completions.

- **:complete word type** — This form returns a list of completions on the string *word*. The *type* argument specifies the style of completion to use and can be any of the following:
 - **:apropos** — Performs completion by matching any substring within a word.
 - **:recognition** — Performs completion by matching the first characters of each word.
 - **:spelling-corrected** — Performs completion by using a spelling correction algorithm.
- **:arglist symbol** — This form returns a string that displays the argument list for *symbol*, if appropriate. This string is used by various help features (such as mouse documentation window help) to display the argument lists of symbols.
- **:help-doc symbol** — This form returns the documentation string for *symbol*, if appropriate. This string is used by various help features (such as the input editor CTRL-SHIFT-D command) to display the long documentation for a symbol.

Instance Variables: The following are instance variables for **ucl:typein-mode**.

ucl:auto-complete-p Instance Variable of **ucl:typein-mode**

Controls auto-completion for your mode. If this instance variable is non-*nil*, the user can press the space bar to perform recognition completion on any symbol for which your **:complete-p** method returns *t*. Ordinarily, **ucl:auto-complete-p** is set to *nil* (the default) to allow expressions to be typed efficiently, but your flavor can set it to a non-*nil* value if your expressions complete quickly and you think auto-completion is appropriate.

ucl:description Instance Variable of **ucl:typein-mode**

A short (one-line) description of your type-in mode, used in the Top Level Configurer and for the Configure Type-In Modes commands. The default value is "".

ucl:documentation Instance Variable of **ucl:typein-mode**

A longer description of how your mode works. It is used for online help in the Top Level Configurer and for the Configure Type-In Modes commands. The default value is "".

After you define your own type-in mode, you can add it to the UCL by doing the following:

1. Set a symbol to an instance of your type-in mode.
2. Add the symbol to the list stored in the variable **ucl:*all-typein-modes***. This allows the user to activate your mode using the Top Level Configurer command.
3. (Optional) If you want your type-in mode to be in all UCL top levels (the Lisp Listener, break, the debugger, and so forth), push the symbol onto

the list stored in the user-customizable global variable `ucl:*default-typein-modes*`.

4. (Optional) If you want your type-in mode to be added only to your application, set the instance variable `ucl:typein-modes` to a list of whatever type-in mode symbols you have defined. For example, the following hypothetical example would enable the processing and completion of typed command names and Prolog expressions in your application:

```
(ucl:command-names ucl:prolog-expressions)
```

ucl:selective-features-mixin Flavor 7.7.3.3 The `ucl:selective-features-mixin` flavor allows you to omit UCL features you do not want for your program.

ucl:selective-features-mixin Flavor

This mixin is provided for use in application flavors that incorporate either `ucl:basic-command-loop` or `ucl:command-loop-mixin`. It gives you the capability to turn off UCL features that are not appropriate for your application. For example, if you feel that the Top Level Configurer command is too complex for your program users, you can remove it from your application by including this mixin in your code.

Not all UCL features can be turned off with this mixin; some are controlled through the instance variables of `ucl:basic-command-loop` and `ucl:command-loop-mixin`. (See paragraph 7.7.1, `ucl:basic-command-loop` Flavor, for further information.)

You can put this mixin anywhere in the list of component flavors. To turn off features, you must supply the `:remove-features` initialization keyword in the `:default-init-plist` for the application command-loop flavor definition because a given flavor always has the same set of features removed.

Note that one goal of the UCL is to provide uniform command interfaces across all Explorer software. No features *must* be removed; before removing a feature, consider whether the intended application user expects it to be there because other UCL interfaces provide the feature.

Example:

```
(deflavor foo ()
  (ucl:basic-command-loop ucl:selective-features-mixin)
  (:default-init-plist
   :remove-features `(:lisp-typein ucl:edit-command-tables))
```

The `:remove-features` initialization keyword accepts a list of features (represented by symbols) to remove. The following table lists the recognized symbols and describes the corresponding features.

Keyword	Description
:lisp-typein	Inhibits Lisp expression processing. The interpreter processes (and allows completion on) only typed command names.
:input-editor-commands	Deselects the input editor command table. This means that at the top level the user is not able to press CTRL-C to copy the previously typed expression, press CLEAR SCREEN to clear the screen, and so forth. The Command Display does not display the input editor commands. However, once the user starts typing an expression, the input editor commands are active. Typically, you turn off this feature for only two reasons: either you have designed your own input editor or you have designed a type-in handler that is activated by only one keystroke (such as the META-X prefix for Zmacs commands).
:all-universal-commands	Removes all the universal commands from your application. This removal is fairly drastic; it is the same as removing all of the features described next.
ucl:display-command-tables	Removes the command that displays all active command tables in a scroll window (the Command Display). You typically should not remove this feature.
ucl:display-previous-commands	Removes the command that displays previously executed commands (the Command History feature).
ucl:command-name-search	Removes the command that accepts a search string from the user and displays all commands whose names match the search string (the Command Name Search feature). You might remove this feature if your application does not include many commands.
ucl:keystroke-search	Removes the command that accepts a keystroke from the user and finds which command (or commands) has the keystroke (the Keystroke Search feature).
ucl:numeric-argument	Removes the command that accepts a numeric argument from the user.
ucl:redo-command	Removes the command that executes the previous command with the same user-supplied arguments (the redo command feature). This command is strongly application-dependent. For example, this command is useless for the graphics editor because commands never obtain arguments from the user by using the :arguments feature of defcommand . However, this command is useful for the RTMS user interface because <i>most</i> of the commands query the user for arguments. For information about the obtaining of arguments, read the documentation on :arguments under defcommand .
ucl:edit-command-tables	Removes the command that allows the user to modify command data fields, such as names and keystrokes (the Command Editor command). It is preferable to let the user modify the command environment, especially by adding or modifying the command keystrokes. However, if your users are naive or your application has only a few commands (such as in Peek), you might remove this feature.

Keyword	Description
ucl:build-command-macro	Removes the command that allows the user to build command macros that execute a sequence of other commands (the Build Command Macro command). The usefulness of command macros is application dependent. Notice that the Build Command Macro command is provided (in addition to the Build Keystroke Macro command) because commands are not always bound to keystrokes.
ucl:build-keystroke-macro	Removes the command that allows the user to build keystroke macros that force a series of keystrokes into the input buffer. The usefulness of keystroke macros is application dependent.
ucl:load-commands	Removes the command that allows the user to load command macros, keystroke macros, and user-modified commands (using the Command Editor) that were saved during a previous session. Remove this feature if and only if you remove the ucl:edit-command-tables , ucl:build-command-macro , and ucl:build-keystroke-macro features.
ucl:save-commands	Removes the command that allows the user to save any modified commands (and macro commands) for loading in subsequent sessions. Remove this feature if and only if you remove the feature ucl:load-commands .
ucl:configure-typein-modes	Removes the command that allows the user to rearrange the type-in modes that process expressions entered from the keyboard and provide completion (the Configure Type-In Modes command). Remove this feature if you set the ucl:typein-handler instance variable to <code>nil</code> or if you have removed the <code>:lisp-typein</code> feature. However, you can optionally leave this feature in your program because its only action is to beep if your application does not process typed expressions and provide completion.
ucl:top-level-configurer	Removes the command that allows the user to configure the processing of typed expressions (the Top Level Configurer command). Remove this feature if and only if you remove the ucl:configure-typein-modes feature. However, you can optionally leave this feature in your program because its only action is to beep if your application does not process typed expressions.
ucl:help-menu	Removes the command that executes when the user presses the HELP key (the Help command). It is advisable not to remove this feature because it lets the user access all the UCL help and environment customization commands. If you do remove it, be sure that your Help command points the user to the various UCL help and environment customization commands. Your help feature might do this by calling the ucl:help-menu function at some point.
ucl:system-menu	Removes the command that brings up the System menu when the user clicks once on the rightmost mouse button. If your program does something else in response to input from the right mouse button, you can include this option to remove the System menu command from the Command Display.

Global Variables 7.7.4 This paragraph discusses the global variables for command history and interpreter defaults.

History Variables 7.7.4.1 The following symbols are global variables that store the most recently typed expressions and returned Lisp values.

Variable	Description
-	Holds the expression the user just typed
/	Holds a list of values returned by the executed command or Lisp expression
*	Holds the car of / for typed Lisp expressions
**	Holds the previous value of *
***	Holds the previous value of **
+	Holds the previous value of -
++	Holds the previous value of +
+++	Holds the previous value of ++

Default Variables 7.7.4.2 The UCL command interpreter uses the following global variables as defaults. You can override any of these variables by using the corresponding instance variables. For example, if an application has not specified a value for its **ucl:prompt** instance variable, the value of **ucl:*default-prompt*** is used to prompt the user. The user can customize all of these variables with the Top Level Configurer command. Thus, the user can customize the default command interface, yet the programmer can override these customizations when required.

ucl:*default-typein-modes* Variable

Default: (command-names implicit-messages functions symbols
pathname-completion)

The default list of type-in modes used to process typed expressions in the interpreter and completion in the input editor. The list holds symbols that are set to instances of flavors that incorporate the **ucl:typein-mode** variable. The value includes modes that process typed UCL command names, Lisp function calls, implicit messages, symbol evaluation, and completion of pathnames. For a detailed description of these modes, refer to paragraph 6.5.1, Typed Expressions. (See also the **ucl:typein-modes** instance variable described in paragraph 7.7.1.2, Instance Variables for Reading Typed Input.)

ucl:*default-prompt* Variable

Default: >

A form whose value is a string used as the default prompt for typed input. (See also the **ucl:prompt** instance variable described in paragraph 7.7.1.2, Instance Variables for Reading Typed Input.)

ucl:*default-read-function* Variable
 Default: **ucl:read-for-ucl**, which permits the reading of *implicit lists* (that is, several atoms on one line).

The default read function used for reading typed expressions from the user. (See also the **ucl:read-function** instance variable described in paragraph 7.7.1.2, Instance Variables for Reading Typed Input.)

ucl:*default-print-function* Variable
 Default: **sys:pprin1**, which pretty-prints the expressions

The default print function used to print the results of executed Lisp forms. (See also the **ucl:print-function** instance variable described in paragraph 7.7.1.3, Instance Variables for Printing.)

ucl:*default-max-command-history* Variable
 Default: 16.

The default size of the command history. (See also the **ucl:max-command-history** instance variable described in paragraph 7.7.1.1, Basic Instance Variables.)

ucl:*default-max-output-history* Variable
 Default: 26.

The default size of the output history. (See also the **ucl:max-output-history** instance variable described in paragraph 7.7.1.3, Instance Variables for Printing.)

Miscellaneous Variables 7.7.4.3 This paragraph describes two other special global variables that are frequently useful to UCL applications.

ucl:this-application Variable

Holds the currently executing UCL application command-loop flavor instance. For example, while the graphics editor program is executing, this variable is bound to an instance of **ged:graphics-editor**. You may find it convenient to reference this variable in various parts of your code, such as command functions.

ucl:preempting? Variable

Non-nil when the current command is preempting a typed expression. This is used in certain commands that must alter their behavior when preempting typed expressions.

Miscellaneous Functions

7.8 The following paragraphs describe useful functions that you can include in your program as required.

ucl:pop-up-command-menu *menu-name* Function

Can be used to implement a command that displays a pop-up menu of other commands.

Argument: *menu-name* — A menu symbol used in the first argument to **build-menu**. It can be either a single-column or multicolumn menu. The menu item list (stored in the value cell of *menu-name*) is used in a pop-up menu. If the user selects a command to execute, the command instance is pushed onto your **ucl:command-execution-queue** and is the next command to be executed. If the user does not select a command, **nil** is returned.

The following example illustrates the use of the **ucl:pop-up-command-menu** function:

```
(defcommand display-sounds-menu ()
  (:description "Pop up a menu of sounds"
   :names      ("Sounds")
   :keys       (#\mouse-r-1)
  )
  (ucl:pop-up-command-menu 'sound-command-menu)
)
```

ucl:get-command *function-spec* Function

Returns the **ucl:command** instance associated with a **defcommand** method, function, or **make-command** logical name. The *function-spec* argument is either a method specification of the form **(:method application-flavor command-method-name)** or a symbol that is either a function name or **make-command** logical name. In any case, *function-spec* corresponds to the first argument of a call to **defcommand** or **make-command**. When no **ucl:command** is associated with *function-spec*, **nil** is returned.

The following examples illustrate the use of the **ucl:get-command** function:

```
(ucl:get-command 'doorbell-command)
(ucl:get-command '(:method sound-demo :help))
```

ucl:display-some-commands *commands-and-doc* Function
&optional *execute? help-doc*

Displays a list of **ucl:command** instances and text documentation strings in a Command Display scroll window.

Arguments: *commands-and-doc* — A list of **ucl:command** instances, **ucl:command-table** instances, and documentation strings. The commands, command tables, and the strings are displayed in the order of this list.

execute? — Specifies whether to execute a command when the user clicks left on a command shown in the display. If this option is non-**nil**, the user can select a command by clicking the left mouse button, which causes **ucl:display-some-commands** to place the command in the currently running UCL application command queue, **ucl:command-execution-queue**, to be executed. If you are not using the command loop with an application, you should set this argument to **nil**. Then, clicking left on a command causes this function to return the selected **ucl:command** instance for your program to execute.

help-doc — A string that is displayed when the user presses the HELP key while the commands are being displayed. It defaults to a string indicating that the user is viewing the active command tables in the current program, which is not necessarily true for your program.

view-documentation &optional *documentation* (*separator* "*****") Function

Displays the string specified by *documentation* in a pop-up Zmacs window, along with any previous documentation strings displayed in calls to **view-documentation**. In other words, this function adds the value of *documentation* to a history of displayed documentation strings and displays the history. Users do not have to search for documentation they have already viewed. If *documentation* is not supplied, the previous documentation entries are still displayed. The *separator* argument is a string to use in separating documentation from the previous documentation entries in the edit window.

A Zmacs window is provided to allow the user to mark regions and perform string searches through the displayed documentation history. The user can edit the history to collect useful documentation and save it in a file.

ucl:build-temporary-command-table *table-symbol* *command-data* Function
&optional *table-options*

Creates a temporary command table of commands for use in the UCL. It is useful for applications that create commands dynamically during execution. For example, the debugger commands for proceeding from errors cannot be built into permanent command tables because each error produces a different set of commands.

This function works like **build-command-table**, setting *table-symbol* to a command table instance. This command table holds commands that are created from data supplied in *command-data*. You can use the command table just as you would use a normal command table, pushing *table-symbol* onto the **ucl:active-command-tables** instance variable of your application flavor to activate the commands.

Arguments: *table-symbol* — A symbol to represent a **ucl:command-table** instance.

command-data — A list with two types of entries:

- **ucl:command** instances that already exist.
- Data to create temporary command instances. This data is arranged in the form of a list of keyword-value pairs that **make-command** uses, such as **:names**, **:keys**, and **:description**, but not **:command-flavor**. Temporary commands are of the flavor **ucl:temporary-command-table**.

table-options — A list of keyword-value pairs allowed by the **:init-options** keyword of **build-command-table**. The only ones you supply (if any) are **:name** and **:documentation**. The **:name** option provides the name of the command table to use in the Command Display, and the **:documentation** option is a string to display when the user wants the command table described. The default for **:name** is **Command Table**.

When your application no longer needs the temporary command table, you should deallocate it using **ucl:deallocate-temporary-command-table**.

ucl:deallocate-temporary-command-table *table-symbol* Function

Returns to their respective resources the temporary command tables and commands built by **build-temporary-command-table**. This function sets *table-symbol* to nil to assure that the pointer to the deallocated command table is lost.

ucl:looping-through-command-tables *all-of-them?* . *body* Macro

Provides a means of looping through command tables. On each iteration of the loop, the forms in *body* are executed with the symbol **ucl:command-table** bound to the current **ucl:command-table** instance. If *all-of-them?* is nil, then only the active command tables are included in the loop. All command tables of the program are looped through. The caller of **ucl:looping-through-command-tables** can terminate the loop by calling the **return** function.

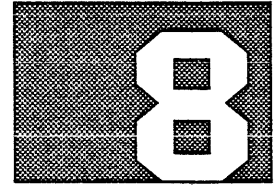
Note that there is more to the task of looping through command tables than looping through either of the variables **ucl:active-command-tables** or **ucl:all-command-tables**. You need to be aware that the UCL adds several command tables to each UCL application, storing them separately from these instance variables. Specifically, the UCL provides tables holding the universal commands, the input editor commands, and the application command macros built by the user. Thus, **ucl:looping-through-command-tables** loops through three instance variables, not one:

1. **ucl:special-command-tables**, which holds the application macro command table.
2. Either **ucl:active-command-tables** or **ucl:all-command-tables**, depending on the value of *all-of-them?*
3. **ucl:system-command-tables**, which holds the universal commands and input editor commands.

ucl:help-menu Function

Used by the UCL Help command to display a menu of help options. You can use it in your own help mechanisms, especially if you replace the UCL Help command with one of your own.

USING SUGGESTIONS



Introduction

8.1 The Suggestions system displays menus appropriate for different contexts of an application; these menus display lists of items that are mouse-sensitive. You can use the menus to display submenus, commands, Lisp symbols, information, and messages. You also can archive Lisp code in a menu for building commands. Figure 8-1 shows four of the different menus that are invoked by selecting items on the Zmacs Basic Menu.

This section discusses the following topics.

- How to access Suggestions — Tells how to invoke the Suggestions menus.
- Suggestions menu windows — Suggestions menus are displayed in a versatile window format that offers pop-up features, scrolling, and submenus. Section 9, Programming Suggestions, describes how to add suggestions to your application. The Suggestions frame reapporitions the window to display the Suggestions menus without interfering with the information displayed by the application. The Suggestions windows handle any word-wrapping required to adjust your application screen to allow the display of Suggestions menus.

The Suggestions system accesses the Universal Command Loop (UCL), so Suggestions can offer a broad range of help documentation and menus of helpful features. Because the Suggestions system runs in its own process, it provides ways for the user to handle other processes that have become arrested or locked during software development.

- Menu examples — Examples of the Suggestions menus for the Lisp Listener, Zmacs, the Inspector, and the debugger are provided.

Sets of Suggestions menus are also provided for the following Explorer applications: break, Stepper, Converse, Flavor Inspector, and namespace editor.

In UCL-based applications, a set of Suggestions menus complements the UCL's Command Display command by showing the commands of an application in menus that are displayed while the user is viewing the application window. These menus can also be arranged into smaller menus according to criteria selected by the application developer.

- Menu tools — The Suggestions Menu Tools menu provides options that let you modify the behavior of Suggestions. For example, you can do the following:
 - Find menus or commands
 - Create new menus or menu items
 - Add, reorder, remove, and save menus
 - Display pop-up keystrokes when you select a command

- Display a history of the menus you have chosen
 - Add a symbol to a Lisp expressions menu
 - Select the applications for which you want to display Suggestions
 - Turn Suggestions off or return to the initial Suggestions menus
- Lisp expressions — The Suggestions system also provides a set of Lisp expressions menus that let the user enter Lisp code from the Suggestions menus. The default set of Lisp expressions menu consists of Lisp functions, global variables, format directives, and so forth. Custom menus consisting of strings of Lisp code can be built and used for rapid assembly of programs and selection of file pathnames; that is, the user can select these items from the menu instead of typing them.

How to Access Suggestions

8.2 You can invoke Suggestions by doing one of the following;

- Select the Suggestions item from the System menu.
- Select the Suggestions item from the Listener permanent command line (that is, the command line directly above the mouse documentation window).
- Press TERM SUPER-HELP.

A menu appears that allows you to turn on Suggestions for several applications. You can specify the applications for which you want Suggestions turned on, or you can select the Do It box to turn on Suggestions for all the applications listed.

To turn Suggestions off, you can select the Suggestions item from the System menu again or press TERM SUPER-HELP. You can also select the Suggestions Menus Off item from the Suggestions menus.

Suggestions Menu Windows

8.3 The Suggestions constraint frame consists of four panes, three of which contain menus, and a fourth pane that displays the name of the menu that is installed in the large pane (see Figure 8-2 and Figure 8-3). Each menu pane can be scrolled when the menu installed in it contains more items than can be displayed at one time. Moving the mouse cursor to the upper or lower boundary of the pane where the message More Above Or More Below is displayed causes the menu to scroll in that direction.

Figure 8-1 Suggestions System Menu Interaction

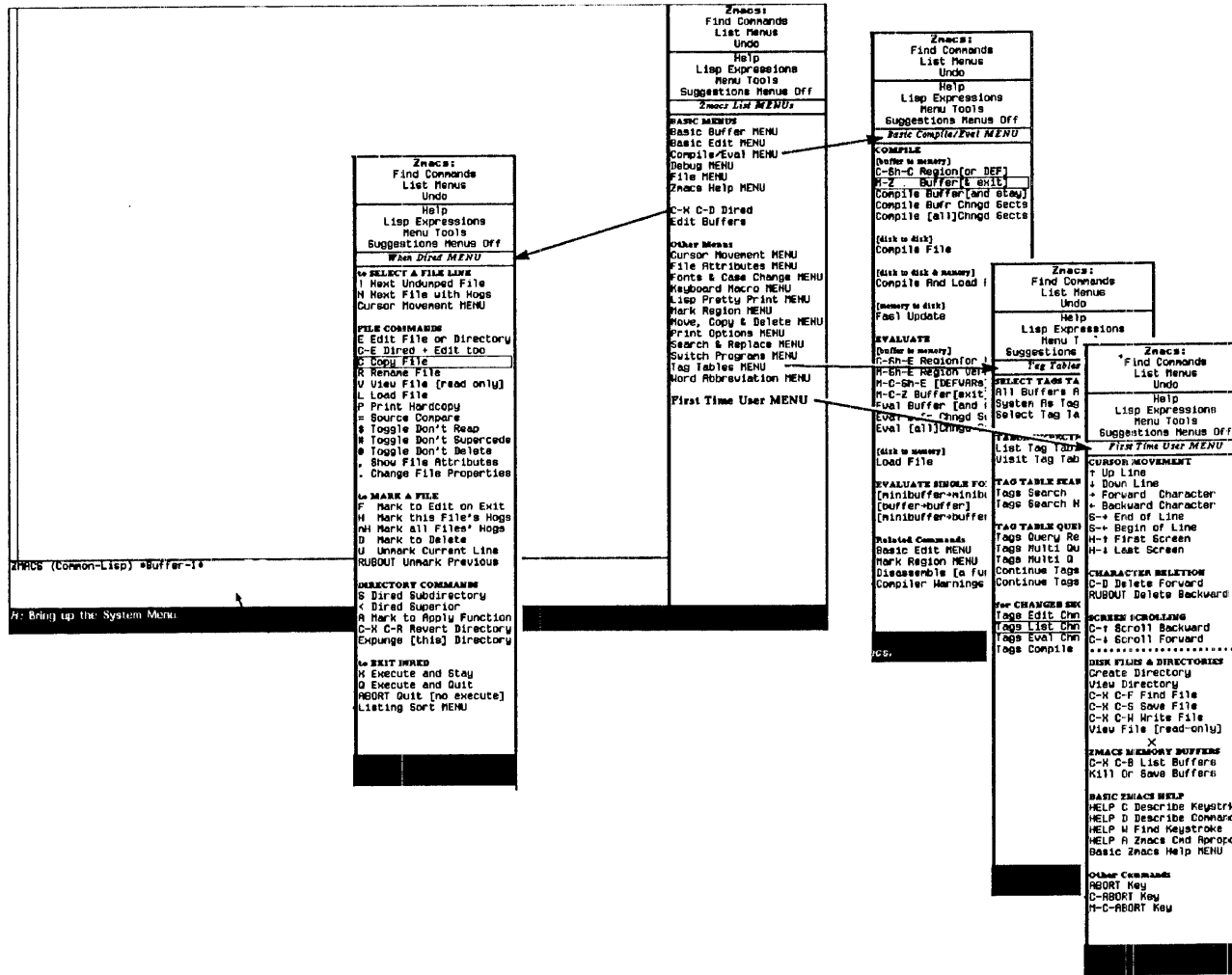
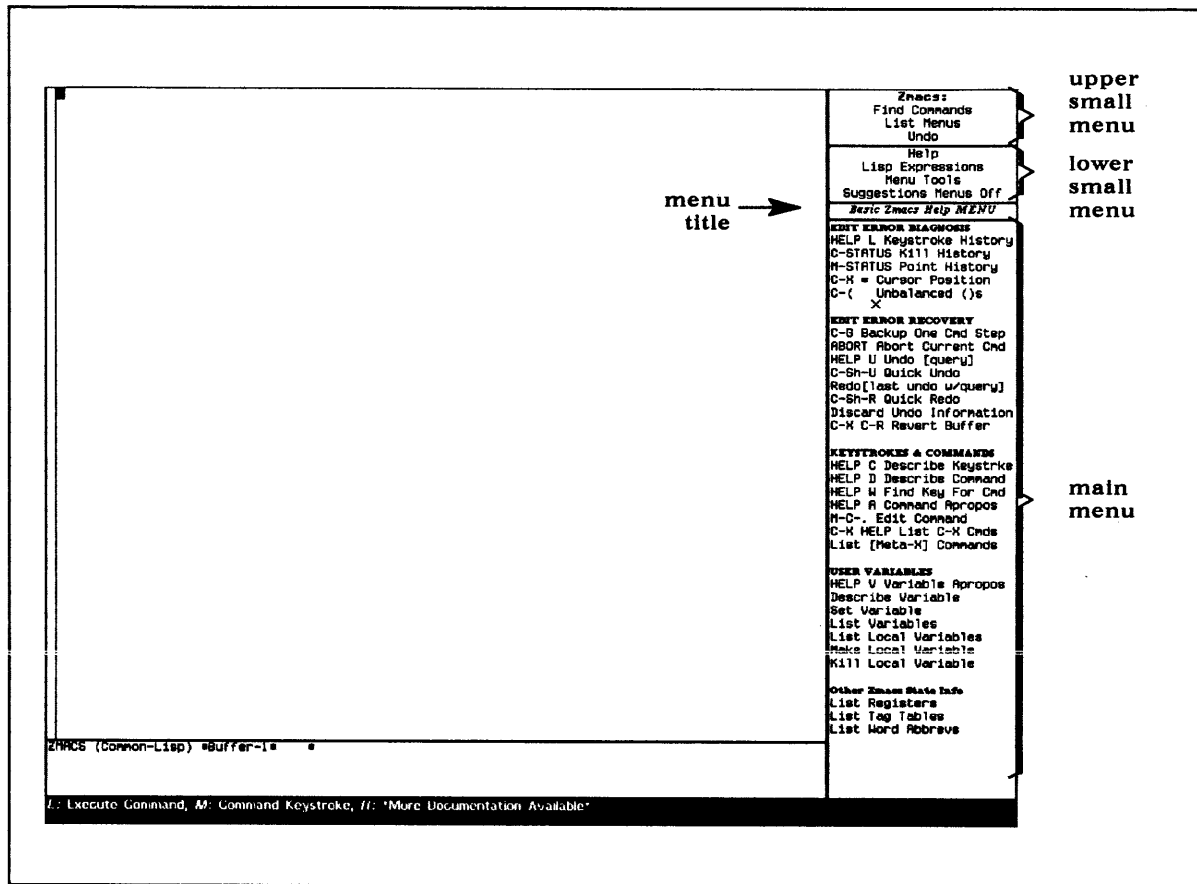


Figure 8-2 Suggestions Frame, Explorer Landscape Configuration



Suggestions Window Configurations

Explorer Landscape Video Display

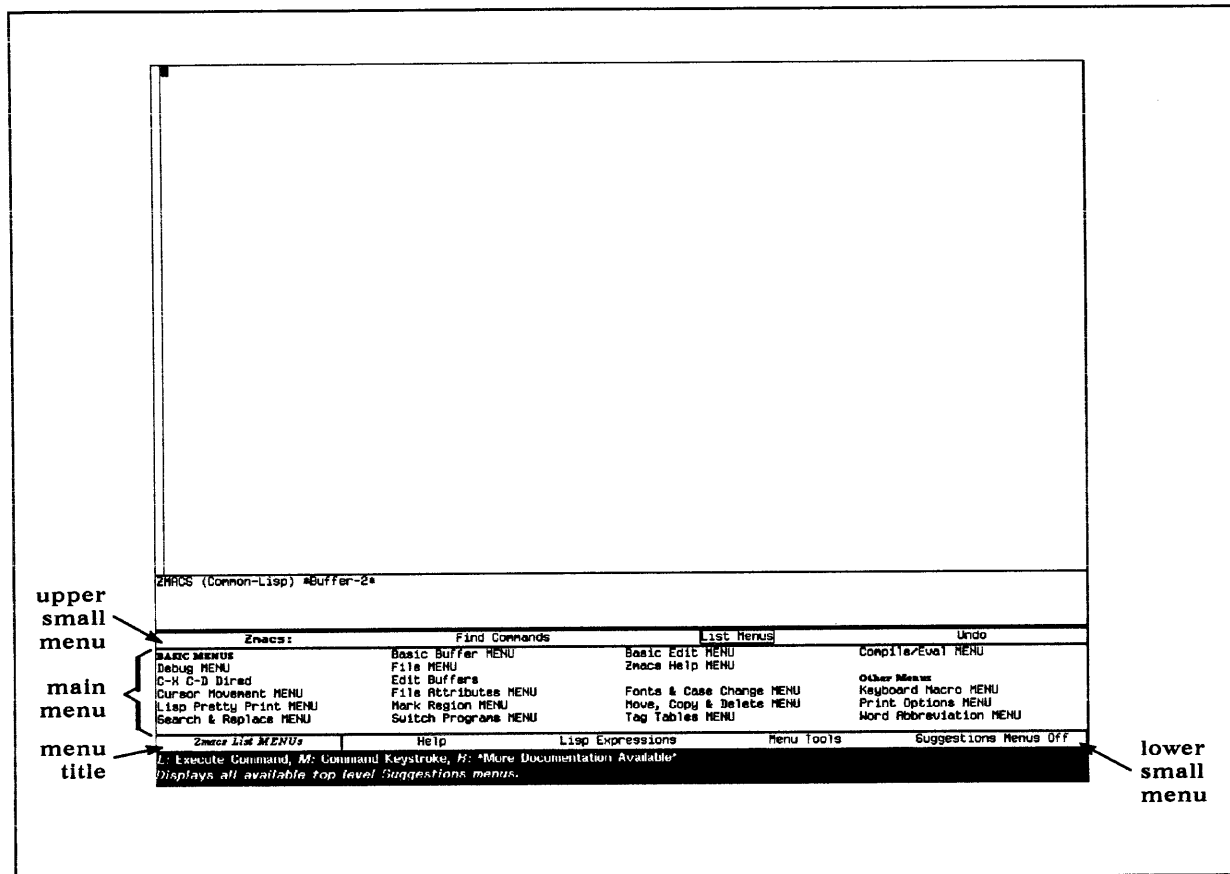
Portrait Video Display

8.3.1 The Suggestions constraint frame definition provides configurations for both the portrait and landscape displays. The appropriate configuration for the video display is set internally when you turn on Suggestions.

8.3.1.1 The Explorer system uses a landscape video display, that is, a screen that is wider than it is high. In the landscape configuration, the Suggestions frame is a vertical window at the right edge of the screen (see Figure 8-2).

8.3.1.2 The portrait configuration of the Suggestions frame accommodates a video display that is higher than it is wide. It extends horizontally across the bottom of the screen and is located immediately above the mouse documentation window (see Figure 8-3).

Figure 8-3 Suggestions Frame, Portrait Configuration



Panes 8.3.2 Both standard configurations of the Suggestions frame contain four panes as follows:

- A menu pane (upper small menu) that contains the name of the application for which the Suggestions menus are currently installed and a set of commands that are set apart, either because they are important to the selected application or because they are generally useful in any context for the application.
- A pane (lower small menu) that contains other frequently selected commands.
- A label pane that displays the name of the menu currently installed in the main-menu pane.
- A main pane (main menu) that contains commands, Lisp expressions, or submenus of commands. The items in this pane change when you select a new menu and also when the state of the system changes in certain ways so that a new set of allowable commands is in effect. (See paragraph 9.5, Triggering Automatic Menu Changes.)

Examples of a Lisp Listener command menu and a Zmacs command menu are shown in Figure 8-4 and Figure 8-5. These examples show different ways of formatting the contents of a pane. Notice that the Zmacs menu has headings for subgroups of commands and several keystrokes displayed in the names of commands.

Figure 8-4 Listener Suggestions Basic Menu

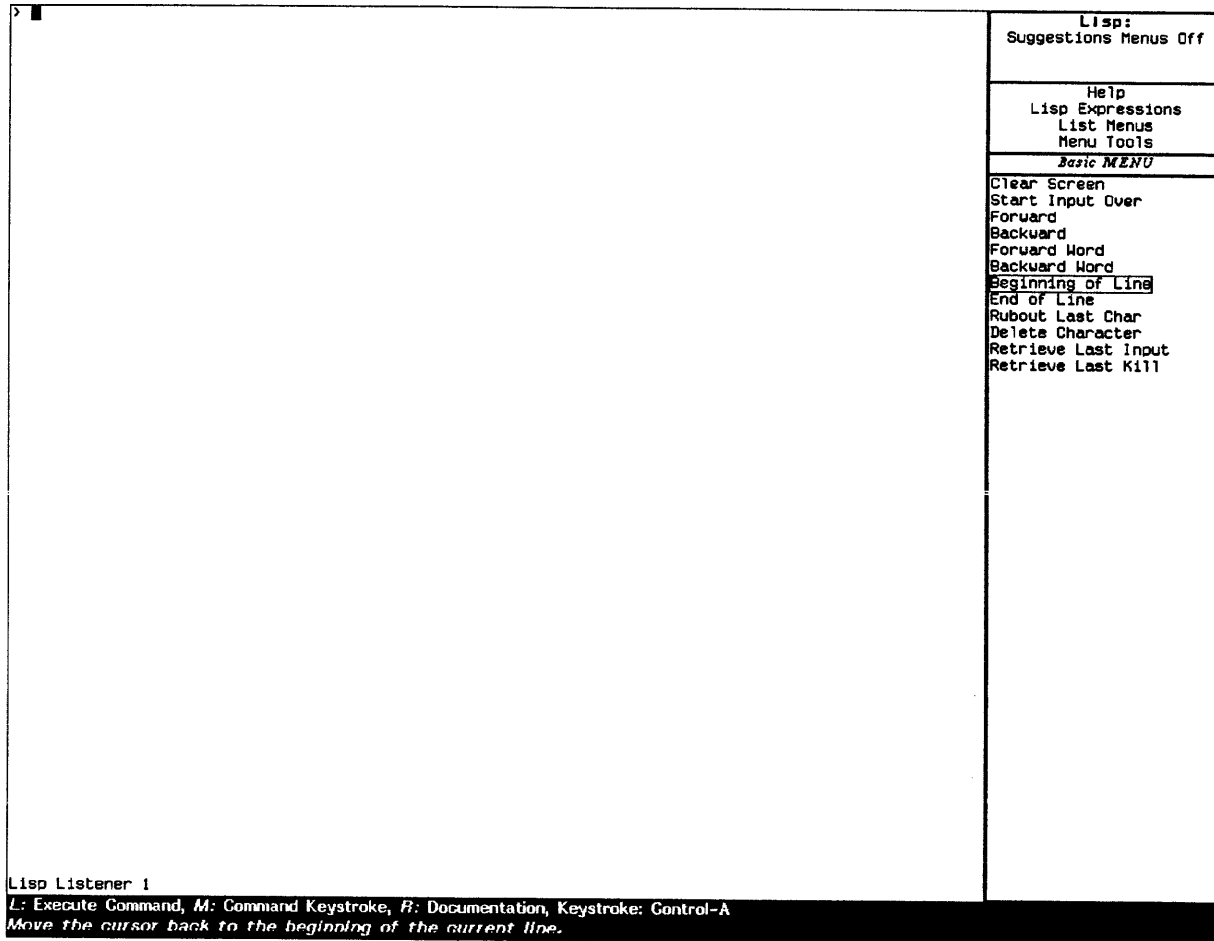


Figure 8-5 Zmacs Suggestions Menu With Keystroke Display

	<pre> ZMACS: Find Commands List Menus Undo Help Lisp Expressions Menu Tools Suggestions Menus Off Cursor Movement MENU by LINE & CHARACTER ↑ Up Line ↓ Down Line → Forward Character ← Backward Character by SENTENCE & WORD M-↑ Up Sentence M-↓ Down Sentence M-→ Forward Word M-← Backward Word by DEF & S-EXPRESSION M-C-↑ Up DEF M-C-↓ Down DEF M-C-→ Forward SEXP M-C-← Backward SEXP by LIST STRUCTURE M-C-D Down List M-C-H Forward List M-C-P Backward List M-C-) Forward Up List M-C-(Backward Up List SCROLLING C-→ Scroll Backward C-↓ Scroll Forward H-↑ Top of Buffer H-↓ Bottom of Buffer to WINDOW EXTREMES S-↑ Top of Screen S-↓ Bottom of Screen S-→ End of Line S-← Begin of Line M-M Begin - Indent Other Cursor Menu </pre>
<pre>ZMACS (Common-Lisp) #Buffer-1#</pre>	
<pre> L: Execute Command, M: Command Keystroke, R: Documentation With an argument n, these commands repeat n times. </pre>	

Menu Examples

8.4 Figure 8-6 shows the video display with Suggestions enabled. The Lisp Listener window, which normally extends to the right margin of the screen, has been shrunk to allow room for the Suggestions frame to be displayed vertically along that edge.

You typically use the Lisp Listener to type Lisp expressions for evaluation by the interpreter. An example of using Suggestions to evaluate a Lisp symbol is shown in Figure 8-6. The user has selected the symbol `SYS:ALL-PROCESSES` from the System Symbols menu. These words are automatically inserted at the keyboard cursor location. In this case, the evaluation of the symbol causes a list of all packages to be printed out immediately below the symbol name.

Figure 8-6 Evaluation of a Symbol Using Lisp Listener Suggestions

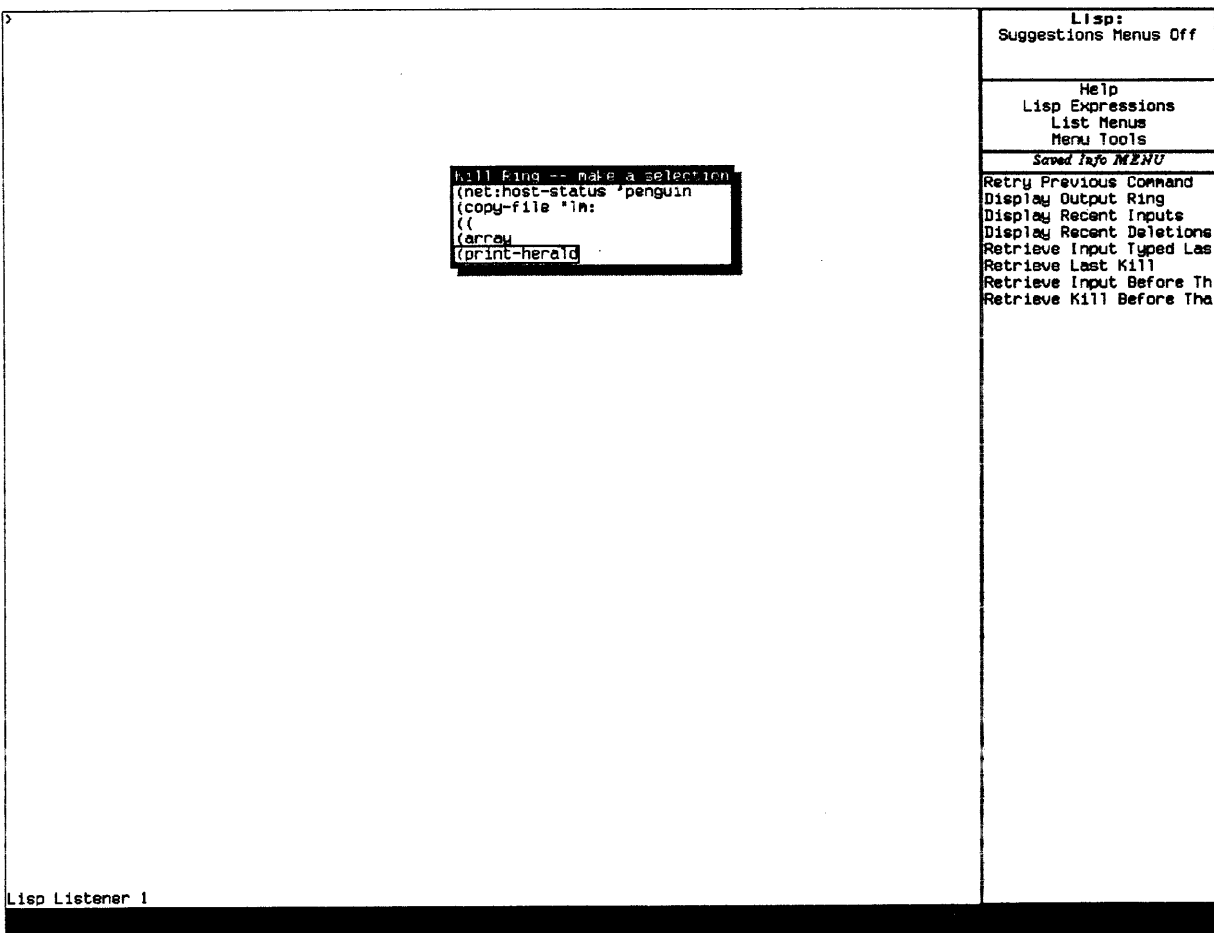
<pre> > SYS:ALL-PROCESSES (#<SYS:PROCESS Inspect Frame 2 72775064> #<SYS::COROUTINING-PROCESS Zmacs Frame 2 71211553> #<SYS:PROCESS Peek Frame 1 70645533> #<SYS:PROCESS Dormant FTP Connection GC 57044463> #<SYS::COROUTINING-PROCESS Zmacs Frame 1 21470141> #<SYS:PROCESS Mail Daemon 21371660> #<SYS:PROCESS TCP Background 21361421> #<SYS:PROCESS UDP Background 21360524> #<SYS:PROCESS IP Packet Fragment timer 21310977> #<SYS:PROCESS ICMP Listener 14303610> #<SYS:PROCESS Suggestions Frame 2 14307361> #<SYS:PROCESS Suggestions Frame 1 51053132> #<SYS:PROCESS Pop-Up Keystrokes 14304071> #<SYS:SIMPLE-PROCESS DC Daemon 14300162> #<SYS:PROCESS Lisp Listener 2 14307461> #<SYS:PROCESS Hardware Monitor 14307021> #<SYS:PROCESS Tn-Update 14307521> #<SYS:PROCESS NUBUS Receiver, NXF0 14303425> #<SYS:PROCESS Chaos Background 14303550> #<SYS:PROCESS Screen Manager Background 14307321> #<SYS:PROCESS Mouse 14303365> #<SYS:PROCESS Keyboard 14302301> #<SYS:PROCESS Initial Process 14300222> #<SYS:PROCESS Page-Background 14307661>) </pre>	<div style="text-align: right;"> <p>Lisp: Suggestions Menu Off</p> <hr/> <p>Help Lisp Expressions List Menus Menu Tools</p> <hr/> <p><i>System Symbols Menu</i></p> <p>TU:ALL-THE-SCREENS SYS:ACTIVE-PROCESSES SYS:ALL-PROCESSES CURRENT-PROCESS TU:MOUSE-PROCESS SUGG:;*ALL-PACKAGES* *PACKAGE* Previously Selected Wind TU:PREVIOUSLY-SELECTED-W TU:WINDOW-RESOURCE-NAMES TU:*TERMINAL-KEYS* TU:*SYSTEM-KEYS* TU:SELECTED-WINDOW TU:SELECTED-IO-BUFFER TU:MOUSE-WINDOW *STANDARD-INPUT* *STANDARD-OUTPUT* TU:INITIAL-LISP-LISTENER TU:DEFAULT-SCREEN TU:MAIN-SCREEN TU:WHO-LINE-SCREEN M:;*SYSTEM-MENU-PROGRAMS M:;*SYSTEM-MENU-THIS-WIN M:;*SYSTEM-MENU-WINDOWS-</p> </div>
<p>Lisp Listener 1 L: Execute Command, M: Command Keystroke, R: *More Documentation Available* A list of all processes that have not been "killed".</p>	

Lisp Listener Suggestions

8.4.1 A number of commands available in a Lisp Listener allow you to correct typing mistakes, request documentation for the items you have typed, ask the system to complete the item that is being typed, and so forth. Normally, you invoke these commands by pressing keystroke sequences, such as CTRL-C, which inserts the most recently entered form at the cursor location as if you had typed it again.

When the Suggestions menus are enabled, however, you have the option of executing those same commands from the menus instead. In Figure 8-7, the user has moved the mouse cursor to the Suggestions menu option Display Recent Deletions.

Figure 8-7 Invoking the Display Recent Deletions Menu Using Listener Suggestions



By moving the mouse cursor over the items in the large menu pane, a new user can investigate each of the commands available when interacting with a Lisp Listener window. Clicking left with the mouse over one of these items—selecting that item—issues the corresponding command to the Lisp Listener; clicking middle temporarily displays a small window that shows the corresponding keystroke command; clicking right displays a window that contains full documentation for the command.

You can click left on the List Menus item in the uppermost Suggestions menu to see what other menus of commands are available for Lisp Listeners. The result is shown in Figure 8-8, which lists the menus that can be selected in addition to the Basic Menu. Clicking left on Completion Menu installs that menu of commands in the Suggestions main menu as shown in Figure 8-9. These commands cause the system to try either to complete the string that you have typed or to display a pop-up menu of possible completions to allow you to select one.

For the example shown in Figure 8-9, suppose you only remember that the name of the desired Lisp function contains the string typed in the Listener. You then select List Apropos Completions from the Suggestions menu to display a list of Lisp functions whose names contain that string. You can use the mouse to select one of the completions listed in the menu; that selection is then automatically inserted at the keyboard cursor location.

The Suggestions interface provides menus of Lisp Listener commands that acquaint new users with available options and help them learn the keystroke sequences that permit them to interact efficiently with the Lisp Listener utility. Besides promoting more efficient interaction, Suggestions also displays the appropriate command to remedy an adverse condition, such as a window lock.

Figure 8-8 Result of Selecting the Listener List Menu Option

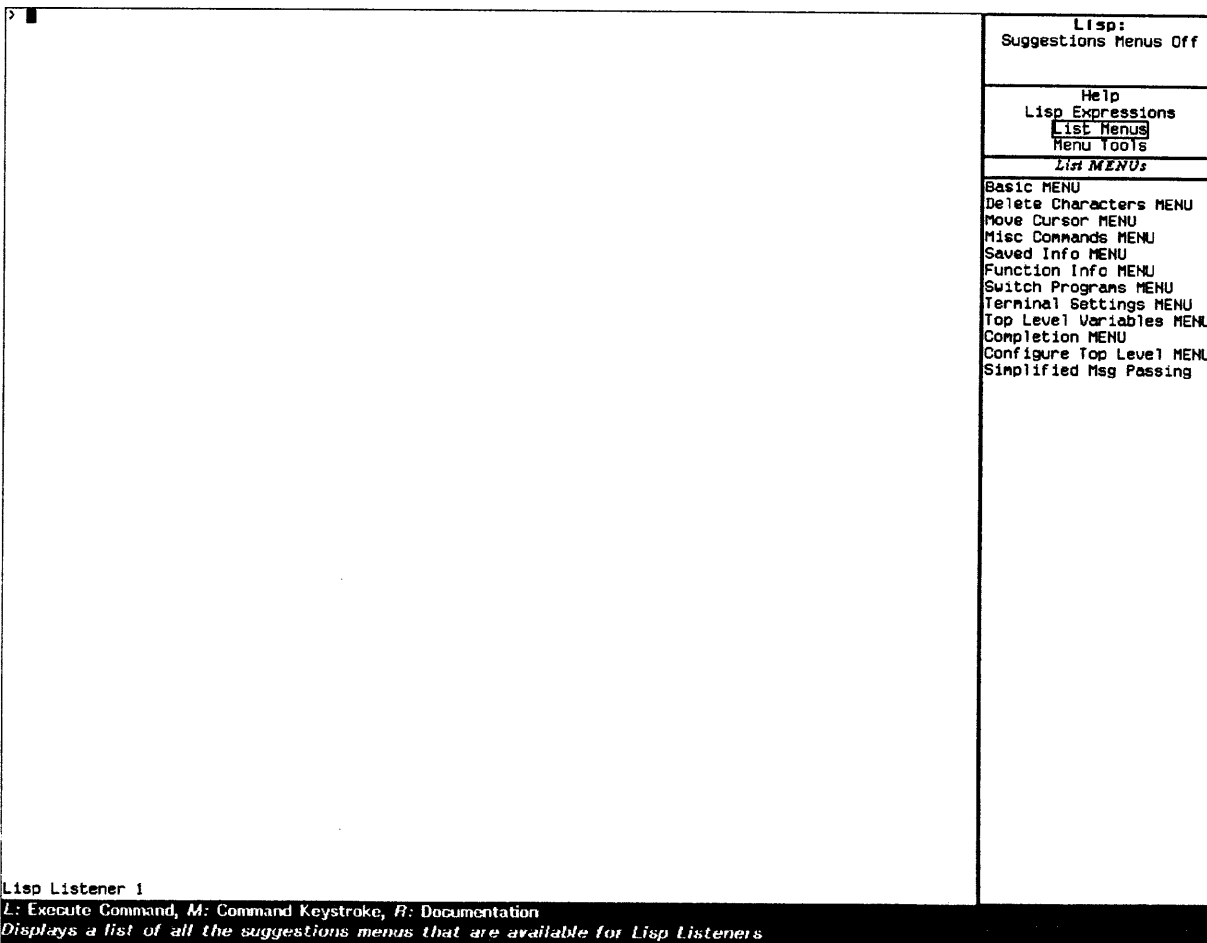


Figure 8-9 Result of Selecting the List Apropos Completions Option

```

> (array
Select one of the following (Functions)
#ARRAY
ADJUST-ARRAY
ADJUST-ARRAY-SIZE
ADJUSTABLE-ARRAY-P
APPEND-TO-ARRAY
ARRAY
ARRAY-N-DIMS
ARRAY-ACTIVE-LENGTH
ARRAY-BITS-PER-ELEMENT
ARRAY-DIMENSION
ARRAY-DIMENSION-N
ARRAY-DIMENSIONS
ARRAY-DISPLACED-P
ARRAY-ELEMENT-SIZE
ARRAY-ELEMENT-TYPE
ARRAY-ELEMENTS-PER-Q
ARRAY-GROW
ARRAY-HAS-FILL-POINTER-P
ARRAY-HAS-LEADER-P
ARRAY-IN-BOUNDS-P
ARRAY-INDEXED-P
ARRAY-INDIRECT-P
ARRAY-INITIALIZE
ARRAY-LEADER
ARRAY-LEADER-LENGTH
ARRAY-LENGTH
ARRAY-POP
ARRAY-PUSH
ARRAY-PUSH-EXTEND
ARRAY-RANK
ARRAY-ROW-MAJOR-INDEX
ARRAY-TOTAL-SIZE
ARRAY-TYPE
ARRAY-TYPES
ARRAYCALL
ARRAYDIMS
ARRAYP
COPY-ARRAY-CONTENTS
COPY-ARRAY-CONTENTS-AND-LEADER
COPY-ARRAY-PORITION
DISPLACED-ARRAY-P
FILLARRAY
GET-LOCATIVE-POINTER-INTO-ARRAY
LIST-ARRAY-LEADER
LISTARRAY
MAKE-ARRAY
MAKE-ARRAY-INTO-NAMED-STRUCTURE
MAKE-PIXEL-ARRAY
NUMBER-INTO-ARRAY
PIXEL-ARRAY-HEIGHT
PIXEL-ARRAY-WIDTH
RETURN-ARRAY
SORT-GROUPED-ARRAY
SORT-GROUPED-ARRAY-GROUP-KEY
STORE-ARRAY-LEADER

```

Lisp Listener 1
Add VALUE as an element at the end of ARRAY.
The fill pointer (leader element 0) is the index of the next element to

Zmacs Suggestions

8.4.2 To take advantage of the advanced capabilities of the Zmacs editor, you need to know a large number of commands and the keystrokes that execute them. Unfortunately, informal observations of moderately experienced users suggest that only a very small fraction of the power of Zmacs is effectively tapped by users.

Suggestions menus were first developed to give online assistance in accessing the hundreds of Zmacs commands; these menus were then provided as a similar interface to all of the major keystroke-driven programming utilities on the Explorer system.

Figure 8-10 shows the menus that are available when you select List Menu while using Zmacs. Besides these menus that you can select whenever you want, one of another set of 20 or so menus is automatically installed whenever you cause a context switch in Zmacs that makes a new set of commands temporarily relevant. For instance, a switch in the context occurs when you enter a command not related to Zmacs editing, such as a file command or the Dired command.

Figure 8-10 Result of Selecting the Zmacs List Menu Option

<pre> -- MODE: lisp ; BASE: 10. ; PACKAGE: user ; FONTS:(cptfont h110b) -- ;; ;; RESTRICTED RIGHTS LEGEND ;; ;;Use, duplication, or disclosure by the Government is subject to ;;restrictions as set forth in subdivision (b)(3)(11) of the Rights in ;;Technical Data and Computer Software clause at 52.227-7013. ;; ;; TEXAS INSTRUMENTS INCORPORATED. ;; P.O. BOX 2909 ;; AUSTIN, TEXAS 78769 ;; MS 2151 ;; ;; Copyright (C) 1985, Texas Instruments Incorporated. All rights reserved. ;; Copyright (c) 1984 by Texas Instruments, Incorporated ;; All rights reserved. # ;; Universal Command Loop ;; Starter Kit ;; This file provides an example of an application which uses the UCL. The file is broken up into sections, with each section building on the previous one and showing more features of the UCL and UCL programming techniques. The first section is a tiny application using only the basic features. Later sections show how to tailor the UCL to meet the applications needs. As you study each section, you should compile the code in the section and call the function TANK to run the section's version of the application. Do not compile the whole file, as that will give you the most complex version after needlessly compiling the simpler versions. In case you are not familiar with Zmacs, the following is a list of commands you will need to use in order to go through each version: CTRL-SPACE -- Sets a mark at the current cursor position. As you then move the cursor, a "region" will be underlined. META-W -- Clears the "region" (doesn't kill it; stops underlining). CTRL-V, META-V -- Scroll forward and backward one page. CTRL-N, CTRL-P -- Move cursor down or up one line. CTRL-SHIFT-C -- Compile "region". You should use the above commands to mark a section and then compile it with this command. Each section contains a call to TANK at the end. If this call is included in your region when you compile it, the application's window will be expose and the application will start executing. For detailed documentation on the Universal Command Loop, refer to the ZMACS (ZetaLisp) STARTER-KIT.LISP#* UCL; MA-X: (2) Font: A (CPTFONT) ↓ Reading MR-X: UCL; STARTER-KIT.LISP#2 -- 93K characters. Type META-CTRL-L to return to Dired </pre>	<pre> Zmacs: Find Commands List Menu Undo Help Lisp Expressions Menu Tools Suggestions Menu Off Zmacs List MENUS BASIC MENUS Basic Buffer MENU Basic Edit MENU Compile/Eval MENU Debug MENU File MENU Zmacs Help MENU C-X C-D Dired Edit Buffers Other Menus Cursor Movement MENU File Attributes MENU Fonts & Case Change MENU Keyboard Macro MENU Lisp Pretty Print MENU Mark Region MENU Move, Copy & Delete MENU Print Options MENU Search & Replace MENU Switch Programs MENU Tag Tables MENU Word Abbreviation MENU First Time User MENU </pre>
<p>L: Execute Command, M: Command Keystroke, R: *More Documentation Available* Displays all available top level Suggestions menus.</p>	

As Figure 8-11 shows, the keystrokes that correspond to commands are often listed explicitly in the menu. This listing of the Fonts & Case Change menu helps you see patterns in the assignment of keystrokes to commands, thus making it easier for you to learn the keystrokes. For example, it is apparent from the figure that the keystrokes for the commands that change fonts differ only in the modifier keys CTRL, META, and SHIFT.

Figure 8-11 Example Keystroke Commands Listed in Zmacs Suggestions

<pre>;; -- Mode:Common-Lisp; Fonts:(CPTFONT HL128 HL128I) --</pre> <p>Using the Change Fonts command, you can display text in several different <i>fonts</i>.</p>	<pre>Zmacs: Find Commands List Menus Undo Help Lisp Expressions Menu Tools Suggestions Menu Off Fonts & Case MENU EXAMINE FONTS List [all loaded] Fonts Display [specified] Font MODIFY BUFFER DEFAULTS Electric Font Lock Mode Electric Shift Lock Mode Set [buffer's] Fonts Reparse Attribute List Update Attribute List File Attribute MENU CHANGE FONTS C-J a Character M-J a Word C-X C-J a Region M-Sh-J X-Y in Region M-C-J the Default <also see SYSTEM F> CHANGE WORD CASE M-C Uppercase Initial M-U Uppercase Word M-L Lowercase Word CHANGE REGION CASE Lisp Uppercase Region Lisp Lowercase Region Related Commands Mark Region MENU C-B Unmark Region</pre>
<pre>ZMACS (Common-Lisp) *Buffer-1* Font: A (CPTFONT) *</pre>	
<p>L: Execute Command, M: Command Keystroke, R: *More Documentation Available*, Keystroke: CTRL-X CTRL-J Change the font between point and the mark.</p>	

**Inspector
Suggestions**

8.4.3 The Inspector, a window-based utility that allows you to examine Lisp objects, contains a permanent command menu pane (the fourth pane down on the left margin in Figure 8-12). Because the commands you use to operate the Inspector are already displayed, the Suggestions menus for the Inspector contain Lisp expressions instead of commands. You typically interact with the Inspector by either typing symbol names or selecting objects with the mouse, so this use of Suggestions saves you from typing the expression. Figure 8-12 shows the Inspector with Suggestions menus enabled. The user has selected the item `w:main-screen` from the Interesting Globals menu, which causes the characters of the item to be entered into the Inspector type-in window (immediately above the mouse documentation window) in the same way as if the user had typed them. The user then pressed RETURN to inspect `w:main-screen`.

Figure 8-12 Inspector Suggestions Menus

<pre>#<STANDARD-SCREEN Main Screen 17540143 exposed> An object of flavor TV::STANDARD-SCREEN. Function is #<EQ-HASH-TABLE (Funcallable) 17171425> TV:SCREEN-ARRAY: #<ART-1B-754-1024 14300473> TV:LOCATIONS-PER-LINE: 32 TV:OLD-SCREEN-ARRAY: #<ART-1B-754-1024 14300473> TV:BIT-ARRAY: NIL TV:NAME: "Main Screen"</pre>		<p>Inspector: List Menus Menu Tools</p>
<pre>a list (#<INSPECT-FRAME Inspect Frame 2 17563654 exposed> #<SUGGESTIONS-FRAME Suggestions Frame 2 17540536 exposed> #<ZMACS-FRAME Zmacs Frame 2 17562604 deexposed> #<BACKGROUND-LISP-INTERACTOR Background Lisp Interactor 1 17560150 deexposed> #<PEEK-FRAME Peek Frame 1 17560720 deexposed> #<LISP-LISTENER Lisp Listener 1 17540000 deexposed> #<LISP-LISTENER Lisp Listener 2 17540656 deexposed> #<ZMACS-FRAME Zmacs Frame 1 17556346 deexposed> #<SUGGESTIONS-OFF-FRAME Suggestions Off Frame 1 17540241 deexposed></pre>		<p>Help Lisp Expressions</p>
<pre>#<ZMACS-FRAME Zmacs Frame 2 17562604 deexposed> An object of flavor ZNEI::ZMACS-FRAME. Function is #<EQ-HASH-TABLE (Funcallable) 47770750> TV:SCREEN-ARRAY: #<ART-1B-754-1024 71204761> TV:LOCATIONS-PER-LINE: 32 TV:OLD-SCREEN-ARRAY: NIL TV:BIT-ARRAY: #<ART-1B-754-1024 71125550> TV:NAME: "Zmacs Frame 2" TV:LOCK: NIL TV:LOCK-COUNT: 0 TV:SUPERIOR: #<STANDARD-SCREEN Main Screen 17540143 exposed> TV:INFERIORS: (#<MODE-LINE-WINDOW Mode Line Window 6 17562706 exposed> #<ZMACS-WI TV:EXPOSED-P: NIL TV:EXPOSED-INFERIORS: (#<ZMACS-WINDOW-PANE Zmacs Window Pane 2 17563401 exposed> #<MODE-L TV:M-OFFSET: 0 TV:Y-OFFSET: 0 TV:WIDTH: 824 TV:HEIGHT: 754 TV:CURSOR-X: 1 TV:CURSOR-Y: 1 TV:MORE-UPDS: NIL TV:TOP-MARGIN-SIZE: 1 TV:BOTTOM-MARGIN-SIZE: 1 TV:LEFT-MARGIN-SIZE: 1 TV:RIGHT-MARGIN-SIZE: 1 TV:FLAGS: 32784 TV:BASELINE: 9 TV:FONT-MAP: #<FONT-D-26 71125615> TV:CURRENT-FONT: # TV:BASELINE-ADJ: 0 TV:LINE-HEIGHT: 13</pre>		<p>Interesting Globals MENU</p> <p>General current-process #all-packages# packages #all-flavor-names# #standard-input# #standard-output#</p> <p>TV u:all-the-screens Previously Selected Wind w:previously-selected-wi u>window-resource-names u:#system-keys# u:#terminal-keys# u:selected-window tv:selected-io-buffer u:mouse-window u:initial-lisp-listener u:default-screen u:main-screen u:who-line-screen u:#SYSTEM-MENU-PROGRAMS- u:#SYSTEM-MENU-THIS-WIND u:#SYSTEM-MENU-WINDOWS-C u:mouse-process</p> <p>SI si:self-mapping-table si:patch-systems-list si:#systems-list# si:active-processes si:all-processes</p> <p>FS fs:#default-pathname-def fs:#pathname-host-alist#</p>
<pre>Flavins Doc Exit Delete Set= Refresh Modify Config Mode Print Edit</pre>	<pre>#<STANDARD-SCREEN Main Screen 17540143 exposed> #<INSPECT-FRAME Inspect Frame 2 17563654 exposed> #<SUGGESTIONS-F #<ZMACS-FRAME Zmacs Frame 2 17562604 deexposed></pre>	
<pre>Inspect: TV:MAIN-SCREEN Inspect:</pre>		
<pre>Inspect Frame 2 I: Inspect the indicated object, M: Remove it from the Inspector, R: Find its function definition if it exists</pre>		

**Debugger
Suggestions Menus**

8.4.4 Suggestions tracks context changes to provide the user with a set of legal commands whenever possible. The goal is to ensure that the user always has a set of commands from which to choose. When the Explorer debugger is invoked, for example, Suggestions assists the user by automatically installing a set of menus that contain debugger commands. Figure 8-13 shows an example of this. The Suggestions menus have changed to replace the menus of Lisp Listener commands with debugger commands.

When the debugger is invoked, it displays an error message, the name of the function in which the error occurred, that function's arguments, and a list of commands bound to the SUPER key. Because of this listing, many users assume that these are the only commands available in the debugger when in fact there are many more. In addition to commands that let you examine and manipulate functions, arguments, and global variables, a window debugger is available, which displays various features of the current execution stack. Lisp expressions can also be evaluated in the error context.

As shown in Figure 8-13, a menu item that invokes the window debugger is in one of the two upper Suggestions panes whose contents are available as long as the user is still interacting with the debugger. The possibility of evaluating a Lisp expression and various ways of returning from the error are indicated in the Basic Debugger menu that Suggestions installs in the main menu pane by default whenever the debugger is invoked. Users can also select the List Menus option to execute or learn about other debugger commands. When the user enters a Lisp expression to be evaluated, input editor Suggestions menus are automatically installed to assist the user in editing the expression.

Figure 8-13 Debugger Suggestions Menu

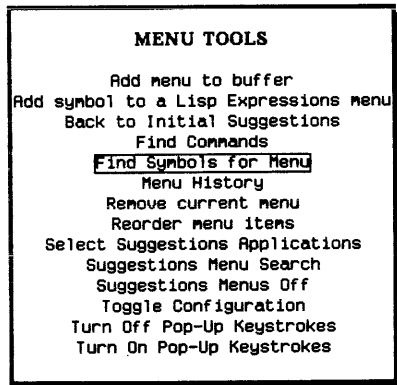
<pre> >>Keyboard break. While in the function PROCESS-WAIT + TV:KBD-IO-BUFFER-GET + (:METHOD TV:STREAM-MIXIN :ANY-TYI) Debugger entered while in the following function: PROCESS-WAIT (P.C. = 28) Arg 0 (WHOSTATE): "Keyboard" Arg 1 (FUNCTION): #<DTP-FUNCTION (:INTERNAL TV:KBD-IO-BUFFER-GET 0) 22050600> Rest arg (ARGUMENTS): (#<IO-BUFFER 76401503: empty, State: NIL>) Commands for proceeding from this particular error: Super-A, ⌘-A: NO-ACTION (undocumented). Super-B: Restart process Initial Process. Super-C: Reset and arrest process Initial Process. ⌘-C: Return to Lisp Listener top level. →■ </pre> <p>Lisp Listener 1 A: Bring up the System Menu.</p>	<p>Debugger: Proceed Types Window Based Debugger Abort</p> <hr/> <p>Help Lisp Expressions List Menus Menu Tools</p> <hr/> <p><i>Basic Debugger MENU</i></p> <p>Eval a Lisp Form Short Backtrace Full Backtrace Return A Value Return Reinvoication Clear And Show All Down Stack Up Stack Edit Frame Function Search Bug Report Repeat Error Message</p>
---	--

Menu Tools

8.5 The Menu Tools set of commands is provided for managing the Suggestions menus themselves. You display a pop-up menu of the tools by clicking left on the Menu Tools option in one of the small panes (see Figure 8-14). The menu tools options include the following, which are described in the following text:

- Back to Initial Suggestions
- Find Commands
- Select Suggestions Applications
- Menu History
- Suggestions Menu Search
- Suggestions Menus Off
- Add Menu to Buffer
- Turn Off Pop-Up Keystrokes
- Turn On Pop-Up Keystrokes
- Reorder Menu Items
- Find Functions for Menu
- Remove Current Menu
- Add a Symbol to a Lisp Expressions Menu

Figure 8-14

Menu Tools Pop-Up Window


-
- Back to Initial Suggestions** 8.5.1 The Back to Initial Suggestions command restores the Suggestions menus that were installed when the current application was initially selected. This command then resets the menu history (described in paragraph 8.5.4) to its initial value.
-
- Find Commands** 8.5.2 Find Commands can be executed in Zmacs or in any UCL-based application. Its purpose is to find all commands of the applications that contain a substring specified by the user. A menu of the commands is also created and installed in the main menu pane, and the name of the newly created menu is added to the List Menus and Menu History listings so that the user can return to the menu easily. A menu created in this way behaves exactly like any other Suggestions menu; items are executed in the usual way, and documentation is obtained for them in the usual way.
-
- Select Suggestions Applications** 8.5.3 You can display the Suggestions menus provided for some applications but not for others by executing the Select Suggestions Applications command, which displays a menu of the window-based applications for which Suggestions menus are defined. This is the same menu that is displayed when you originally enable Suggestions.
-
- Menu History** 8.5.4 Often, you want to return to a menu that you used earlier in the session. To help you find that menu more easily, the Menu History command displays in the main menu pane a listing of the names of menus you previously selected. You can return to any of these menus by boxing its name and clicking left. The Menu History can be installed either by selecting it through the Menu Tools option or by clicking left in the label pane when the option is displayed there.
-
- Suggestions Menu Search** 8.5.5 The Suggestions search capability permits the system to traverse the menu hierarchy and to stop at the first occurrence of a menu item that contains a string of characters you specify. You then decide whether to use this item or press HYPER-SUPER-S to continue to search for the next occurrence of that same string.
-
- Suggestions Menus Off** 8.5.6 The Suggestions Menus Off command turns off Suggestions for all applications. This option is also displayed in the lower small pane of the Listener Suggestions.
-
- Add Menu to Buffer** 8.5.7 The Add Menu to Buffer command is useful when some of the menus have been customized or newly created during a session and you want to use them at a later session. The Add Menu to Buffer command creates a new Zmacs buffer and inserts into it a Lisp form that corresponds to the menu currently installed in the main menu pane. You save this buffer to a file in your user directory. You restore it during a later session by explicitly loading the file, using the load function or by including a load form in your LOGIN-INIT file so that the file is automatically loaded when you log in.

Turn Off Pop-Up Keystrokes 8.5.8 Executing the Turn Off Pop-Up Keystrokes command suppresses the small window that momentarily displays the corresponding keystroke for a menu item that you have selected.

Turn On Pop-Up Keystrokes 8.5.9 Executing the Turn On Pop-Up Keystrokes command enables the display of a small window that briefly shows the keystroke for a menu item that you have selected.

Reorder Menu Items 8.5.10 The Reorder Menu Items command permits you to reorder the items installed in the main menu pane. The command pops up a menu of the items and shows instructions in the mouse documentation window that describe the mouse clicks used to reorder the menu. You can then use the Add Menu to Buffer command to store the menu in its new order in a file to be reloaded in the next session (see Add Menu to Buffer described previously).

Find Functions for Menu 8.5.11 The Find Functions for Menu command searches the Lisp environment for function names that contain a text string you specify. The symbols that are found are installed in the main menu pane, and the name of the menu is added to the Lisp Expressions menu.

Remove Current Menu 8.5.12 The Remove Current Menu command removes the menu that is installed in the main menu pane and removes its name from the List Menus listing for the selected application.

Add a Symbol to a Lisp Expressions Menu 8.5.13 The Add a Symbol to a Lisp Expressions Menu command allows you to add a symbol that you specify to the Lisp Expressions menus.

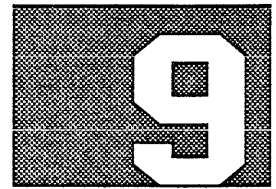
Lisp Expressions 8.6 Lisp Expressions menus contain symbols that you would otherwise type into an application input window. When editing and testing software, you often type certain symbol names repeatedly. To help you avoid both typographic errors and waste programming time caused by repeatedly typing the same expressions, a set of menus called *Lisp Expressions* is included with the standard set of Suggestions menus. When you select a symbol from a Lisp Expressions menu, the symbol is entered in the application window at point, as if it were typed. If you are editing a file, for example, the symbols that you select from a Lisp Expressions menu are inserted in the editor buffer at the keyboard cursor position.

In any context in which Lisp forms can be evaluated, those forms can be built by selecting items from Lisp Expressions menus. The default set of Lisp Expressions menus provides commonly used system function names and global variables; for example, Figure 8-15 shows the System Symbols menu. You can also create new Lisp Expressions menus or add symbols to existing ones. As with command menus, the changed Lisp Expressions menus can be saved in a file and reloaded at the beginning of each new session so that they only need to be created once.

Figure 8-15 Lisp Expressions Menu

<p>> ■</p>	<p>Lisp: Suggestions Menu Off</p>
	<p>Help Lisp Expressions List Menus Menu Tools</p>
<p>Lisp Listener 1 L: Execute Command, M: Command Keystroke, R: Documentation</p>	<p>System Symbols Menu TV:ALL-THE-SCREENS SYS:ACTIVE-PROCESSES SYS:ALL-PROCESSES CURRENT-PROCESS TV:MOUSE-PROCESS SUGG::*ALL-PACKAGES* *PACKAGE* Previously Selected Wind TV:PREVIOUSLY-SELECTED-W TV:WINDOW-RESOURCE-NAMES TV:*TERMINAL-KEYS* TV:*SYSTEM-KEYS* TV:SELECTED-WINDOW TV:SELECTED-IO-BUFFER TV:MOUSE-WINDOW *STANDARD-INPUT* *STANDARD-OUTPUT* TV:INITIAL-LISP-LISTENER TV:DEFAULT-SCREEN TV:MAIN-SCREEN TV:WHO-LINE-SCREEN W:*SYSTEM-MENU-PROGRAMS W:*SYSTEM-MENU-THIS-WIN W:*SYSTEM-MENU-WINDOWS-</p>

PROGRAMMING SUGGESTIONS



Introduction

9.1 Suggestions menus provide users with menus of commands appropriate for each phase of an application program. As the user moves through the various utilities, Suggestions provides menus of relevant commands for each utility, organized by topic.

The menus allow the user to view the keystrokes that execute the command, view command documentation, and execute a command from the menu with the mouse. Menu items can be submenus, lists of commands or Lisp expressions, or helpful messages. The user can customize the menus to meet a particular need.

This section tells how to add Suggestions to your application. The following topics are discussed:

- Incorporating Suggestions in an application — Provides an overview on incorporating Suggestions into your application. An example is provided.
- `sugg:suggestions-build-menu` function — Tells how to build a menu to use in Suggestions.
- `sugg:initialize-suggestions-for-application` function — Tells how to provide the information that the command loop must have so that it can supply a set of Suggestions menus for an application.
- Triggering automatic menu changes — Describes how to program the automatic changing of Suggestions menus depending on context (that is, what the user is doing at the time).

Incorporating Suggestions in an Application

9.2 Any application program that uses commands can use Suggestions menus. Typical commands include UCL-based commands built using the `defcommand` macro and Zmacs editor commands built using the `zwei:defcom` function. Your application should be built before you attempt to incorporate Suggestions.

Incorporating Suggestions menus in an application program is essentially a two-part process. After you build the application, you build the Suggestions menus, and then you initialize them. Optionally, you can trigger automatic menu changes based on context.

Building the Suggestions Menus

9.2.1 You should use the `sugg:suggestions-build-menu` function to build the menus for the application. This function allows you to create the item lists for each of the Suggestions menus and to organize them into a hierarchy of menus.

The symbol at the peak of the hierarchy has as its item list the user-selectable menus of the application. Your program must pass the symbol for the name of this menu to the `sugg:initialize-suggestions-for-application` macro. (However, if your application has a simple menu structure that does not have a hierarchy, you do not need to supply this symbol.)

Initializing Suggestions

9.2.2 Use the `sugg:initialize-suggestions-for-application` macro to initialize Suggestions for your application. You should specify the following, which are described in detail in paragraph 9.4, `sugg:initialize-suggestions-for-application` Macro:

- The flavor of the application window or, if there is none, the name of the command loop function.
- Names of the initial Suggestions menus to be displayed for your application.
- Optionally, the symbol that represents the list of menus that the user can select (defined by the `sugg:suggestions-build-menu` function). If your application has a simple menu structure that does not have a hierarchy, you do not need to supply this symbol.
- Optionally, a predicate function to be evaluated that indicates whether Suggestions should be displayed for the application under a particular set of circumstances.

Optionally, you can trigger automatic menu changes based on context. You can choose from three different procedures:

- You can use the inline macros `sugg:declare-suggestions-for` and `sugg:with-suggestions-for` within a command definition. When the user executes the command, the menus are changed.
- You can use `advise` declarations, which wrap around a command definition (`defcommand` in this case) without changing it. Again, when the user executes the command, the menus are changed.
- If the execution of a `defcommand` installs a new set of active command tables by calling the `ucl:set-active-command-tables` function, you can install a corresponding set of Suggestions menus whose items are the commands of the newly installed command table. You define an association list that maps command tables to sets of Suggestions menus. While a command table on the list is active, the menus mapped to it are installed in Suggestions menu panes. The symbol to which the association list is bound is given as the `context-switch-tables-alist` argument to the `sugg:initialize-suggestions-for-application` macro.

**Example of
Building Suggestions**

9.2.3 The following code shows a simple example of implementing Suggestions menus. The example uses UCL to create the commands and command tables.

Suppose that you have defined the following commands using the `defcommand` macro and the `make-command` macro:

```
write buffer to a specified filename
save buffer to a (default) file
clear the buffer
print the buffer
load the buffer with a compiled file
load the buffer with an uncompiled program file
clear the screen (but keep the contents of the buffer)
insert a timestamp into the buffer
scroll down a window
scroll up a window
scroll down a line
scroll up a line
scroll to the beginning of the buffer
scroll to the end of the buffer
```

Further suppose that you decide to group these commands as follows:

Read and Write Commands

```
write buffer to a specified filename
save buffer to a (default) file
load the buffer with a compiled file
load the buffer with an uncompiled program file
```

Scrolling Commands

```
scroll down a window
scroll up a window
scroll down a line
scroll up a line
scroll to the beginning of the buffer
scroll to the end of the buffer
```

Miscellaneous Commands

```
clear the buffer
print the buffer
clear the screen (but keep the contents of the buffer)
insert a timestamp into the buffer
```

Then, you create the menus for these various groups of commands. Each of these groups is called from a single superior menu called `all-the-commands`.

```
(sugg:suggestions-build-menu 'read-and-write-commands
  "Commands to read or write information to or from the buffer."
  `(write-buffer save-buffer load-compiled-file load-program-file)
  'all-the-commands
  "Read and Write Commands")

(sugg:suggestions-build-menu 'scrolling-commands
  "Commands to scroll within the buffer."
  `(scroll-down-wd scroll-up-wd
    scroll-down-line scroll-up-line
    scroll-to-beginning scroll-to-end)
  'all-the-commands
  "Scrolling Commands")
```

```
(sugg:suggestions-build-menu 'miscellaneous-commands
  "Miscellaneous commands, such as clearing the screen or the buffer,
   or inserting a timestamp in the buffer."
  '(clear-buffer print-buffer clear-screen insert-time)
  'all-the-commands
  "Miscellaneous Commands")
```

Next, create the symbol at the peak of the menu hierarchy that your program will use to initialize Suggestions. This symbol has as its item list the user-selectable menus of your application.

```
(sugg:suggestions-build-menu 'all-the-commands
  "All the commands available for this application."
  '(read-and-write-commands scrolling-commands
    miscellaneous-commands)
  "All the Commands")
```

Now, create the small upper and lower panes of the Suggestions menu for your application:

```
(sugg:suggestions-build-menu 'my-top-menu-contents
  ("My Application" :no-select ignore))

(sugg:suggestions-build-menu 'my-bottom-menu-contents
  ("Help" #\HELP))
```

Finally, you initialize the Suggestions menu for your application. The `simple-example` argument is the application command loop function. Notice that the symbol at the peak of your menu hierarchy (`all-the-commands`) is one of the arguments to this function. In this case, the symbol is also used for the menu that is installed in the main menu pane.

```
(sugg:initialize-suggestions-for-application 'simple-example
  'my-top-menu-contents 'my-bottom-menu-contents
  'all-the-commands 'all-the-commands)
```

The Suggestions menus appear as follows:

My Application
Help
All the Commands
Read and Write Commands Scrolling Commands Miscellaneous Commands

Detailed Example
of Building
Suggestions

9.2.4 The following code shows a detailed example of implementing Suggestions menus. The example creates Suggestions menus for a Lisp Listener. Notice that the menu item lists for the `sugg:suggestions-build-menu` function are more complicated than the menu item lists in the example in paragraph 9.2.3, Example of Building Suggestions.

The items in these lists can be in the variety of different formats acceptable to the `w:menu-choose` function, which is described in the *Explorer Window System Reference*. Note that the Suggestions menus allow the following alternative format. The suggestions and `w:menu-choose` items are equivalent:

```
(string #\char option1 arg1)      ;; Suggestions item
(string :kbd #\char option1 arg1) ;; w:menu-choose item
```

Create the menus for the small upper and lower panes:

```
(sugg:suggestions-build-menu
  ^my-lisp-listener-top-menu ""
  ^(("Lisp:" :no-select ignore :font sugg:*bold-suggestions-font*)
    ("Suggestions Menu Off" :process-eval
      (sugg:turn-all-suggestions-menus-off)
      :documentation "Switch to a configuration without suggestions menus."
      :keystroke-string "TERM - SUPR-HELP"))
  nil)

(sugg:suggestions-build-menu
  ^my-lisp-listener-bottom-menu ""
  ^(("Help" #\Help :documentation "Explains how to get help on available
    programs,
    console operations, and command editing.")
    ("Lisp Expressions" :suggestions-menu sugg:lisp-expressions)
    ("List Menus" :suggestions-menu w:com-list-menus)
    ,sugg:*menu-tools-menu-item*)
  nil "Lisp Bottom Menu")
```

Then, create the symbol at the peak of the menu hierarchy that your program will use to initialize suggestions:

```
(sugg:suggestions-build-menu
  ^w:my-com-list-menus
  "Displays a list of all the suggestions menus that are available
  for Lisp Listeners."
  nil nil)
```

Next, create the menus for the various groups of commands:

```
(sugg:suggestions-build-menu
  ^my-basic-menu
  "The most basic command editing operations for Lisp Listeners"
  ^(("Clear Screen" #\Page :documentation
    "Clear all old input off the screen so that only the current input
    is visible.")
    ("Start Input Over" #\Clear-input :documentation
    "Delete the current input so that this command can be started over from
    scratch.")
    ("Forward" #\c-f :documentation "Move the cursor one character forward.")
    ("Backward" #\c-b :documentation "Move the cursor one character backward.))
  ^(w:my-com-list-menus))
```

```
(sugg:suggestions-build-menu
  ^my-delete-characters-menu
  "Move over or delete lisp forms."
  ^(("Rubout Last Char" #\Rubout :documentation
    "Delete the character before the cursor.")
    ("Delete Character" #\c-d :documentation
    "Delete the character currently under the cursor."))
  ^(w:my-com-list-menus))

(sugg:suggestions-build-menu
  ^my-move-cursor-menu
  "Move cursor on screen."
  ^(("Forward" #\c-f :documentation "Move the cursor one character forward.")
    ("Backward" #\c-b :documentation "Move the cursor one character backward.")
    ("Forward Word" #\m-f :documentation "Move the cursor forward one word.")
    ("Backward Word" #\m-b :documentation "Move the cursor backward one word.")
    ("Beginning of Line" #\c-a :documentation
    "Move the cursor back to the beginning of the current line.")
    ("End of Line" #\c-e :documentation
    "Move the cursor all the way to the end of the current line."))
  ^(w:my-com-list-menus))
```

Next, create a window flavor for the application:

```
(defflavor my-lisp-listener ()
  (w:lisp-listener)
  :settable-instance-variables)
```

Finally, initialize the Suggestions menus for your application. In this example, `my-lisp-listener` is the name of the application window flavor. The `my-basic-menu` is the menu displayed in the main menu pane when the user initially selects the application. The symbol `w:my-com-list-menus` is the symbol at the peak of the menu hierarchy. When the user selects List Menus from the small lower pane, `w:my-com-list-menus` is called. Then, instead of the `my-basic-menu` being displayed, the `my-delete-characters-menu` and the `my-move-cursor-menu` are displayed.

```
(sugg:initialize-suggestions-for-application my-lisp-listener
  my-lisp-listener-top-menu
  my-lisp-listener-bottom-menu my-basic-menu
  w:my-com-list-menus nil "My Lisp Listener")
```

Now, turn the Suggestions menus on by pressing `TERM SUPER-HELP` and selecting `Do It`. Notice that `My Lisp Listener` is one of the applications listed on the menu.

To access the application window, you can use the following:

```
(w:select-or-create-window-of-flavor ^my-lisp-listener)
```

Also, to create and access a new instance of the application window, you can use the following:

```
(send (make-instance ^my-lisp-listener) :select :expose)
```


The following display appears:

Lisp: Suggestions Menu Off
Help Lisp Expressions List Menu Menu Tools
<i>My Basic MENU</i>
Clear Screen Start Input Over Forward Backward

If you select List Menus, the following display appears:

Lisp: Suggestions Menu Off
Help Lisp Expressions List Menu Menu Tools
<i>My Com List MENUs</i>
My Basic MENU My Delete Characters MEN My Move Cursor MENU

**sugg:suggestions-
build-menu
Function**

9.3 The `sugg:suggestions-build-menu` function builds a menu that will be used in the Suggestions menus for your application.

`sugg:suggestions-build-menu` *menu-name* Function
&optional *menu-documentation items superior-menu*
menu-name-string filter

Creates a Suggestions menu by attaching all of the supplied information to the property list of the specified symbol *menu-name*.

Arguments: *menu-name* — The symbol name that is associated with the information for this Suggestions menu.

menu-documentation — A general description of the menu contents and other helpful hints about this menu or the application it supports. This string appears when the user requests documentation.

The menu documentation is a string of arbitrary length. The mouse documentation window displays as much of the documentation string as will fit in the window or up to the first newline character (whichever comes first). If the mouse documentation window does not display all the documentation, the message `*more documentation available*` appears beside the R mouse button symbol. The entire documentation appears in a pop-up window if the user clicks right.

items — The menu item list, where items can be in a variety of different formats acceptable to the `w:menu-choose` function. (For information on the `w:menu-choose` function, refer to the *Explorer Window System Reference*.)

Note that the Suggestions menus allow the following alternative format. The suggestions and `w:menu-choose` items are equivalent:

`(string #\char option1 arg1)` ;; Suggestions item

`(string :kbd #\char option1 arg1)` ;; w:menu-choose item

superior-menu — The name of the menu that is this menu's immediate superior in the menu hierarchy.

menu-name-string — The name to be displayed for this menu. Supply this option when the displayed menu name should be different from the symbol passed in *menu-name*, or when that symbol is too long to fit in the label pane.

filter — A function name that you can include to check the items in *items* to ensure they are of the desired form.

Refer to paragraph 9.2.3, Example of Building Suggestions, for an example of using `sugg:suggestions-build-menu`.

sugg:initialize-suggestions-for-application Macro

9.4 The `sugg:initialize-suggestions-for-application` macro provides the information needed by the command loop to supply a set of Suggestions menus for an application.

`sugg:initialize-suggestions-for-application` Macro
selectable-window-or-command-loop-function
 &optional *upper-small-menu lower-small-menu main-menu*
list-menus-symbol context-switch-tables-alist name-for-display
 &key (:predicate '(suggestions-visible-for-applications?))

Adds the information specified by its arguments to the lists to which Suggestions refers when updating the menus in the Suggestions frame. (The lists are contained in the variables `sugg:function-wrapped-functions` and `sugg:suggestions-advised-functions`.)

Arguments: *selectable-window-or-command-loop-function* — Either the application window flavor or the application command loop function.

The UCL supports two types of applications:

- Window-based applications (such as the Lisp Listener, Zmacs, and the Inspector), which use a special window flavor created specifically for the application
- Applications that are not window-based (such as `break` and the Stepper), which run a command loop in a window of a generic flavor

If the flavor of the window is specific to the application (that is, you want the set of Suggestions menus to be displayed whenever a window of that flavor is selected), you should specify the window's flavor as the first argument. In this case, any time a full-sized instance of that window flavor is selected and Suggestions is enabled, the window is shrunk to allow room for the Suggestions.

If you do not specify a window flavor as the first argument, the argument should be the name of the application command-loop function.

upper-small-menu, lower-small-menu, main-menu — The symbols that name the menus that you want installed in the top small pane, in the lower small pane, and in the main menu pane when a user initially selects the application. You create these menus with the `sugg:suggestions-build-menu` function.

list-menus-symbol — The symbol name associated with the list of all menus to be used by the application. You create this symbol with the function `sugg:suggestions-build-menu`. (If your application has a simple menu structure that does not have a hierarchy, you do not need to supply this symbol.)

Find Commands adds new menus to this list of menus.

context-switch-tables-alist — An association list of the following form:

```
(defvar alist-name '((command-table-1 upper-small-menu-1
                             lower-small-menu-1
                             main-menu-1)
                    (command-table-2 upper-small-menu-2
                             lower-small-menu-2
                             main-menu-2)
                    .
                    .
                    (command-table-n upper-small-menu-n
                             lower-small-menu-n
                             main-menu-n)))
```

In the preceding, *alist-name* is a unique variable name, defined by the user, *command-table-1* through *command-table-n* are command tables of the application, and the menus (*upper-small-menu*, *lower-small-menu*, and *main-menu*) are application menus that are installed when the corresponding command tables are made active. (You create these menus with the `sugg:suggestions-build-menu` function.) The use of this association list is described in more detail in the paragraph 9.5.3, Changing Menus Through the Active Command Table.

name-for-display — The string to be displayed at the top of the upper small pane. Usually, this string is the name of the application.

`:predicate` — A function to be evaluated that indicates whether Suggestions should be displayed for the application under a particular set of circumstances.

For example, you may not want Suggestions displayed unless a certain variable *foo* has a value. In this case, pass the `:predicate` option with the argument `(variable-boundp foo)`.

Refer to paragraph 9.2.3, Example of Building Suggestions, and paragraph 9.2.4, Detailed Example of Building Suggestions, for examples of using `sugg:initialize-suggestions-for-application`.

Triggering Automatic Menu Changes

9.5 Your program should automatically install menus in the Suggestions menus panes to help the user when an application allows contexts where:

- Commands are allowed that are not normally part of the standard set of commands
- Certain standard commands are not allowed

Suggestions provides facilities to update menus in response to a context change triggered when the user invokes a certain command. You can use one of the following to change the Suggestions menus in response to the context of the current command:

- Using inline macros to change menus automatically — You can include the `sugg:declare-suggestions-for` and the `sugg:with-suggestions-menus-for` macros within your command definition to change menus. These macros expand inline.

- Advising commands to change menus automatically — You can use the `sugg:advise-function-to-push-all-menus` and `sugg:advise-function-to-push-one-menu` macros to advise a command (created by `defcommand`) to change menus automatically. These macros are wrapped around existing code without changing it as opposed to being inline.

Using macros that expand inline to change menus has one advantage over using the `advise` declarations. When you are attempting to debug code, inline macros are immediately apparent when you examine the source code for a function or command (for example, when you invoke a `META-` on a function from the Zmacs editor). The Suggestions `advise` declarations, on the other hand, are not obvious when you use the `META-` facility. Thus, code written using inline macros is often easier to debug than code written using `advise` declarations. You can examine the `sugg:function-wrapped-functions` and the `sugg:suggestions-advised-functions` variables to discover which functions have been advised.

However, when you change the code containing the inline macro, you must recompile both the inline macro and the function itself. If you change the code of an `advise` declaration, you need compile only the code for the `advise`. You can work around this difficulty by having the inline macro immediately call a helper function. Then, if you need to change the code of the helper function, you need compile only that helper function rather than both the inline macro and the function itself.

- Changing menus through the active command table — When a command definition of an application contains a call to `ucl:set-active-command-tables` to change command tables, you can have the Suggestions menus updated when the command is executed to display the commands for the new command table. You specify the Suggestions menus to be displayed in the `context-switch-tables-alist` argument to the `sugg:initialize-suggestions-for-application` macro.

In each case, the menus that are used in Suggestions menus are the menus that you previously created with the `sugg:suggestions-build-menu` function.

When you use the inline macros or the `advise` macros to change menus, the macros push one or more menus onto the Suggestions menu panes when the command is called and pop them off when the command terminates execution. When you use the `context-switch-tables-alist` argument for the `ucl:set-active-command-tables` case, the menus specified in the association list are pushed onto the Suggestions menu panes when a command adds a certain command table to the list of active command tables; the menus are popped off when some command removes that command table from the list.

**Using Inline Macros
to Change Menus
Automatically**

9.5.1 This paragraph describes how to use the `sugg:declare-suggestions-for` and the `sugg:with-suggestions-menus-for` inline macros to change menus automatically. Also, you can use the `sugg:undeclare-suggestions-for` function to undo the effects of the inline macros.

sugg:declare-suggestions-for *function*

Macro

```
&key :upper-menu :lower-menu :main-menu
:predicate :before :after :use-arglist :around
```

Provides a way to execute code for the benefit of Suggestions when *function* is called. The macro first executes `:before`, then executes the code within *function* that the `sugg:with-suggestions-menus` macro is wrapped around, and then executes `:after`.

All of the optional arguments are keyword-value pairs. For `:before`, `:after`, and `:around`, the values should simply be calls to functions. By putting the actual code in a function that is called, you can correct and compile the code separately, without compiling the higher-level function that contains the `sugg:with-suggestions-menus` macro.

:upper-menu, **:lower-menu**, and **:main-menu**— Symbols that name the menus that you want installed in the top small pane, in the lower small pane, and in the main menu pane when a user initially selects the application. You create these menus with the `sugg:suggestions-build-menu` function.

:predicate — An optional, additional test that a user might want performed before installing Suggestions. For example, a particular menu would not be installed unless a variable *foo* were bound to a certain value. The `:predicate` would be `(when (boundp foo 'value))`.

:use-arglist — When non-nil, the functions specified as `:before`, `:after`, and `:around` can use the arguments of the function for which you are defining suggestions. Suppose you define the following function:

```
(defun foo (arg1 list) (...))
```

The following shows how you can use *foo*'s arguments:

```
(sugg:declare-suggestions-for 'foo
 :before (before-foo function-arglist) ; function-arglist is
 ; foo's (arg1 list).
 :use-arglist t)
```

:around — If specified, `:do-it` *function* should be part of the code, referring to the *function* for which you are defining Suggestions.

sugg:undeclare-suggestions-for *function*

Function

Provides a way to remove Suggestions code that was executed when `sugg:declare-suggestions-for` was invoked for *function*.

sugg:with-suggestions-menus-for *function &body body*

Macro

Used with the **sugg:declare-suggestions-for** macro to provide Suggestions for a particular function. You declare Suggestions for a function before you define it, as shown in the following example:

Example 1: This example shows how you can avoid having to recompile both the inline macro and the function itself when you change the code of the inline macro. You can make the inline macro immediately call a helper function. Then, if you need to change the code of the helper function, you need compile only that helper function rather than both the inline macro and the function itself. (Because this helper function is called with the `:around` keyword, you must include a `:do-it` in the body of the helper function.)

For example, for the function `my-function1`, you could use the following code to switch contexts for the Suggestions menus:

```
;; First, declare the suggestions for my-function1.
(declare-suggestions-for my-function1
 :around '(helper-function))

(defun my-function1 ()
  (with-suggestions-menus-for my-function1
    (...body of my-function1...)))

(defun helper-function ()
  (body ...
   :do-it my-function1 ; The body must include a :do-it somewhere.
   ...))
```

Example 2: The next example provides a more complicated case.

```
(sugg:declare-suggestions-for my-function2
 :around '(wrapper-function-around-my-function2)
 :before '(do-this-function-first)
 :after '(do-this-function-last)
 :upper-menu 'my-top-menu
 :lower-menu 'my-bottom-menu
 :main-menu 'all-the-commands
)

(defun my-function2 ()
  (sugg:with-suggestions-menus-for my-function2
    (body-of-my-function2. . .)))

(defun wrapper-function-around-my-function2 ()
  (body. . .)) ; The body must include a :do-it somewhere.

(defun do-this-function-first ()
  (body. . .))

(defun do-this-function-last ()
  (body. . .))
```

When you execute `my-function2` with Suggestions enabled, the system first invokes `do-this-function-first`, then `wrapper-function-around-my-function2`, next `body-of-my-function2`, and finally `do-this-function-last`. (Notice that the `:around` function `wrapper-function-around-my-function2` must include `:do-it` in the body somewhere.) By putting the before, after, and around code in separate functions, you can easily change and recompile any of these functions without recompiling `my-function`. This advantage becomes very important when the code supporting Suggestions is in one system and the code containing `my-function2` is in a different system.

Advising Commands to Change Menus Automatically

9.5.2 If an application command definition (a `defcommand`) contains a loop in which commands are allowed that would not otherwise belong to the set of legal commands for the application, use either the `sugg:advise-function-to-push-all-menus` macro or the `sugg:advise-function-to-push-one-menu` macro to install those special commands in the Suggestions menu panes.

As mentioned earlier, debugging code written using `advise` declarations is often harder than debugging code written using inline macros. Inline macros are immediately apparent when you examine the source code for a function or command (by using `META-`), but the `advise` declarations are not. However, you can examine the `sugg:function-wrapped-functions` and the `sugg:suggestions-advised-functions` variables to discover which functions have been advised.

`sugg:advise-function-to-push-all-menus` *command* Macro
upper-small-menu lower-small-menu main-menu
 &optional (*predicate* '(suggestions-visible-for-application?))

Pushes the menus onto all of the Suggestions menu panes when the specified *command* (created by a `defcommand`) is invoked by the user. Thus, the previous menus are saved while the command is executing. When the command finishes executing, `sugg:advise-function-to-push-all-menus` restores the previous Suggestions menu.

Arguments: *command* — Either a function or a method defined in a `defcommand` or referenced in the `:definition` field of a `make-command`.

upper-small-menu, lower-small-menu, main-menu — The menus that you want installed in the top small pane, the lower small pane, and the main menu pane when the specified *command* is invoked by the user. You create these menus with the `sugg:suggestions-build-menu` function.

predicate — If a *predicate* argument is specified, the menus are installed only if the predicate evaluates to a non-nil value.

Example: The following example pushes the `debugger-top-menu`, `debugger-bottom-menu`, and `com-basic-debugger-menu` onto the Suggestions menu panes when the `eh:signal-microcode-error` command is invoked:

```
(sugg:advise-function-to-push-all-menus eh:signal-microcode-error
  debugger-top-menu debugger-bottom-menu com-basic-debugger-menu)
```

`sugg:advise-function-to-push-one-menu` *command main-menu* Macro
 &optional (*predicate* '(suggestions-visible-for-application?))

Similar to `sugg:advise-function-to-push-all-menus` except that a single menu is installed in the main menu pane when *command* is executed, while the two smaller panes remain unchanged. The menu is removed from the main menu pane when the command finishes executing. The arguments are the same as those for `sugg:advise-function-to-push-all-menus`.

This function is useful for installing Suggestions that are relevant when a particular function or method is running. (For example, when a user is performing an Incremental Search in the Zmacs editor.)

Example: The following example installs a single menu (`eh-help-characters-menu`) in the main menu pane when the `com-help` command is executed:

```
(advise-function-to-push-one-menu com-help eh-help-characters-menu)
```

**Changing Menus
Through the Active
Command Table**

9.5.3 You can define a variable, such as *alist-name*, to map command tables to menus. If a command definition of the application contains the following function call, you may want to have the Suggestions menus updated when this command is executed by a user so that the Suggestions panes contain commands for the newly installed command tables:

```
(ucl:set-active-command-tables 'one-or-more-command-tables)
```

To do this, you need to define an association list and pass the symbol for the association list to the *context-switch-tables-alist* argument to the `sugg:initialize-suggestions-for-application` macro.

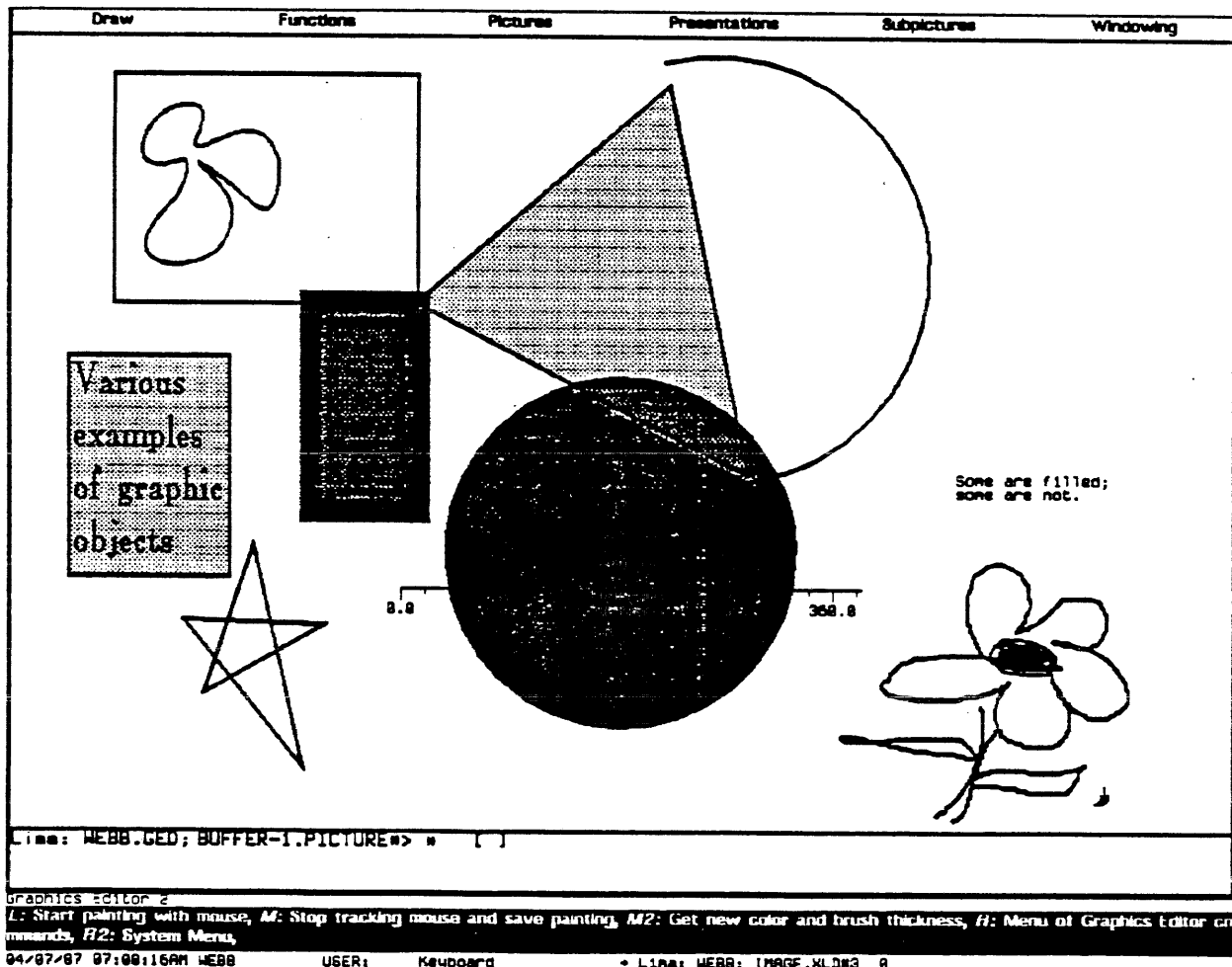
NOTE: For this association list to work properly, you must use the function `ucl:set-active-command-tables` in your command code to change the active command tables.

GRAPHICS EDITOR

Introduction

10.1 The graphics editor is an interactive utility for creating and modifying pictures consisting of graphic objects. Objects, once created, are both drawn on the video display and exist in a graphics database known as a *world*. Thus, when you clear the screen, the image of the object disappears, but the object itself still exists in memory. To display the object again, you draw it rather than recreating the object.

The graphics editor is based on the graphics window system. Pictures created with one can be used by the other. However, each has different uses. For instance, you could create a picture interactively using the graphics editor, and then use the picture in an application program which uses the graphics window system.



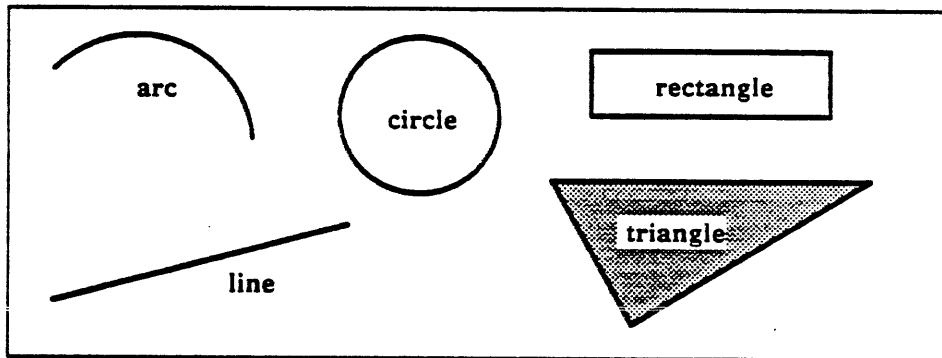
Worlds, Windows, and Transformations

10.1.1 A *world* is an infinite, two-dimensional plane that can contain graphic objects. World coordinates are arbitrary units that you can determine. Distances, sizes, offsets, and thicknesses are measured in world coordinate units. In the graphics editor, each buffer displays a distinct world.

A window gives a limited view of a world; it literally provides a viewport in which you can see the contents of the world. A world can contain objects that are positioned or sized so you cannot display all the objects in a window simultaneously.

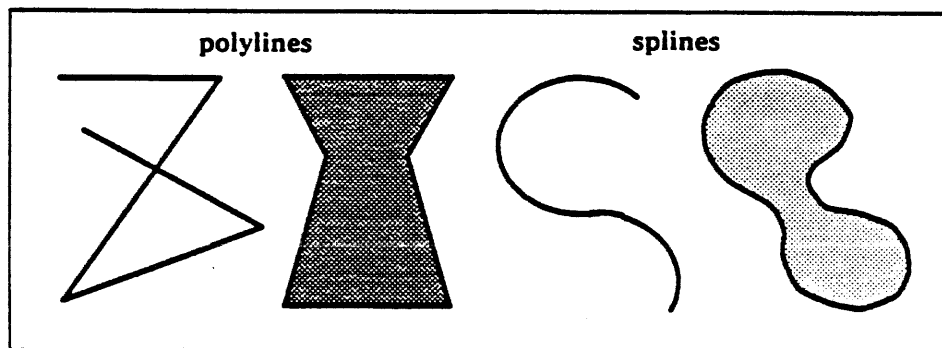
Kinds of Objects

10.1.2 The graphics editor manipulates objects. Some objects are simple geometric shapes: arcs, circles, lines, rectangles, and triangles.



Other objects require some explanation:

- *Paintings* are raster objects that you create by *painting* the image on the video display.
- *Polylines* are several straight lines joined end to end. When a polyline is closed (that is, when the end point of the last line is the same as the starting point of the first line), it is a polygon.
- *Rulers* are graphic objects that aid you in drawing and can be used to label graphs.
- *Splines* are smooth curves, such as sine curves.
- *Text objects* are strings of text on a background used to label figures, provide comments, and so on.



Grouping Objects 10.1.3 Objects in a picture can be grouped together into collections called *subpictures*. For example, you might group the various objects that comprise the image of a house into a single subpicture. You do operations, such as move, delete, copy, or save, on a subpicture as a unit.

In addition, every object—including subpictures—in the current window is part of a single *picture*. You can group a sequence of pictures into a *presentation* that can be saved as a single unit for later use.

Aids for Drawing Objects 10.1.4 The graphics editor offers two additional features to help you manipulate objects: a grid and background pictures. The grid is a grid of dots that appears on the video display to aid you in aligning objects. You can specify whether the grid appears and the spacing of the units. If you print the contents of a picture, the grid is not printed.

Background pictures are pictures loaded into the world for reference or as template figures. A background picture is similar to a subpicture except that background pictures cannot be selected or edited.

Loading and Entering the Graphics Editor 10.2 To use the graphics editor, you must first load the graphics window system (GWIN), the graphics editor (GED) system, and then enter the editor.

Creating the GWIN and GED Systems 10.2.1 Before you can enter the graphics editor, you must load both GWIN and GED by using the `make-system` function. In general, you load the systems by evaluating the following forms:

```
(make-system 'gwin :noconfirm)
(make-system 'ged :noconfirm)
```

The forms cause the system to load the files that comprise the graphics window system and the graphics editor. Once the files are loaded and the GED system is created, you can enter the graphics editor.

Note that you can avoid having to create the graphics editor each session by saving a band that includes the graphics editor. See the *Explorer Input/Output Reference* for information about saving a band.

ged:*save-bits-for-buffers*

Variable

When this variable is `t`, a bit-save array is created when new graphics editor buffers are constructed. A new graphics editor buffer is created for each picture file that you load (read in).

When this variable is `nil`, a bit-save array is not created. Therefore, memory allocation can be eliminated. Bit-save arrays are very large on color systems, but they are not a problem on monochrome systems. Also note that if you set this variable to `nil`, redrawing the graphics editor screen may take longer than usual.

For more information on bit-save arrays, refer to the *Explorer Window System Reference*.

Entering the Graphics Editor 10.2.2 You can enter the graphics editor in two ways: by using the SYSTEM key or by using the `ged:ged` function.

Using the SYSTEM Key 10.2.2.1 Press SYSTEM G to start the graphics editor or to enter the current graphics editor window. Press SYSTEM CTRL-G to create a new graphics editor window.

Using the ged:ged Function 10.2.2.2 You can use the `ged:ged` function to enter or create graphics editor windows.

`ged:ged` &optional *pathname-or-object* Function

Enters and creates graphics editor windows. *pathname-or-object* can be either the pathname of a file to be read into a graphics buffer, or an object or list of objects to be inserted into a new buffer. If you do not specify *pathname-or-object*, the function creates a new graphics editor window.

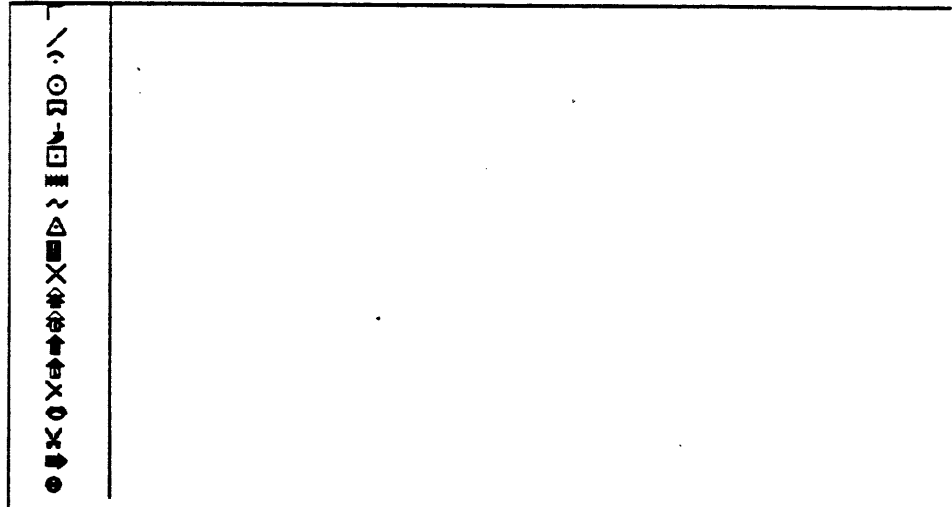
Graphics Editor Window 10.3 The window that the graphics editor uses has several features that you should be familiar with. You can change the layout of the window so that different menus and panes are displayed. The mouse and keyboard can both be used to select commands and to move the mouse cursor. In addition, you can select commands in four different ways. Finally, the graphics editor uses buffers to hold the pictures that you are working on.

Window Layout 10.3.1 The graphics editor has several different layouts that you can select with the Status command. Three components can be combined with the graphics pane: icon menu, submenus, and file and error information. Refer to paragraph 10.5.8, Status Variables, for information about selecting the window layout.

The following figure shows the default layout after the user clicked right to invoke the command menu. Clicking on an entry in the menu of submenus displays only that submenu of commands; you can then select from that submenu. The file and error information pane, similar to a Zmacs mode line and minibuffer, contains the name of the current buffer, error messages, the name of the current presentation, and prompts for information.

Draw	Functions	Pictures	Presentations	Subpictures	Windowing
<i>Draw</i>	<i>Functions</i>	<i>Pictures</i>	<i>Presentations</i>	<i>Subpictures</i>	<i>Windowing</i>
Arc Mode	Copy Objects Mode	Clear Backgrnd Picture	Define Presentation	Define Subpicture	Pan Down
Circle Mode	Delete Objects Mode	Clear Picture	Display Presentation	Explode Subpicture	Pan Down Half
Line Mode	Drag Move Mode	Find Picture	Kill Presentation	Insert Subpicture	Pan Left
Paint Mode	Drag-Copy Objc Mode	Insert Picture	Kill/Save Presentations	Restore Subpicture	Pan Left Half
Polyline Mode	Edit Parameters	Kill or Save Pictures	List Presentations	Save Subpicture	Pan Right
Rectangle Mode	Edit	Kill Picture	Load Presentation	Undefine subpicture	Pan Right Half
Ruler Mode	Move Objects Mode	List Pictures	Modify Presentation		Pan Up
Spline Mode	Restore User Status	Next Picture	<u>Restore Presentation</u>		Pan Up Half
Text Mode	Revert Picture Status	Previous Picture	Save Presentation		Set rewindow area
Triangle Mode	Revert User Status	Print Graphics			Show default window
	Save User Status	Read Background Picture			Show entire picture
	Scale Objects Mode	Redraw Picture			Zoom In
	Undo	Reorder Pictures			Zoom In Two
	View or Modify Status	Revert Picture			Zoom Out
		Save Picture			Zoom Out Two
		Write Picture			
Line: WEBB; BUFFER-I.PICTURE> ()					
GRAPHICS EDITOR Restore presentation definition from file Key assignments: SUPER-H					
04/07/87 05:46:41AM WEBB USER: Menu_choose * Line: WEBB; IMAGE.XLDR1 0					

When you select a combination that includes the icon menu, shown in the following figure, you can select commands that use icons by clicking on the icon for that command. You can use the keyboard or the graphics editor command menu to select other commands. Use the graphics only layout when you want to see how the picture will look on the full screen.



After you become familiar with the keystrokes for most of the commands and modes, you probably will not need the icon menu or submenu, but the file and error information is always useful.

Buffers 10.3.2 The graphics editor uses buffers the way that the Zmacs editor does; while you are using a picture, that picture is stored in a graphics editor buffer. Each graphics editor can have several buffers. You can save the pictures in these buffers to files for later use.

A buffer contains a picture and some information that is related to drawing that picture. This information includes the following:

- The current command mode
- The list of commands that can be undone
- Whether the buffer has been saved, what file the buffer has been saved in, and whether the buffer has been modified since it was read into a buffer
- Both the initial and current status variable values
- Whether you have panned or zoomed

This information is saved with the picture when you save the buffer to a file. Status variable values are automatically restored when you read the file into a buffer.

The buffer commands are described in paragraph 10.6, Pictures; the status variable commands are described in paragraph 10.5.8.

-
- Types of Commands 10.3.3 Commands in the graphics editor fall into six groups:
- The Draw commands select the type of object you will draw.
 - The Function commands alter the parameters of graphics objects, buffers, and commands.
 - The Pictures commands perform operations on the whole picture.
 - The Presentations commands are used for keeping sets of related files. The files in a presentation can be loaded into individual buffers by using one command.
 - The Subpictures commands are used to define several objects to be one object. You can save subpicture definitions for access during a later session.
 - The Windowing commands change the position of your window into the graphics world.
-
- Selecting Commands 10.3.4 You can select most commands in four ways: using the graphics editor command menu, using the graphics editor command submenus, using the icon menu, or using the keyboard.
- Using the Graphics Editor Command Menu* 10.3.4.1 Click right once to display the graphics editor command menu. Click on any command name to select that command, or move the mouse cursor off to remove the menu. The commands are displayed in six groups: Draw, Functions, Pictures, Presentations, Subpictures, and Windowing. Most commands are on this menu.
- Using the Graphics Editor Submenus* 10.3.4.2 If the layout of your graphics editor window includes the submenu menu, you can click on the submenu names to display submenus. Six submenus are available. Each submenu contains the commands from that group in the graphics editor command menu. Click on any command name to select that command, or move the mouse cursor off to remove the submenu. Most commands are available through the submenus.
- Using the Icon Menu* 10.3.4.3 If your window layout includes the icon menu, you can click on the icons to select commands. All commands that have icons are available through the icon menu.
- Using the Keyboard* 10.3.4.4 All commands can be selected from the keyboard. Refer to Table 10-5 at the end of this section for a list of key sequences associated with graphics editor commands.
-
- Mouse and Keyboard Operation 10.3.5 You can move the mouse cursor in the graphics editor by using either the mouse or the arrow keys. The buttons on the mouse and the LEFT, MIDDLE, and RIGHT keys on the keyboard can be used to select points and objects and to make selections on menus.
- The Mouse Cursor* 10.3.5.1 To move the mouse cursor, you can use the mouse or the arrow keys. The arrow keys move the mouse cursor one *unit*. A unit is a pixel when the grid is off or a grid point when the grid is on. Using SUPER with an arrow key moves the mouse cursor two units, and using HYPER with an arrow key moves the mouse cursor three units.
-

Several different icons are used to represent the mouse cursor; for instance, when you are drawing circles, the mouse cursor is a circle icon.

The Mouse Buttons

10.3.5.2 The LEFT, MIDDLE, and RIGHT keys on the keyboard correspond to the left, middle, and right buttons on the mouse. For example, to click right, you can press the RIGHT key on the keyboard or click the right button on the mouse.

The specific role of the mouse buttons is determined by the current function. The operations performed by the mouse buttons are listed in the mouse documentation window.

In general, each mouse button has the same role throughout the graphics editor:

Mouse Click	Description
Clicking right once	Invokes the graphics editor command menu
Clicking right twice	Invokes the System menu
Clicking middle once	Selects an object In some drawing modes, indicates that the figure is open
Clicking middle twice	Releases an object In some drawing modes, indicates that the figure is closed
Clicking left once	Selects a point or object
Clicking left twice	Releases a point or object

When the mouse cursor is over a menu item, you can usually click any of the buttons to select that menu item.

Objects

10.4 The graphics editor allows drawing of ten types of objects: arcs, circles, lines, paintings, polylines, rectangles, rulers, spline curves, text, and triangles.

The following pages describe the general information you need to know about how to draw an object. Be sure to read paragraph 10.4.2, Drawing an Object, to learn about the different ways you can draw an object. Then, read a particular paragraph to learn specifics about drawing a particular kind of object.

Characteristics of Objects

10.4.1 The characteristics of objects that you draw are determined by several features of the graphics editor. First, the status variables for the buffer determine how thick the edge is, what color the edge is, whether the object is filled, what font the text is in, and so on. Second, the type of object that you draw is determined by the command you have selected from the Draw menu. Last, the points that you select determine the position and size of the object.

Filled and Unfilled Objects

10.4.1.1 Most objects can be filled (their interiors are shaded) or unfilled (their interiors are clear). Some objects, such as text or rulers, are not actually filled; instead, a background rectangle large enough to enclose the object is filled instead. This rectangle can have both a fill color and an edge color and thickness.

Filled Objects The interior of filled objects is drawn in a single color called the *fill color*. The fill color can be the same as the edge color, but it is usually different.

Unfilled Objects Only the edges of unfilled objects are drawn. This edge can have an *edge color* and a *thickness*. The thickness is the number of units in the edge of the object. You can specify a thickness so large that the edges overlap and the object appears to be filled.

Color of Objects 10.4.1.2 The possible values for these colors in a monochrome environment are listed in Table 10-1.

In a color environment, you can specify the value or name of any color in the color map. The first 32 slots of the color map, which are listed in Table 10-2, contain eight shades of gray (for compatibility with the eight gray or stipple patterns in the monochrome environment) plus commonly named colors. The color map contains numerous other colors that you can choose.

Table 10-1









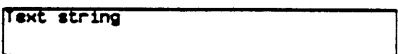
Color Values for Graphic Methods		
Value	% Gray	Screen Representation
8, w:black	Black	 100%-black
7, w:88%-gray-color	88% gray	 88%-gray
6, w:75%-gray-color	75% gray	 75%-gray
5, w:66%-gray-color	66% gray	 66%-gray
4, w:50%-gray-color	50% gray	 50%-gray
3, w:33%-gray-color	33% gray	 33%-gray
2, w:25%-gray-color	25% gray	 25%-gray
1, w:12%-gray-color	12% gray	 12%-gray
0, w:white	White	 100%-white
nil	No color, transparent	

Table 10-2

Named Colors in the Default Color Map	
Index Number	Color Name
0	w:white
1	w:12%-gray-color
2	w:25%-gray-color
3	w:33%-gray-color
4	w:50%-gray-color
5	w:66%-gray-color
6	w:75%-gray-color
7	w:88%-gray-color
8	w:black
9	w:dark-green
10	w:blue
11	w:red
12	w:orange
13	w:purple
14	w:pink
15	w:cyan
16	w:magenta
17	w:yellow
18	w:dark-blue
19	w:green
20	w:dark-brown
21	w:blue-green
22	w:light-brown
23	w:red-purple

Note:

Locations 24 through 31 are reserved for future use.













Note the following points when using color in your pictures:

- If you are on a color system and you load picture files created on a monochrome system, the gray shades (stipple patterns) from the monochrome system become true gray shades on the color system. Thus, the pictures may appear different. Pictures that are created on a color system can be loaded on a monochrome system; the colors are mapped to the monochrome stipple patterns.
- A picture that is created on a color system and that contains paint (raster) objects or rasterized subpictures does not display properly on a monochrome system.
- If you intend to display a picture on both color and monochrome systems, the colors and ALUs for each object should be carefully considered. For example, the w:opposite ALU should not be used when creating a picture on a monochrome system because of the way w:opposite works on a color system.

ALU Value 10.4.1.3 The arithmetic logic unit (ALU) value determines what happens when the system combines the pixels already on the display with the pixels in the object to be drawn.

Table 10-3 discusses the effects of the most common ALU values. In the center column, the first square shows the input that is drawn on the screen; the second square shows what the screen looks like to begin with; the third square shows the effects of the ALU values on the display.

Table 10-3 Common ALU Values

ALU Value and Boolean Name	Input, Screen, Result	Description for a Monochrome System
w:combine or Inclusive-or	  	Merges the object being drawn with the existing object on the window. w:combine turns pixels on if either input value is on; otherwise, the pixel remains off.
w:erase And-with-complement	  	Turns pixels off if the input is on; otherwise, w:erase has no effect on the pixels. w:erase is useful for erasing areas on the window, particular characters, or graphics.
w:normal Set-all	  	Copies the object being drawn without regard to the existing image on the window.
w:opposite Exclusive-or	  	Turns on pixels if only one of the input values is on; otherwise, w:opposite turns the pixel off. A common use of w:opposite is to erase an object after it has been drawn on the screen.

The default ALU value for all the commands is **w:normal**. Other ALU values are available, including color ALU values; these are discussed in the *Explorer Window System Reference*.

The same ALU operations that are available in the monochrome environment are also available in the color environment, as well as other operations. However, the standard monochrome operations become less valuable because they are designed as binary operations. In the color environment, binary operations are also performed on a bit-for-bit basis. Thus, the source and destination are well-defined mathematically. When the result of this operation is interpreted as a color, however, the result is not so well-defined. For example, XORing an ON pixel with an ON bit yields an OFF pixel. But, what does it mean to XOR a green pixel with a pink value?

These color ALU operations depend on four values: the (color) value of the pixel already drawn on the window, the (color) value of the pixel to be drawn, the ALU operation, and the color map. In general, color ALU operations take the two color values, apply an ALU operation to them, then use the resulting value as an index to the color map table. Thus, in a color environment, the same values combined using the same ALUs can result in different colors depending on the color map.

Color ALU operations are add, subtract, max, min, add with saturation, subtract with clamping, transparency (**w:combine**), and background (**w:erase**). These operations are briefly described in Table 10-4.

Table 10-4 Color ALU Operations

Operation	Description
w:alu-add	The source and destination values are added, and the result is allowed to wrap around 255.
w:alu-adds	The result is the smaller of the saturation value or the sum of the source and destination.
w:alu-avg	The source and destination values are averaged. Any fractions are truncated.
w:alu-back	The result is the background color, regardless of the value of the source or destination.
w:alu-max	The result is the larger value of the source or destination.
w:alu-min	The result is the smaller value of the source or destination.
w:alu-sub	The source and destination values are subtracted, and the result is allowed to wrap around 0.
w:alu-subc	The result is the larger of the clamp value or the difference between the source and destination values.
w:alu-transp	The result combines the source and destination values according to the rules of transparency. Refer to the Graphics and Using Color sections in the <i>Explorer Window System Reference</i> for details on transparency.

Status Variables 10.4.1.4 For a complete list of the status variables that affect a type of object, use the Edit Parameters command to display a window that lists the parameters for an object of that type. Parameters that are determined by status variable values are listed in this window. Other parameters are also listed in this window, for example, coordinates for the object.

Refer to paragraph 10.5.8, Status Variables, for complete information on setting the status variables and on the effects of the status variables.

Drawing an Object 10.4.2 The characteristics of objects that you draw are determined by several features of the graphics editor. First, the status variables for the buffer determine how thick the edge is, what color the edge is, whether the object is filled, what font the text is in, and so on. Second, the type of object that you draw is determined by the command you have selected from the Draw menu. Last, the points that you select determine the position and size of the object.

Choosing a Function 10.4.2.1 When you need to add an object to your picture, select a drawing function. Choose one from the Draw submenu, select an icon from the icon menu, or enter a command from the keyboard. At this point, the mouse cursor changes to the icon that represents the selected drawing mode, and the mouse documentation window shows the assignments for each mouse button.

Selecting and Positioning the Object 10.4.2.2 Use the mouse to select points and to replace points on the screen. You can also move points until you are satisfied with the object.

For information about drawing a specific type of object, refer to the paragraph in this section on that type of object.

Selecting the Points Use the mouse to select points on the screen. To select a point, position the mouse cursor, and click left. When you select a point, a marker appears at the location you selected.

Replacing a Point If you want to change the location of the last point you selected, click left twice, and the last point you selected is moved to the current position of the mouse cursor.

Rubber Banding Rubber banding lets you move points on the screen while you are drawing some types of objects. While the button is held down, the object changes shape as you move the mouse cursor. To use rubber banding, select the first point, and then hold the left button down while you move the mouse cursor to the next point. As you move the mouse, the shape drawn on the screen will change. Release the left button when the shape is correct. You can also use rubber banding on the succeeding points if the object requires more than two points

Illustration Conventions 10.4.3 The illustrations that describe how to use the graphics editor commands use the following conventions:

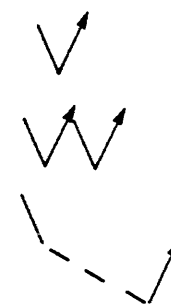
- Numbers in the illustration correspond to specific mouse operations. These operations are explained in numbered steps beneath the art. The instructions for each step tell you which mouse button to use. A typical instruction is "Click left to set the center of the circle."

- Arrows indicate how you use the mouse button.

Clicking a mouse button once is shown by a single angled arrow.

Clicking a mouse button twice is shown by two angled arrows.

Pressing a mouse button and holding it while you move the mouse is shown by a dashed line joining the two parts of the angle.



Arcs 10.4.4 To start drawing arcs, click on Arc in the Draw submenu, select the arc icon from the icon menu, or press META-A. The mouse cursor becomes the arc icon. The dot in the arc icon indicates the point that is set when you click left.

Arc Definition 10.4.4.1 An arc is defined by the following:

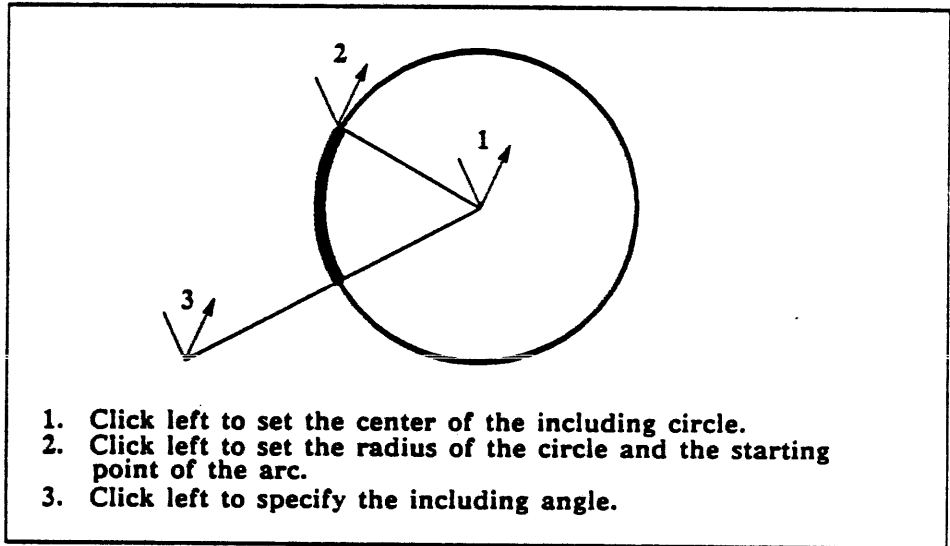
- The center of the circle that includes the arc
- The radius of that circle and the point where the arc begins
- The angle of the arc

The Fill Color status variable determines whether the object drawn is an arc or a sector. Refer to paragraph 10.5.8, Status Variables, for information on the Fill Color status variable.

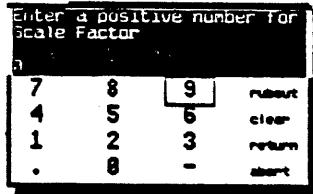
Drawing an Arc 10.4.4.2 The first left click sets the center of the circle that contains the arc. The second left click sets the radius of the circle and the starting point for the arc. Arcs are drawn counterclockwise from the second point. You can use rubber banding while you are selecting the second point to see the circle.

The third step is to specify the included angle. You can use the mouse or the keyboard to specify the angle.

- To use the mouse, select a third point; the arc ends at the point where the circle intersects the line from the center of the circle to the third point, as shown in the following figure. You can use rubber banding to select the third point.



- To specify the angle numerically, click middle to pop up a numeric pad (shown in the following figure). Specify the angle in degrees.

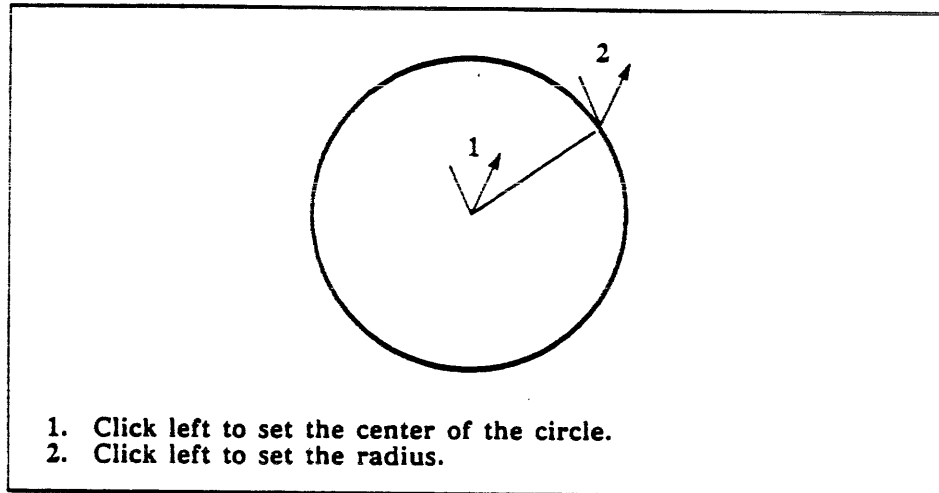


Circles 10.4.5 To start drawing circles, select Circle on the Draw submenu, click on the circle icon in the icon menu, or press META-C. The mouse cursor becomes the circle icon. The dot in the center of the icon indicates which point is selected when you click left.

Circle Definition 10.4.5.1 A circle is defined by two points:

- The center of the circle
- A point on the circumference of the circle

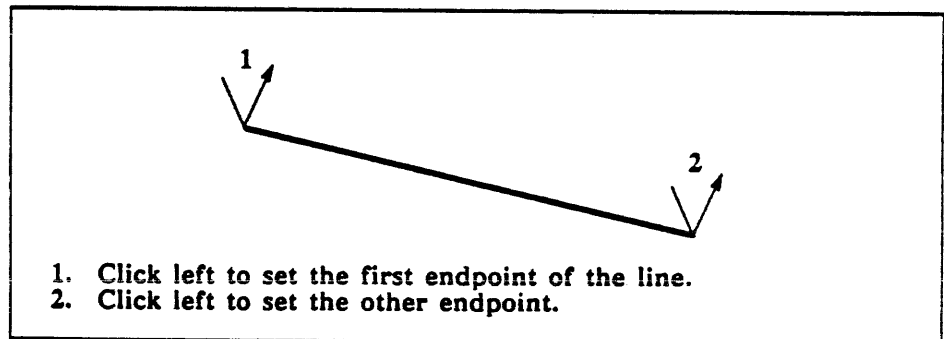
Drawing a Circle 10.4.5.2 The first left click sets the center of the circle. The second left click sets a point on the circumference of the circle, shown in the following figure. You can use rubber banding to see the circle while you are selecting the second point.



Lines 10.4.6 To start drawing lines, select Line on the Draw submenu, click on the line icon on the icon menu, or press META-L. The mouse cursor becomes the line icon. The center of the line indicates the point that is set when you click left.

Line Definition 10.4.6.1 The line segment is defined by the two endpoints.

Drawing a Line 10.4.6.2 Click left to set the first point, and click left a second time to set the second point, as shown in the following figure. You can use rubber banding to select the second point.



Paintings 10.4.7 To start to draw paintings, select Paint from the Draw submenu, click on the paint icon on the icon menu, or press META-B. The mouse cursor becomes a paint icon. The tip of the paintbrush indicates the point that is set when you click left.

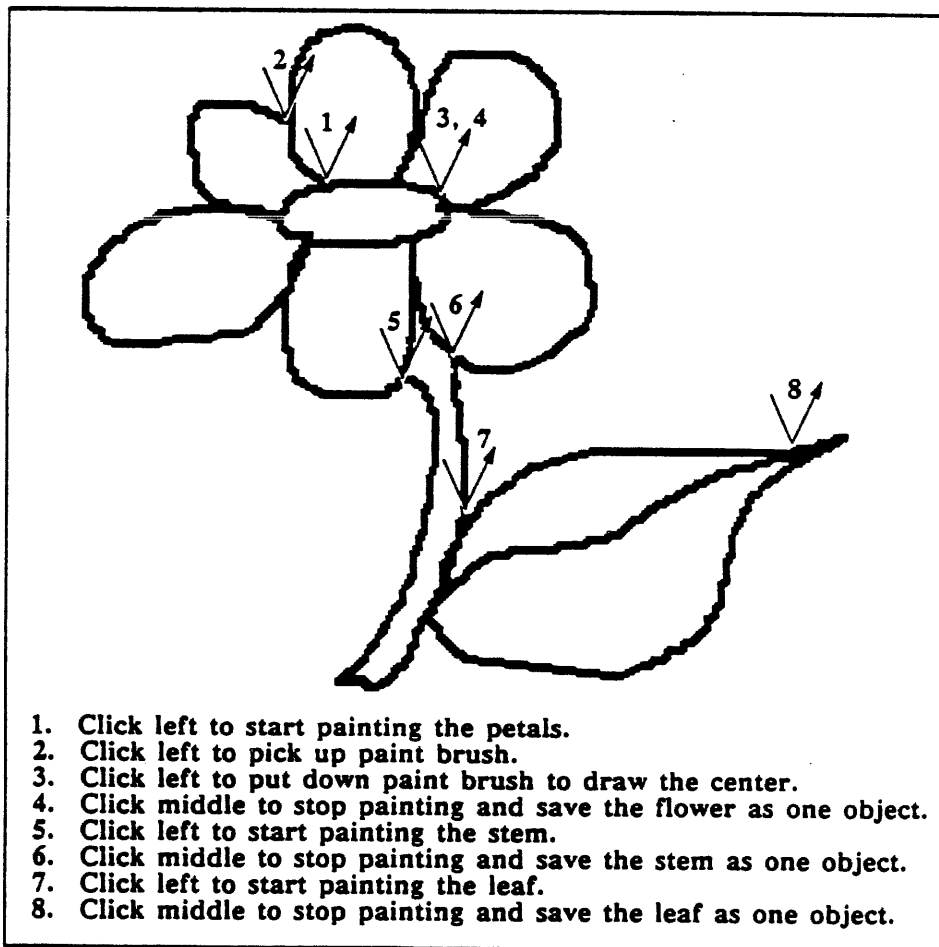
Painting Definition 10.4.7.1 A painting is one graphics object, even if the painting is composed of several distinct pieces. Each time you save a painting, every piece that has not been saved previously is defined as one graphics object.

When painting on a color system, it is recommended that your painting be done in one complete session (that is, as one graphics paint object). The reason for this recommendation is that large amounts of memory are consumed for individual paint objects. It is also advised that you set `gwin:*current-cache-for-raster-objects*` to nil if your painting is done in the graphics editor. Paintings also consume large amounts of memory so their use should be limited. Refer to the Graphics section in the *Explorer Window System Reference* for more information on raster objects.

Drawing a Painting

10.4.7.2 To begin a painting, click left. To lift the paintbrush and start another piece of the same painting, click left to lift, and click left to begin painting again. Even though you see brush strokes on the screen, a painting that has not been saved is not part of the picture.

To save a painting, click middle. When you click middle, all of the pieces of the painting are defined as one raster object, as shown in the following figure. If you paint again, pieces that have been saved will not be included in the current object.



*Status Variables
for Paintings*

10.4.7.3 The thickness of the brush stroke and the color of the paint are determined by the Thickness and Edge Color status variables. You can change the values for these status variables by clicking middle twice or by using the Status function.

Clicking Middle Twice Clicking middle twice displays a pop-up status window; change the values in this window, and then click on Update. This changes the status variables for the buffer. For example, suppose you change the edge thickness to 50 and the edge color to 75 percent gray. The painting you draw will be 50 pixels thick and 75 percent gray. If you stop painting and begin to draw circles, the circles will also have an edge thickness of 50 and will be drawn in 75 percent gray.

Using the Status Function You can also change the Thickness and Edge Color variables by using the Status function. Refer to paragraph 10.5.8, Status Variables, for further information.

**Polylines**

10.4.8 To begin to draw polylines, select Polyline on the Draw submenu, click on the polyline icon on the icon menu, or press META-P. The mouse cursor becomes the polyline icon. The gap in the bottom of the icon indicates the point that is set when you click left.

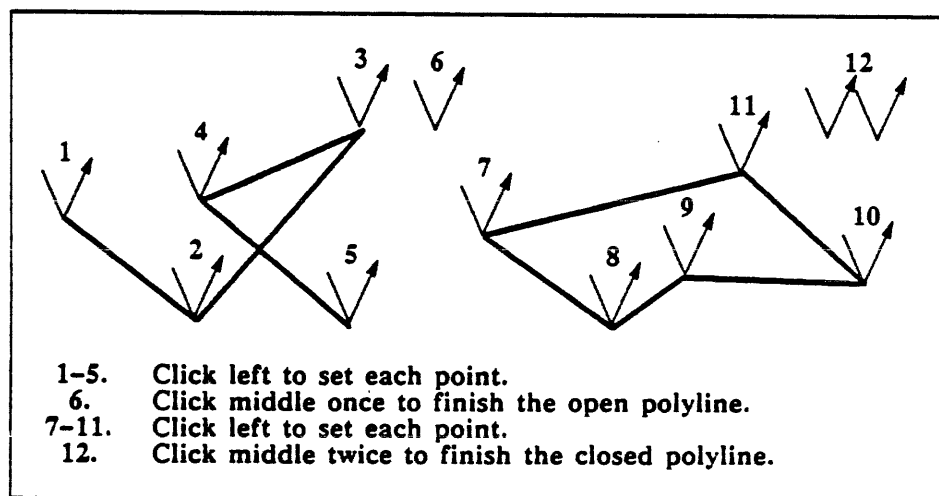
Polyline Definition

10.4.8.1 A polyline is defined by an ordered set of points. You can also draw polygons by indicating that you want the figure to be closed.

Drawing a Polyline

10.4.8.2 To draw a polyline or polygon, click left to set the first point. Then click left to specify each succeeding point. You can use rubber banding to see the lines as you specify the points.

After you have selected all the points for the figure, you must complete the polyline by using the middle button; the polyline is not part of the picture until you do. When you have specified all the points for your figure, click middle once for an open polyline, or click middle twice for a closed polyline, as shown in the following figure.





Rectangles

10.4.9 To start drawing rectangles, select Rectangle from the Draw sub-menu, click on the rectangle icon on the icon menu, or press META-R. The mouse cursor becomes the rectangle icon. The dot in the center indicates the point that is selected when you click left.

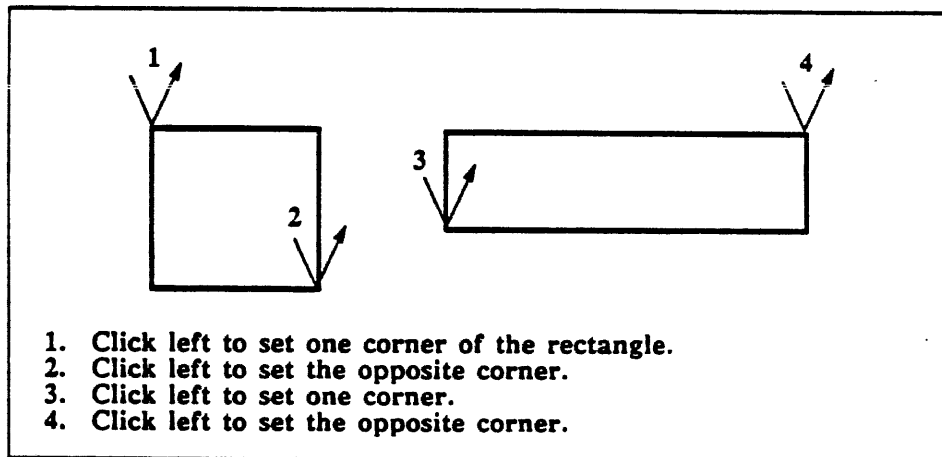
Rectangle Definition

10.4.9.1 A rectangle is specified by the position of the opposite corners. Rectangles are drawn such that the sides are parallel to the edges of the window.

Drawing a Rectangle

10.4.9.2 To draw a rectangle, click left to specify one corner of the rectangle. Click left again to specify the opposite corner, as shown in the following figure. You can use rubber banding to see the rectangle while you are selecting the second point.

The grid can help you draw squares. If the X Grid Spacing and Y Grid Spacing status variables have the same value, you can click on a grid point and then click on the grid point that is the same distance from the first point in both the x and y directions. Refer to paragraph 10.5.8, Status Variables, for information on the grid.





Rulers 10.4.10 To start drawing rulers, select Ruler on the Draw submenu, click on the ruler icon on the icon menu, or press META-N. The mouse cursor becomes the ruler icon. The midpoint of this ruler indicates the point that is selected when you click left.

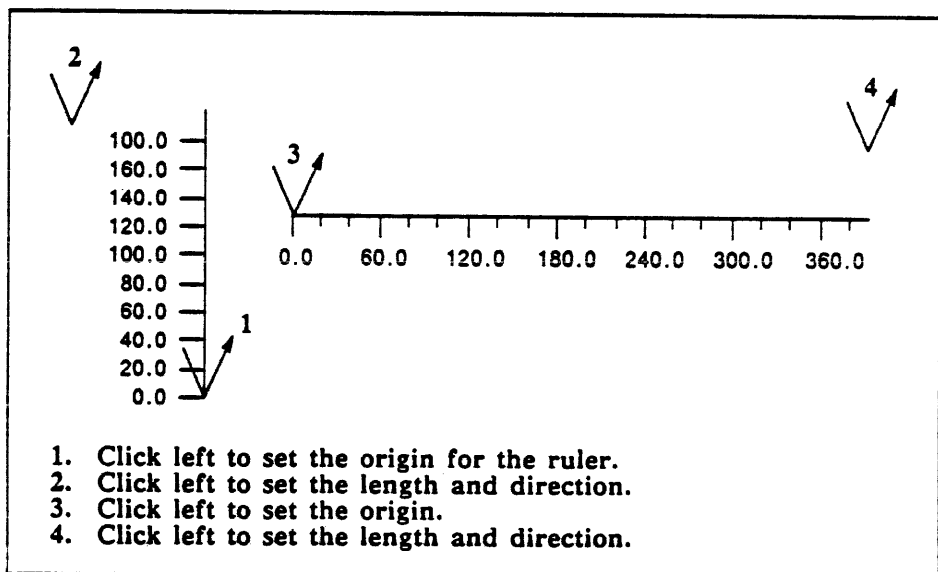
Ruler Definition 10.4.10.1 Rulers are specified by two points; the first point sets the origin, and the second point determines the following:

- The direction that the numbers run on the ruler
- The length of the ruler
- The axis that the ruler is parallel to

Rulers are always drawn parallel to one of the edges of the window.

Drawing a Ruler 10.4.10.2 To draw a ruler, set the origin by clicking left. The ruler will be numbered starting at the origin.

Select the second point by clicking left again, as shown in the following figure. You can use rubber banding to select the second point.



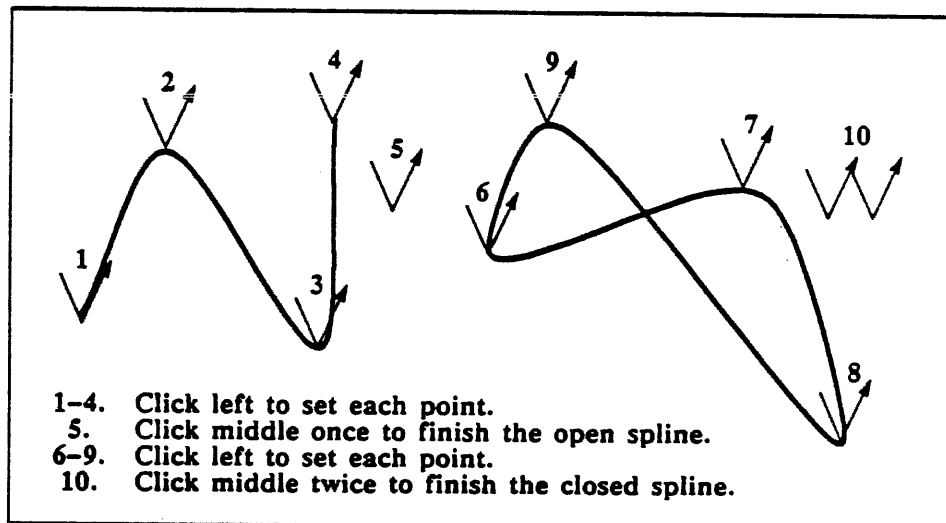
If the absolute value of the difference between the x coordinates of the first and second points is greater than the absolute value of the difference between the y coordinates, the ruler is parallel to the x axis, and the length of the ruler is the absolute value of the difference between the x coordinates. Otherwise, the ruler is parallel to the y axis, and the length of the ruler is the absolute value of the difference between the y coordinates.

Splines 10.4.11 To start drawing splines, select Spline in the Draw submenu, click on the spline icon on the icon menu, or press META-S. The mouse cursor becomes the spline icon. The hole in the center indicates the point that is set when you click left.

Spline Definition 10.4.11.1 A spline is defined by an ordered set of points. You can draw an open or closed figure; indicate whether the figure is open or closed after you have selected all the points.

A parametric cubic curve is drawn through each successive pair of points, using a fixed number of points for each curve. The cubics are computed so that they match in position and first derivative at each of the points. At the endpoints of an open figure, the derivative is zero.

Drawing a Spline 10.4.11.2 To draw a spline, select the points that the curve will pass through by clicking left in the order that the curve will pass through them. When all points are selected, use the middle button to specify whether the spline is open or closed, as shown in the following figure. Click once for an open spline; click twice for a closed spline. Rubber banding is not available for drawing splines.

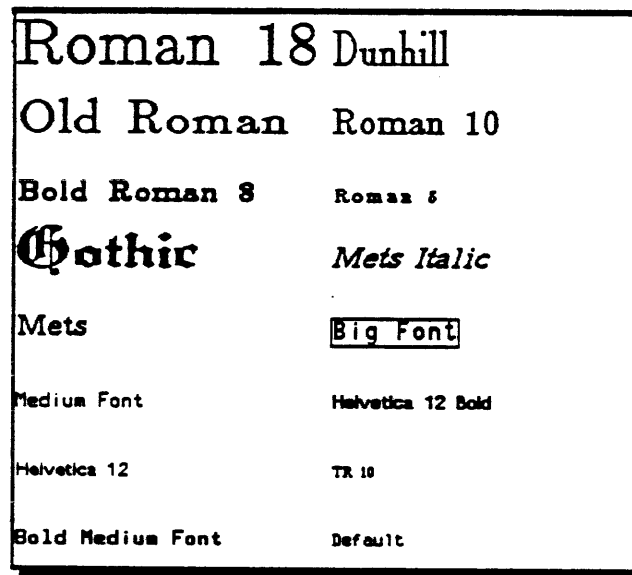


T

Text 10.4.12 To start drawing text objects, select Text from the Draw submenu, click on the text icon on the icon menu, or press META-W. The mouse cursor becomes a text icon. The upper left corner of the text icon indicates the point that is set when you click left.

Text Definition 10.4.12.1 Text is defined by a point and a sequence of character codes that you input from the keyboard. Because the representation for each code is different in each font, the characters you see are determined by the font used for that text.

The font can be selected by using the Status function or by pressing CTRL-F. The system displays a menu of fonts that you can select, as shown in the following figure:



Drawing Text 10.4.12.2 To begin entering text, click left. A flashing cursor appears. Next, use the keyboard to input the text. If the text is long, you may want to use the editing figure to input most of the text.

Note that the character on the keycap is not necessarily the character that appears on the screen. What appears on the screen depends on the current font's definition of the code that corresponds to the key you have pressed.

The description of the object in the edit parameters window is the representation of the characters in standard font. By default, gwin:cptfont-font is the standard font.

To stop entering text, either click left to start entering text at another location, or select another command.

Editing Text 10.4.12.3 If you want to edit text in a text object, click middle over that object. A Zmacs window appears. The text in the window is identical to the text in the buffer; the font is the same, and the window is positioned so that the text is in the same position on the screen.

Press the END key to exit the editing window. The text object is updated and redrawn on the screen.



Triangles

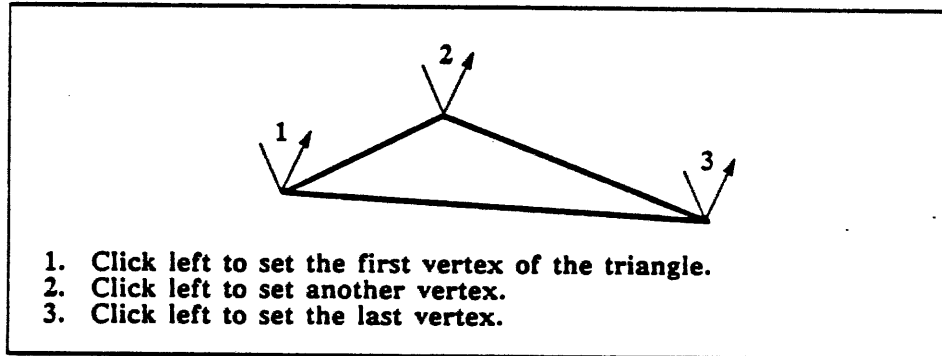
10.4.13 To start drawing triangles, select Triangle on the Draw submenu, click on the triangle icon on the icon menu, or press META-T. The mouse cursor becomes the triangle icon. The dot in the center indicates the point that is set when you click left.

Triangle Definition

10.4.13.1 A triangle is defined by three points.

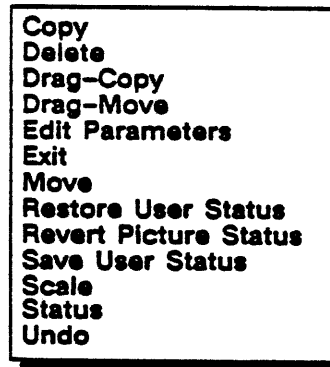
Drawing a Triangle

10.4.13.2 Click left to select a vertex of the triangle. Then select another vertex by clicking left. Finally, select the last vertex by clicking left, as shown in the following figure. You can use rubber banding to see the line segments you are creating while you are selecting the second and third points.



Functions

10.5 This section discusses the commands that change the parameters of graphics objects, buffers, and commands. The Function submenu is as follows:



Selecting Objects

10.5.1 You select one or more objects for commands in this group. The only command that works for only a single object is the Edit Parameters command. If you want to select more than one object for a command, either select the objects one at a time, or select several objects using rubber banding. Selected objects are indicated by the highlighting hand.

Selecting Objects Individually

10.5.1.1 You select an individual object by clicking left or middle, depending on the function, over that object. Repeat this click over each object you want selected.

*Selecting Objects
by Rubber Banding*

10.5.1.2 You can select several objects in the same area by rubber banding a box that encloses the objects. Select one corner of the box by pressing the middle or left button, depending on the function, but do not release the button. Hold the button down as you move to the opposite corner. As you move the mouse, a box appears to show you what objects you are selecting. Any object lying entirely inside the box will be selected. Any object lying partly or entirely outside the box will not be selected.

Copying Objects

10.5.2 You can copy any graphics object in a picture by using one of two options: Copy and Drag-Copy. Copy creates a copy at a specified location; Drag-Copy creates a copy that you move with the mouse cursor to the desired location.



Copy

10.5.2.1 To start copying objects from one location to another, select Copy in the Function submenu, click on the copy icon in the icon menu, or press META-F. The mouse cursor becomes the copy icon. The tip of the white arrow indicates the point that is selected when you click left or middle.

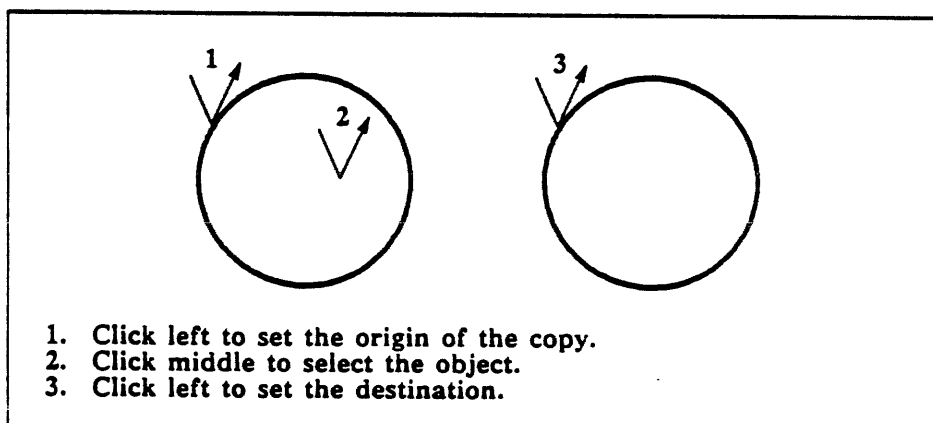
To copy objects, you must specify the following:

- The objects that will be copied
- The origin of the copy
- The destination of the copy

First, select the object or objects using the middle button. Refer to paragraph 10.5.1, *Selecting Objects*, for further information.

Next, select a point for the origin. When you select the destination, the copies will be drawn in relation to the destination as the objects are drawn in relation to this origin.

Finally, select a point for the destination. The copies are drawn in relation to the destination, as shown in the following figure.

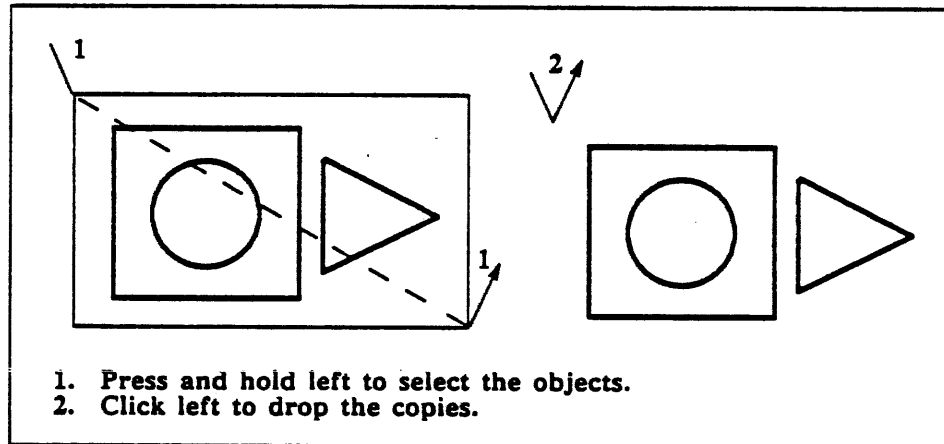




Drag-Copy 10.5.2.2 To start drag-copying objects, select Drag-Copy on the Function submenu, click on the drag-copy icon in the icon menu, or press META-CTRL-E. The mouse cursor becomes the drag-copy icon. The tip of the white arrow of the drag-copy icon indicates the point that is selected when you click left.

First, select the object or group of objects using the left button. Refer to paragraph 10.5.1, *Selecting Objects*, for further information. A blinking copy of the object appears and follows the cursor as you move the mouse.

When the blinking copy is at the correct location, click the mouse left again. The copy is then drawn at that point, as shown in the following figure.



If you decide not to copy a selected object, click left twice to release the object.

Deleting Objects



10.5.3 To start deleting objects from the picture, select Delete from the Function submenu, click on the delete icon in the icon menu, or press META-D. The mouse cursor becomes the delete icon. The hole in the middle indicates the point that is selected when you click middle.

First, select the objects, either individually or as a group. Refer to paragraph 10.5.1, *Selecting Objects*, for further information.

Next, click left to delete the selected objects. If you decide not to delete the selected object, click middle to abort the delete operation.

If you accidentally delete an object, you can usually restore the object by executing the Undo command.

Editing Parameters



10.5.4 Each graphics object has a set of parameters that define that object. The parameters include the object's location, edge and fill colors, the thickness of its lines, and the ALU that is used to draw the object on the screen. To edit these parameters, select Edit Parameters from the Functions submenu, click on the edit parameters icon in the icon menu, or press META-E. The mouse cursor becomes the edit parameters icon. The center of the crosshair indicates the point that is selected when you click left.

To edit the parameters of an object, click left. A pop-up window with a list of the current parameter values appears.

In a color environment, you can specify any color from the color map for the edge and fill colors. Clicking on these items displays a menu of color names from which you can choose. Also on this menu is the Show Color Map item, which displays the color map from which you can choose a color. You can also select the Enter Color Number item and enter a number for a color.

The Draw ALU item allows you to specify color ALUs as well as the standard ALUs in a color environment. The standard and color ALUs are described in paragraph 10.4.1.3, ALU Value.

Click on the values you want to change, and enter new values. When you are finished, click on the UPDATE box. The object is redrawn with the new values.

If you decide not to make any changes, click on the ABORT box. Refer to paragraph 10.4.1, Characteristics of Objects, for information on ALU and color values.

Exiting 10.5.5 To exit from the graphics editor and enter the previous window, press the END key, or select Exit from the Functions submenu.

Moving Objects 10.5.6 You can move objects in a buffer by using one of two commands in the graphics editor: Move and Drag-Move. Move redraws the object at a specified location; Drag-Move allows you to use the cursor to move the object to the desired location.



Move 10.5.6.1 To start moving objects from one location to another, select Move on the Functions submenu, click on the move icon in the icon menu, or press META-M. The mouse cursor becomes the move icon. The tip of the arrow indicates the point that is selected when you click left or middle.

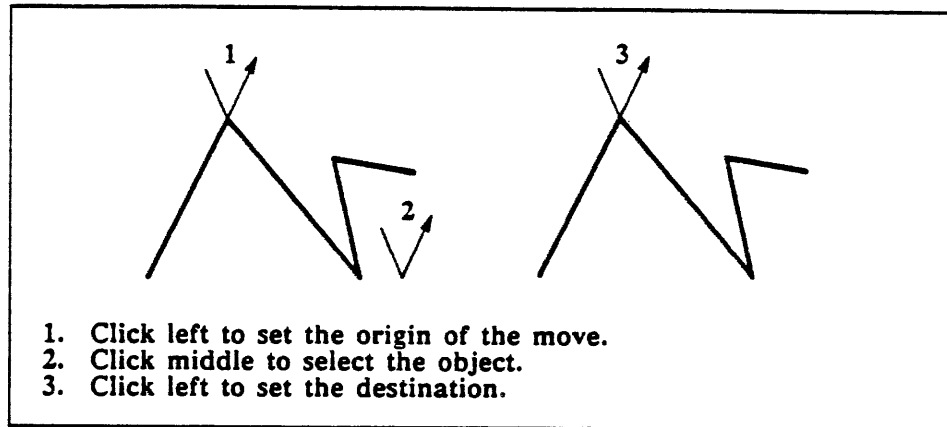
To move objects, you must specify the following:

- The objects to be moved
- The origin of the move
- The destination of the move

First, select the objects using the middle button. Refer to paragraph 10.5.1, Selecting Objects, for further information.

Next, select a point for the origin. When you select the destination, the objects will be drawn in relation to the destination as the objects are drawn in relation to the origin.

Finally, select a point for the destination. The objects are drawn in relation to the destination, as shown in the following figure. The original is deleted.

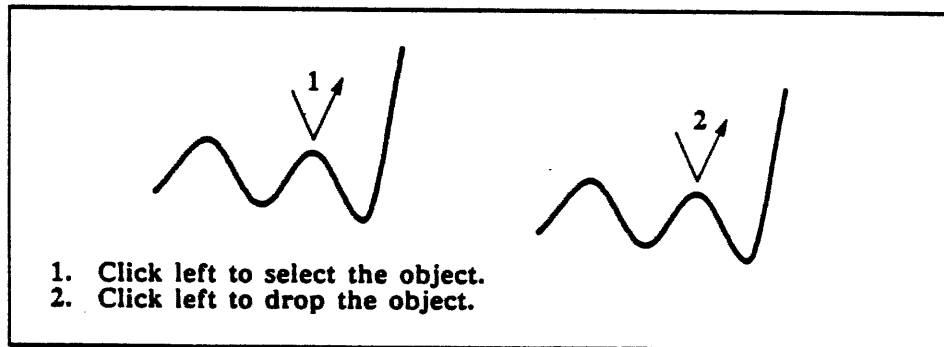


Drag-Move

10.5.6.2 To start drag-moving objects, select Drag-Move on the Functions submenu, click on the drag-move icon on the icon menu, or press META-CTRL-M. The mouse cursor becomes the drag-move icon. The tip of the arrow on the drag-copy icon indicates the point that is selected when you click left.

First, select one object or one group of objects. Refer to paragraph 10.5.1, Selecting Objects, for further information. A blinking copy of the object appears and follows the object as you move the mouse cursor.

Click left again when the copy is in the desired position. The original is deleted, and the copy is drawn in the new location, as shown in the following figure.



If you decide not to move a selected object, click left twice to release the object.



Scaling Objects

10.5.7 To start scaling objects or groups of objects, select Scale from the Functions submenu, click on the scale icon on the icon menu, or press META-Q. The mouse cursor becomes the scale icon. The hole in the center of the icon indicates the point that is selected when you click left or middle.

First, select the objects using the middle button. Refer to paragraph 10.5.1, *Selecting Objects*, for further information.

When you finish selecting objects, set the origin for the scale by clicking left. Each object is scaled in relation to the origin. For example, scaling by 2 causes every point in the object to be twice as far from the origin, and scaling by 0.5 causes all points in the object to be half as far from the origin.

When the origin and the objects are selected, click left. Use the pop-up numeric pad or the keyboard to enter the scale factor. The original objects are erased, and the scaled versions are drawn.

Status Variables 10.5.8 You can modify, save, restore, and revert status variables. The following table explains the status variables. The status variables are listed in the same order that the status menu uses.

Status Variable	Meaning
Edge Color and Fill Color	The Edge Color variable specifies the color of the delimiting edge for objects that you draw. The default is black. The Fill Color variable specifies the color of the area inside the object. When the value is nil, the object is only an outline. The default is nil.
ALU	In a color environment, you can specify any color from the color map for the edge and fill colors. Clicking on these items displays a menu of color names from which you can choose. Also on this menu is the Show Color Map item, which displays the color map from which you can choose a color. You can also select the Enter Color Number item and enter a number for a color.
Crosshair Cursor	Specifies the ALU used to draw the object on the screen. In a color environment, there are several color ALU values available as well as the standard ALU values. The standard and color ALU values are discussed in paragraph 10.4.1.3, <i>ALU Value</i> . The default is w:normal.
X Grid Spacing	Used for aligning points horizontally and vertically. The default is not to display the crosshair.
Y Grid Spacing	Determines the distance between grid points on the x axis. The default is 20.
Grid	Determines the distance between grid points on the y axis. The default is 20.
Tab Spacing	When on, the grid is displayed as a series of dots, and the mouse cursor can only select these grid points. The default is not to have the grid.
Thickness	Determines how many spaces to insert when the TAB key is pressed during text entry. The default is eight.
Text Font	Determines the width of objects that you draw. The default is two pixels.
Window Layout	Determines the font for text objects. The default is standard-font.
	Determines which panes are displayed in the graphics editor window. The default is the submenu and file layout.

Continued

Status Variable	Meaning
Draw Blob Threshold	Determines the minimum size for drawing objects in detail. Objects with fewer pixels in all directions are drawn as blobs. The default is six pixels.
No Draw Threshold	Determines the minimum size for visible objects. Objects that have fewer pixels in all directions are not displayed. The default is two pixels.
Picking Radius	Determines how close the mouse cursor must be to an object before you can select the object. The default is 20 pixels.
Scale Thickness	Determines whether the line thickness of objects is scaled when the object is scaled. The default is to have the thickness scaled.
Zoom Grid	Determines whether the grid zooms with the window or the world. The default is to have the grid zoom with the world.
Interrupt Drawing	Determines whether drawing is completed when a command that draws a different view is selected. The default is not to interrupt.
Save Picture Compiled	Determines whether the Lisp forms for the objects are compiled when a picture is saved to a file. Compiled files are read in faster, but they cannot be examined. The default value is compiled.

*Modifying
Status Variables*

10.5.8.1 You can begin to modify the status variable values for a buffer by either pressing the STATUS key or selecting Status from the Functions submenu.

For values that have a numeric value, such as line thickness or grid size, click on the current value. A pop-up numeric pad is displayed. You can enter a new value either from the numeric pad or from the keyboard. Press the RETURN key or click on Return when you have finished.

For variables that have only a few possible values, a list of the values is displayed with the current value in a bolder font. Select the new value by clicking over it with the mouse. The new value is then displayed in the bold font to show that it has been selected.

If a variable can assume many values, only the current value is displayed in the window. Clicking on the current value brings up a menu of selectable values. Choose one with the mouse, and the new value will show in the window.

*Saving
Status Variables*

10.5.8.2 You can save the modifications you have made to the status variables for use in another buffer or during other sessions. To save the modifications you have made, select Save User Status from the Functions submenu, or press HYPER-SUPER-S. You are then prompted to specify the file in which to save the modifications.

Saving the status variables makes the changed values the defaults for the rest of the current graphics editor session.

Restoring Status Variables 10.5.8.3 To read in the variable values you saved with Save User Status, select Restore User Status from the Functions submenu, or press HYPER-SUPER-R. You are then prompted for the name of the file that the values are stored in.

Restoring the status variables makes the new values the defaults for the rest of the current session.

Reverting Status Variables 10.5.8.4 You can undo modifications to the status variables two different ways. You can revert to the previous default values when you have modified the status variables, or you can revert to the values that were saved with the file that has been read into the current buffer.

Use Revert User Status to return to the default values if you have modified the status variables or if the status variables were reset when a file was read into the buffer.

If you have modified the status variables since you read a file into a buffer, you use Revert Picture Status in that buffer to revert to the values that were saved with the file.

Reverting User Status To revert the status variables to the defaults, select Revert User Status on the Functions submenu, or press HYPER-SUPER-T. The status variables for the buffer are reset to the default values.

Reverting Picture Status To revert the status variables to the values saved with the file, select Revert Picture Status from the Functions submenu, or press HYPER-SUPER-V.

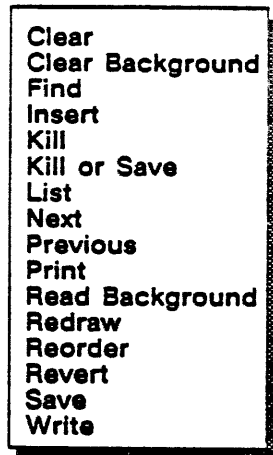
Undoing Commands 10.5.9 You can undo the effects of your previous commands. To undo the last command, select Undo from the Functions submenu, or press the UNDO key. A confirmation window is displayed. If you want to undo the command that is listed in the window, click on the window; otherwise, move the mouse cursor off the window after you move out of the confirmation window. If the last command was UNDO, you are asked if you want to UNDO the UNDO.

Not all commands can be changed: only the ones that alter the composition of the picture, such as drawing, deleting, or moving.

Another restriction is that commands must be undone in reverse order. You must undo the last command before you can undo the next-to-the-last command.

Pictures

10.6 This section discusses the commands for the graphics editor buffers and the picture files. The Pictures submenu is as follows:



```

Clear
Clear Background
Find
Insert
Kill
Kill or Save
List
Next
Previous
Print
Read Background
Redraw
Reorder
Revert
Save
Write

```

Background Pictures 10.6.1 You can have objects on the screen that are not part of the picture. These objects can be used for comparison or for guides in drawing. Objects that appear on the screen but are not part of the picture form a background picture.

Reading a Background 10.6.1.1 A background picture is copied into the current buffer from a file. Any previously saved picture can be used as a background picture. When you are in the buffer that needs the background, select Read Background from the Pictures submenu, or press META-CTRL-B. You are then prompted to supply the name of the file to use for the background.

When the background is read into the buffer, the objects are visible, but they are not selectable. The background objects might appear larger or smaller or shifted along some axis when they are drawn. They are drawn according to the transformation of the buffer they have been read into. The objects are the same, but the window used to view them has changed position.

Clearing a Background 10.6.1.2 To remove the background objects from the screen, select Clear Background on the Pictures submenu, or press META-CTRL-CLEAR INPUT. All background objects are erased from the screen.

Changing Buffers 10.6.2 The graphics editor buffers that you are using are kept in a circular list. You can select one of these to be your current buffer in three ways: selecting a buffer from a list, moving to the next buffer in the list, or moving to the previous buffer in the list. You can also specify the order of the buffers in the list.

Selecting Buffers From the List 10.6.2.1 To see the list of buffers that you can select, select List from the Pictures submenu, or press META-CTRL-L. When the list of buffers appears, click on the one you want to have as the current buffer.

Moving to the Next Buffer 10.6.2.2 To move to the next buffer, select Next from the Pictures menu, or press META-CTRL-N. The next buffer in the circular list becomes the current buffer.

Moving to the Previous Buffer 10.6.2.3 To move to the previous buffer, select Previous from the Pictures submenu, or press META-CTRL-P. The previous buffer in the circular list becomes the current buffer.

Ordering the Buffers 10.6.2.4 Buffers are placed in the circular list in the order they are created. You can change the order of buffers to make Next and Previous more useful. To order the buffers, select Reorder from the Pictures submenu, or press META-CTRL-O. The current list of buffers appears.

Select the buffer that you want to move by clicking middle when the underline is beneath the name of that buffer. Selected buffers are in reverse video. You can select more than one buffer.

When you have finished selecting buffers, position the underline at the place in the list where you want the buffers moved, and click left. The list is reordered. You can repeat the process until the buffers are in the proper order.

Click on Done when the list is in the proper order. If you want to discard the modifications, click on Start Again.

Clearing the Foreground 10.6.3 When you want the objects in the foreground of the current buffer deleted, select Clear from the Pictures submenu, or press the CLEAR INPUT key. Click on the confirmation window, or move the mouse cursor off the window to abort the clear operation. When you confirm the clear operation, all the objects in the foreground are deleted.

Inserting Pictures 10.6.4 To insert all the objects from a saved picture into the current buffer, select Insert from the Pictures submenu, or press META-CTRL-I.

You are prompted to provide the name of the file that contains the picture. Then you set the location for the origin of the picture being inserted by clicking left. The origin of a picture is at the upper left corner of the default window.

Killing and Saving Buffers 10.6.5 When you are finished with a buffer, you can kill the buffer, and you can save the picture to a file. Use the Kill, Save, Write, and Kill or Save commands to kill and save buffers.

Killing a Buffer 10.6.5.1 To kill the current buffer, select Kill on the Pictures submenu, or press META-CTRL-K. Click on the confirmation window, or move the mouse cursor off the window to abort the operation.

If you confirm that you want the buffer killed, select the buffer that you will enter when the current buffer is killed.

Saving a Picture 10.6.5.2 To write a picture to the default file, select Save from the Pictures submenu, or press META-CTRL-S. You are not prompted for a filename; the file that the picture was read from is updated. If you use Save on a picture that has not been saved before, you are prompted for a filename.

Writing a Picture to a Specific File 10.6.5.3 To write a picture to a specific file, select Write from the Pictures submenu, or press META-CTRL-W. You are then prompted to provide a filename.

Killing or Saving Buffers 10.6.5.4 To kill or save several buffers at once, select Kill or Save from the Pictures submenu, or press META-CTRL-STATUS.

A window listing all the buffers is displayed. If changes have been made to any of these buffers since they were last saved, the corresponding save boxes are marked. An asterisk (*) in the left column means that the buffer has been modified. A plus sign (+) in the left column means that the contents of the buffer have not been saved to a file.

You can choose whether to kill, save, and unmodify each buffer. Select these options by marking the corresponding boxes; to mark or unmark a box, click over that box.

When the kill box is marked, that buffer is killed. When the save box is marked, the picture in that buffer is saved. When the unmodify box is marked, the save box for that buffer will not be marked the next time this window is displayed; marking the unmodify box does *not* change the contents of the buffer.

A buffer can be both killed and saved by marking both boxes. The picture in the buffer is saved in a file, and then the buffer is killed.

When the boxes are marked properly, click on the Do It box. If you decide not to make any changes, click on the Abort box.

Printing Pictures 10.6.6 You can print pictures from the screen or from a file.

Printing From the Screen 10.6.6.1 Press TERM Q to print a picture that is on the screen or select the Hardcopy Menu option from the System menu. Refer to the *Explorer Input/Output Reference* for more information on printing from the screen.

Printing From a File 10.6.6.2 Use the print-graphics function to print a picture that has been stored in a file. Refer to the *Explorer Input/Output Reference* and the *Explorer Window System Reference* for further information on this function. The *Explorer Window System Reference* describes printing color screens on a monochrome printer.

To use the print-graphics function while you are in the graphics editor, select Print from the Pictures submenu, or press CTRL-P. When you call the print-graphics function this way, you can only supply the pathname; the defaults for the other arguments are used.

Reading a Picture File 10.6.7 To read a saved picture into a buffer, select Find from the Pictures submenu, or press META-CTRL-F. You are then prompted to provide a filename.

If the file does not exist, an empty buffer is created, and the ****New File**** message appears in the file pane if the current layout includes the file pane.

If the file exists, a new buffer is created, and the existing file is inserted in the new buffer. The default window and default status variable values are used for this new buffer, unless you use the Revert Picture Status command. If you use the Revert Picture Status command, the buffer will have the same zooming, panning, and status variable values that were in effect when the picture was saved.

Redrawing the Picture 10.6.8 To redraw the picture, select Redraw from the Pictures submenu, or press the CLEAR SCREEN key. Objects in the picture are redrawn in the order they were created. This can be useful for briefly seeing objects that are covered by overlying objects.

Reverting a Buffer 10.6.9 To discard modifications you have made to a retrieved picture, select Revert from the Pictures submenu, or press META-CTRL-R. Click on the window to confirm that you want to clear the buffer and read in the picture that was saved. Move the mouse cursor off the window to abort the operation.

Presentations

10.7 This section discusses the commands for the graphics editor buffers and the picture files. The Presentations submenu is as follows:



```

Define
Display
Kill
Kill or Save
List
Load
Modify
Restore
Save

```

Graphics Editor Presentations 10.7.1 If you are using a collection of pictures, you can save time by defining them to be a group called a presentation. Specifically, a presentation is a group of files where each file is single picture. When you define the presentation, you specify which files are to be included, their order in the group, and the name of the presentation. After you define a presentation, the definition can be saved, edited, and killed. The component files can be read with one command. A file can be in more than one presentation, or in the same presentation more than once.

Buffers cannot be included in a presentation definition; you must save a buffer to a file and include the file in the definition.

A graphics editor can have several presentations defined at a time. The name of the current presentation is displayed in the file pane when the current layout includes the file pane. The name appears on the first line of the pane in square brackets.

Defining Presentations 10.7.2 To define a list of files as a presentation and give the list a name, select Define from the Presentations submenu, or press SUPER-D. You are then prompted for the names of the files that are included in the presentation. When you have entered all the pathnames, press the RETURN key.

Then you are prompted to provide a name for the presentation. This name is not a pathname; do not include a file extension or any other pathname components. The name has the same limitations as any symbol name in the Explorer system (for example, spaces and periods must be protected by a slash, there is no limit on length, case is not significant, and so on, as explained in the *Explorer Lisp Reference*).

Defining a presentation does not make that presentation available for later sessions. You must save a presentation if you want to have the definition available at future sessions.

**Restoring
Presentation
Definitions**

10.7.3 To use presentations that were defined and saved in previous sessions, select Restore from the Presentations submenu, or press SUPER-R. You are then prompted to provide the pathname for the presentation definition. That definition becomes the current definition.

If you have made modifications to a presentation definition and want to undo them, you can use Restore to read in the most recently saved definition for that presentation.

**Viewing
Presentations**

10.7.4 You can use two functions to look at a presentation: Display or Load. If the presentation was not defined during the current session, you must restore the presentation for the current session.

*Displaying
Presentations*

10.7.4.1 Once a presentation is defined or restored from a file, it can be displayed. To show the current presentation, select Display from the Presentations submenu, or press SUPER-Y.

Pictures are displayed one at a time. To move to the next picture, press any key or button. If the pictures have already been read in, you can move immediately to the next picture; otherwise, the next picture is read in when you press the key.

*Loading
Presentations*

10.7.4.2 To show the presentation without pressing keys to move from picture to picture, select Load from the Presentation submenu, or press SUPER-F.

The presentation is displayed. The pictures in the presentation are displayed in succession without pressing a key between pictures. If the pictures have already been read in, the presentation is displayed rapidly; otherwise, the pictures are displayed as they are read in.

**Killing and
Saving
Presentation
Definitions**

10.7.5 Presentation definitions can be killed and saved. If you want to use a presentation during a later session, you must save the presentation definition. When you are through using a definition, you can kill the presentation definition.

*Killing a
Presentation
Definition*

10.7.5.1 To kill the current presentation definition, select Kill from the Presentation submenu, or press SUPER-K. Click on the confirmation window to kill the definition. To abort the operation, move the mouse off the window. When you kill the presentation, the next presentation definition becomes the current definition. If you had only one definition, no presentations are defined for the buffer.

To kill a definition that is not the current definition, use Kill or Save Presentations.

*Saving a
Presentation
Definition*

10.7.5.2 To save the current presentation, select Save from the Presentations submenu, or press SUPER-S. You are prompted to provide a pathname if the definition has not been saved before. If you are updating a definition that has been saved before, the definition is written to the same file name with an incremented version number.

To save a definition other than the current one, use the Kill or Save Presentation command.

*Killing or Saving
Presentation
Definitions*

10.7.5.3 To kill and save several presentation definitions, select Kill or Save from the Presentation submenu, or press SUPER-STATUS.

A window listing all the definitions is displayed. If changes have been made to a definition, the save box is marked.

You can choose whether to kill, save, or unmodify each presentation definition. Select these options by marking the corresponding boxes; to mark or unmark a box, click over that box.

When the kill box is marked, that definition is killed. When the save box is marked, that definition is saved. When the unmodify box is marked, the save box for that definition will not be marked the next time this window is displayed; marking the unmodify box does not change a presentation.

A definition can be both killed and saved by marking both boxes. The definition is saved, and then that definition is killed so that it is no longer available in the current session.

When the boxes are marked properly, click on the Do It box. If you decide to make no changes, click on the Abort box.

**Listing and
Selecting
Presentations**

10.7.6 To select another presentation, select List from the Presentations submenu, or press SUPER-L. A window listing defined presentations appears. Click on a presentation to make it the current presentation.

**Modifying
Presentations**



10.7.7 After you define a presentation, you can include other files or delete a picture from the list. To modify the current presentation, select Modify from the Presentations submenu, click on the modify icon on the icon menu, or press SUPER-M. The mouse cursor becomes the modify icon.

To add more files to the definition, click left. You are prompted to provide a filename. You can add several files. Press the RETURN key when you are finished adding files.

To delete files from a presentation, click middle. A window with a list of the component files appears. Mark the delete box corresponding to the files you want to remove from the presentation. Click over a box to mark or unmark it. Click on the Do It box to remove the files from the presentation, or click on the Abort box if you decide not to delete any files.

To change the order of the files in the presentation, click left twice. A window with a list of the files in the current definition appears. Select the file that you want to move by clicking middle when the underline is beneath the name of that file. Selected filenames are displayed in reverse video. You can select more than one file.

When you have finished selecting files, position the underline at the place in the list where you want the files moved, and click left. The list is reordered. You can repeat this process until the buffers are in the proper order.

Click on Done when the list is in the proper order. If you want to discard the modifications, click on Start Again.

Subpictures

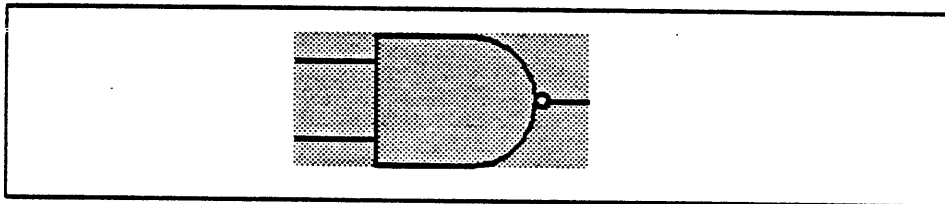
10.8 This section discusses the commands for the graphics editor buffers and the subpicture files. The Subpictures submenu is as follows:



Using Subpictures

10.8.1 You can define an object that is constructed of several objects. For example, you could use lines, curves, and triangles to define a NAND gate object, and then use this subpicture to draw circuits. Using subpictures, you can define a group of objects to be one object.

Subpictures can have edge and fill colors. The edge and fill colors are initially nil, but you can edit the parameters for a subpicture after it is created. When the edge color is non-nil, a box is drawn around the extents of the objects that form the subpicture. When the fill color is non-nil, the area outside the objects but inside the extents of the subpicture is colored.



Defining Subpictures



10.8.2 To define a subpicture, select Define from the Subpicture submenu, click on the subpicture icon on the icon menu, or press META-G. The mouse cursor becomes the subpicture icon. The place where the ends cross indicates the point that is selected when you click middle or left.

First, select the objects that form the subpicture using the middle key. Refer to paragraph 10.5.1, *Selecting Objects*, for further information.

Next, set the origin of the subpicture. This point is the point you will specify when you insert the subpicture. Click left to set the origin.

Finally, click left again to define the subpicture. A window appears prompting you to provide a name for the subpicture. You also select whether the object is rasterized.

A subpicture can be either one raster object or a list of component objects. A rasterized subpicture is drawn much faster, but a rasterized picture is scaled by pixel replication. A rasterized subpicture cannot be exploded; that is, it cannot be modified.

Furthermore, rasterized subpictures, which are actually GWIN raster objects, are very expensive in terms of disk file and memory usage. This problem mainly occurs on color systems. Refer to the Graphics section in the *Explorer Window System Reference* for a discussion on GWIN raster objects.

If you intend to scale or explode the subpicture and you do not need fast drawing, save the subpicture as a list of the component objects.

Click on the Update box when you are finished. The original objects are removed from the buffer and replaced with the subpicture object.



Exploding Subpictures

10.8.3 If you need to perform an operation on an object that is part of a subpicture, you can explode the subpicture. Exploding undefines the subpicture and makes the component objects individual objects. If the subpicture has been scaled, the component objects are returned to the size that they were when the subpicture was defined. Any edge or fill color for the subpicture is removed. You cannot explode a rasterized subpicture.

To explode a subpicture, select Explode from the Subpictures submenu, click on the explode icon (the scissors) on the icon menu, or press META-X. The mouse cursor becomes the explode icon. The point where the blades cross indicates the point that is selected when you click middle.

First, select the subpicture by clicking middle over one of the component objects. Only one component of the subpicture will have a highlighting hand. You can select more than one subpicture.

When the subpictures are selected, click left to explode the subpicture. No visual changes occur but the individual components of the subpicture can now be selected.



Inserting Subpictures

10.8.4 To insert a subpicture into a buffer, select Insert from the Subpictures submenu, or press META-I.

If you have defined more than one subpicture, a list of subpictures appears. Select the one you want to insert. The mouse cursor becomes the insert icon. The tip of the arrow is the point that is selected when you click left.

Click left at the location you want the origin of the subpicture placed. You can insert the subpicture more than once.

If you want to insert a different subpicture, you must start the insert process again.

Saving Subpicture Definitions

10.8.5 To save the subpicture definitions for use in later sessions, select Save on the Subpictures submenu, or press HYPER-S. You are then prompted to provide the name of a file to save all the current subpicture definitions in.

Restoring Subpicture Definitions

10.8.6 To restore subpicture definitions that were saved in a previous session, select Restore from the Subpictures submenu, or press HYPER-R. You are then prompted to provide the name of a file that the subpicture definitions were saved in. The definitions are read and added to the list of current subpicture definitions.

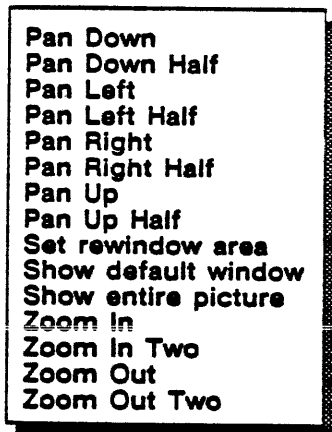
Undefining Subpictures

10.8.7 To remove a subpicture definition from the list of current definitions, select **Undefine** from the **Subpictures** submenu, or press **META-U**. A window with a list of the current definitions appears. Click on the definitions that you want to undefine.

Undefining a subpicture has no effect on the subpictures of that type already drawn.

Windowing

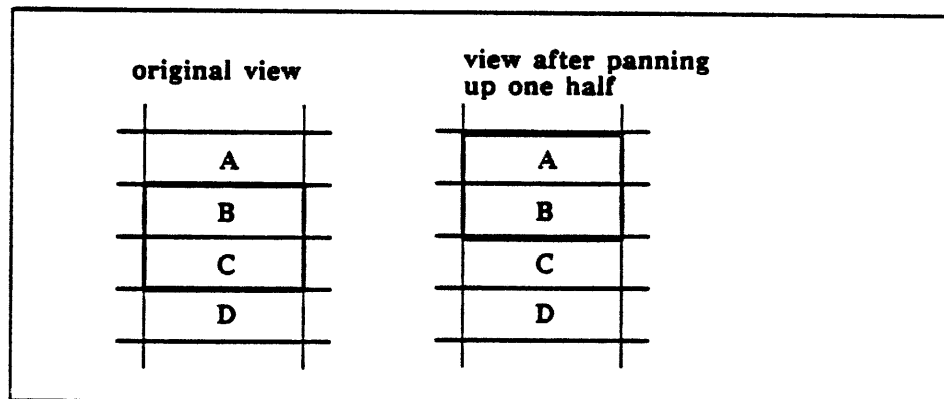
10.9 This section discusses the commands for determining which part of the world is visible through the window. The **Windowing** submenu is as follows:



In addition to the commands on the **Windowing** submenu, there are also commands for panning one-third screen and commands for zooming by 8 and 0.125. These commands are available only through the keyboard.

Panning

10.9.1 Panning moves the window to view adjacent portions of the world. For example, suppose you have a window that is displaying parts of a world labeled **B** and **C**. If you pan up half, the window now displays parts **A** and **B** of the world.



The following table lists the commands that pan the window up, down, right, or left, in increments of one screen, one-half screen, and one-third screen.

Increment To Move the Window	Command on the Windowing Submenu	Keystroke Sequence
One screen	Pan Up	CTRL-↑
One screen	Pan Down	CTRL-↓
One screen	Pan Left	CTRL-←
One screen	Pan Right	CTRL-→
One-half screen	Pan Up Half	META-↑
One-half screen	Pan Down Half	META-↓
One-half screen	Pan Left Half	META-←
One-half screen	Pan Right Half	META-→
One-third screen	-	META-CTRL-↑
One-third screen	-	META-CTRL-↓
One-third screen	-	META-CTRL-←
One-third screen	-	META-CTRL-→

Zooming

10.9.2 The following table lists the commands that zoom the window by factors of 8, 4, 2, 0.5, 0.25, and 0.125. Zooming by a factor means that all objects appear to be scaled by that factor with the center of the window as the origin.

Factor To Zoom the Window	Command on the Windowing Submenu	Keystroke Sequence
In by 8	-	META-CTRL->
In by 4	Zoom in Two	META->
In by 2	Zoom In	CTRL->
Out by .5	Zoom Out	CTRL-<
Out by .25	Zoom Out Two	META-<
Out by 0.125	-	META-CTRL-<

Defining a Rewindow Area

10.9.3 You can set the window in three additional ways: setting the rewindow area with the mouse, showing all the objects in the graphics world, or returning to the default window.



Setting the Area With the Mouse

10.9.3.1 To begin rewindowing, select Set Rewindow Area on the Windowing submenu, click on the rewindow icon on the icon menu, or press META-V. The mouse cursor becomes the rewindow icon. The upper left corner of the icon indicates the point that is selected when you click left or middle.

You can either specify a rectangle that you want included in the new window or specify the upper left or lower right corner for the new window. If you select a rectangle that is too narrow or too short, some area that you did not select is added to the right side or the bottom of the window to prevent distortion. If you specify only a corner, no scaling is performed.

Selecting a Rectangle To specify a rectangle that you want included in the new window, click left on one corner, and then click left or use rubber banding to set the opposite corner.

Selecting a Corner To select a point for the upper left corner, click middle. Your window is moved so that the point you have selected is the upper left corner of the screen.

To select a point for the lower right corner, click middle twice. Your window is moved so that the point you have selected is the lower right corner of the screen.

Showing the Entire World

10.9.3.2 To set the window so that all objects in the world are on the screen, select Show Entire Picture on the Windowing submenu, or press CTRL-CLEAR SCREEN. The picture is redrawn with the necessary zooming and panning so that all objects are on the screen.

Showing the Default Window

10.9.3.3 To return to the default window, either select Show Default Window from the Windowing submenu or press META-CLEAR SCREEN. Any panning or zooming is cancelled so that the window returns to the original position.

Command Names, Keystrokes, and Menus

10.10 Table 10-5 lists the various commands in the graphics editor. Each command is listed along the keystroke sequence that invokes it and the menu or submenu that it appears in. All commands have an assigned keystroke sequence, but some commands do not appear on a menu.

Table 10-5 Graphics Editor Keystroke Assignments

Command Name	Assigned Keystroke	Menu or Submenu
Arc Mode	META-A	Draw
Circle Mode	META-C	Draw
Clear Background Picture	META-CTRL-CLEAR INPUT	Picture
Clear Picture	CLEAR INPUT	Picture
Copy Objects Mode	META-F	Function
Crosshair On or Off	CTRL-H	Status
Define Presentation	SUPER-D	Presentation
Define Subpicture	META-G	Subpicture
Delete Objects Mode	META-D	Function
Display Presentation	SUPER-Y	Presentation
Drag-Move Mode	META-CTRL-M	Function
Drag-Copy Mode	META-CTRL-E	Function
Edit Parameters	META-E	Function
Exit	END	Function
Explode Subpicture	META-X	Subpicture
Find Picture	META-CTRL-F	Picture
Grid On or Off	CTRL-G	Status
Insert Picture	META-CTRL-I	Picture
Insert Subpicture	META-I	Subpicture
Interrupt Drawing On or Off	CTRL-I	Status
Kill or Save Pictures	META-CTRL-STATUS	Picture
Kill or Save Presentations	SUPER-STATUS	Presentation
Kill Picture	META-CTRL-K	Picture
Kill Presentation	SUPER-K	Presentation
Line Mode	META-L	Draw
List or Select Presentation	SUPER-L	Presentation
List Pictures	META-CTRL-L	Picture
Load Presentation	SUPER-F	Presentation
Modify Presentation	SUPER-M	Presentation

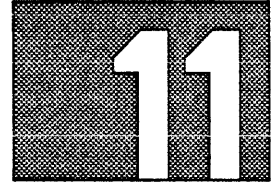
Table 10-5 Graphics Editor Keystroke Assignments (Continued)

Command Name	Assigned Keystroke	Menu or Submenu
Mouse Down One	↓	—
Mouse Down Three	HYPER-↓	—
Mouse Down Two	SUPER-↓	—
Mouse Left One	←	—
Mouse Left Three	HYPER-←	—
Mouse Left Two	SUPER-←	—
Mouse Right One	→	—
Mouse Right Three	HYPER-→	—
Mouse Right Two	SUPER-→	—
Mouse Up One	↑	—
Mouse Up Three	HYPER-↑	—
Mouse Up Two	SUPER-↑	—
Move Objects Mode	META-M	Function
Next Picture	META-CTRL-N	Picture
Paint Mode	META-B	Draw
Pan Down	CTRL-↓	Windowing
Pan Down Half	META-↓	Windowing
Pan Down Third	META-CTRL-↓	—
Pan Left	CTRL-←	Windowing
Pan Left Half	META-←	Windowing
Pan Left Third	META-CTRL-←	—
Pan Right	CTRL-→	Windowing
Pan Right Half	META-→	Windowing
Pan Right Third	META-CTRL-→	—
Pan Up	CTRL-↑	Windowing
Pan Up Half	META-↑	Windowing
Pan Up Third	META-CTRL-↑	—
Polyline Mode	META-P	Draw
Previous Picture	META-CTRL-P	Picture
Print Graphics File	CTRL-P	Picture
Read Background Picture	META-CTRL-B	Picture
Rectangle Mode	META-R	Draw
Redraw Picture	CLEAR SCREEN	Picture
Reorder Pictures	META-CTRL-O	Picture
Restore Presentation	SUPER-R	Presentation
Restore Subpicture	HYPER-R	Subpicture
Restore User Status	HYPER-SUPER-R	Function
Revert Picture	META-CTRL-R	Picture
Revert Picture Status	HYPER-SUPER-V	Function
Revert User Status	HYPER-SUPER-T	Function
Ruler Mode	META-N	Draw
Save Picture	META-CTRL-S	Picture
Save Picture Compiled Mode	CTRL-E	Status
Save Presentation	SUPER-S	Presentation
Save Subpicture	HYPER-S	Subpicture
Save User Status	HYPER-SUPER-S	Function
Scale Objects Mode	META-Q	Function
Scale Thickness On or Off	CTRL-Q	Status
Set Fill Color, Edge Color, ALU	CTRL-C	Status
Set Font	CTRL-F	Status
Set Layout	CTRL-L	Status

Continued

Table 10-5 Graphics Editor Keystroke Assignments (Continued)

Command Name	Assigned Keystroke	Menu or Submenu
Set Min Dot Delta	CTRL-B	Status
Set Min Nil Delta	CTRL-N	Status
Set Pick Tolerance	CTRL-R	Status
Set Rerindow Area	META-V	Windowing
Set Tab Width	CTRL-S	Status
Set Thickness	CTRL-T	Status
Set X Grid Size	CTRL-X	Status
Set Y Grid Size	CTRL-Y	Status
Show Default Window	META-CLEAR SCREEN	Windowing
Show Entire Picture	CTRL-CLEAR SCREEN	Windowing
Spline Mode	META-S	Draw
Text Mode	META-W	Draw
Triangle Mode	META-T	Draw
Undefine Subpicture	META-U	Subpicture
Undo	UNDO	Function
View or Modify Status	STATUS	—
Zoom Grid On or Off	CTRL-Z	—
Zoom In	CTRL->	Windowing
Zoom In Three	META-CTRL->	—
Zoom In Two	META->	Windowing
Zoom Out	CTRL-<	Windowing
Zoom Out Three	META-CTRL-<	—
Zoom Out Two	META-<	Windowing



TREE EDITOR

Overview

11.1 The tree editor can display any kind of data organized in a tree structure. You can then pan or zoom on the tree, click on nodes to see what they represent, and alter the structure of the tree. The tree editor can display trees either horizontally or vertically to maximize the amount of information in the window.

The tree editor can be used for any tree when you provide an interface that enables the tree editor to traverse the tree structure. Two sample interfaces, one that displays strings and another that displays flavors and their components, are included in the software and are discussed in this section.

The tree editor can operate on the nodes in the tree. You provide an interface for handling mouse clicks so that you can operate on the nodes in your tree. Paragraph 11.2, The Accessor File, and paragraph 11.3, Editing Methods, provide information on creating this interface.

When editing your tree using the tree editor, you must provide the methods described in paragraph 11.3, Editing Methods, to update data in the nodes for each editing function.

Paragraph 11.4, Tree Editor Functions, and paragraph 11.5, Tree Editor Variables, discuss functions and variables that are available to you for use in your interface code.

All of the functions, flavors, and variables discussed in this section are in the TREE package unless otherwise specified.

Tree Editor Display 11.1.1 Figure 11-1 shows the tree editor display for one of the sample interfaces, the string displayer. The tree for the string is displayed vertically, which is better for trees with more levels and fewer nodes at each level.

Figure 11-1 Vertical Window Display

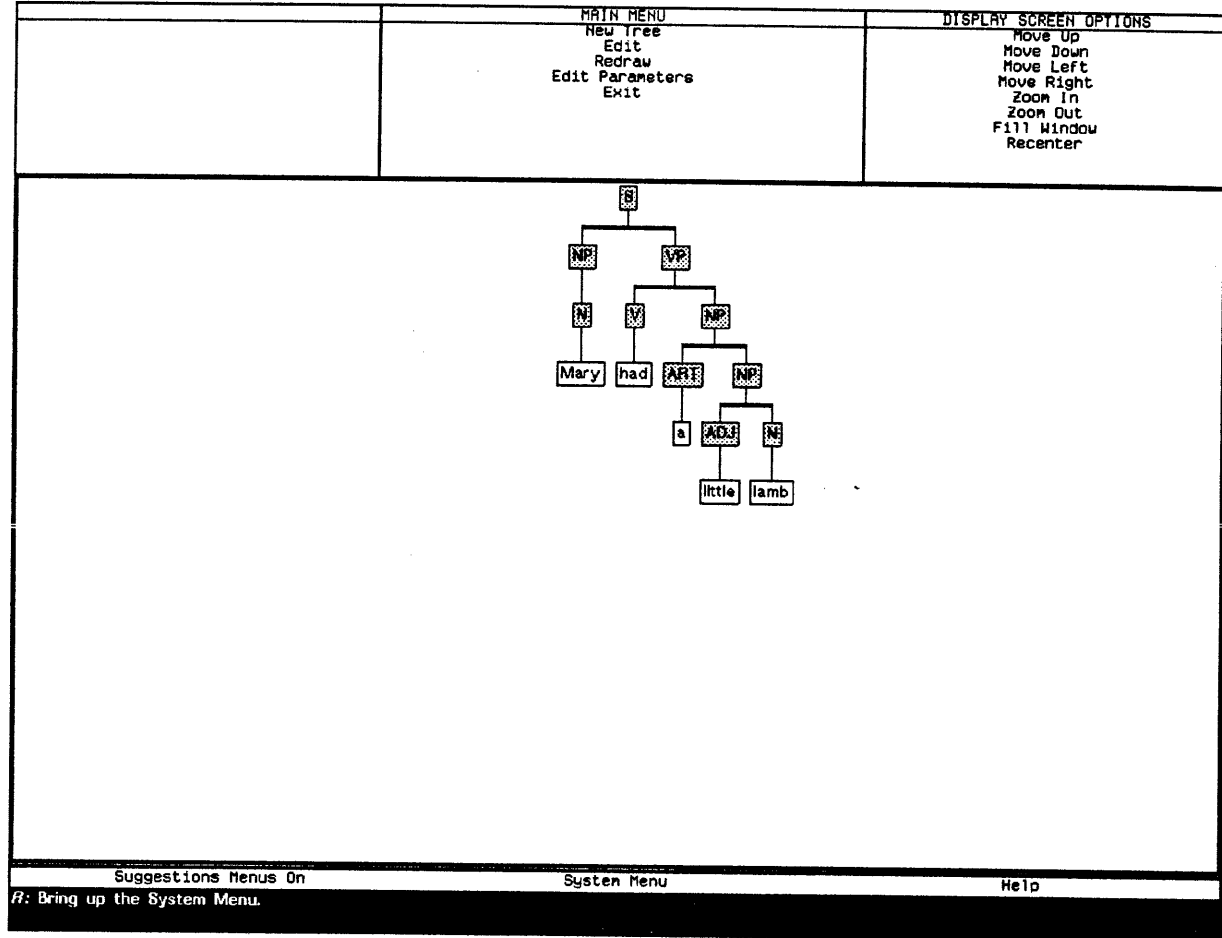
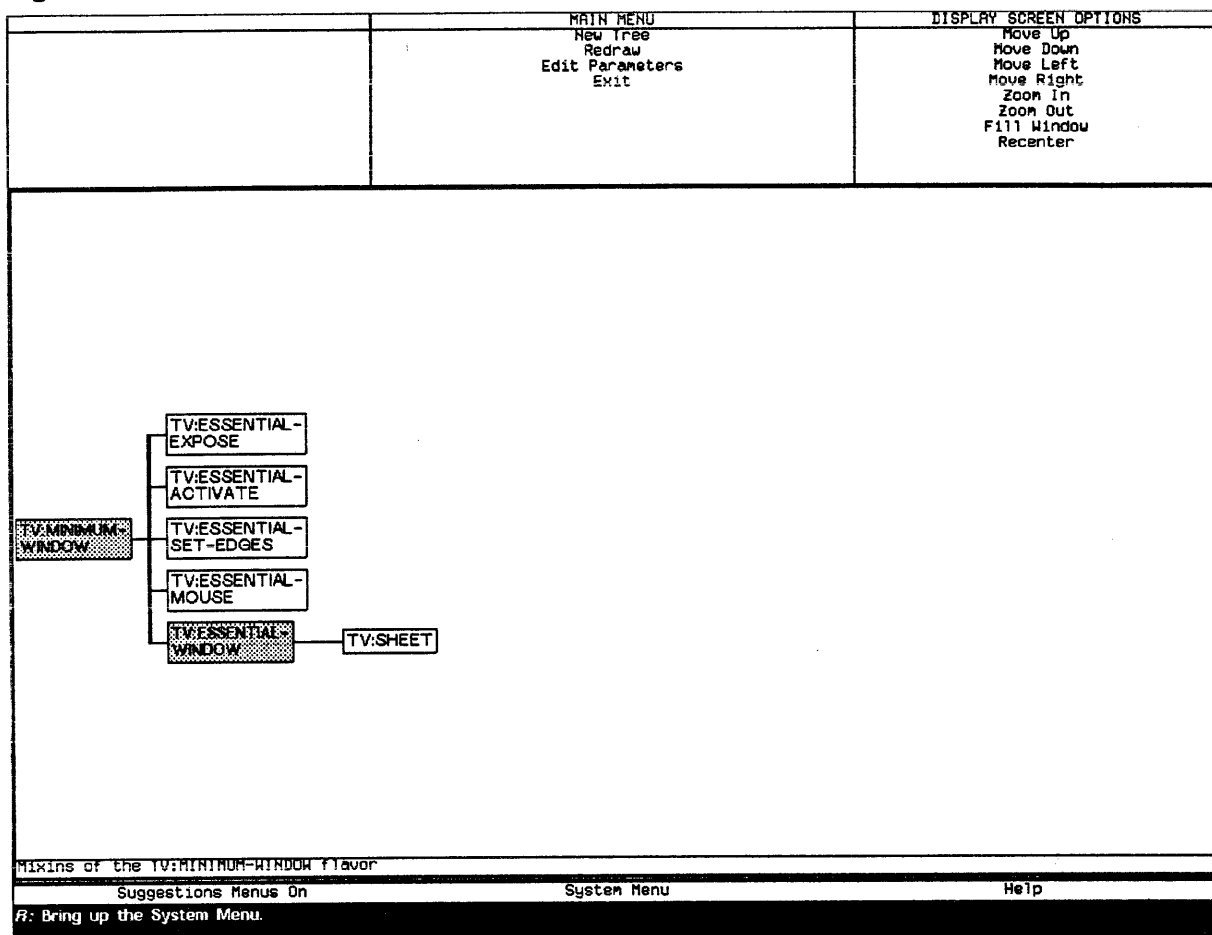


Figure 11-2 shows the tree editor display for the other sample interface, the flavor displayer. The tree for the flavor and its components is displayed horizontally, which is better for trees with fewer levels and more nodes at each level.

Figure 11-2 Horizontal Window Display



Loading the Tree Editor

11.1.2 Enter the following to load the tree editor software:

```
(make-system 'tree :noconfirm)
```

The Sample Interfaces

11.1.3 To see how the tree editor can be used, look at the sample interfaces.

String Displayer

11.1.3.1 To look at the example that displays strings, enter the following:

```
(run-string-displayer)
```

Two kinds of nodes are displayed: terminal and nonterminal. Both are the same shape, but the nonterminal ones are highlighted. The mouse documentation window lists the functions that are associated with the mouse buttons when you position the mouse cursor over a node. The functions are different for terminal and nonterminal nodes.

Two menu panes at the top of the window list available operations.

This interface displays symbols or strings, and lists of symbols or strings. To see the code for the interface, look at the following files:

- SYS:TREE-EDITOR.STARTER-KIT;STRING-ACCESSORS.LISP
- SYS:TREE-EDITOR.STARTER-KIT;STRING-EDIT-METHODS.LISP

Flavor Displayer

11.1.3.2 The other sample interface displays flavors and their components. You can click on nodes to see the methods, the instance variables, and the files that the definitions came from. To look at this example, enter the following:

```
(run-flavor-displayer)
```

You are then prompted to provide a flavor name. For example, try `w:minimum-window`.

This interface does not allow editing. To see the code for this interface, look at the SYS:TREE-EDITOR.STARTER-KIT;FLAVOR-ACCESSORS.LISP file.

The mouse documentation window lists the functions associated with the mouse buttons when the mouse cursor is over a node. The functions are different for the root node. Notice that the file uses the string displayer to display the methods of flavors.

Running the Tree Editor

11.1.4 To run the tree editor, use the `tree:display` function. Both `run-string-displayer` and `run-flavor-displayer` call this function.

```
tree:display tree &key (:superior w:selected-window) Function
(:application-type *default-application-type*) (:level 0)
(:vertical? *default-vertical*) (:edit? nil)
(:init-label *default-init-label*)
(:adjust-to-sup-size *default-adjust-to-sup-size*)
(:frame nil) (:return-window superior) (:allow-new-tree t)
```

This function is the top-level driver for the tree editor. The *tree* argument is the tree to be displayed and/or edited. This is the only required argument.

:superior — The window from which the tree editor obtains its size and position if the `:adjust-to-sup-size` keyword is true. The default is `w:selected-window`.

:application-type — The application flavor name that defines all the accessor functions to handle this kind of data. If you do not specify `:application-type`, the value of `*default-application-type*` is used; if `*default-application-type*` is nil and there is only one value in `*known-application-types*`, the value of `*known-application-types*` is used.

:level — The maximum depth to be displayed. The top node displayed is at level 1; 0 displays the entire tree. The default is 0.

:vertical? — This keyword controls the orientation of the display. The value true produces a vertical (top-down) orientation; nil produces a horizontal orientation.

:edit? — This keyword controls whether destructive editing is allowed on the tree. The default is nil.

- :init-label:** — The label you want displayed in the bottom of the display pane. If you do not specify a label, the value of ***default-init-label*** is used.
- :adjust-to-sup-size** — If this keyword is true, the tree window takes on the size and position of its superior. Otherwise, it fills the screen. The default is the value of ***default-adjust-to-sup-size***.
- :frame** — This keyword contains the value of the tree frame process, if it is being run in a process. If this keyword is nil, the tree frame is allocated as a resource without a process. The default is nil.
- :return-window** — The window that is to be exposed when the tree frame is deactivated. The default is the value of the **:superior** keyword.
- :allow-new-tree** — If this keyword is true, the tree editor includes the New Tree option in its menu. If creating new trees does not concern your application, this keyword should be nil. The default is t.

The Accessor File

11.2 You must provide an accessor file so that the tree editor can create and display the tree.

To allow you to display different types of trees, the tree editor must have a flavor for each type of tree. The tree editor accesses each tree using methods for that flavor.

Your accessor file also includes forms that add your flavor to the list of tree flavors and determine the function of the mouse buttons while displaying a tree of that flavor.

Complete the steps described in the following numbered paragraphs to create your accessor file. The discussion of each form that you need to create is followed by a template for that form.

The following files contain examples that can be helpful for understanding the forms that you provide in your accessor file:

- SYS:TREE-EDITOR.STARTER-KIT;STRING-ACCESSORS.LISP
- SYS:TREE-EDITOR.STARTER-KIT;FLAVOR-ACCESSORS.LISP

No examples of working code are included in this section of the manual.

Adding to the List of Tree Flavors

11.2.1 The first form in your accessor file adds the name of the flavor to the global list of defined flavors:

```
(push your-flavor *known-application-types*)
```

The variable ***known-application-types*** is the list of known flavor names.

Defining the Function of the Mouse Buttons

Node Types

Association List for the Node Types

11.2.2 The next step is to decide what values to return when you click on a node in the tree. Clicking on a node can cause the tree editor to perform an action or return information.

11.2.2.1 Before you specify the actions the tree editor can take, you must decide what types of nodes are to be displayed in the tree. You can have as many types of nodes as necessary for your tree.

In the string displayer, two types of nodes appear: terminal and nonterminal. In the flavor displayer, the three types of nodes are depends-on-flavors, depended-on-by-flavors, and component-flavor.

11.2.2.2 The type of the node and how many times a particular button was clicked determine the action taken. For each node type, you need to have an association that defines the actions to take for each mouse click.

You can define any number of actions for each node type; four actions can be assigned to clicking left or middle once or twice, and you can also define a list of items that appear in a menu when the mouse is clicked right once.

The form that you use is a `defparameter` to define the association list as a constant. The `defparameter` form should have the following structure:

```
(defparameter application-alist-name
  '((node-type-1-info) (node-type-2-info) ... (node-type-n-info)))
```

The `(node-type-n-info)` forms have the following structure:

```
(node-type-name
  (left-once left-twice middle-once middle-twice)
  (:mouse-l-1 "Action taken when you click left once"
   :mouse-l-2 "Action taken when you click left twice"
   :mouse-m-1 "Action taken when you click middle once"
   :mouse-m-2 "Action taken when you click middle twice"
   :mouse-r-1 "Menu of available functions")
  ("First Menu Item" :value first-item
   "Action taken when you click on first menu item")
  ("Second Menu Item" :value second-item
   "Action taken when you click on second menu item")
  ...
  ("Nth Menu Item" :value nth-item
   "Action taken when you click on nth menu item"))
```

You must include a `(node-type-n-info)` form for each type of node.

The name for the type of node that the form describes is the value for `node-type-name`. Next, you provide the four keywords that are returned for the corresponding mouse clicks. Use `nil` instead of a keyword for any mouse click that does not have an associated action. The action associated with these keywords is actually specified in the method `:handle-node` that you create in a later step.

The next list specifies the documentation that is visible in the mouse documentation window when the mouse cursor is positioned over a node of the `node-type-name` type. Alternate the `:mouse-n-m` keywords with the strings that describe the action that is to be taken when that mouse click is used. Do

not include the `:mouse-r-1` keyword if you do not plan to have a right click menu for this type of node.

The remaining lists determine the menu that is displayed when you click right once. Each list contains three things:

- The string that appears in the menu
- The keyword `:value` and the value that is returned when this item is clicked on
- The documentation that appears in the mouse documentation window when the mouse cursor is positioned over this menu item

You can have as many menu items as necessary. You can also choose to not have a right click menu. If you do not use the right click menu, your (*node-type-n-info*) form should not have any lists after the documentation list for the mouse documentation window.

Defining the Flavor

11.2.3 The next step is to define the flavor for the type of tree. This flavor is used only to collect the methods for your tree; therefore, no components or instance variables are necessary.

However, you can make `item-type-alist` a gettable instance variable with the default value of *application-alist-name*. If you do not, you must create a method `:item-type-alist` that returns the value of *application-alist-name*.

Your flavor should have the following form:

```
(defflavor your-flavorname
  ((item-type-alist application-alist-name))
  ()
  :gettable-instance-variables)
```

Building a Displayable Tree

11.2.4 To build a displayable tree of your data, you need to define several methods for the flavor.

The `:first-node` Method

11.2.4.1 The method `:first-node` returns the data that is stored in the root node. This data must include enough information to find the children of the root node for your tree:

```
(defmethod (your-flavorname :first-node) (tree) body)
```

The `:children-from-data` Method

11.2.4.2 The second method, `:children-from-data`, returns a list of the children of *node*. This method is first called with the data that `:first-node` returns:

```
(defmethod (your-flavorname :children-from-data) (node) body)
```

This method is called recursively on each member of the list it returns, and each node passed back needs to contain enough information to find its children. The first child returned is the one drawn leftmost or topmost. If *node* has no children, it should return `nil`.

The :print-name Method 11.2.4.3 The third method, `:print-name`, returns either a string or a list that is used to display *node*:

```
(defmethod (your-flavorname :print-name) (node) body)
```

If a string is returned, the tree editor makes a text object from the string.

However, if the method returns a list, the first element is the name of that node, and the rest of the list is a list of graphics objects.

The origins of the graphics objects are used only for placing the objects relative to each other. The group of objects is placed in the correct spot on the display automatically. For example, if a node were to be represented by a trashcan drawing on the display, this method might return the following:

```
("trashcan" (circle-instance line-instance1 line-instance2))
```

For information on creating graphics entity instances, refer to the *Explorer Window System Reference*.

The :font-type Method 11.2.4.4 The fourth method, `:font-type`, returns the graphics window system font that is used to display the text for *node*:

```
(defmethod (your-flavorname :font-type) (node) body)
```

Evaluating `gwin:*font-list*` shows the names of the fonts you can use. This method must return one of these font names when given the data stored in *node*.

If you are not using text strings or want all nodes to use the standard font, return `nil`.

The :highlight-function Method 11.2.4.5 The fifth method, `:highlight-function`, returns a form that, when evaluated, indicates whether *node* should be highlighted:

```
(defmethod (your-flavorname :highlight-function) (node) body)
```

Highlighting is implemented by coloring the inside of the rectangle in which the node is drawn.

Each node has a highlighting form that is evaluated before it is drawn. If the evaluation of the form returns true, the node is highlighted; otherwise, it is not highlighted.

This method returns a form to be stored with the node because the form must be evaluated when the node is drawn. Different types of nodes can have different highlighting forms.

The :find-type Method 11.2.4.6 The sixth method, `:find-type`, returns the type of *node*. The types of nodes are the same ones you defined in the association list constant:

```
(defmethod (your-flavorname :find-type) (node) body)
```

The `:handle-node` Method 11.2.4.7 The seventh method, `:handle-node`, handles the information in the association list:

```
(defmethod (your-flavorname :handle-node)
  (node type choice instance) body)
```

This method takes four arguments: *node* is the data that represents the current node; *type* is the type for *node*; *choice* is the keyword that was returned from the mouse click according to your association list constant; and *instance* is the flavor instance that represents the current node. Note that *instance* includes the information in *node* and *type*, but the data and type for the current node are passed as separate parameters for easier access.

Some actions require knowledge about a node's parent and children. This information is available in the gettable instance variables `parent` and `children` of *instance*.

The body of the `:handle-node` method determines the action that is taken after a mouse button is clicked. For example, you can use `case`:

```
(case type
  (type1 (case choice
          (choice1 (action1))
          (choice2 (action2))
          ...
          (choicew (actionw))))
  (type2 (case choice
          (choice1 (action1))
          (choice2 (action2))
          ...
          (choicex (actionx))))
  ...
  (typen (case choice
          (choice1 (action1))
          (choice2 (action2))
          ...
          (choicex (actionz))))))
```

Be sure that all the keywords you defined in your association list constant are handled here. This method can return four types of values:

- `nil` — If `nil` is returned, the monitor beeps to let you know that no action was taken.
- `t` — If `t` is returned, no further action is taken.
- `'tree:new-tree` — If `'tree:new-tree` is returned, a new tree is displayed using the special variable `tree:*tree` to build the new tree. Be sure to put the new value into `tree:*tree` before returning `'tree:new-tree`.
- An item for a scroll window — If any other value is returned, the object is displayed in a pop-up scroll window. Make certain that this item is in a form acceptable to the `w:scroll-window` flavor. Refer to the *Explorer Window System Reference* for further information.

You can use two functions, `tree:string-item` and `tree:grind-item`, to make sure that the item is acceptable to the scroll window. The first function converts strings into acceptable items for the scroll window. The second function converts arbitrary forms to items for the first function.

To return a list of items, put the items into a list as follows:

```
(list nil item1 item2 ...)
```

For information on functions you can use to write your methods, see paragraph 11.4, Tree Editor Functions.

The :get-new-tree Method 11.2.4.8 If you want to display a different tree without exiting the tree editor, you must provide the `:get-new-tree` method.

```
(defmethod (your-flavorname :get-new-tree) ( ) body)
```

You can have the `:get-new-tree` method get data for a tree in several ways. You can use a pop-up window, call another program, or use any method or function. The data this method returns is used as a parameter in a call to the `:first-node` method.

The main menu for the tree editor includes the New Tree option if the `:allow-new-tree` keyword for the `tree:display` function is `t`, the default. When you click on New Tree, `:get-new-tree` is called.

If you do not use this feature, the `:allow-new-tree` keyword of the `tree:display` function must be `nil`.

Editing Methods

11.3 The tree editor allows several kinds of editing. You can insert or delete nodes, and you can delete subtrees. If you want to store data in the nodes to reflect these changes, you need to provide the methods as described in this numbered paragraph.

Applications that allow editing must provide these methods. If you want the user to edit the display without changing the actual data, the methods must return `nil`.

Each method is associated with the flavor you defined for that type of tree.

Each node in a tree displayed by the tree editor is an instance of the flavor `tree:tree-node`. The flavor has the instance variables `parent`, `children`, and `data`. The data that the node represents is in the instance variable `data`.

If the change affects the way the node is drawn in the display, you can either reset the node's list of graphics objects or call the `tree:update-node` function to make sure that the node has the right name and highlight function. All editing changes must be reflected in the data for a node before calling `tree:update-node`.

For an example of working code, refer to the file `SYS: TREE-EDITOR.STARTER-KIT; STRING-EDIT-METHODS.LISP`.

The :add-node-before Method

11.3.1 The first editing method you must provide is `:add-node-before`:

```
(defmethod (your-flavorname :add-node-before)
  (new-node selected-node) body)
```

This method updates the tree when `new-node` is inserted before `selected-node`. You need to change your data to reflect the resulting changes in parents and children.

**The :add-node-after
Method**

11.3.2 The second editing method you must provide is `:add-node-after`:

```
(defmethod (your-flavorname :add-node-after)
  (new-node selected-node) body)
```

This method updates the tree when *new-node* is inserted after *selected-node*.

**The :add-brother-node
Method**

11.3.3 The third editing method you must provide is `:add-brother-node`:

```
(defmethod (your-flavorname :add-brother)
  (new-node selected-node) body)
```

This method adds *new-node* as a brother to *selected-node*.

To find out if the brother was inserted above (or to the left) or below (or to the right) of the selected node, send the `:parent` message to one of them. The order in which *new-node* and *selected-node* appear in the list of children determines how the nodes are displayed. The one that appears first in the list is displayed above or to the left.

**The :delete-subtree
Method**

11.3.4 The fourth editing method you must provide is `:delete-subtree`:

```
(defmethod (your-flavorname :delete-subtree)
  (selected-node) body)
```

This method deletes the entire subtree headed by *selected-node*. You need to remove any data that the ancestors have about the selected node and its children.

**The :delete-yourself
Method**

11.3.5 The fifth editing method you must provide is `:delete-yourself`:

```
(defmethod (your-flavorname :delete-yourself)
  (selected-node) body)
```

This method deletes *selected-node* from the tree. Your data must be edited in this function, so the parent takes on the children of *selected-node* as its children.

**The :get-user-data
Method**

11.3.6 The sixth editing method you must provide is `:get-user-data`:

```
(defmethod (your-flavorname :get-user-data)
  () body)
```

This method returns the data for a new node. The parent and children for the new node are determined by where the node is being inserted in the tree.

Typically, the user is asked to supply the data through a pop-up window.

**Tree Editor
Functions**

11.4 The tree editor provides many functions that you can use to build the accessor files. These functions perform formatting for the scroll window, redraw the tree, display different parts of the tree, pan or zoom on the tree, print error messages, and update nodes.

**Formatting for the
Scroll Window**

11.4.1 Use `tree:string-item` and `tree:grind-item` to format a string or a Lisp form into acceptable output for the scroll window. You can use these functions in your `:handle-node` method.

`tree:string-item` *value* &optional *format-string* Function

Formats a string into the form needed by the scroll window. Newlines are eliminated. The *value* argument must be either a string or something that can be coerced into a string.

The *format-string* argument is a control string as used by the `format` function. The default for *format-string* is "-A".

`tree:grind-item` *form* Function

Creates a string out of the form and then passes the string to `tree:string-item`.

Redrawing the Tree

11.4.2 You can call one of two functions to redraw the tree: `tree:tree-redraw` or `tree:tree-draw-after-small-changes`.

`tree:tree-redraw` Function

Recalculates the position of the nodes and then redraws the tree. To force the tree to recalculate the positions of the nodes, set the special variable `tree:*force-recalculate` to `t` before calling `tree:tree-redraw`.

`tree:tree-draw-after-small-changes` Function

Redraws the tree without repositioning any nodes.

**Expanding and
Contracting Nodes**

11.4.3 You can use three functions to expand or contract nodes in the tree. Expansion and contraction allows you to focus on the parts of the tree you need to see.

Contracting a node means that descendants are not displayed; however, the descendants are not deleted. When the node is expanded, the descendants are displayed.

`tree:expand-node-with-redraw` *instance* *amount* &optional *strict-level* Function

Adds the descendants of the node *instance* to the display. If *amount* is 0, all descendants are displayed. If *amount* is not 0, the node is expanded the specified number of levels. If *strict-level* is true, no descendants beyond *amount* levels in the subtree headed by *instance* are displayed. The default for *strict-level* is nil.

tree:contract-node-with-redraw *instance* &optional *full-redraw* Function

Contracts the node represented by *instance* so that all its descendants are not displayed. If *full-redraw* is true, the tree is rebalanced before it is drawn. The default for *full-redraw* is nil. This function returns true if the node has descendants.

tree:expand-contract-with-redraw *instance amount*
&optional *strict-level full-redraw* Function

Contracts the node represented by *instance* if the node has been expanded. Otherwise, this function expands the node. The arguments are passed to the **tree:expand-node-with-redraw** or **tree:contract-node-with-redraw** function, whichever is appropriate.

Panning and Zooming 11.4.4 You can pan or zoom on the pane in which the tree is displayed by using the following functions.

tree:fill-window Function

Scales the display so that the tree fills the window.

tree:move-to-front *node* Function

Pans the display pane so that *node* is at the center of either the left edge or the top edge, depending on whether the tree is displayed horizontally or vertically.

tree:zoom-window *direction button* Function

Zooms the window. If *direction* is 'tree:in, the window is zoomed by a factor of 2. If *direction* is 'tree:out, the window is zoomed by a factor of 0.5. If the value of *button* is 4 (in other words, if the right button was clicked), the zoom is increased; the factor is either 4 or 0.25, depending on the direction.

tree:pan-window *dx dy button* Function

Pans by the distances specified by *dx* and *dy* if the right button was clicked (in other words, if the value of *button* is 4). Otherwise, this function pans by one-third of the specified distances.

tree:return-to-default-window Function

Undoes any panning or zooming and redraws the tree display pane.

Displaying Error Messages 11.4.5 Error messages appear in the utility pane, which is the window at the top left corner.

tree:complain *message* Function

Prints an error message in the utility pane in reverse video and makes the monitor beep.

The *message* argument is a list of strings; each string contains one line printed in the utility pane. The string "click any button to continue" is appended to the end of your message.

Changing How a Node is Drawn 11.4.6 Changing the data in a node can alter how it should be represented on the screen.

For example, if you have two node types, terminal and nonterminal, and you delete all of the children of a node, the node's type changes from nonterminal to terminal. If you draw nonterminal nodes as circles and terminal nodes as triangles, you need to tell the edited node to change graphics representation from a circle to a triangle.

tree:update-node *node* Function
 This function updates the name, highlight function, and graphics objects for *node*.

Tree Editor Variables 11.5 You can change the values of some of the tree editor's special variables in your access methods. Some special variables are local to each tree editor, and others are global variables common to all tree editors.

Local Variables 11.5.1 Special variables that affect the tree display are local to each tree editor. You can change their values when necessary.

tree:*force-recalculate Variable
 If this variable is true, the positions of the nodes are recalculated when the tree is redrawn.

tree:*max-level Variable
 The value of this variable is the lowest level of the tree that can be displayed. When the value is 0, the whole tree can be displayed.

tree:*root-node Variable
 This variable contains the data for the root of the tree. It is an instantiation of the **tree:tree-node** flavor.

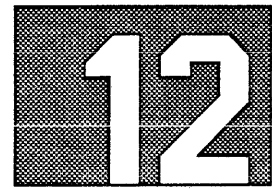
tree:*tree Variable
 The data passed to the editor when the tree display was created is stored in this variable. This is the data you gave as the first argument to the **tree:display** function.

tree:*tree-window Variable
 This is the instance of the current tree editor frame.

tree:*vertical? Variable
 If this variable is true, the tree is displayed vertically. If this variable is nil, the display is horizontal.

Global Variables 11.5.2 Special variables that affect the tree editor window are common to all tree editors. You can change the values of the global variables when necessary.

- tree:*default-adjust-to-sup-size*** Variable
 The value of this variable is the default for the **:adjust-to-sup-size** keyword when you call the **tree:display** function. When this variable is true, the size of the tree editor window is the size of the superior window. Otherwise, the tree editor window fills the screen. The initial value is **t**.
- tree:*default-init-label*** Variable
 The value for this variable is the default for the **:init-label** keyword in the **tree:display** function. If you do not specify a value for **:init-label**, the value of this variable is the label shown at the bottom of the tree editor. The initial value is **nil**.
- tree:*default-vertical*** Variable
 This is the default for the **:vertical?** keyword for the **tree:display** function. When the value of this variable is true, the tree is drawn with the root node at the top. Otherwise, the tree grows from left to right. The initial value is **nil**.
- tree:*default-application-type*** Variable
 The value of this variable is the default for the **:application-type** keyword to the **tree:display** function. The initial value is **nil**.
- tree:*known-application-types*** Variable
 This variable is a list of known application types. If the **:application-type** keyword is **nil** for the **tree:display** function, a menu of the known application types is displayed. This variable is used to create that menu. When you define an application flavor, you **push** the application flavor name onto this list.
- tree:*minimum-breadth-spacing*** Variable
 The value of this variable determines the minimum spacing between brother nodes in the tree editor display. The value is in pixels. The default is 10.
- tree:*minimum-depth-spacing*** Variable
 The value of this variable determines the minimum spacing in pixels between each level of the tree. The default is 30.
- tree:*scroll-window-height*** Variable
 The value of this variable determines how many lines of text fit in the scroll window. The default is 10 lines.
- tree:*scroll-window-width*** Variable
 The value of this variable determines the number of characters that fit on one line of the scroll window. The default is 50 characters.
- tree:*starting-point-offset*** Variable
 The value of this variable determines how far the first node is from the edge of the pane. The default is 5 pixels.
- tree:*truncation-for-scroll-window*** Variable
 When this variable is true, items displayed in the scroll window with strings too long to fit on one line are truncated at the last visible character. When the variable is **nil**, the remainder of the string wraps around and prints on the next line. The default is **nil**.



FONT EDITOR

Introduction

12.1 A *font* is an assortment of type all of one size and style. Fonts are used to make text used for different purposes appear different. For example, the System menu uses a large font for its title and smaller fonts for the items in the menu.

Explorer System Menu			
USER AIDS:	PROGRAMS:	WINDOWS:	DEBUG TOOLS:
Glossary	Backup	Arrest	Flavor Inspector
New User	Converse	Bury	Inspector
Profile	Font Editor	Change Layouts	Network
Suggestions	Hardcopy Menu	Create	Peek
	Lisp Listener	Edit Attributes	Trace
	Mail	Edit Screen	
	Namespace Editor	Kill	
	Telnet	Refresh	
	VT100 emulator	Reset	
	Zmacs Editor	Select	
		Split Screen	

Many programmers put their code in one font and their comments in a different font. The Zmacs editor has a minor mode, the Electronic Font Lock mode, that changes fonts automatically so that code and comments are in different fonts. For example, the following buffer shows a program with the code in medfnt and most of the comments in hl10b.

```
;;; -- Mode:Common-Lisp; Package:USER; Fonts:(MEDFNT HL10B HL12BI); Base:10 --
;;;
;;; This file contains Lisp forms that demonstrate how to use the graphics window system. To execute the examples, evaluate or compile the Lisp forms
;;; from a Zmacs buffer. Instructions for evaluating and compiling will appear immediately before the form. Note that little error checking is included
;;; in these forms, so that the essential code is not obscured.
;;;
;;; Step 1: Creating a Graphics Window
;;;
;;; This form splits the screen between a graphics window and this Zmacs window. They will be nonoverlapping so that both windows can be exposed
;;; and drawn on at the same time.
;;;
;;; To evaluate this form, first click middle over the opening parenthesis to select the form, and then evaluate it by pressing CTRL-SHIFT-E.
;;;
(LET ((zmacs (w:sheet-superior w:selected-window)))
  (MULTIPLE-VALUE-BIND (left top right bottom)
    (SEND (w:sheet-superior zmacs) :edges)
    ;;
    ;; Send a message to the Zmacs window to resize it.
    ;;
    (SEND zmacs :set-edges left (- bottom 391.) right bottom)
    ;;
    ;; Create a graphics window and set the global symbol window to point to it.
    ;;
    (setq window (make-instance 'gwin:graphics-window
      :bottom (- bottom 391.)
      :left left
      :right right
      :top top
      :expose-p t)))
  ;;
  ;; Set the global variable world to point to the world for the newly created graphics
  ;; window.
  ;;
  (setq world (SEND window :world)))
```

The Explorer system includes fonts for most uses. Almost all users and most programmers can find a font that fits their needs. If you need a special font, however, you can create one using the font editor.

This section describes, in general, what a font is and how to create a font using the font editor. For information about using fonts in files and buffers, see the *Explorer Zmacs Editor Reference*. For details about how a font is implemented and stored on the Explorer system, see the *Explorer Window System Reference*. For a list of all the fonts loaded on the Explorer system, see Appendix A, Explorer Fonts.

Properties of Fonts

12.1.1 A font is an array or list of up to 256 characters. Not all characters in a font need to be specified.

Each character or *glyph*, is both displayed and printed as an array of pixels, or dots. A character can be a letter, number, symbol, or special icon such as a mouse cursor. When a font is displayed in the font editor, the font you are working with (the *selected* font) is displayed in rows and columns next to a sample font. Within a font, the different glyphs are referred to by index numbers that indicate their position in the array. These index numbers appear in the font editor display as column and row heads and are displayed in decimal by default.

Some characters in a font cannot be redefined. These characters, which are between character codes 128. and 160., inclusive, are associated with control characters such as TAB, SYSTEM, NETWORK, and NEWLINE.

The characters within a font can be defined as a specific width—*fixed-width* fonts. When you print or display information in a fixed-width font, all the characters align in columns. In *variable-width* fonts, each character has its own width. When you print or display information in a variable-width font, the characters are proportionately spaced.

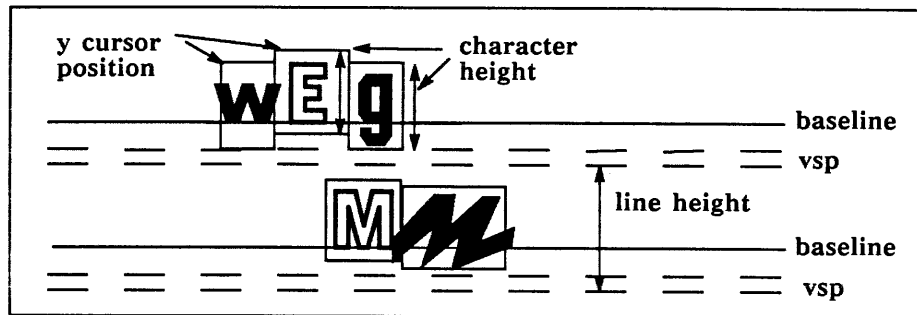
```
;; -*- Mode:Common-Lisp; Fonts:(COURIER HL12) -*-
Lines of fixed-width fonts
form columns of letters
because the letters are all
the same width.
...
Lines of variable-width fonts
do not form columns because
the letters are not all the
same width.
```

Each font also defines a *blinker* or *cursor* as a solid rectangle of a specified height. The width of the blinker varies according to the character it is highlighting. If the blinker is not highlighting a character, the blinker is the size specified by the blinker width and height.

Fonts in the Video Display

12.1.2 At any time, the information that a user types on the video display is shown using the *current font* for that window. If you are using several fonts in the window, the current font changes as the fonts displaying information change. The current font is one of the fonts contained in a *font map*, an attribute of a window. The font map is a list of all the fonts used to display information within the window. The fonts used to display specific tags and labels—for example, the **More** tag or the label for the window—are specified by global variables.

If you use several fonts to display information in a window, the window system computes a *line height*—the largest height of the fonts in the font map, plus the *vertical spacing*, the amount of space left between lines. The line height determines the amount of space allowed for each line of text. When fonts of different heights are shown on the same line, the characters are aligned by their *baselines*.



Families of Fonts

12.1.3 Although you can use many different fonts that vary greatly in size and style, you may want fonts that are very similar—fonts that belong to the same *family*. A family of fonts usually includes a font that is the basis of the family and variations of the original font, such as a boldface font, an italic font, and a bold italic font. By convention, the name of the original font is modified with suffixes to indicate the various fonts in the family. For example, in the family of fonts shown in the following figure, the Helvetica regular font that is twelve pixels in height is named hl12; the boldface font, hl12b; the italic font, hl12i; and the bold italic font, hl12bi.

FAMILIES OF FONTS:

This line is displayed in hl10 font.
 This line is displayed in hl12 font.
This line is displayed in hl12b font, a bold font.
This line is displayed in hl12i font, an italic font.
This line is displayed in hl12bi font, a bold Italic font.
This line is displayed in hl6 font, a very small font.

Each font in this family shares the same height, but some fonts differ in width. The font editor includes commands that help you create these families of fonts after the regular font is created.

Not all fonts belong to a family, however. For example, the mouse font, which is used to store the mouse cursors, is not a member of a family of fonts. It defines the various mouse cursors used in different applications. See the *Explorer Window System Reference* for more information about using a special mouse cursor for your application.

Font Directory

12.1.4 Fonts defined and used by the system software are stored in the FONTS directory in files of type XLD. You can also create fonts in other formats for use on other types of machines. Table 12-1 lists some of the commonly used fonts:

Table 12-1

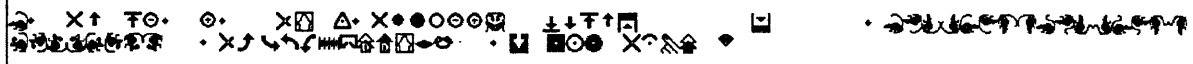
Some Commonly Used Fonts

Font	Description
cptfont	The default font for the U.S. market, used for almost everything.
medfntb	A boldface version of medfnt; the default font for the European market. When you use the Split Screen, for example, the Do It and Abort items are in this font.
medfnt	The default font in menus. It is a fixed-width font with characters somewhat larger than those of cptfont.
hl10	A very small font used for items in choose-variable-values windows that are currently not selected.
hl10b	A boldface version of hl10, used for selected items in choose-variable-values windows.
hl12i	A variable-width italic font. It is useful for italic items in menus.
tr10i	A very small italic font.
mouse	A font that contains the glyphs used as mouse cursors and icons.

```

;;; -- Mode:Common-Lisp; Fonts:(CPTFONT MEDFNT MEDFNTB HL12I TR10I HL10 HL10B MOUSE); Base:10 --
A sample of cptfont. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}
A sample of medfnt. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}
A sample of medfntb. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}
A sample of hl10. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}
A sample of hl10b. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}
A sample of hl12i. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}
A sample of tr10i. abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890-=' !##$%^&*()_+{}

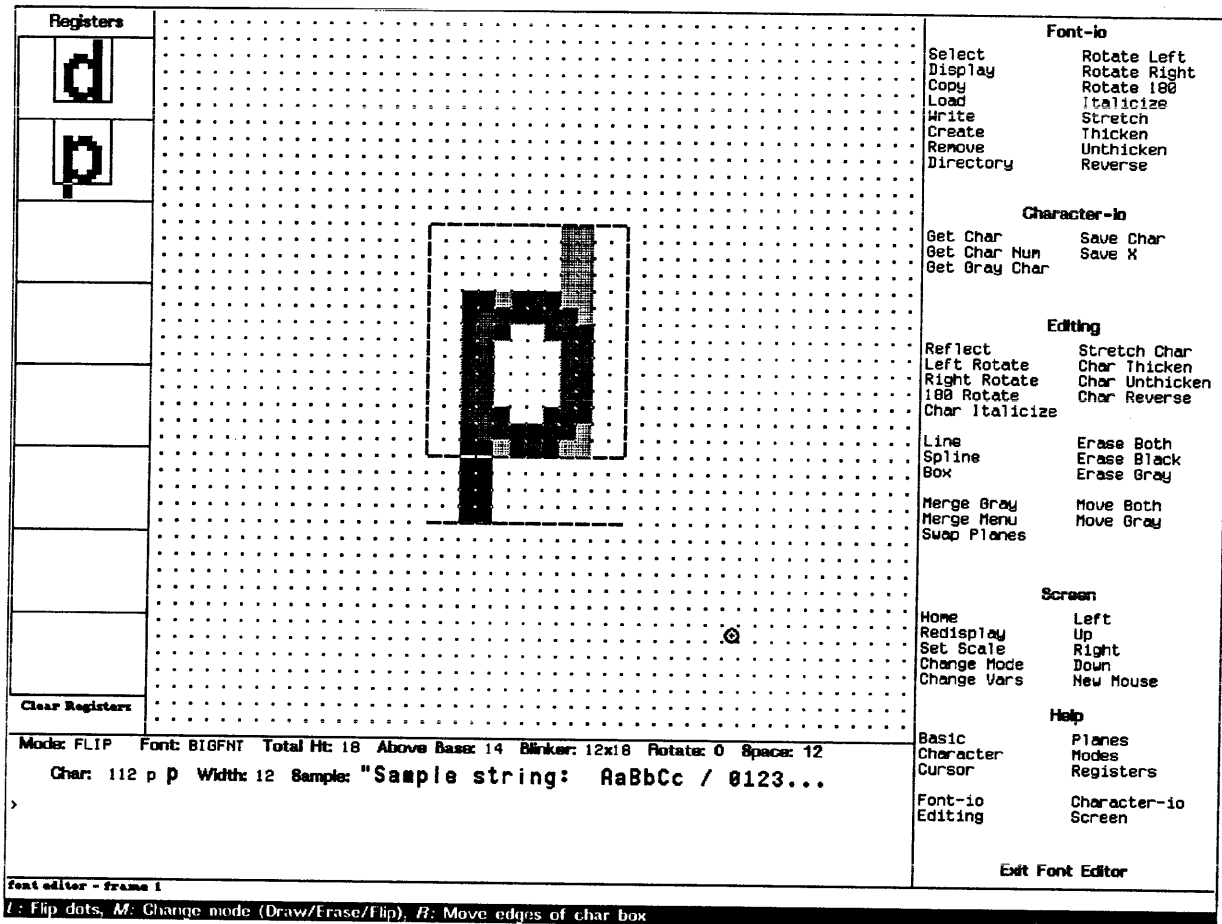
```



Certain fonts, specified by a list in the current load band, are loaded during boot operations. Additional fonts must be loaded before they can be used.

Font Editor Window

12.2 The font editor is the utility that enables you to manipulate fonts. The font editor window, as shown in the following figure, contains several special panes: command menus, an editing area, registers, a label pane, and a Lisp Listener pane. The following paragraphs describe these panes.



Command Menus

12.2.1 The font editor includes permanent command menus. The commands are grouped by function to help you use the editor. You can obtain more information about each group of commands by selecting its label. A typeout window appears over the editing area and displays information about the commands in that group.

When you execute a command by selecting an item from a menu and the command requires more information, you are prompted for the information in the Lisp Listener pane. This Lisp Listener pane is described in paragraph 12.2.5

You can also execute these commands by pressing a keystroke sequence rather than by selecting the command name from the menu. As you move the mouse over a command in the command menu, a short description of the command and its associated keystroke sequence is displayed in the mouse documentation window. Table 12-2 lists these keystroke sequences. You can also press CTRL-HELP to get a list of commands with their associated keystrokes; you can select and execute a command from this list.

Editing Area 12.2.2 The editing area, the large central area in the font editor window, contains the character box and grid that you use to edit individual characters in a font. The following paragraphs describe how the editing area appears when you are editing a character.

In addition, the editing area is used to display:

- A list of the loaded fonts when the font editor is first invoked, as well as when certain commands are invoked.
- The table of the characters in the selected and sample fonts when a font is selected or when certain commands are invoked.

If you have not selected a character to edit, the editing area of the font editor displays only a grid of dots and a character box.

Grid 12.2.2.1 The grid of dots within the editing area defines the pixels used to create a character. Each dot is a corner of a pixel; by moving the mouse cursor into the center area of a pixel and clicking left with the mouse, you can set, erase, or *flip* (change the state of) a pixel. To help you see a character, you can reduce or increase the scale of the grid.

Character Box 12.2.2.2 In the center of the editing area is a box with a line underneath it. This *character box* indicates the size of a character. A character's total *height* is the number of pixels from the top of the character box to the underscore; a character's height above baseline is the height of the character box. Height is constant for the entire font. The bottom of the box is the *baseline* of the character. Some characters, such as the lowercase letters p or g, have *descenders*, parts of the character that fall below the baseline.

A character's *width*, which is determined by the width of the character box, defines how far the cursor moves after the character has been displayed. Typically, the width of the character box is the same size or slightly larger than the width of the actual glyph. If a font is a fixed-width font, the width of the character box remains constant for all characters in the font. If a font is a variable-width font, the width of the character box changes with the characters. Fonts that have a width of 32 or more are called *fat* fonts. The Explorer system stores and displays fat fonts differently from other fonts. In addition, fat fonts require more time to display than other fonts. For more information about how fonts are stored internally and how they are displayed, see the *Explorer Window System Reference*.

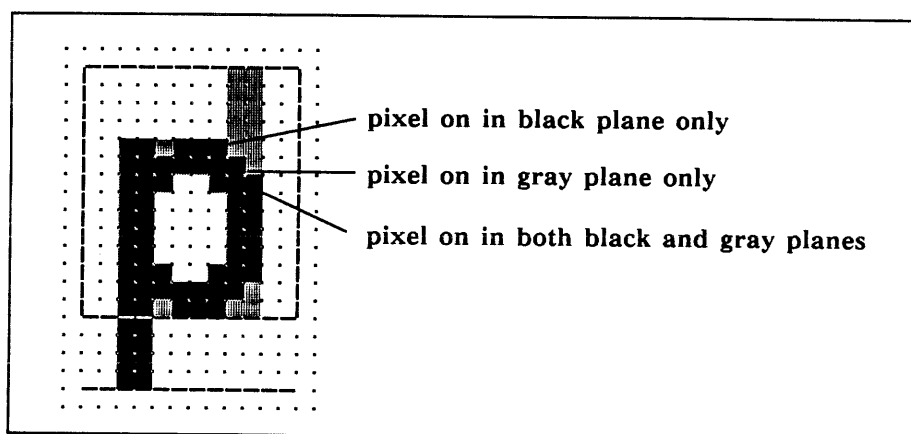
Most pixels within a character occur within the character box. Descenders fall below the baseline but above the underscore bar of the character box. In some fonts, such as italic fonts, pixels may occur outside the left and right margins of the box. You cannot, however, have any pixels in a character above the top line of the box or below the underscore. If any characters in a font have pixels set left of the character box, the font editor creates a kerning table for the font. The Explorer system requires more time to display fonts that include kerning tables than to display fonts that do not have kerning tables. For more information, refer to left kerning in the *Explorer Window System Reference*.

Black Plane 12.2.2.3 When you set a pixel in the editing area, you are actually setting a pixel in the *black plane*, a buffer that contains the character you are editing. Pixels set only in the black plane are displayed in black.

The black plane holds the current character, which can include editing changes that are not in the font itself. In the black plane, you can draw or modify characters, merge characters from registers or the gray plane, save characters into fonts, and so on.

Gray Plane 12.2.2.4 The *gray plane*, similar to the black plane, is a buffer that displays characters in the editing area. You can use a character in the gray plane as a reference or template for the character you are editing in the black plane. Pixels set in the gray plane only are displayed in a light gray; pixels set in both the black and gray planes are displayed in a medium gray.

You can exchange the contents of the black and gray planes, merge the contents of the planes, erase one or both of the planes, put or recover contents to a register, and so on.



Registers 12.2.3 The font editor window includes registers that can hold characters or parts of characters. By moving the mouse cursor into a register, you can store the contents of the black plane into that register, put the contents of that register into the black or gray plane, merge the contents with one of the planes, erase the contents, or save the contents into the font. You can also clear all the registers by executing the Clear Registers command.

Label Information

12.2.4 The label pane contains information about the font you are editing. If you are not editing a font, many of these fields are blank. You can change the contents of most of the fields by boxing the field and clicking left with the mouse. The system prompts you for the new information in the Lisp Listener pane. You can obtain information about a field by selecting its label.

Mode: FLIP Font: COURIER Total Ht: 13 Above Base: 11 Blinker: 9x13 Rotate: 0 Space: 9 Char: nnn None Width: 9 Sample: 'Sample string: RaBbCc / 0123...

The fields display the following information:

- **Mode** — One of three modes of drawing: *flip*, where the pixels are toggled to the opposite setting; *draw*, where the pixels are set; *erase*, where the pixels are erased.
- **Name of font** — The name of the font you are currently editing.
- **Total height** — The height in pixels of the character box.
- **Height above baseline** — The height in pixels of the character box above the baseline. (This height does not include descenders.)
- **Blinker size** — The width and height of the blinker for the font.
- **Rotation** — The rotation value of the font. For example, a font that you use to label the side of a graph might have a rotation value of 270.
- **Space** — The width of the space character for the font.
- **CHG flag** — A flag (the letters CHG, for changed) that appears if you have edited the character you are currently displaying.
- **The character displayed** — The decimal index number for the symbol, the symbol in this position in the standard Lisp character set, and the character in the black plane.
- **Width** — The width of this character.
- **Sample** — A sample string displayed in the font you are currently editing.

Lisp Listener Pane

12.2.5 The font editor window includes an interactive Lisp Listener pane that accepts keyboard input. Typically, messages and prompts for more information appear in this pane. You can use the same input editor commands in this Lisp Listener pane as you use in any Lisp Listener.

Help Features 12.2.6 The font editor includes several help features:

- You can select an item in the Help menu to display more information. Selecting any of these commands displays a brief introduction to the topic listed as its title. Some items also include more detailed help available by selecting the title of a subtopic.
- You can select the title of a menu, pane, or field, such as Font-io, to display more information. Selecting any of these titles displays a brief description of all the commands within that group. You can select the name of the command to display more information about that command, as shown in the following figure:

Registers	<p>*** Mouse a FONT-IO command name for additional documentation. ***</p> <p>FONT EDITOR HELP -FONT-IO-</p> <p>The following is a brief description of the commands relating to font-io that can be activated. These are located in the top-portion of the the Menu Pane and each of these commands are mouse-sensitive. The key assignments for each of these commands are displayed in the Mouse Documentation Line when the commands are highlighted with the mouse-cursor.</p> <p>SELECT - displays a menu of the currently loaded fonts. You can choose one of these fonts for editing with the mouse.</p> <p>DISPLAY - displays all characters in the current font.</p> <p>COPY - copies the current font to a new font name.</p> <p>LOAD - prompts for a filename of a font to be loaded.</p> <p>WRITE - prompts for a filename to which it will write the current font.</p> <p>CREATE - prompts for a name for a new font to be created.</p> <p>REMOVE - removes the specified font from the FONTS package.</p> <p>DIRECTORY - lists the fonts on a directory you specify. You can select a font from this list to be loaded.</p> <p>ROTATE, ITALICIZE, STRETCH, THICKEN, UNTHICKEN, REVERSE - modifies all characters of the current according to the specific action implied by the command name and saves the altered characters in a new font.</p> <p>*** Press the space bar to remove this output. ***</p>	<p>Font-io</p> <p>Select Rotate Left Display Rotate Right Copy Rotate 180 Load Italicize Write Stretch Create Thicken Remove Unthicken Directory Reverse</p> <p style="text-align: center;">Character-io</p> <p>Get Char Save Char Get Char Num Save X Get Gray Char</p> <p style="text-align: center;">Editing</p> <p>Reflect Stretch Char Left Rotate Char Thicken Right Rotate Char Unthicken 180 Rotate Char Reverse Char Italicize</p> <p>Line Erase Both Spline Erase Black Box Erase Gray</p> <p>Merge Gray Move Both Merge Menu Move Gray Swap Planes</p> <p style="text-align: center;">Screen</p> <p>Home Left Redisplay Up Set Scale Right Change Mode Down Change Vars New Mouse</p> <p style="text-align: center;">Help</p> <p>Basic PLine Character Modes Cursor Registers</p> <p>Font-io Character-io Editing Screen</p> <p style="text-align: right;">Exit Font Editor</p>
Clear Registers	<p>Mode: FLIP Font: HL12BI Total Ht: 13 Above Base: 10 Blinker: 7x13 Rotate: 0 Space: 7</p> <p>Char: 121 y y Width: 9 Sample: "Sample string: AaBbCo / 0123..."</p>	
font editor - frame 1		
Display additional documentation on this font-io command.		

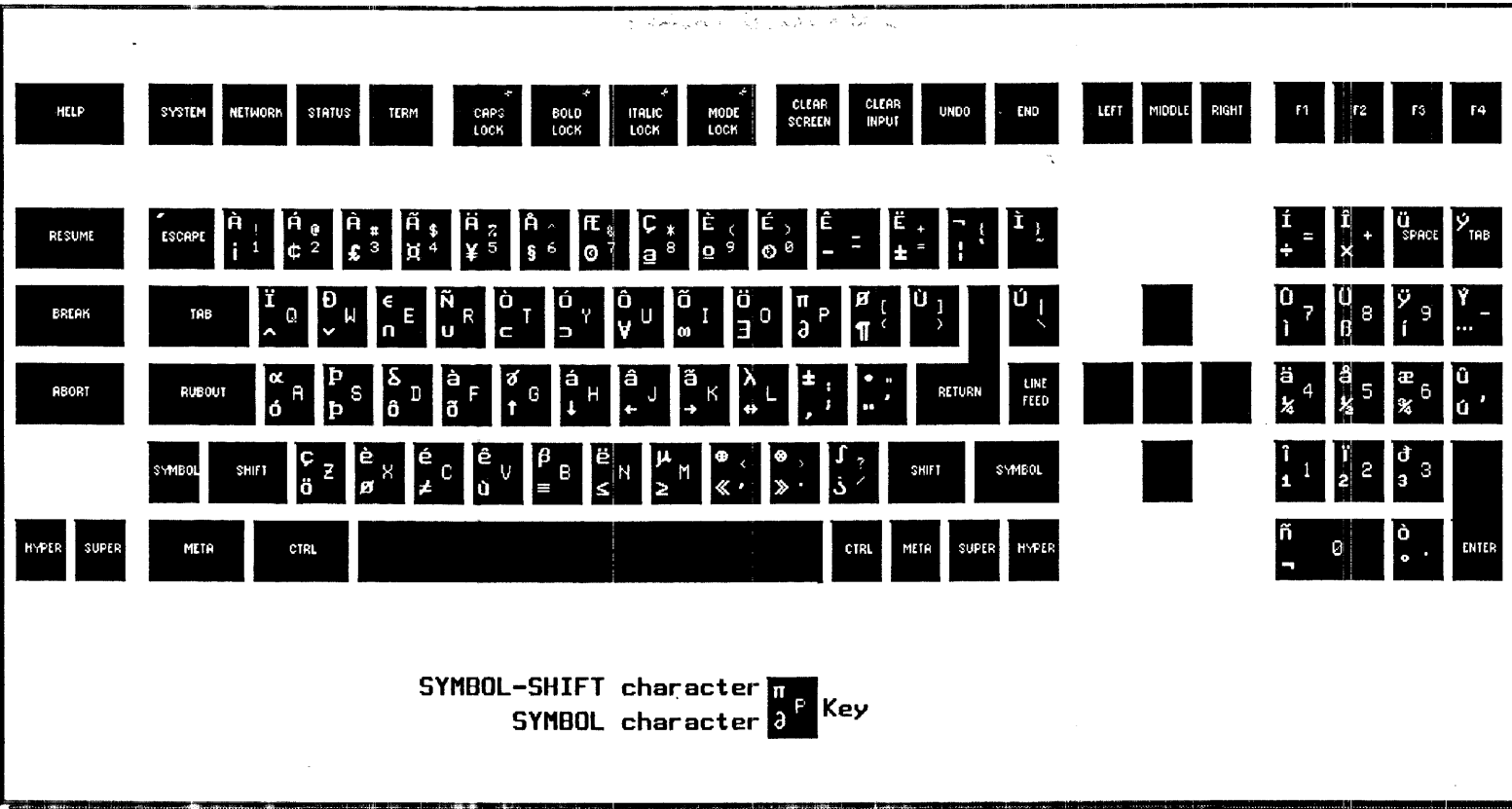
- You can press HYPER-CTRL-HELP to display a list of all the commands that you can execute by pressing a keystroke sequence. From this list, shown in Figure 12-1, you can execute or describe a command.

Figure 12-1 Font Editor Command Display

FED COMMAND TABLE:		
Command Name	Assigned Keystroke	Description
180 Rotate	META-↓ or META-↑	Rotate character 180 degrees clockwise.
Basic	META-HELP	Display information about each Fed pane.
BOX	META-CTRL-B	Create a rectangle (a box) specified by two opposite corners.
Change Mode	CTRL-M	Toggle the mode to draw, flip or clear. (The current mode is displayed)
Change Vars	HYPER-V	Change the variables that control how a font is displayed.
Char Italicize	META-I	Form an italic version of the current character.
Char Reverse	META-V	Reverse-video this character
Char Thicken	META-T	Make the character appear more BOLD by adding one pixel in width to all
Char Unthicken	META-U	Make the character appear less BOLD by deleting one pixel in width to
Character	HYPER-B	Display information about the character box.
Character-io	HYPER-A	Display information about char-io functions.
Clear Registers	SUPER-E	Clear the contents of all eight registers.
Copy	SUPER-C	Copy selected font to a new font.
Create	SUPER-M	Create a new font.
Cursor	HYPER-C	Display information about non-mouse cursor commands.
Cursor Down	↓	Move the non-mouse cursor down.
Cursor Down Set	CTRL-↓	Move the non-mouse cursor down after setting the square.
Cursor Left	←	Move the non-mouse cursor left.
Cursor Left Set	CTRL-←	Move the non-mouse cursor left after setting the square.
Cursor Right	→	Move the non-mouse cursor right.
Cursor Right Set	CTRL-→	Move the non-mouse cursor right after setting the square.
Cursor Up	↑	Move the non-mouse cursor up.
Cursor Up Set	CTRL-↑	Move the non-mouse cursor up after setting the square.
Directory	SUPER-F	Display a menu of fonts in a FONT directory to view/edit.
Display	SUPER-D	Display the currently selected font.
Down	META-CTRL-↓	Move the character and box down in the grid window.
Editing	HYPER-E	Display information about character editing functions.
Erase Black	CTRL-E	Erase only the black plane.
Erase Both	META-CTRL-E	Erase both the black and gray planes without changing the size of the
Erase Gray	META-E	Erase only the gray plane.
Exit Font Editor	END	Exit the Font Editor.
Flip Cursor	CTRL-SPACE	Flip, set, or clear the mouse cursor point.
Font-io	HYPER-F	Display information about font-io functions.
Get Char	CTRL-G	Get a character from the selected font into the black plane.
Get Char Num	META-CTRL-G	Get a character specified by number from the selected font into the bl
Get Gray Char	META-G	Get a character from the selected font into the gray plane.
Help	HYPER-H	Display information about general help functions.
Home	CTRL-H	Move the character and box to the center of the grid window.
Italicize	SUPER-I	Italicize all characters in the specified font.
Left	META-CTRL-←	Move the character and box to the left in the grid window.
Left Rotate	META-↶	Rotate character 90 degrees counter-clockwise.
Line	META-CTRL-L	Create a line specified by two endpoints.
Load	SUPER-L	Specify a font file to be loaded into the editor and selected.
Merge Gray	META-X	Merge the gray plane into the black plane.
Merge Menu	META-CTRL-X	Merge the gray plane into the black plane with menu for copy, set, cle
Modes	HYPER-M	To switch modes, click middle while over grid window.
Move Both	META-CTRL-M	Move both the black and gray planes from one specified reference point
Move Gray	META-M	Move the gray plane from one specified reference point to another spec
New Mouse	HYPER-N	Select a new mouse character for drawing in the grid plane.
Command Display	(Press END to exit)	

L:Execute command M:Describe command R:Command Menu R2:System menu

Figure 12-2 SYMBOL-HELP Keyboard Map



- You can press SYMBOL-HELP to display a list of keystroke sequences that produce the special Lisp characters, as shown in Figure 12-2.

Invoking the Font Editor

12.3 The font editor enables you to view, create, modify, and store fonts.

You can invoke the font editor by doing one of the following:

- Pressing `SYSTEM F`
- Selecting the Font Editor item in the System menu
- Executing the `fed` function in a Lisp Listener window

Exiting the Font Editor

12.4 You can exit the font editor by doing any of the following:

- Selecting the Exit Font Editor item in the command menu or by pressing the `END` key
- Invoking a different program or utility, either by pressing the `SYSTEM` key and a letter other than `F`, or by pressing `TERM S`
- Selecting another program or utility by using the System menu

Executing Commands

12.5 In addition to executing a command by selecting it from the menu, you can also use a keystroke sequence to execute many of the commands. You can list the commands and their associated keystroke sequences by pressing `HYPER-CTRL-HELP`.

For your convenience in the rest of this section, the keystroke sequence associated with a menu command—where such a keystroke sequence exists—is listed after the name of the command in parentheses. For example, the Home command (`CTRL-H`) is used to move the character to the center of the grid area.

Selecting a Font to Edit

12.6 When you first invoke the font editor, it displays a list of loaded fonts. You can then select one of the fonts to edit by boxing the name of the font and clicking left with the mouse. If, during your session, you want to select a different font to edit, you can do so by executing the Select command (`SUPER-A`).

If you want to edit a font whose name is not on the list of loaded fonts (that is, a font that is not currently loaded), you must first load the font and then select it.

**Using the
Select Command**

12.6.1 When you want to list all the loaded fonts, possibly to select a new font to edit, you use the Select command (SUPER-A). The following figure shows a typical listing of loaded fonts:

Registers	Loaded Fonts - Select one to edit it		
	43VKMS	5X5	BIGFNT
	BOTTOM	CMB8	CMDUNH
	CMOLD	CMR10	CMR10
	CMR5	COURIER	CPTIFONT
	CPTFONTB	CPTFONTBI	CPTIFONTI
	HIGHER-MEDFNB	HIGHER-TR8	HL10
	HL10B	HL12	HL12B
	HL12BI	HL12I	HL6
	HL7	ICONS	MEDFNB
	MEDFNT	METS	METSB
	METSBI	METSI	MOUSE
	SEARCH	TI-LOGO	TINY
	TOP	TR10	TR10B
	TR10BI	TR10I	TR12
	TR12B	TR12BI	TR12I
	TR10	TR10B	TR8
	TR8B	TR8I	TVFONT
	VT-GRAPHICS	VT-GRAPHICS-BOTTOM	VT-GRAPHICS-TOP
	VT-GRAPHICS-WIDER-FONT	WIDER-FONT	WIDER-MEDFNT

*** Press the space bar to remove this output. ***

After you select one of the listed fonts, the font editor displays that font along with a sample font in a table format. From this table, you can select a character to edit. This table of characters is described in paragraph 12.7.1, The Font Display.

**Using the
Directory Command**

12.6.2 When you want to view the fonts available in a directory—and perhaps load one of them—do the following:

1. Select the Directory command (SUPER-F).

The system prompts you for the name of the directory that contains the font.

2. Supply the name of the directory. Either type the pathname or, if the default pathname supplied by the system is correct, press RETURN.

The system pops a menu that asks whether you want to see all the fonts in the directory, or only the fonts that are *not* currently loaded.

3. Select one of the choices.

The system lists the fonts.

You can then select one of the fonts to load and edit by boxing the name of the font and clicking left with the mouse.

**Using the
Load Command**

12.6.3 When you want to load a font whose filename you know, do the following:

1. Select the Load command (SUPER-L).

The system prompts you for the type of a font to load and edit.

2. Select one of the file types. Currently, you can load fonts with types AST or XLD.

- AST files contain text-formatted font information. The pixels in each character are represented by asterisks and spaces. AST files are used to move fonts between various vendors' machines.
- XLD files contain binary files, which are the type of files created by default by the font editor.

3. The system then prompts you for the name of the file to load.

Supply the pathname. Either type the pathname or, if the default pathname supplied by the system is correct, press RETURN.

**Using the
load Function**

12.6.4 In addition to using the commands in the font editor menu to load a font, you can also use the **load** function to load fonts. For example, if you know that you will be developing a font named `my-font` over several sessions, you may want to include the following form in your login-initialization file so that this font is automatically loaded each time you log in:

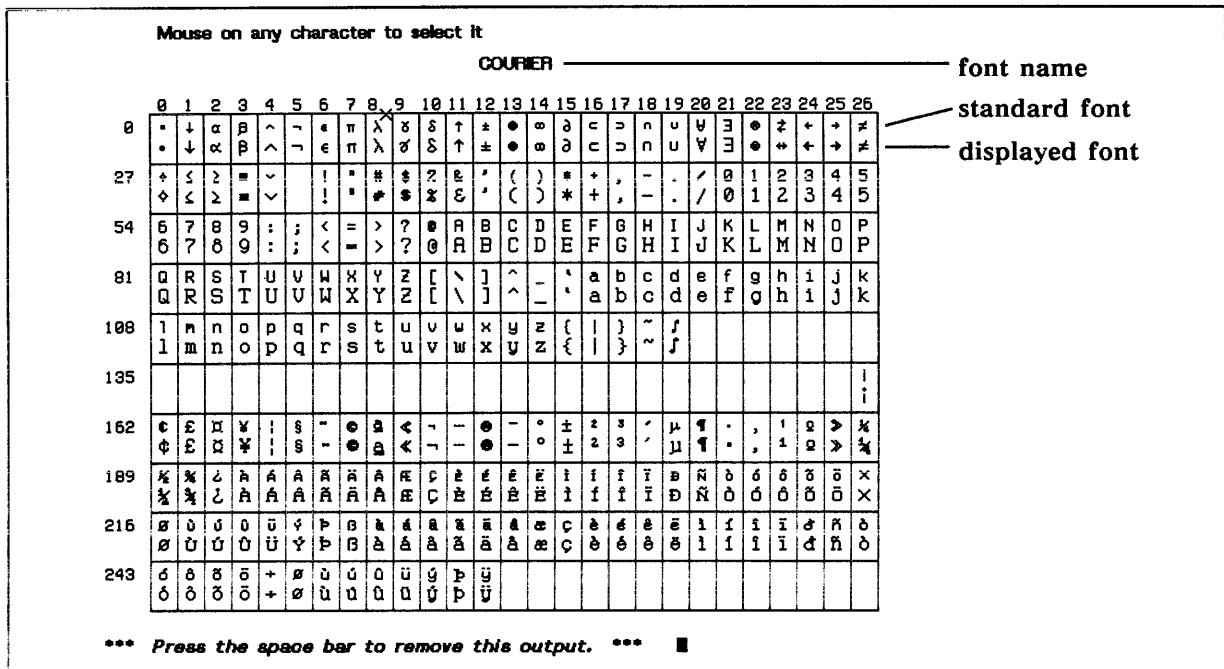
```
(load "my-host:my-directory;my-font.xld")
```

**Selecting a
Character
to Edit**

12.7 After you select a font to edit, you can select a character to edit. The character may or may not already exist. When you select a character to edit, you discard the current contents of the black plane. To select a character, you can use any of the following procedures:

- Display the entire font by using the Display command (SUPER-D); then, select one of the characters to edit.
- Specify a character from the selected font by using the Get Char command (CTRL-G) or the Get Char Num command (META-CTRL-G).
- Select the character code in the label pane (box the character code and click left with the mouse), then type the code for the desired character. The code is accepted in the default input base for the Explorer system and is displayed in decimal numbers. The system converts the number if necessary.
- Select the symbol in the label pane (box the symbol and click left with the mouse), then type the standard symbol. For characters that are not represented on the Explorer keyboard, you can press a keystroke sequence combining the SYMBOL key and a letter.

The Font Display 12.7.1 When the font editor displays an entire font, it appears similar to the following:



The top line of each pair of lines is a sample font used for comparisons. By default, this is cptfont. The second line of each pair of lines is the actual font being displayed. This comparison is useful if you want to compare two similar fonts, or if you want to verify the glyphs in a nonalphabetic font with the characters in a sample font (for example, mouse font and cptfont).

In addition to the standard alphanumeric and punctuation characters that appear on a typewriter, most Explorer fonts that are used to display text include the following:

- Characters with character codes less than 32 are special Lisp characters or symbols, such as Greek characters, mathematical symbols, and logic symbols. You typically produce these characters by pressing a combination of the SYMBOL key and an alphanumeric key. You can display a list of these combinations by pressing SYMBOL-HELP.
- Characters with character codes in the range of 128. and 160., inclusive, are associated with control characters such as TAB and SYSTEM and cannot be redefined.
- Characters with character codes greater than 152 are part of the international character set defined by the International Standards Organization. These characters are commonly called ISO characters. You typically produce these characters by pressing a combination of the SYMBOL key and an alphanumeric key. You can display a list of these combinations by pressing SYMBOL-HELP.

**Using the
Change Variables
Command**

12.7.2 You can change the appearance of a font listing by using the Change Variables command (HYPER-V). When you select this command, the system pops up a menu that enables you to change any of several variables: the sample font, the number of columns displayed, and the base of the numbers used to label the columns.

*Changing
the Sample Font*

12.7.2.1 You can change the sample font to display a font that you want to compare to the one you are currently editing. For example, if you are creating a bold version of a font, you may want the medium version of the font as your sample font.

*Changing
the Number
of Columns*

12.7.2.2 You can change the number of columns of the font displayed by default. The following are valid values:

- Any negative integer — The system uses as many columns of the current font as fit in the available space.
- Zero — The system uses the number of columns that is the largest multiple of two that fits in the available space. This is the default value.
- A specific positive integer — The system uses this number of columns to display the font. If a font is too large to be displayed using this number of columns, the system puts as many columns in the display as possible.

*Changing
the Base
of the Number Labels*

12.7.2.3 By default, the system uses decimal numbers to label the columns and rows of the displayed font. You can change the base of the numbers to any positive integer.

**Changing
the Configuration
by Setting Variables**

12.7.3 You can change any of the values that you change with the Modify Variables command by setting variables. Thus, you can include these changes in a login-initialization file and have new defaults set for your font editor whenever you log in. The variables to set are as follows:

```
fed:*sample-font*
fed:*label-base*
fed:*columns*
```

For example, if you always want the labels for the columns to be in hexadecimal, you could include the following form in your login-initialization file:

```
(setq fed:*label-base* 16)
```

Editing a Character

12.8 You can modify an existing character by using a command, by drawing, by merging parts of other characters, or by moving the contents of the editing area. The commands available to edit a character include thicken, unthicken, italicize, stretch, rotate, reflect, box, and reverse, among others.

The *drawing mode* determines whether a pixel is drawn (set), flipped, or erased when you click on that pixel. The drawing mode is also used in some drawing operations.

Remember that a character cannot have pixels set above the character box or below the line below the character box. Most of the editing operations allow you to set such pixels; however, when you attempt to save such a character in a font, the system warns you that some pixels will not fit within the character and asks you whether to continue with saving the character.

Changing Drawing Mode

12.8.1 You can change the drawing mode of the editing area—draw, flip, or erase—by doing any of the following:

- Clicking middle with the mouse.
- Selecting the Mode label in the label pane.
- Invoking the Change Modes command (CTRL-M).

By default, the shape of the mouse cursor changes as you change drawing mode.

Choosing Mouse Cursors

12.8.2 When you first invoke the font editor, the mouse cursor is a different shape for each of the drawing modes.

Drawing Mode	Icon
Flip mode	
Draw mode	
Erase mode	



You can select different mouse cursors by using the New Mouse command (HYPER-N). This command invokes a menu of cursor icons. You select an icon for each of the three modes. These icons become your mouse cursors when the cursor is in the editing area.

Drawing 12.8.3 You create or modify a character by setting or erasing pixels in the editing area. The drawing mode determines whether you draw (set), flip, or erase each pixel. To draw a character, you can use the mouse or the keyboard to set pixels, or you can use a drawing operation.

Using the Mouse to Set Pixels 12.8.3.1 Using the drawing mode, you can do the following:

- Click the mouse to affect the individual pixel pointed to by the mouse cursor.
- Click and hold the mouse, and then move the mouse cursor to affect a series of pixels.

Using the Keyboard to Set Pixels 12.8.3.2 You can use the keyboard instead of the mouse to set pixels and to move the cursor, as follows:

- Set the pixel that the cursor is over by pressing CTRL-space bar.
- Move the mouse cursor pixel-by-pixel by using the arrow keys on the keyboard.
- Set the current pixel and move to the next pixel by pressing CTRL-arrow key.

You can move or set several pixels in a row by using a numeric argument with an arrow key. You provide a numeric argument with the CTRL-U keystroke sequence, which supplies four or a multiple of four. For example, to move four pixels up, you would press CTRL-U CTRL-↑. To set four pixels to the left of the current cursor position, you would press CTRL-U CTRL-←. To set sixteen (four times four) pixels, you would press CTRL-U CTRL-U CTRL-←.

Drawing Operations 12.8.3.3 The font editor includes several drawing operations that enable you to draw a line, a rectangle (called a box), or a spline (a curved line) by specifying only a few points. Each of these operations set, flip, or erase pixels according to the current drawing mode.

- The Line command (META-CTRL-L) requires you to specify a beginning and an ending point. If the line is neither vertical nor horizontal, the editor attempts to draw a straight line between the two points.
- The Box command (META-CTRL-B) requires you to specify two opposite corners.
- The Spline command (META-CTRL-S) requires you to specify several points on the curved line. When you have specified the last point, you click right to identify that point as the last point. The font editor connects the points with a curved line.

Swapping and Merging

12.8.4 You can swap and/or merge the contents of the planes with each other, or you can merge the contents of a register with either plane. You can do the following:

- Select the Swap Planes command (CTRL-X). This command swaps the contents of the black and gray planes.
 - Select the Merge Gray command (META-X). This command merges the gray character with the black character according to the current drawing mode.
 - Select the Merge Menu command (META-CTRL-X). This command invokes a menu that offers you several options. You can either choose to copy the pixels from the gray plane into the black plane while keeping the gray character intact, or you can choose to merge the gray character and either set, flip, or erase the pixels in the black plane.
 - Position the cursor in the register containing the character you want to merge and then select from the menu of commands by clicking right.
-

Moving the Contents of the Editing Area

12.8.5 You can change the position of the character box within the editing area in several ways. These commands do *not* change the position of the character relative to the position of the character box.

- Use the Home command (CTRL-H) to position the character box, with its contents, in the center of the visible editing area.
 - Use the Up, Down, Left, or Right commands (META-CTRL-arrow keys) to move the box and character a specific number of pixels in one direction.
 - Use the Move Both command (META-CTRL-M) to move the character box, with its contents, from one position to another. The system prompts you for a reference point on the current character and for another point to which you want the original point moved.
-

Adjusting the Character Placement

12.8.6 You can change the position of the box in relation to the character or you can move the character in the gray plane in relation to the box and the character in the black plane.

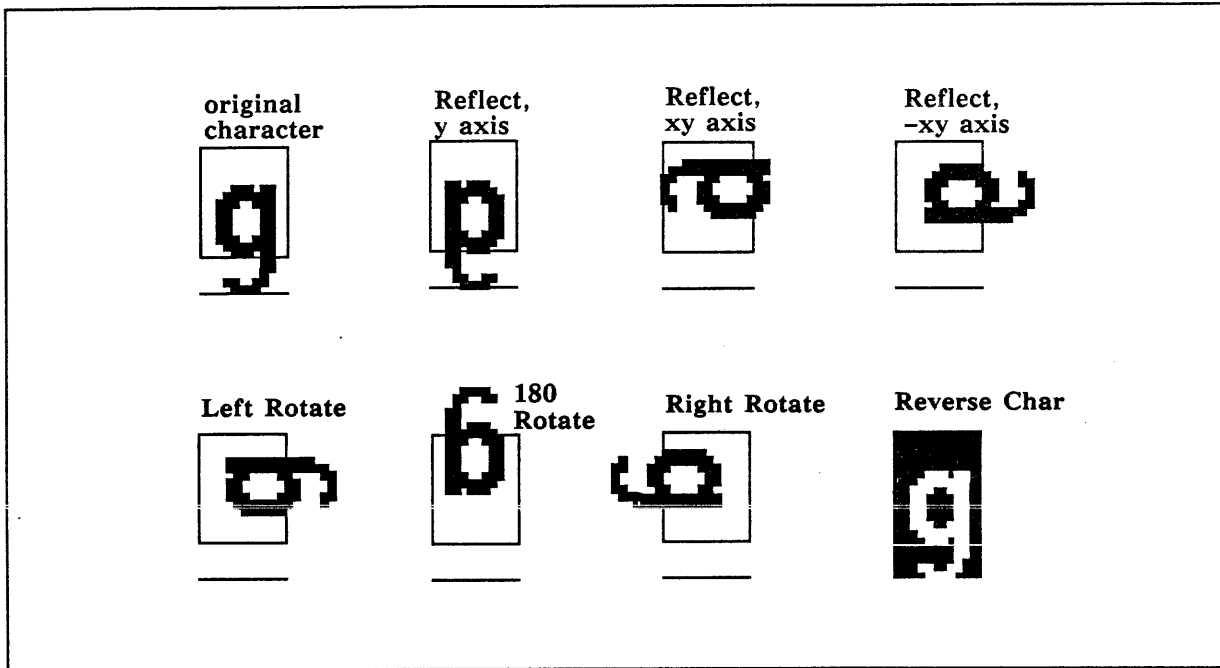
Moving the Box

12.8.6.1 You can move the box in relation to the character by dragging the character box with the mouse. To move the box, press and hold the right mouse button as you move the mouse in the direction desired. When you are in the new position, release the button.

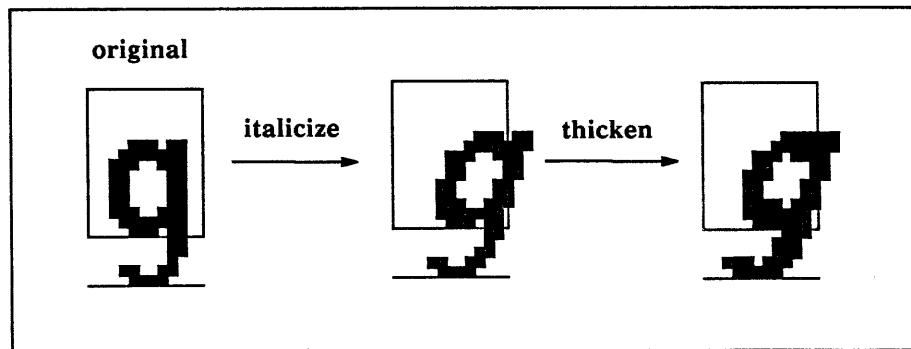
Moving the Character in the Gray Plane

12.8.6.2 You can move the character in the gray plane in relation to the box by using the Move Gray command (META-M). The system prompts you for a reference point on the current character and another point to which you want the original point moved.

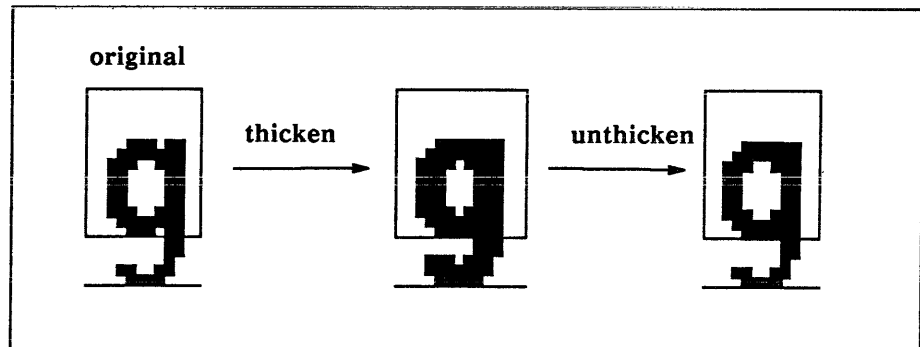
Editing Operations 12.8.7 For any character, you can thicken, unthicken, italicize, stretch, rotate, reflect, box, and reverse it, among other operations. Most of these operations are self-explanatory. Each of the following figures illustrates an operation performed on the original letter. The term above each figure indicates what operation was performed on the original character to create the new figure.



You can, of course, do multiple operations to the same character. For example, to create a bold italic character, you could first italicize the character and then thicken it.



However, inverse operations do not necessarily return the character that you started with. For example, if you thicken a character and then unthicken it, you may not obtain the original character again:



Erasing the Contents 12.8.8 You can erase the contents of either or both planes:

- The Erase Black command (CTRL-E) erases the contents of the black plane.
- The Erase Gray command (META-E) erases the contents of the gray plane.
- The Erase Both command (META-CTRL-E) erases the contents of both planes.

These commands do *not* affect the position or size of the character box.

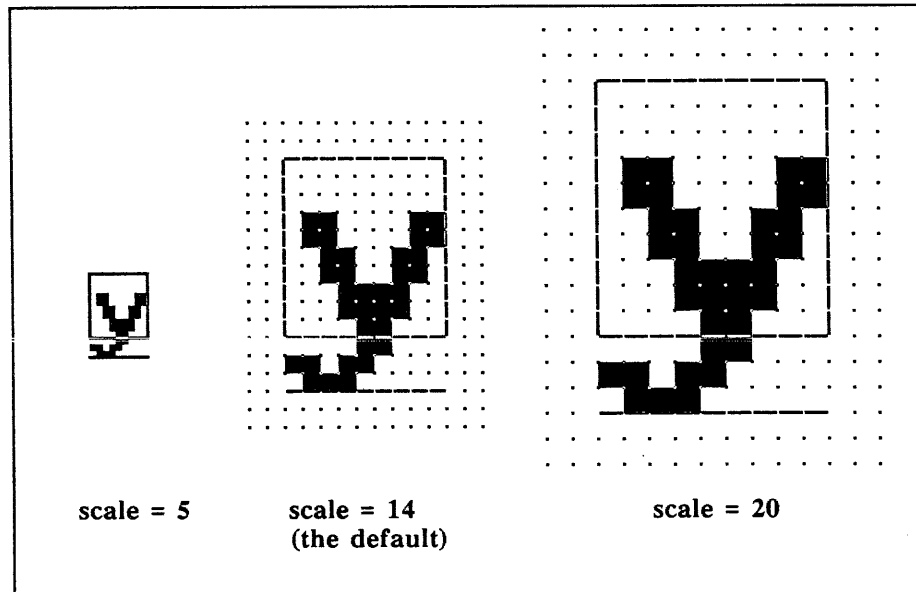
You can also erase part of the black plane by flipping or erasing pixels. The Box command (META-CTRL-B), used with a drawing mode of Erase, is very useful for this.

Examining a Character

12.9 After you edit or create a character, you can view it using different scales and next to other characters in the font.

Adjusting the Grid Scale

12.9.1 You can specify the scale of the editing area by selecting the Set Scale command (CTRL-@). When the font editor is first accessed, the scale is 14. You can reduce the scale in increments of one to a scale of one, or increase the scale in increments of one to a scale of 50. Displays with a scale of five or less do not include the grid of dots. Working with a smaller scale enables you to see large characters easily, but editing operations execute more slowly.



Modifying the Sample String

12.9.2 To view the character alongside other characters in the font, you can specify a sample string. The number of characters that can fit in the sample string depends on the width of the characters. If the string is too long to be entirely displayed, the font editor displays as many characters as fit. The remaining characters are not displayed, but they remain in memory. If you edit a smaller font, the font editor displays more of the string.

You can specify a new sample string by selecting the current string in the label pane. You can then type the new string and press RETURN. The new sample string appears immediately. As you edit characters, the characters displayed in the sample string include the changes.

Saving a Character

12.10 After you edit a character, you can hold the changes in a register or save the edited character in a font. These procedures store a character during the current session only; to permanently save characters, you must write the font containing the characters to a file. However, those characters are available in that font whenever it is used in the current session.

The system will not save a character that includes pixels above or below the character box or that has a total height or a height above the baseline that is incompatible with the current font. When you attempt to save such a character, the system warns you that some pixels will not fit within the character and asks you whether to continue with saving the character. If you confirm the save operation, the system forces the character to meet font characteristics by discarding pixels above or below the character box and by adjusting font parameters as needed.

Holding in the Register

12.10.1 The contents of the eight registers remain as long as you do not clear the registers, replace their contents, or boot the system.

- To hold a character in a register, you can move the mouse cursor into the register and copy the contents of the black plane into the register by clicking left.
- To restore a character from a register back to the editing area, copy or merge the contents of the register into the black plane by clicking middle.
- To invoke a menu that enables you to clear the register or merge the register contents with the contents of one or both planes, click right.

Saving in the Font

12.10.2 Saving a character in a font retains the changes only during the current session; you must write the font to a file to save the characters permanently.

To save a character in a font, you can do either of the following:

- Select the Save Char command (CTRL-P). The system saves your edited character into the selected font.
- Select the Save X (for eXplicit) command (META-CTRL-P). The system prompts you for the name of a font and a character in which to save your edited character. You can also use this command to save an edited character in a different character position in the same font by specifying the name of the current font and a different character position.

Writing a Font to a File

12.11 To save a changed font so that you can recall it during another session or to use the font within a display, you must write the font into a file.

1. To write a font, you can use the Write command (SUPER-W). This command first invokes a menu of file types:
 - For fonts used on the Explorer system, you should choose the XLD item. Fonts are typically stored on the Explorer system in binary files of type XLD.
 - For fonts that you intend to move to other types of machines, you should choose the AST item. See the paragraph 12.15, AST Files, for more information.
 - The Other item pops up a menu of other file types that are included for compatibility with other types of machines or with previous releases of the Explorer system.

After you choose a file type, the system prompts you for the name of the file in which to store the font.

2. Supply the pathname. Either type the pathname or, if the default pathname supplied by the system is correct, press RETURN.

After you have written a font to a file, you can load the font each time you log in by including a **load** function in your initialization file. For more information about initialization files, see Section 3, Login Initialization Files.

To have the system load the font automatically during boot operations, you must save a band that includes the font. For information about saving a load band, see the *Explorer Input/Output Reference*.

Creating a Modified Font

12.12 If the font you want to create is similar to another font, you can usually modify the existing font to create one of your own. The following paragraphs describe how to create a modified font.

Using the Font Editing Operations

12.12.1 After you select a font, you can modify all the characters in the font to create a similar font. You can stretch, rotate, italicize, thicken, unthicken, or reverse the font by using commands from the command menus. These commands modify each of the characters in the font in turn. The resulting characters may not be identical with characters created by manipulating the individual characters. Specifically, each character of a font so created is forced to a valid character (that is, pixels that occur above or below the character box are discarded and the total height and the height above the baseline are forced to fit within the constraints of the font as a whole).

In general, follow these steps to create a font similar to an existing font:

1. Create the new font.
 - a. Select the Create command (SUPER-M, for make).
 - b. When the system prompts you, type the new font name.

2. If desired, edit the font parameters as explained in paragraph 12.13, Creating a Font From Scratch.
3. Modify the characters, either by modifying the font as a whole or by editing each character separately.
4. Write the font to a file to save it permanently, as explained in paragraph 12.11, Writing a Font to a File.

For each font editing operation, you are prompted for the name of the new font. You can either accept the default value by pressing RETURN or supply a different pathname. For example, if you select the courier font and then select the Italicize command (SUPER-I), the font editor displays the following prompt in the Lisp Listener pane:

Name for new Italic version of COURIER (default COURIERI):

You can either press RETURN to accept the default value of COURIERI, or you can supply a different pathname.

The following font was created by using the Italicize command (SUPER-I) on the courier font. The sample font in this display is courier.

COURIERI

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	
0	•	↓	α	β	^	~	ε	π	λ	σ	δ	†	±	•	•	•	•	•	•	•	•	•	•	•	•	•	•	
27	◊	<	>	≡	∇	!	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	
54	6	7	8	9	:	:	<	=	>	?	0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
81	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-	'	a	b	c	d	e	f	g	h	i	j	k	
108	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	f								
135																											i	
162	φ	ε	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο	π	ρ	σ	τ	υ	φ	χ	ψ	ω	•	
189	ξ	ξ	ζ	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α	Α
216	ø	ù	ú	û	ü	ý	þ	ß	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	
243	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ															

*** Press the space bar to remove this output. ***

nt: COURIERI Total Ht: 13 Above Base: 11 Blinker: 14x13 Rotate: 180 Space: 14
 one Width: 14 Sample: 'Sample string: RaBbCc / 0123...

As another example, the following font was created by using the Thicken command (SUPER-T) and then the Reverse command (SUPER-V) on bigfnt. The sample font in this display is bigfnt. Note that this font is so large that only the first 98 characters are displayed.

BIGFNTBV

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	•	↓	α	β	^	-	ε	π	λ	δ	↑	±	⊕	⊗	⊙	⊚	⊛	⊜	⊝	⊞	⊟	⊠
22	⊡	⊢	⊣	⊤	⊥	⊦	⊧	⊨	⊩	⊪	⊫	⊬	⊭	⊮	⊯	⊰	⊱	⊲	⊳	⊴	⊵	⊶
44	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A
66	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
88	X	Y	Z	[\]	^	_	'	a	b	c	d	e	f	g	h	i	j	k	l	•
110	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ				
132																						
154										i	†	‡	§	¶	•	ˆ	˜	˘	˙	˚	˛	˜
176	•	±	²	³	´	µ	¶	•	,	¹	º	»	¼	½	¾	¿	À	Á	Â	Ã	Ä	Å

◆*MORE**

nt: BIGFNTBV Total Ht: 18 Above Base: 14 Blinker: 13x18 Rotate: 0 Space: 13
 one Width: 13 Sample: "Sample string: AaBbCc / 0123..."

Modifying a Font

12.12.2 If the changes you want to make in the font cannot be done with the font editing operations, but the new font is similar to an existing font, you can copy the existing font and change each character as you like. Follow these steps to modify an existing font without using the font editing commands:

1. Copy the existing font that is closest to the font you want to create.
 - a. Select the Copy command (SUPER-C).
 - b. When the system prompts you, type the name of the new font.

The font editor displays the selected font.

2. If desired, edit the font parameters as explained in paragraph 12.13, Creating a Font From Scratch.
3. Edit the characters within the font, as explained in paragraph 12.8, Editing a Character. If you are editing a fixed-width font, do *not* change the Width field for any of the characters. If you do so, the font becomes a variable-width font.
4. Write the font to a file to save it permanently, as explained in paragraph 12.11, Writing a Font to a File.

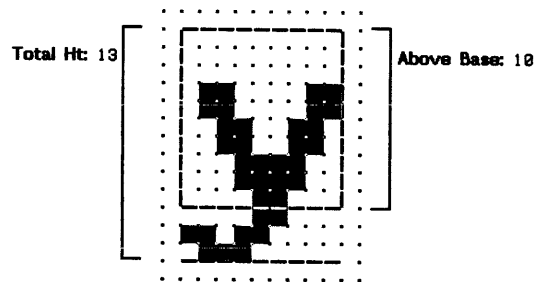
Creating a Font From Scratch

12.13 Most fonts are similar in shape. In some instances, however, you may want to create an entirely new font that contains unusual characters. To create a totally new font, follow these steps:

1. Prepare the editing area for a new font.
 - a. Select the Create command (SUPER-M, for make).
 - b. When the system prompts you, type the name of the new font.

The font editor displays the table of characters with the sample font but no characters in the selected font. The font editor also supplies default values for the font parameters.

2. Modify the parameters for the font by selecting the values in the label pane and typing new values when the system prompts you. You should verify and/or modify the values for the following parameters:
 - a. Total height
 - b. Height above the baseline
 - c. Blinker width
 - d. Blinker height



3. Specify the default width of each character in the font by setting the width of the space character (character code 32). Even when the default space width of seven is acceptable, you should follow these steps to specify the width of the space character, and thus the default width of all characters in the font:
 - a. Select the Get Char command (CTRL-G) to request a character to edit.
 - b. When the system prompts you for the character to edit, press the space bar.
 - c. Select the Width field in the label pane and, when prompted, supply the value for the width.
 - d. Save the new space character in the font by selecting the Save Char command (CTRL-P).

4. Create the characters within the font, as explained in paragraph 12.8, Editing a Character. To create a fixed-width font, do not change the Width field for any of the characters. To create a variable-width font, adjust the Width field of each character to fit the size of the character.
5. Write the font to a file to save it permanently, as explained in paragraph 12.11, Writing a Font to a File.

Performance Considerations

12.14 Some fonts require more time to display than others. Some fonts require more memory than others.

Execution Time

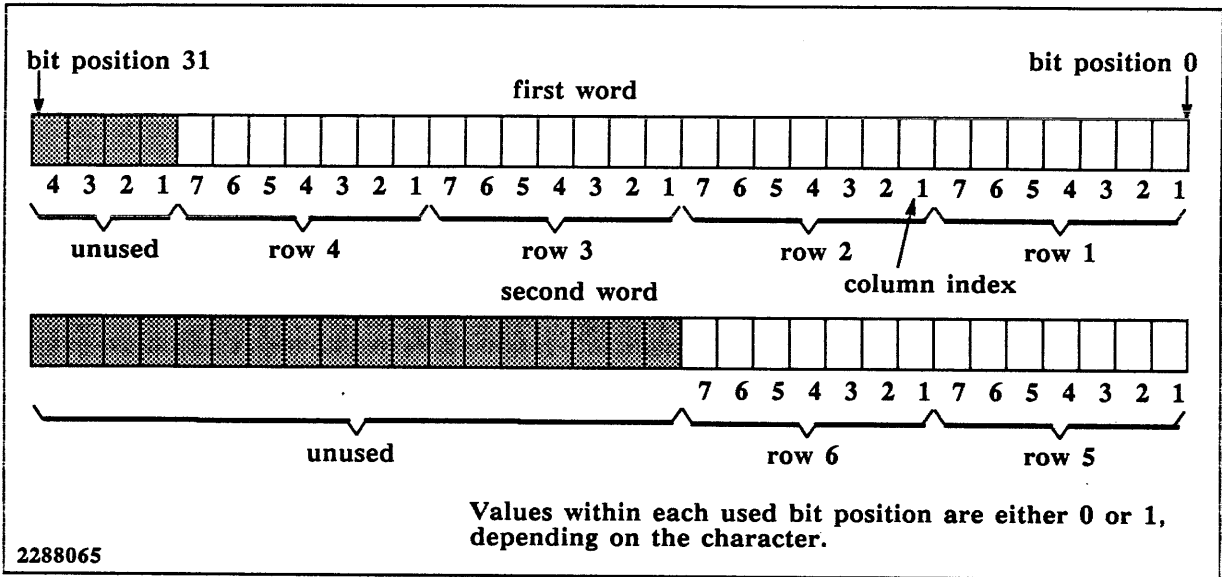
12.14.1 Some fonts require more time to display than other fonts. In general, the larger and more complicated a font is, the longer it takes to display. The Explorer system requires more time to display fonts that:

- Are more than 32 pixels wide. These *fat* fonts are stored using a subcharacter scheme, and each subcharacter is accessed and drawn separately. In addition, some output primitives do not work with fat fonts.
- Include a kerning table. The Explorer system attempts to minimize the amount of space required to store a font by using kerning. In general, a character has a left kern if any pixel of the character glyph extends outside the left edge of the character box. In some cases, a character can have a kern value if the width of the glyph is less than or equal to the raster width (the number of bits used to store the width of the character) but the character is offset to the right of the left edge of the character box (that is, the leftmost column(s) of the character box are empty).

Memory Allocation

12.14.2 Each character in a font can be thought of as having an array of bits stored for it. The dimensions of this array are called the *raster width* and *raster height*. The raster width and raster height are the same for every character of a font; they are properties of the font as a whole, not of each character separately. Consecutive rows are stored in the array; the number of rows per character is the raster height, and the number of bits per row is the raster width. An integral number of rows is stored in each word of the array. If there are any bits left over, those bits are unused. Thus, no row is ever split over a word boundary. When there are more rows than will fit into a word, the next word is used. Remaining bits to the left of the last word are ignored, and the next row is stored right-adjusted in the next word, and so on. An integral number of words is used for each character.

For example, consider a font in which the widest character is seven bits wide and the tallest character is six bits tall. The raster width of the font is seven and the raster height is six. Each row of a character is seven bits, so four of them fit into a 32-bit word, with four bits wasted, as shown in the following figure.



The remaining two rows require a second word, the rest of which are unused because the number of words per character must be an integer. Thus, this font has four rows per word and two words per character.

If you edited this font and created characters that increased its raster width by one, the font would still require the same number of words of storage. The additional bits of information would be stored in the previously unused bits of the words. Suppose, however, you doubled the size of each character. The new raster width is 14, and the new raster height is 12. Each character requires six words to store instead of two words.

AST Files

12.15 Files of type AST are text files that contain font information. Typically, AST files are used to port fonts between different types of machines. However, you can also edit characters in an AST file directly using the Zmacs editor. The font editor can both load and write files of type AST.

A character in an AST file is represented as a two-dimensional array of asterisks and spaces, with each asterisk a pixel that is drawn in the character. At the beginning of the file is information about the font as a whole. In addition, each character includes certain information such as character height, character width, kerning, and so on. For example, the following is the AST file for bigfnt for the initial information and the first character:

```
18 HEIGHT
14 BASE LINE
0 COLUMN POSITION ADJUSTMENT
<PAGE> 0 CHARACTER CODE MH: WEBB; BIGFNT.AST#>
8 RASTER WIDTH
12 CHARACTER WIDTH
0 LEFT KERN
```

```
****
****
****
****
```

The following is the AST representation of the letter g in bigfnt.

```
<PAGE> 147 CHARACTER CODE MH: WEBB; BIGFNT.AST#>
10 RASTER WIDTH
12 CHARACTER WIDTH
0 LEFT KERN
```

```
*** **
*****
*** **
** **
** **
** **
** **
*** **
*****
*** **
**
**
** **
****
```

Command Summary 12.16 Table 12-2 lists the font editor keystroke assignments.

Table 12-2 Font Editor Keystroke Assignments

Menu Name	Keystroke	Menu Name	Keystroke
<i>Font-io</i> ¹			
Select	SUPER-A	Rotate Left	SUPER-←
Display	SUPER-D	Rotate Right	SUPER-→
Copy	SUPER-C	Rotate 180	SUPER-↓ or SUPER-↑
Load	SUPER-L	Italicize	SUPER-I
Write	SUPER-W	Stretch	SUPER-S
Create	SUPER-M	Thicken	SUPER-T
Remove	SUPER-K	Unthicken	SUPER-U
Directory	SUPER-F	Reverse	SUPER-V
<i>Character-io</i>			
Get Char	CTRL-G	Save Char	CTRL-P
Get Char Num	META-CTRL-G	Save X	META-CTRL-P
Get Char Gray	META-G		
<i>Editing</i> ²			
Reflect	META-R	Stretch Char	META-S
Left Rotate	META-←	Char Thicken	META-T
Right Rotate	META-→	Char Unthicken	META-U
180 Rotate	META-↓, META-↑	Char Reverse	META-V
Char Italicize	META-I		
<i>Line</i> ³			
Spline ³	META-CTRL-S	Erase Both ⁴	META-CTRL-E
Box ³	META-CTRL-B	Erase Black ⁴	CTRL-E
		Erase Gray ⁴	META-E
<i>Merge Gray</i> ⁴			
Merge Menu	META-X	Move Both ⁴	META-CTRL-M
Swap Planes ⁴	META-CTRL-X	Move Gray ⁴	META-M
	CTRL-X		
<i>Screen</i>			
Home	CTRL-H	Left	META-CTRL-←
Redisplay	CLEAR SCREEN	Up	META-CTRL-↑
Set Scale	CTRL-@	Right	META-CTRL-→
Change Mode	CTRL-M	Down	META-CTRL-↓
Change Var	HYPER-V	New Mouse	HYPER-N
Clear Registers	SUPER-E		

NOTES:

¹ All the font-io commands use SUPER.

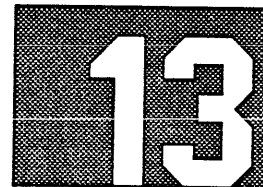
² All the character editing commands use META.

³ The drawing operations (line, spline, and box) use META-CTRL and the first letter of the command.

⁴ Most of the commands that affect the planes are on separate modifier keys:

- Black plane only — CTRL key
- Gray plane only — META key
- Both planes — META-CTRL keys

DEBUGGER (ERROR HANDLER)



Introduction

13.1 The debugger allows you to examine the environment in which an error condition is signaled in your program; then you can either abort your program or take corrective action and resume the execution of your program. The debugger is also known as the error handler (EH), although the EH is a separate entity.

For a full discussion of how programs can detect, signal, handle, and proceed from errors or other unusual situations encountered during processing, refer to the Error Handling section in the *Explorer Lisp Reference*. The EH and condition handlers are discussed there.

Entering the Debugger

13.2 When an error condition is signaled and no condition handlers are defined for the error, the Explorer system enters an interactive debugger that allows you to investigate the source of the problem and then continue or abort your program. Errors are signaled by Explorer microcode and by Lisp programs (using `fferror` or related functions). When there is a microcode error, the debugger prints out a message such as the following:

```
>>TRAP 5543 (TRANS-TRAP)
The symbol FOOBAR is unbound.
While in the function SYS:*EVAL ← SYS:LISP-TOP-LEVEL1
```

The first line of this error message indicates entry to the debugger and contains the following internal microcode information:

- A microprogram address
- The microcode trap name and parameters
- An optional microcode backtrace

Users can ignore this first line in most cases. The second line contains a description of the error. The third line indicates where the error happened by printing an abbreviated *backtrace* of the stack. The previous example shows that the error was signaled inside the function `*eval`, which was called by `sys:lisp-top-level1`.

An error signaled by a Lisp program using `fferror` enters the debugger and prints a message similar to the following one:

```
>>ERROR: The argument X was 1, which is not a symbol,
While in the function FOO ← SYS:*EVAL ← SYS:LISP-TOP-LEVEL1
```

Debugger entered while in the following function:

```
FOO (P.C. = 102)
  Arg 0 (x): 1
  Arg 1 (y): 10
```

The first line contains the description of the error message, and the second line contains the abbreviated backtrace of the functions called. The `foo`

function signaled the error by calling `error`; however, `error` is censored out of the backtrace. You can use the following variable to control the length of the backtrace.

eh:*error-backtrace-length*

Variable

This variable controls the length of the abbreviated backtrace (the functions currently on the stack) that are initially printed when the debugger is entered. Initially, the value is 3. The following is an example of an abbreviated backtrace:

```
While in function FOO ← BAR ← BAZ
```

Next, the debugger prints the arguments for the function that produced the error. Then it prints a list of commands you can use to proceed from the error or to abort to various command loops. The possibilities depend on the kind of error and where it happened, so the list is different each time, which is why the debugger prints it. The commands in the list all start with SUPER-A and continue as far as is necessary. For example:

```
SUPER-A, RESUME Ask for a replacement argument and proceed.
SUPER-B, ABORT Return to the top level of Lisp Listener 1.
SUPER-C      Restart the initial process.
SUPER-D      Reset and arrest the initial process.
```

If you do not want to display this list, set `eh:*inhibit-debugger-proceed-prompt*` to `t`.

eh:*inhibit-debugger-proceed-prompt*

Variable

If this variable is non-nil, the list of SUPER commands is not printed when the debugger is entered. Press META-CTRL-Q to display the list of commands for proceeding.

You can manually enter the debugger by explicitly causing an error. For example, you can enter an illegal divide-by-zero expression such as `(/ 4 0)` at the Lisp read-eval-print loop. Pressing META-BREAK enters the debugger while the currently running program is reading from the keyboard. Pressing META-CTRL-BREAK forces the currently running program into the debugger immediately, even if it is not reading from the keyboard.

NOTE: Pressing the BREAK key or CTRL-BREAK enters a read-eval-print loop rather than the debugger.

To invoke the debugger on a process, a window's process, or a stack group, use the `eh` function.

eh process

Function

The `eh` function stops *process* and invokes the debugger on it so that you can look at its current state. If you exit the debugger by pressing the ABORT key,

`eh` releases the process and returns. The *process* argument can be a window, in which case the window's process is used.

If *process* is not a process but a stack group, the current state of the stack group is examined. The caller should ensure that no one tries to resume this stack group while the debugger is looking at it.

To enter the debugger when a particular function is called, use `breakon` (see Section 22, `Breakon`, of this manual).

How to Use the Debugger

13.3 Once inside the debugger, the user has access to a variety of powerful commands. There are commands to examine the stack frames, move around inside the stack, resume execution, step, and transfer to other systems. This paragraph describes how to specify the commands and then explains them in approximate order of usefulness. A summary is provided at the end of the section.

The debugger also warns you about certain unusual circumstances that can cause paradoxical results. If `default-cons-area` is anything except `working-storage-area`, a message is printed indicating the area involved. If `*print-base*` and `*read-base*` are not the same, a message is printed.

When the debugger is waiting for a command, it prompts with an arrow:

→

After the arrow prompt appears, you can type either a Lisp expression or a command; a CTRL or META character is interpreted as a command, whereas most normal characters are interpreted as the first character of an expression. If you press the HELP key, the debugger gives you some introductory help. If you press a key that is not a debugger command, the debugger responds with the EVAL: prompt and waits for a Lisp expression or an input editor command.

If you type a Lisp expression, it is interpreted as a Lisp form and is evaluated in the context of the function that received the error. That is, all bindings that were in effect at the time of the error are in effect when your form is evaluated, with certain exceptions explained later. The result of the evaluation is printed, and the debugger prompts again with an arrow. Like the Lisp top-level loop, this read-eval-print loop maintains the values of the following variables: `+`, `++`, `+++`, `*`, `**`, `***`, `-`, `/`, `//`, and `///`. (Refer to Section 26, `Lisp Listener`, for a description of these variables.)

If, during the typing of the form, you change your mind and want to return to the debugger's command level, press the ABORT key; the debugger responds with an arrow prompt. In fact, any time that keyboard input is expected from you while you are in the debugger, you can press ABORT to flush what you are doing and return to command level.

If an error occurs in the evaluation of the Lisp expression you typed, you enter a second (or a deeper) invocation of the debugger, looking at the new error. The number of arrows in the prompt indicates the depth at which the error occurred. You can abort the computation and return to the previous error by pressing the ABORT key. However, if the error is trivial, the abort is performed automatically, and the original error message is reprinted.

Various debugger commands ask for Lisp objects, such as a value to return or a name of a catch-tag. Whenever the debugger tries to get a Lisp object from you, it expects you to type a *form*; it then evaluates what you type in. This feature provides greater generality because some objects to which you might want to refer cannot be typed (such as arrays). If the form you type is non-trivial (not merely a constant form), the debugger shows you the result of the evaluation and asks you if this result is what you intended. It expects a *y* or *n* answer (see the function *y-or-n-p* in the *Explorer Input/Output Reference*), and if you answer negatively, it asks you for another form. To quit the command, simply press ABORT.

Note that the bindings in effect are those of the frame being viewed, *not* those in effect at the point of the error. Evaluating these symbols within the debugger shows the binding in the current frame. The only exceptions are the bindings made by the debugger itself (refer to Table 13-1). To see the actual value of these bindings in a particular frame, you can press META-CTRL-S to see all special bindings in this frame or META-S to specify a particular special binding in this frame.

Table 13-1 summarizes these variable bindings.

Table 13-1

Special Variable Bindings in the Debugger

Variable	Binding at the Point of Error
terminal-io	This variable is rebound to the stream the debugger is using.
standard-input *standard-output*	Both of these variables are rebound to be synonymous with *terminal-io* .
+ *	Both + and * are rebound to the debugger's previous form and previous value. When the debugger is first entered, + is the last form typed, which is typically the one that caused the error, and * is the value of the <i>previous</i> form. The ++ , +++ , - , / , // , and /// variables are also rebound as expected.
evalhook sys:*applyhook*	These variables (see Section 20, Evalhook) are rebound to nil , turning off the step facility if it was in use when the error occurred.
eh:*condition-handlers* eh:*condition-default-handlers*	These variables are rebound to nil so that errors occurring within forms you type while in the debugger do not inadvertently resume execution of the erring program.

Table 13-1

Special Variable Bindings in the Debugger (Continued)

Variable	Binding at the Point of Error
----------	-------------------------------

eh:*condition-resume-handlers*

To prevent resume handlers established outside the error from being invoked automatically by deeper levels of error, this variable is rebound to a new value, which has an element *t* added in the front.

debug-io

Variable

The debugger uses this stream for its I/O. Normally, the value is a synonym stream that indirects to the value of *terminal-io*. The most important value of this variable occurs in the stack group in which the error was signaled.

**Debugger
Commands**

13.3.1 All debugger commands are single characters, usually with the CTRL or META keys. The single most useful command is ABORT (or CTRL-Z), which exits from the debugger and throws out of the computation that produced the error. This command is invoked using the ABORT key. Often you are not interested in using the debugger at all and merely want to return to Lisp top level; this key allows you to do so quickly.

If the error happened while you were innocently using a system utility such as the editor, then this error represents a bug in the system. Report the bug using the debugger command CTRL-M. This command provides an editor preinitialized with the error message and a backtrace of *n* frames. (The default value of *n* is 5; however, a numeric argument such as CTRL-1 CTRL-5 with CTRL-M creates a bug report containing 15 frames of detailed backtrace.) You should type in a *precise* description of what you did that led up to the problem; then send the message by typing END. Be as complete as possible, and always give the exact commands you typed, exact filenames, and so on, rather than general descriptions. The person who investigates the bug report has to try to make the problem recur.

Pressing ABORT signals the sys:abort condition, returning control to the most recent command loop. This loop can be Lisp top level, a break, or the debugger command loop associated with another error. Pressing ABORT numerous times throws back to successively older read-eval-print or command loops until top level is reached. Pressing META-ABORT, on the other hand, always throws to top level. META-ABORT is not a debugger command but a system command that is always available no matter what program you are in.

Note that pressing ABORT as you are typing a form to be evaluated by the debugger aborts this form and returns to the debugger's command level; however, pressing ABORT as a debugger command returns out of the debugger and the erring program to the *previous* command level. Pressing ABORT after entering a numeric argument simply discards the argument.

Pressing the META-HELP key types out documentation about the debugger commands, including any special commands that apply to the particular error currently being handled.

*Examining
Stack Frames*

13.3.1.1 The RESUME key is usually synonymous with SUPER-A. RESUME only proceeds, never aborts. If there is no way to proceed, only ways to abort, then RESUME does not do anything and is therefore no longer synonymous with SUPER-A.

The debugger knows about the current stack frame, and several commands use it. When the debugger is called, the current stack frame is the one that signaled the error and is either the one that received the microcode-detected error or the one that called `ferror`, `cerror`, or `error`. When the debugger is called, it shows you this frame in the following format:

```
FOO:
  Arg 0 (X): 13
  Arg 1 (Y): 1
```

In this example, `foo` was called with two arguments whose names in the Lisp source code are `x` and `y`. The current values of `x` and `y` are 13 and 1, respectively. These may not be the original values if `foo` sets its argument variables with `setq`.

The CTRL-L command (or CLEAR SCREEN) clears the screen, retypes the error message that was initially printed when the debugger was entered, and prints out a description of the current frame in the preceding format.

Several commands are provided to allow you to examine the Lisp control stack and to make current a frame that did not produce the error. The control stack (or *regular pdl*) keeps a record of all functions currently active. If you call `foo` at Lisp's top level and this function calls `bar`, which in turn calls `baz`, and `baz` produces an error, then a backtrace (a backwards trace of the stack) shows all of this information. The debugger has two backtrace commands.

CTRL-B simply prints out the names of the functions on the stack; for the previous example, it prints the following:

```
BAZ ← BAR ← FOO ← SYS:*EVAL
← SYS:LISP-TOP-LEVEL1 ← SYS:LISP-TOP-LEVEL
```

The arrows indicate the direction of calling. The META-B command prints a more extensive backtrace, indicating the names of the arguments to the functions and their current values; for the preceding example, it might look like the following:

```
BAZ:
  Arg 0 (X): 13
  Arg 1 (Y): 1
BAR:
  Arg 0 (ADDEND): 13
FOO:
  Arg 0 (FROB): (A B C . D)
```

The CTRL-N (or LINE) command moves *down* to the *next* frame (that is, it changes the current frame to be the frame that called it) and prints out the frame's function and arguments in the same format previously shown.

CTRL-P (or RETURN) moves *up* to the *previous* frame (the one that this one called) and prints out the frame in the same format.

META-< moves to the top of the stack (where the error occurred). It selects the same frame that was shown when the debugger was first entered. META-> moves to the bottom of the stack, to the initial stack frame. Both print the new current frame.

CTRL-S asks you for a string and searches down the stack for a frame whose executing function's name contains that string. This frame becomes current and is printed. These commands are easy to remember because they are analogous to editor commands.

The META-CTRL-N, META-CTRL-P, and META-CTRL-B commands are like the corresponding CTRL commands but do not censor the stack. When running interpreted code, the debugger usually tries to skip over frames that belong to functions of the interpreter, such as `*eval`, `prog`, and `cond`, and only shows *interesting* functions. META-CTRL-N, META-CTRL-P, and META-CTRL-B show all frames and functions. They also show frames that are not yet active, that is, frames whose arguments are still being computed for functions that are yet to be called. The META-CTRL-U command moves down the stack to the next interesting function and makes this function the current frame (if the current frame is *uninteresting*).

META-L, META-N, and META-P are similar to their corresponding CTRL commands. They print out the frame in *full screen* format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. For help in reading this machine code, see the Disassembler section in the *Explorer Lisp Reference*.

Many debugger commands are meaningful when repeated and take a prefix numeric argument by pressing CTRL or META (or both) and a number (as in the editor). For example, pressing CTRL-5 CTRL-N moves down 5 frames. Pressing CTRL-5 CTRL-B shows a backtrace of 5 frames. Other commands that can use this same format include META-B, CTRL-M, META-N, CTRL-P, META-P, META-CTRL-B, META-CTRL-N, and META-CTRL-P.

*Examining
Arguments,
Locals, Functions,
and Values*

13.3.1.2 CTRL-A prints the argument list of the function in the current frame.

META-CTRL-A takes a numeric argument (CTRL-*n*) and prints out the value of the *n*th argument of the current frame. If no numeric argument is given, 0 is assumed (that is, the first argument). This command leaves `*` set to the value of the argument so that you can use the Lisp read-eval-print loop to examine it. It also leaves `+` set to a locative pointing to the argument on the stack so that you can change this argument (by calling `rplacd` on the locative).

META-CTRL-L is similar to META-CTRL-A but refers to the *n*th local variable of the frame.

META-CTRL-V refers to the *n*th value this frame has returned. However, META-CTRL-V is meaningful only when you use it on a trap-on-exit (or a throw-tag-not-found error) while looking at a frame that is about to return. For information about trap-on-exit, see CTRL-X in paragraph 13.3.1.5, Stepping Through Function Calls and Returns.

META-CTRL-F refers to the function executing in the frame; it ignores its numeric argument and does not allow you to change the function.

Another way to examine and set the arguments, locals, functions, and values of a frame is with the **eh-arg**, **eh-loc**, **eh-val** and **eh-fun** functions. Use these functions in expressions you evaluate inside the debugger; they refer to the arguments, locals, values, and function, respectively, of the debugger's current frame.

eh-arg &optional (*arg-number-or-name* 0) (*errorp* nil) Function

When used in an expression evaluated in the debugger, **eh-arg** returns the value of the specified argument (*arg-number-or-name*) in the debugger's current frame. The default of *arg-number-or-name* is 0, which returns the value of Arg 0. If you specify a name for *arg-number-or-name*, argument names are compared ignoring packages; only the print name of the symbol you supply is relevant. The *errorp* argument, when set to **nil** (the default), prevents another error handler from being entered if **eh-arg** cannot handle your input.

The **eh-arg** function can appear in **self** and **locf** to set an argument or retrieve its location.

The **eh-arg** function with an argument of **nil** returns a list of all the arguments.

Examples: The following examples, using META-CTRL-BREAK to enter the debugger, show how **eh-arg** works. When you press META-CTRL-BREAK, a screen similar to the following appears:

```
>> Keyboard break.
While in the function PROCESS-WAIT ← TV:KBD-IO-BUFFER-GET ← (:METHOD TV:STREAM-MIXIN :ANY-TYI)
Debugger entered while in the following function:

PROCESS-WAIT (P.C. = 28)
  Arg 0 (WHOSTATE): "Keyboard"
  Arg 1 (FUNCTION): #<DTP-FUNCTION (:INTERNAL TV:KBD-IO-BUFFER-GET 0) -66232411>
  Rest arg (ARGUMENTS): (#<IO-BUFFER 530621: empty, State: NIL>)

Commands for proceeding from this particular error:
Super-A, (RESUME):   Proceed
Super-B:           Restart Initial Process.
Super-C:           Reset and arrest Initial Process.
(ABORT):           Return to BREAK ZMACS.
→
```

The first example returns the value "Keyboard":

```
(eh-arg 0)
```

The next example also returns the value "Keyboard":

```
(eh-arg 'whostate)
```

The following example tells you there is no argument named **foo** but does not enter another error handler because the default of the *errorp* argument is **nil**:

```
(eh-arg 'foo)
```

The next example sets the *errorp* argument to **t** and enters another error handler when no argument named **foo** is found:

```
(eh-arg 'foo t)
```

The following example returns a list of all the arguments:

```
(eh-arg nil)
```

eh-loc &optional (*local-number-or-name* 0) (*errorp* nil) Function

The **eh-loc** function is exactly like **eh-arg** but accesses the current frame's locals instead of its arguments.

The **eh-loc** function with an argument of **nil** returns a list of all the locals.

Examples: The following examples show how **eh-loc** works. After pressing META-CTRL-BREAK to enter the debugger, pressing META-L displays the local variables, as follows:

```
Stack frame index: 262
```

```
PROCESS-WAIT (P.C. = 28)
```

```
Arg 0 (WHOSTATE): "Keyboard"
```

```
Arg 1 (FUNCTION): #<DTP-FUNCTION (:INTERNAL TV:KBD-IO-BUFFER-GET 0) -66232411>
```

```
Rest arg (ARGUMENTS): (#<IO-BUFFER 530621: empty, State: NIL>)
```

```
Local 1 (STATE): 7
```

```
Disassembled code:
```

```
...
```

```
→
```

The first example returns the value (`#<IO-BUFFER 530621: empty, State: NIL>`). (In this example, the Rest arg is implemented as a local variable.)

```
(eh-loc 0)
```

The next example returns the value 7:

```
(eh-loc 1)
```

The following example also returns the value 7:

```
(eh-loc 'state)
```

eh-val &optional (*value-number-or-name* 0) (*errorp* nil) Function

When the current frame is returning multiple values, the **eh-val** function is used in an expression evaluated in the debugger to examine these values. This convention is useful only if the function has already begun to return some values (as in a trap-on-exit), because otherwise there are not any. For information about trap-on-exit, see CTRL-X in paragraph 13.3.1.5, Stepping Through Function Calls and Returns. If you specify a name, the **eh-val** function searches for the name in the function's `:values` declaration, if any. The default of *value-number-or-name* is 0, which accesses the first value this frame is returning.

The **eh-val** function can be used with **setf** and **locf**. With an argument of **nil**, the **eh-val** function returns a list of all the values this frame is returning.

eh-fun Function

The **eh-fun** function can be called in an expression being evaluated inside the debugger to return the function-object being called in the current frame. It can be used with **setf** and **locf**.

*Examining
Special Variables*

13.3.1.3 META-S prompts you for the name of a special variable and returns its value in the current frame. You can also specify instance variables of the `self` variable even if they are not special. Refer to flavors in the *Explorer Lisp Reference*.

META-CTRL-S prints a list of special variables bound by the current frame and the values that they are bound to by the frame. If the frame binds `self`, all the instance variables of `self` are listed, even if they are not special.

META-CTRL-H prints the condition handlers in effect for the current frame. It includes both the resume and default handlers. For information about handlers, see the Error Handling section in the *Explorer Lisp Reference*.

Resuming Execution

13.3.1.4 Often you want to try to proceed from the error. When the debugger is entered, it prints a table of commands you can use to proceed or to abort to various levels. The commands are SUPER-A, SUPER-B, and so on. The number of these commands and what they do is different each time there is an error, but the table indicates what each one is for. If you want to see the table again, press META-CTRL-Q.

META-C is similar to RESUME, but in the case of an unbound variable or undefined function, it actually calls `setq` on the variable or defines the function so that the error does not happen again. RESUME provides a replacement value but does not actually change the variable. META-C proceeds using the `:store-new-value` proceed type and is available only if that proceed type is provided.

CTRL-R is used to return a value or values from the current frame. The frame that called the current frame continues running as if the function of the current frame had returned. This command prompts you for each value that the caller expects. You can either type a form that evaluates to the desired value or press END if you want to return no more values.

META-CTRL-R reinvokes the current frame. It starts execution at the beginning of the function with the arguments currently present in the stack frame. These are the same arguments with which the function was originally called unless the function itself has changed them with `setq` or unless you have set them in the debugger. If the function has been redefined (perhaps you edited it and fixed its bug), the new definition is used. META-CTRL-R asks for confirmation before resuming execution.

META-R is similar to META-CTRL-R, but it allows you to change the arguments. You are prompted for the new arguments one by one. You can type a form that evaluates to the desired argument, press the space bar to leave that argument unchanged, or press END if you do not want further arguments. The space bar is allowed only if this argument was previously passed, and END is not allowed for a required argument. Once you have finished specifying the arguments, you must confirm the new arguments before execution resumes.

The CTRL-T command performs a `throw` to a given tag with a given value; you are prompted for the tag and the value.

*Stepping Through
Function Calls
and Returns*

13.3.1.5 You can request the debugger to trap on exit from a particular frame or the next time a function is called.

The CTRL-X command toggles the trap-on-exit bit of the current stack frame.

The META-X command sets the trap-on-exit bit for the current frame and all outer frames. If a program is in an infinite loop, this command is a good way to find out how far back on the stack the loop is taking place.

The META-CTRL-X command clears the trap-on-exit bit for the current frame and outer frames.

The CTRL-D command proceeds like RESUME but requests a trap the next time a function is called.

The META-D command toggles the trap-on-next-call bit for the erring stack group. It is useful if you wish to set the bit and then resume execution with something other than RESUME (see Section 22, Breakon, of this manual).

The **breakon** function is used to request a trap on calling a particular function. Trapping on entry to a frame automatically sets the trap-on-exit bit for that frame; use CTRL-X to clear it if you do not want another trap.

*Transferring to
Other Systems*

13.3.1.6 CTRL-E puts you into the editor, looking at the source code for the function in the current frame. This command is useful when you have found the function that caused the error and that needs to be fixed. Pressing the END key returns to the debugger, if it is still there.

CTRL-M calls the editor to mail a bug report. The error message and a backtrace are put into the editor buffer for you. A numeric argument indicates how many frames to include in detail in the backtrace. (The default is 5.)

META-CTRL-W calls the window debugger, a display-oriented debugger. It performs the same kind of commands as the debugger but displays graphically. For more information on the window debugger, see Section 14, Window-Based Debugger, of this manual.

**Summary of
Debugger Commands**

13.3.2 Table 13-2 summarizes the available debugger commands.

**Debugging After
a Warm Boot**

13.4 After a warm boot, the process that was last running can be debugged if you type no when the system asks whether to reset this process.

sys:debug-warm-booted-process

Function

This function invokes the debugger on the process that was running at the time of the warm boot.

Table 13-2 Summary of Debugger Commands

Key Sequence	Explanation of Command
CTRL-A	Prints the argument list of the function in the current frame.
META-CTRL-A	Sets * to the <i>n</i> th argument of the function in the current frame and + to the locative to the argument.
CTRL-B	Prints a brief backtrace of the stack.
META-B	Prints a more detailed backtrace of the stack.
META-CTRL-B	Prints a more detailed backtrace of the stack with no censoring of interpreter functions.
META-C	Attempts to continue, like RESUME, but executes setq on the unbound variable or defines the undefined function. This command uses the :store-new-value proceed type and is available only if this proceed type is also available.
CTRL-D	Attempts to continue, like RESUME, but traps on the next function call.
META-D	Toggles the flag that causes a trap on the next function call after you continue or otherwise exit the debugger.
CTRL-E	Edits the source code for the function in the current frame.
META-CTRL-F	Sets * to the function in the current frame.
META-CTRL-H	Prints a list of the condition handlers for the current frame.
CTRL-L or CLEAR SCREEN	Clears the screen and redisplay the error message and current frame.
META-L	Clears the screen and redisplay the error message and the current frame's function, arguments, locals, and compiled code.
META-CTRL-L	Sets * to the <i>n</i> th local of the function in the current frame and sets + to the locative to the local.
CTRL-M	Sends a bug report containing the error message and a backtrace of <i>n</i> frames (the default is 5).
CTRL-N or LINE FEED	Moves down to the next frame. With an argument, this command moves down <i>n</i> frames.
META-N	Moves down to the next frame with full-screen typeout. With an argument, this command moves down <i>n</i> frames.
META-CTRL-N	Moves down to the next frame even if it is uninteresting or still accumulating arguments. With an argument, this command moves down <i>n</i> frames.
CTRL-P or RETURN	Moves up to the previous frame. With an argument, this command moves up <i>n</i> frames.
META-P	Moves up to the previous frame with full-screen typeout. With an argument, this command moves up <i>n</i> frames.
META-CTRL-P	Moves up to the previous frame even if it is uninteresting or still accumulating arguments. With an argument, this command moves up <i>n</i> frames.
CTRL-R	Returns a value (or values) from the current frame.
META-R	Reinvokes the function in the current frame with possibly altered arguments.
META-CTRL-R	Reinvokes the function in the current frame (throws back to it and starts it over at its beginning).
CTRL-S	Searches for a frame containing a function with the specified string.
META-S	Prints the value of a special variable within the context of the current frame. Instance variables of self can also be specified even if not special.
META-CTRL-S	Prints a list of special variables bound by the current frame and the values they are bound to by the frame. If the frame binds self, all the instance variables of self are listed even if not special.
CTRL-T	Throws a value to a tag.
META-CTRL-U	Moves down the stack to the next interesting frame if the current frame is uninteresting.
CTRL-X	Toggles the flag in the current frame that causes a trap on exit.
META-X	Sets the flag that causes a trap on exit for the current frame and all the frames outside of it.
META-CTRL-X	Clears the flag that causes a trap on exit for the current frame and all the frames outside of it.
META-CTRL-W	Enters the window-based debugger.
META-<	Moves to the top of the stack.
META->	Moves to the bottom of the stack.

Table 13-2 Summary of Debugger Commands (Continued)

Key Sequence	Explanation of Command
ABORT	Quits to command level. This is not a command but something you can press to escape from typing a form.
HELP	Prints a help message.
CTRL-0 through CTRL-9, META-0 through META-9, META-CTRL-0 through META-CTRL-9	Numeric arguments to a debugger command are specified by typing a number (in base 10) with CTRL and/or META held down.
RESUME	Attempts to continue, using the first proceed type on the list of available ones for this error.
SUPER-A to SUPER-Z	The debugger assigns these commands to all the available proceed types for this error. The assignments are different each time the debugger is entered, so it prints a list of them when it starts up.

WINDOW-BASED DEBUGGER

14

Introduction

14.1 The window-based debugger is an alternative to the regular debugger; it performs the same functions but displays graphically rather than using sequential stream I/O. You invoke the window-based debugger by pressing META-CTRL-W while in the regular debugger. You can switch back and forth between the two debuggers any number of times while handling a single error. (Pressing the END key in the window-based debugger returns to the regular debugger.)

If you always want to enter the window-based debugger without first entering the regular debugger, set the `eh:*enter-window-debugger*` variable to `:always`.

`eh:*enter-window-debugger*`

Variable

If this variable is `nil`, the system uses the regular debugger. If `eh:*enter-window-debugger*` is `:always`, the system uses the window-based debugger. Finally, if the variable is `t`, then you are queried as to which debugger to use, or whether to set the variable to `:always`.

How to Use the Window-Based Debugger

14.2 The debugger window is divided into seven panes (see Figure 14-1).

At the bottom of the window-based debugger is a Lisp Listener pane, which ordinarily provides a read-eval-print loop similar to the regular keyboard debugger. More commands are available by using the mouse in the other panes as described in the following paragraphs.

At the top is an inspection pane, which usually displays the disassembled or interpreted code for the currently selected stack frame, depending on whether or not it is compiled. Items in this display are mouse-sensitive and can be inspected. Also, this pane has a scroll bar.

Next are the arguments and locals panes, side by side, displaying the names and values of the arguments to the current stack frame and its local variables; the panes are grayed out if there are none. Items in these panes are mouse-sensitive and can be inspected. The panes also have scroll bars. Clicking on the value of an argument prints it in the top inspection pane. The Modify command allows you to change the values of these arguments and local variables.

Next is the stack pane, which displays in a pseudo-list format the functions and arguments on the stack. Clicking on a function or argument, or sublists of them, causes them to be printed in the Lisp Listener pane as in the argument or local panes. Clicking the mouse to the left of a line containing a particular stack frame makes the debugger select this frame, which updates the above three panes to display information about this frame. Also, items in this stack pane are mouse-sensitive and can be inspected. Moreover, this pane has a scroll bar.

Figure 14-1 Window-Based Debugger

```

PROCESS-WAIT
=> 28. PUSH LOCAL10. ; ARGUMENTS
    29. PUSH ARG11. ; FUNCTION
    30. (AUX) APPLY-TO-INDS
    31. BR-NULL 33.
    32. (AUX) RETURN-NIL
    33. TEST FEF110. ; SYS::SCHEDULER-EXISTS
    34. BR-NULL 62.
    35. PUSH FEF19. ; SYS::SCHEDULER-STACK-GROUP
    36. EQ FEF18. ; SYS::ZCURRENT-STACK-GROUP
    37. BR-NULL 43.
    38. TEST FEF17. ; CURRENT-PROCESS
    39. BR-NULL 43.
    40. PUSH FEF111. ; *SYS::PROCESS-WAIT-IN-SCHEDULER
    41. SET-NIL PDL-PUSH
    42. (AUX) ZTHROW

```

```

Arg 0 (WHOSTATE): "Keyboard"
Arg 1 (FUNCTION): #<DTP-FUNCTION (:INTERNAL TV:KBD-IO-BUFFER-0
Rest arg (ARGUMENTS): (#<IO-BUFFER 17007202: empty, State: NIL
Local 1 (STATE): 7.
Specials:
INHIBIT-SCHEDULING-FLAG: NIL

```

```

(SYS::CALL-STACK-GROUP #<DTP-STACK-GROUP EH::ERROR-STACK-GROUP 72340056> #<BREAK :CONDITION-NAMES (BREAK CONDITION) :FORMAT
(EH:INVOKE-DEBUGGER #<BREAK :CONDITION-NAMES (BREAK CONDITION) :FORMAT-STRING "break.">))
(SIGNAL-CONDITION #<BREAK :CONDITION-NAMES (BREAK CONDITION) :FORMAT-STRING "break."> (:NO-ACTION) T)
(TV:KBD-INTERCEPT-ERROR-BREAK #<c-n-BREAK #<ZMACS-WINDOW-PANE Zmacs Window Pane 1 41713431 exposed>))
(EH::FOOTHOLD)
-> (PROCESS-WAIT "Keyboard" #<DTP-FUNCTION (:INTERNAL TV:KBD-IO-BUFFER-0ET 0.) -57251755> #<IO-BUFFER 17007202: empty, State:
(TV:KBD-IO-BUFFER-GET #<IO-BUFFER 17007202: empty, State: NIL>))
(#<ZMACS-FRAME Zmacs Frame 1 41712634 exposed> :ANY-TYI T)
(ZMEI::MACRO-STREAM-DEFAULT-HANDLER :ANY-TYI (T))
(ZMEI::MACRO-STREAM-IO :ANY-TYI T)
(TV:READ-ANY #<DTP-CLOSURE 17653251> T NIL NIL)
(#<ZMACS-FRAME Zmacs Frame 1 41712634 exposed> :READ-ANY)
(ZMEI::MACRO-READ :READ-ANY)
(ZMEI::MACRO-STREAM-IO-READ :READ-ANY NIL)
(ZMEI::MACRO-STREAM-IO-READ :READ-ANY)
(ZMEI::INPUT-WITH-PROMPTS #<DTP-CLOSURE 17653251> :READ-ANY)
((:METHOD ZWEI::WINDOW :EDIT) :EDIT)
((:INTERNAL (:METHOD ZWEI::ZMACS-WINDOW :COMBINED :EDIT) SYS::CONTINUATION))
(FUNCALL #<DTP-FUNCTION (:INTERNAL (:METHOD ZWEI::ZMACS-WINDOW :COMBINED :EDIT) SYS::CONTINUATION) -60271351>)
((:METHOD ZWEI::DISPLAYER :AROUND :EDIT) :EDIT #<DTP-FUNCTION (:INTERNAL (:METHOD ZWEI::ZMACS-WINDOW :COMBINED :EDIT) SYS::

```

Examine	Inspect	Resume	Bk Next	#<Stack-Frame PROCESS-WAIT PC=34>
Doc	Edit	Retry	Bk Exit	
Search	Report	Resume	Bk All	
Error	Arglist	Return	Step	
Stay	Modify	Abort	End	

```

>>Keyboard break.
>
Debugger Frame 2
//: Menu of all window-based debugger commands

```

Below the stack pane, and above the Lisp Listener pane, is the command menu and the Inspector history pane. The Inspector history pane maintains a list of all the items that have been previously inspected in the inspection pane, exactly as the standalone Inspector utility does.

The debugger commands in the command menu are divided into two categories: Examine and Resume. Refer to Table 14-1. You can also press HELP and select Command Display from the Help menu to view a listing of the commands.

Table 14-1 Window-Based Debugger Commands

Name	Keystroke	Description
<i>Examine commands:</i>		
Doc	CTRL-HELP	Shows documentation for each of the window-based debugger panes.
Search	CTRL-S	Prompts you for a string, then searches for this string in a stack frame. Search then selects this frame and displays its arguments, variables, and code. This works like CTRL-S in the regular debugger.
Error	CTRL-L	The Error command reprints the error message for the current error in the Lisp window. This works like CTRL-L in the regular debugger.
Stay	HYPER-S	Toggles the value of <code>eh:*enter-window-debugger*</code> to use the regular or window-based debugger.
Inspect	CTRL-I	Inspects either a mouse-sensitive item that you have selected or the value of a form that you enter from the keyboard. The display appears in the top window.
Edit	CTRL-E	The Edit command invokes the editor on a function after reading the name of the function or allowing you to select a function with the mouse. This works like CTRL-E in the regular debugger.
Report	CTRL-M	Mails a bug report containing the error message and a backtrace of the stack. This works like CTRL-M in the regular debugger.
Arglist	CTRL-A	The Arglist command asks for the name of a function that can be typed on the keyboard or selected with the mouse if it is on the screen. When you pick a flavor instance or a closure, the Arglist command asks for the message name to this flavor and prints its arguments. When you pick a line of a stack frame from the stack window, the Arglist command tries to align the printout of the arguments with the value supplied in this line in this frame. This works like CTRL-A in the regular debugger.
Modify	META-CTRL-A or META-CTRL-L	Prompts you to select an argument or local variable with the mouse and allows you to type (or select with the mouse) a new value to be substituted.

Table 14-1 Window-Based Debugger Commands (Continued)

Name	Keystroke	Description
<i>Resume commands:</i>		
Retry	META-CTRL-R	Attempts to restart the current frame, like the META-CTRL-R key sequence in the regular debugger.
Resume	RESUME	Resumes from the error. Clicking left on Resume is like pressing RESUME in the regular debugger.
Return	CTRL-R	Asks for the name of a value (which can be selected with the mouse) and returns it from the current frame, like CTRL-R in the regular debugger.
Abort	META-CTRL-ABORT	Leaves the debugger and aborts the program.
Bk Next	META-D	Toggles the flag that causes a trap on the next function call after you continue or otherwise exit the debugger. This works like META-D in the regular debugger.
Bk Exit	CTRL-X	Toggles the flag in the current frame that causes a trap on exit or throw through the frame. This works like CTRL-X in the regular debugger.
Bk All	SUPER-X	Toggles the flag that causes a trap on exit for the current frame and all outer frames. This works like META-X in the regular debugger.
Step	CTRL-D	Steps from the current point of execution in the current frame, but traps on the next function call. This works like CTRL-D in the regular debugger. Note that this command does not use the Stepper on the function; actually, it steps through the stack frame by frame, reinvoking the debugger on every function call.
End	END	Exits the debugger window, returning to the regular debugger.

Deexposed Windows and Background Processes

14.3 If the debugger is entered in a window that is not exposed, you are notified of this situation. The notification appears either as a brief message printed inside square brackets or as a small window that pops up with a brief message. The notification reminds you that you can select and expose the window in which the error happened by pressing TERM 0 S.

If the debugger is entered in a process that has no window or other suitable stream to type out on, the window system tries to assign it a *background window* and prints a notification to tell you it is there. Since the window is not exposed initially, a notification is printed (as described in the previous paragraphs) and you can press TERM 0 S to see the window.

If the system cannot notify you that an error occurred in another process because windows on the screen are locked, it displays a string containing flashing asterisks in the mouse documentation window. This display indicates that errors exist in the background.

If an error occurs in the keyboard process or the mouse process (either of which needs the window system), or in the scheduler's stack group, you are given several options. You can enter the debugger in a cold-load stream. The cold-load stream is a primitive facility for terminal I/O that bypasses the window system. When selected, it can also forcibly unlock the locks and allow the notification to be printed normally. (This can also be done by pressing TERM CTRL-CLEAR-INPUT.)

Introduction

15.1 The Inspector is a window-based utility for observing and modifying Lisp objects. Inspecting a particular object displays the components of this object. The type of object determines exactly what is displayed. For example, the components of a list are its elements; the components of a symbol are its value binding, function definition, property list, package, and print-name.

The components of displayed objects are mouse-sensitive; positioning the mouse cursor over an object causes a box to appear around that object. Once the object is *boxed*, you can press (click) the proper mouse button (see the mouse documentation window) to inspect the boxed object, modify it, or give the object as an argument to a command.

To access the Inspector, press SYSTEM I, select Inspector from the System menu, or use one of the following functions:

`inspect` &optional *x*

[c] Function

The `inspect` function enters the Inspector, displaying the contents of *x* (if specified). This function invokes the Inspector in its own process instead of in the calling process just as if you pressed SYSTEM I.

An inspector invoked with `inspect` or SYSTEM I does not have access to locally bound variables.

`inspect*` &optional *x*

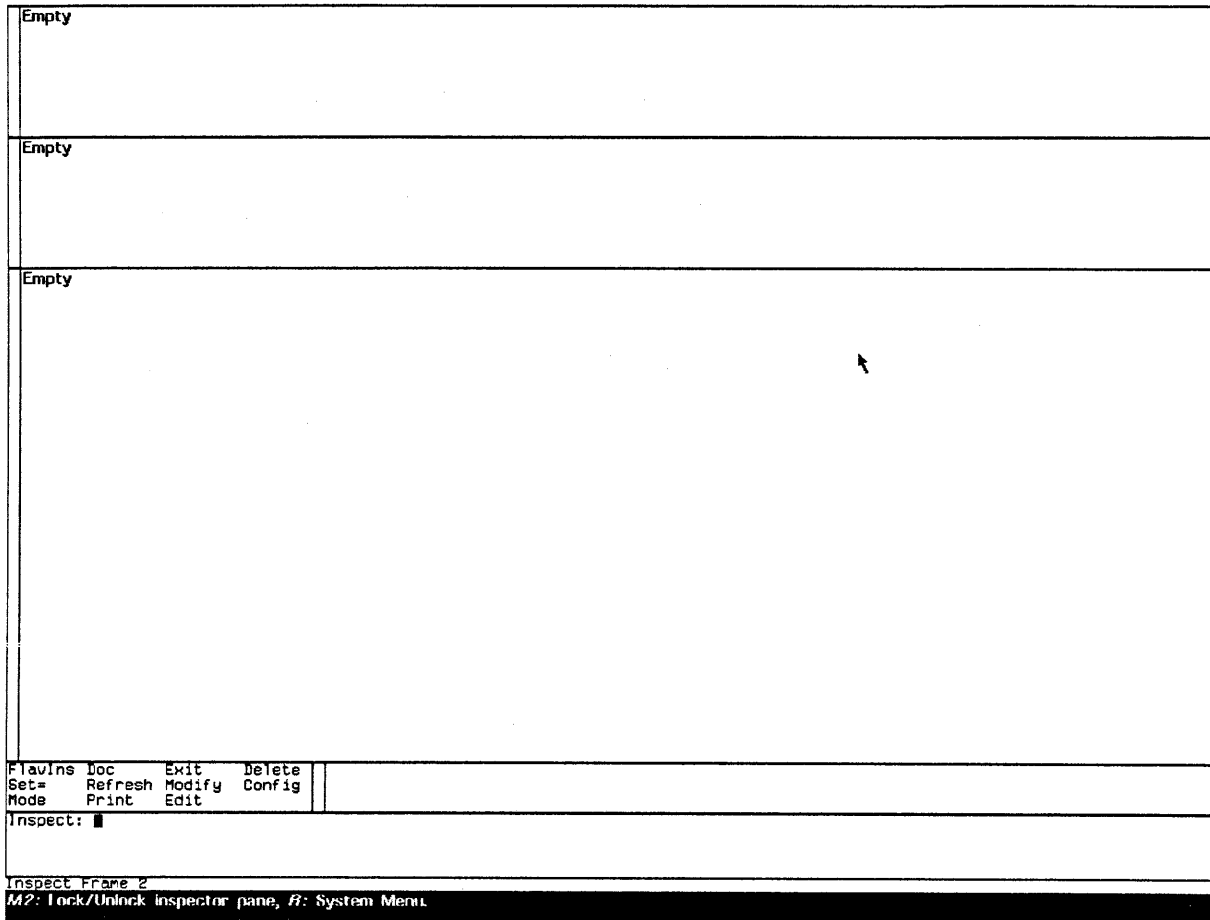
Function

The `inspect*` function enters the Inspector, displaying the contents of *x* (if specified). This function runs in the stack group of the calling process, rather than in a separate stack (as when you press SYSTEM I or use `inspect`).

This function maintains the environment. It has access to locally bound variables.

Once you enter the Inspector, a frame with 6 panes appears on your screen (see Figure 15-1).

Figure 15-1 Inspector Frame



The six panes include the following:

- Lisp Listener pane — Allows you to enter Lisp forms to inspect in the inspection panes, evaluate Lisp forms as in a regular Lisp Listener, and execute Inspector commands by typing their names.
- Three inspection panes — Display the contents of the objects you are inspecting. You can change this configuration from three panes to one large inspection pane, two horizontal inspection panes, or two vertical inspection panes. Depending on which configuration you use, the bottom or the leftmost pane shows the most recently inspected object.
- History pane — Maintains a list of the objects you have inspected and allows you to select them.
- Command menu pane — Provides several Inspector commands.

Lisp Listener Pane 15.2 Any form that you enter is echoed in the Lisp Listener pane, and then evaluated. The result of this evaluation is not printed; instead, it is inspected in the bottom or leftmost inspection pane.

In situations where the user invokes a menu operation, the Lisp Listener pane also serves to prompt the user for input.

The Inspector has several commands, most of which you can select by clicking on them in the command menu with the mouse. You can also invoke commands by typing the name of the command in the Lisp Listener pane or pressing a designated keystroke. A summary of the Inspector commands is listed in Table 15-1.

Table 15-1 Summary of Inspector Keyboard Commands

Name	Keystroke	Description
Bottom	META->	Scrolls the bottom or leftmost inspection pane to the bottom.
Break	BREAK	Runs a break loop in the typeout window of the bottom or leftmost inspection pane.
Config	SUPER-C	Invokes a menu that allows you to choose between four configurations: one with three small inspection panes, one with a single large inspection pane, one with two horizontal inspection panes, and one with two vertical inspection panes. You can set the default configuration in the Profile variable <code>tv:*inspector-configuration*</code> .
Delete	CTRL-CLEAR SCREEN	Erases the contents of the inspection panes and deletes the items in the history pane and history cache.
Doc	CTRL-HELP	Displays a pop-up window briefly describing each inspection pane.
Down	CTRL-V or CTRL-↓	Scrolls the bottom or leftmost inspection pane's display forward by one page.
Edit	HYPER-E	Edits the definition of the object at which you are pointing with the mouse or the definition of the form that you evaluate.
Exit	END	Exits the Inspector, returning the value of = (if you entered the Inspector with the <code>inspect*</code> function). Refer to Set=.
FlavIns	HYPER-F	Invokes the Flavor Inspector on a flavor or method. Refer to Section 16, Flavor Inspector, for details.
Mode	SUPER-M	Enters/exits the Lisp mode. This command causes the Lisp Listener pane to evaluate Lisp forms as in a regular Lisp Listener.

Table 15-1 Summary of Inspector Keyboard Commands (Continued)

Name	Keystroke	Description
Modify	CTRL-M	Allows you to change the value of an object. You are prompted to select the object by clicking on it with the mouse; then enter its replacement value through the Lisp Listener pane or click on another value. You can also invoke this command by pressing HYPHER while clicking on an item.
Print	SUPER-P	Invokes a menu that allows you to modify variables that control the printing format, such as <code>*print-base*</code> .
Refresh	CTRL-R	Redisplays the inspected objects reflecting any value changes to the components of those objects.
Set=	CTRL-=	Sets the value of the symbol = (equal sign) to an object that you select with the mouse or to the result of a form that you type in the Lisp Listener pane, for later reference and use. You can then do something such as <code>(setq foo (car =))</code> , if = was bound to a list. Also, the value of the = symbol is returned when exiting the Inspector if the Inspector was invoked with <code>inspect*</code> .
Up	META-V or CTRL-↑	Scrolls the bottom or leftmost inspection pane's display back by one page.
Top	META-<	Scrolls the bottom or leftmost inspection pane to the top.

Inspection Panes

15.3 When you enter a Lisp form to inspect, its contents appear in the bottom or leftmost inspection pane. The following paragraphs describe important features of the inspection panes.

Label At the top of an inspection pane is a label (the printed representation of the object being inspected in that pane) or else the words *a list*, meaning that a list is being inspected. The main body of an inspection pane displays the components of the object whose name, if any, appears in the label.

Mouse-Sensitive Items The following mouse operations are available when you position the mouse cursor on a mouse-sensitive item in an inspection pane:

- Click left — Inspects the object, displaying its contents in the bottom or leftmost inspection pane. In a three-pane configuration, the previous contents of the bottom inspection pane are moved to the middle inspection pane, while the previous contents of the middle inspection pane are moved to the top inspection pane. The previous contents of the top inspection pane disappear, but you can recover them by using the history pane. (See paragraph 15.4, History Pane.)
- Click right — Finds and inspects the function definition associated with the selected object (symbol), if it has one.

Changing the Configuration You can change the configuration of these three inspection panes to one large inspection pane, two horizontal inspection panes, or two vertical inspection panes by selecting the Config command from the command menu pane or by pressing SUPER-C. You can set the default configuration in the Profile variable `tv:*inspector-configuration*`.

Scrolling When the inspection pane is too small to display all of the information available, you can use the mouse to scroll through the information. Next to the left margin is a *scroll bar*. The scrolling icons in this scroll bar region indicate whether you need to scroll forward or backward to find the additional material. For example, a down arrow appears in the lower left corner if there is more information below. Similarly, an up arrow appears in the upper left corner if there is more information above. The mouse documentation window describes the scrolling operations you can perform when the mouse is positioned in the scroll bar region. For specific information about scrolling with the mouse, refer to the *Introduction to the Explorer System*.

Locking Inspection Panes You can lock up to $n - 1$ of the n existing inspection panes to allow the continued inspection of items from that pane without the original item scrolling out of the history pane. (For example, if there are three inspection panes, you can lock up to two of them.) You can toggle the locked status by double clicking middle anywhere within the inspection pane. The symbol of a padlock is printed in the upper right corner of the inspection pane to indicate it is locked.

, **, and *** Variables** If you are in inspect mode, the last three objects that you inspected are stored in `*` (most recently inspected), `**`, and ``. If you are in Lisp mode, the values of these variables are bound to the results of evaluated forms, as in a standard Lisp Listener. However, if you switch modes, these values may be intermixed: one may be the result of an evaluated form and another may be the value of an inspected object.

Again, the type of object being inspected determines what is displayed in the inspection panes. The following table correlates objects and their respective displays.

Table 15-2

Inspector Display According to Object Type	
Object	Contents of Display
Symbol	The print name, value, function, property list, and package are displayed. All but the name and the package are modifiable.
List	The list is displayed. Any piece of substructure is selectable, and any car or atom in the list can be modified.
Instance	The flavor of the instance, the method table, and the names and values of the instance variable slots are displayed. The instance variables are modifiable.
Closure	The function and the names and values of the closed variables are displayed. The values of the closed variables are modifiable.
Named structure	The name and values of the slots are displayed. The values are modifiable.
Array	The leader of the array, if present, is displayed. For one-dimensional arrays, the elements of the array are also displayed. The elements are modifiable.
FEF	The disassembled code is displayed.
Select method	The keyword/function pairs are shown in alphabetical order by keyword. The function associated with a keyword is settable via the keyword.
Stack group	The name and values of the slots of a stack group, including regular and special PDLs, are displayed.

History Pane

15.4 The history pane maintains a list of all objects that have been inspected. Of the objects listed in the history pane, the object inspected first tops the list, and the object inspected last is at the bottom of the list. The last lines of the history pane are always the objects currently being inspected in the inspection panes.

You can click the mouse on any mouse-sensitive object in the history pane to inspect that object.

There is a line region at the left margin of the history pane that allows other operations. When you move the mouse blinker to this line region (the area immediately to the left of the history lines), the blinker changes shape to a right-pointing arrow. At this time, you can click left on the mouse to inspect the object (the contents of the object appear in the bottom or leftmost inspection pane). This feature can be helpful when the object is a list and you

cannot conveniently place the mouse blinker at the opening parenthesis. If you click middle on the mouse, the Inspector deletes the object you have selected with the right-pointing arrow from the history pane list. Clicking right on the mouse causes the Inspector to find and inspect a function definition, if one exists, for the object.

When the number of objects that have been inspected exceeds the available length of the history pane, the history pane employs scrolling icons (up and down arrows) in the scroll bar region. You may need to scroll lines or pages to find the appropriate object previously inspected.

The history pane also maintains a cache that allows quick redisplay of previously displayed objects.

NOTE: Merely reinspecting an object does not reflect any changes in its state. The Refresh command in the Inspector menu clears everything from the cache and retrieves the current contents of each object in the history pane. Clicking middle while the mouse blinker is in the line region deletes an object from the cache as well as from the history pane.

Command Menu Pane

15.5 The command menu pane appears towards the lower left portion of the Inspector pane (see Figure 15-1). The menu contains a set of commands that you can select with the mouse. (For descriptions of these commands, see the name column of Table 15-1).

16

FLAVOR INSPECTOR

Overview

16.1 The Flavor Inspector is a window-based utility for observing and modifying flavors, including their component flavors, instance variables, and methods. The Flavor Inspector works in much the same way as the Inspector.

The components of displayed objects are mouse-sensitive; positioning the mouse cursor over an item causes a box to appear around that item. Once the item is *boxed*, you can press (click) the proper mouse button (indicated in the mouse documentation window) to inspect the item or modify it.

To access the Flavor Inspector, select Flavor Inspector from the System menu or use the following function:

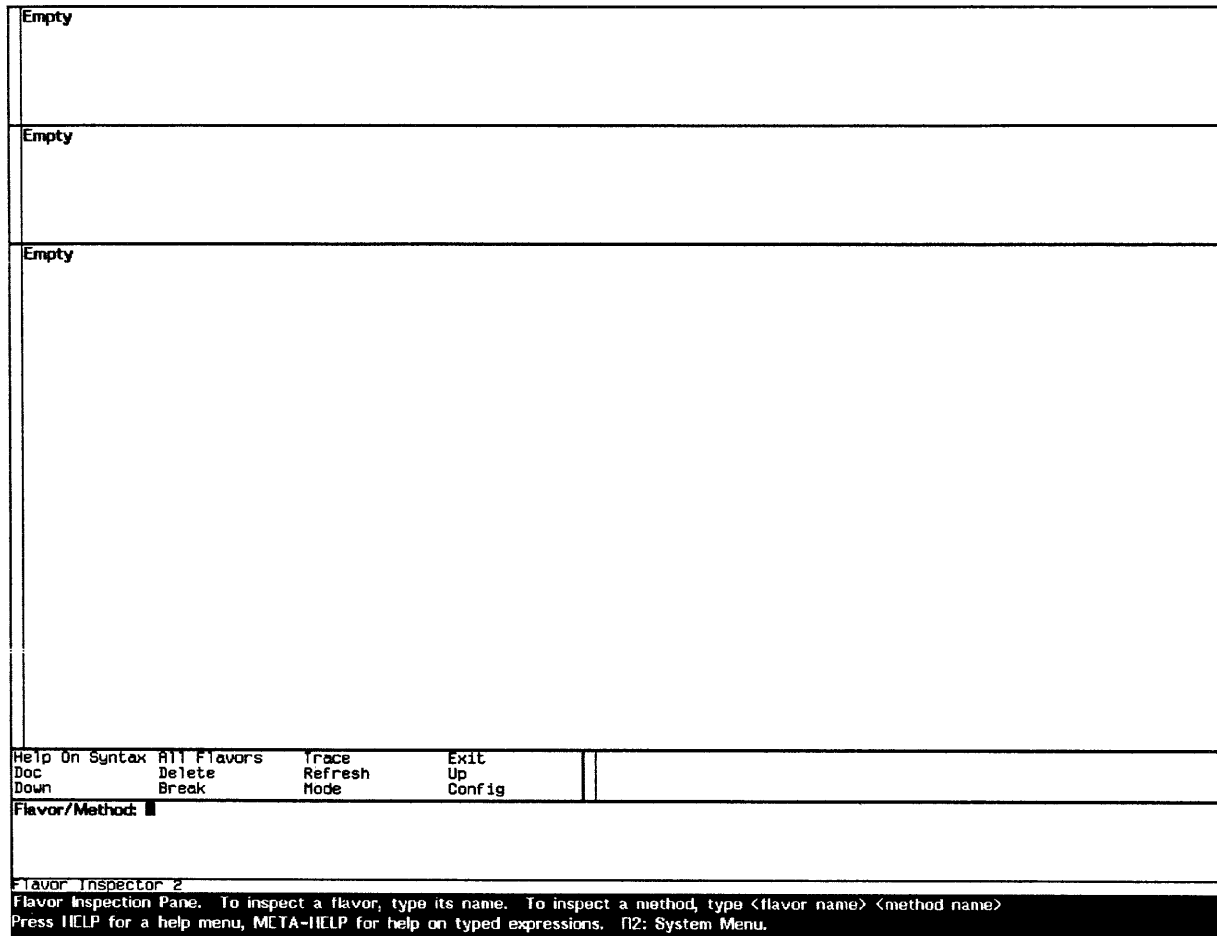
<code>inspect-flavor</code> &optional <i>x</i>	Function
	Enters the Flavor Inspector, displaying the contents of <i>x</i> (if specified). This function invokes the Flavor Inspector in its own process instead of in the calling process.

Once you enter the Flavor Inspector, a frame with six panes appears on the screen (see Figure 16-1).

The six panes include the following:

- Lisp Listener pane — Allows you to enter flavor and method names to inspect, Lisp forms, and the names of the commands that are in the command menu pane.
- Three inspection panes — Display the contents of the objects you are inspecting. You can change this configuration from three panes to one large inspection pane, two horizontal inspection panes, or two vertical inspection panes. Depending on which configuration you use, the bottom or the leftmost pane shows the most recently inspected object.
- History pane — Maintains a list of the objects you have inspected and allows you to select them.
- Command menu pane — Provides general Flavor Inspector commands.

Figure 16-1 Flavor Inspector Frame



Lisp Listener Pane 16.2 As in the Inspector, two modes are available in the Lisp Listener pane: inspect mode and Lisp mode. In inspect mode (the default mode), the expression that you enter is echoed in the Lisp Listener pane and then evaluated. The result of this evaluation is not printed; it is inspected in the bottom or leftmost inspection pane. Lisp mode works like a regular Lisp Listener. You can toggle between these two modes.

Pressing META-HELP displays help on the processing of typed expressions. (This is the Help on Syntax command in the command menu pane.)

You can type one of the following expressions in the Lisp Listener pane:

- A flavor name to inspect, terminated by pressing RETURN.
- A method specification to inspect. The syntax is one of the following:

```
(flavor-name method-name)
```

```
(flavor-name method-type method-name)
```

```
flavor-name method-name
```

```
flavor-name method-type method-name
```

You must press RETURN to terminate the last two types of expressions. The method-type is one of the following:

```
:after      :default
:and        :or
:around     :override
:before     :wrapper
:case
```

While typing these expressions, you can press the space bar to complete a flavor or method name. You can also use the following input editor completion commands:

- ESCAPE — Recognition Completion (same as the space bar)
- CTRL-/ — List Recognition Completions
- SUPER-ESCAPE — Apropos Completions (complete word as an inner substring)
- SUPER-/ — List Apropos Completions
- HYPER-ESCAPE — Spelling Corrected Completions (corrects minor typing errors)
- HYPER-/ — List Spelling Corrected Completions

The Flavor Inspector provides several commands, most of which you can execute by clicking on them with the mouse in the command menu. You can also invoke these commands by typing their name in the Lisp Listener pane or by pressing a keystroke. These commands are described in Table 16-1.

Table 16-1 General Flavor Inspector Commands

Name	Keystroke	Description
All Flavors	SUPER-A	Displays all flavor names currently defined in the system in an inspection pane. You might want to browse through flavor structures.
Bottom	META->	Scrolls the main inspection pane to the bottom.
Break	BREAK	Enters the break read-eval-print loop.
Config	SUPER-C	Invokes a menu that allows you to choose between four configurations: one with three small inspection panes, one with a single large inspection pane, one with two horizontal inspection panes, and one with two vertical inspection panes. You can set the default configuration in the Profile variable <code>tv:*flavor-inspector-configuration*</code> .
Delete	CTRL-CLEAR SCREEN	Deletes all inspected objects from the history and inspection panes.
Down	CTRL-V or CTRL-↓	Scrolls the main inspection pane's display forward one page.
Exit	END	Exits the Flavor Inspector.
Help on Syntax	META-HELP	Provides input editor help.
Mode	SUPER-M	Toggles between Lisp mode and inspect mode.
Refresh	CTRL-R	Redisplays the inspected objects, updating any fields that have changed values.
Top	META-<	Scrolls the main inspection pane to the top.
Trace	SUPER-T	Turns on Trace for a specified method. This command is equivalent to <code>(trace method-spec)</code> . Use <code>(untrace)</code> to turn off all tracing.
Up	META-V or CTRL-↑	Scrolls the main inspection pane's display back one page.

Inspection Panes

16.3 When you enter a flavor or method name to inspect in the Lisp Listener pane, its contents appear in the bottom or leftmost inspection pane. Similarly, if you click *left* on a mouse-sensitive object in any of the inspection panes or in the history pane, its contents appear in the bottom or leftmost inspection pane.

In the three-pane configuration, the previous contents of the bottom inspection pane are moved to the middle inspection pane, while the previous contents of the middle inspection pane are moved to the top inspection pane. The previous contents of the top inspection pane disappear, but you can recover them by using the history pane. (See paragraph 16.4, History Pane.)

The following paragraphs describe important features of the inspection panes.

Label At the top of an inspection pane is a label (the printed representation of the object being inspected in that pane). The main body of an inspection pane displays the components of the object whose name, if any, appears in the label.

Changing the Configuration You can change the configuration of the three inspection panes to one large inspection pane, two horizontal inspection panes, or two vertical inspection panes by selecting the Config command from the command menu pane or by pressing SUPER-C. You can set the default configuration in the Profile variable `tv:*flavor-inspector-configuration*`.

Scrolling When the inspection pane is too small to display all of the information available, you can use the mouse to scroll through the information. Next to the left margin is a *scroll bar*. The scrolling icons in this scroll bar region indicate whether you need to scroll forward or backward to find the additional material. For example, a down arrow appears in the lower left corner if there is more information below. Likewise, an up arrow appears in the upper left corner if there is more information above. The mouse documentation window describes the scrolling operations you can perform when the mouse is positioned in the scroll bar region. For specific information about scrolling with the mouse, refer to the *Introduction to the Explorer System*.

Locking Inspection Panes You can lock up to $n - 1$ of the n existing inspection panes to allow the continued inspection of items from that pane without the original item scrolling out of the history pane. (For example, if there are three inspection panes, you can lock up to two of them.) You can toggle the locked status by double clicking middle anywhere within the inspection pane. The symbol of a padlock is printed in the upper right corner of the inspection pane to indicate that it is locked.

, **, and *** Variables** If you are in inspect mode, the last three objects that you inspected are stored in `*` (most recently inspected), `**`, and ``. If you are in Lisp mode, the values of these variables are bound to the results of evaluated forms, as in a standard Lisp Listener. However, if you switch modes, these values may be intermixed: one may be the result of an evaluated form and another may be the value of an inspected object.

Mouse Operations on Flavor Names The following mouse operations are available when you position the mouse cursor on a flavor name in one of the inspection panes:

- Click left — Inspects this flavor, displaying its contents in the bottom or leftmost inspection pane.
- Click right — Displays a menu of other operations, as described in Table 16-2.

Table 16-2 Flavor Commands

Name	Description
Instance Variables	<p data-bbox="560 310 1421 401">Inspects all instance variables of this flavor. The display tells you whether the variables are local to the flavor you are inspecting or inherited from another flavor.</p> <p data-bbox="560 432 1421 522">You can click left on one of the variables to inspect the methods of <i>this</i> flavor that reference the variable, or you can click middle to inspect the methods of a <i>specified</i> flavor that reference the variable.</p> <p data-bbox="560 548 1421 667">The display also tells you whether the variable is gettable, settable, initable, or special, and what its default value is. You can click left on the default value to pretty print it in the typeout window. Also, you can click middle to inspect the value using a standard Inspector window.</p>
Local Methods	Inspects methods defined locally for this flavor.
All Methods	Inspects all the methods defined locally for this flavor and the methods inherited by this flavor.
All Methods, Sorted	Sorted version of the All Methods command. The methods are listed in alphabetical order.
All Handled Messages	<p data-bbox="560 924 1421 1014">Inspects all messages handled by this flavor. The display tells you the names of the messages and the component flavors providing the handlers.</p> <p data-bbox="560 1045 1421 1104">You can click any button on a message name to display the method combination for the message handler for the flavor.</p>
Component Flavors	Inspects flavors that make up this flavor (not a hierarchical display).
Dependent Flavors	Inspects flavors that directly or indirectly depend on this flavor.
Miscellaneous Data	Displays miscellaneous data on this flavor. For example, the display tells you the package where the flavor is defined, its included flavors, its properties, and its method hash table if it has one.
Edit	Edits this flavor in a Zmacs buffer.

Mouse Operations on Method Names The following mouse operations are available when you position the mouse cursor on a method name in one of the inspection panes:

- Click left — Inspects method details.
- Click middle — Displays the combined methods used in handling the message for this flavor.
- Click right — Displays a menu of other operations, as described in Table 16-3, which follows:

Table 16-3 Method Commands

Name	Description
Inspect	Displays information about this method. The display tells you the source file, method combination type, arglist, and documentation for the method. It also tells you what instance variables and messages are referenced.
Show Combined Methods	Displays the combined methods used in handling the message for this flavor.
Disassemble	Uses a standard Inspector window to display disassembled code. Press END to return to the Flavor Inspector.
Edit Source	Edits this method in a Zmacs buffer.
Trace	Invokes a trace window to trace this method. For information on Trace options, refer to Section 18, Trace, of this manual.

History Pane

16.4 The history pane maintains a list of all objects that have been inspected. Of the objects listed in the history pane, the object inspected first tops the list, and the object inspected last is at the bottom of the list. The last lines of the history pane are always the objects currently being inspected in the inspection panes.

You can click the mouse on any mouse-sensitive object in the history pane to inspect that object.

There is a line region at the left margin of the history pane that allows other operations. When you move the mouse blinker to this line region (the area immediately to the left of the history lines), the blinker changes shape to a right-pointing arrow. At this time, you can click left on the mouse to inspect the object (the contents of the object appear in the bottom or leftmost inspection pane). If you click middle on the mouse, the Inspector deletes the object you have selected with the right-pointing arrow from the history pane list.

When the number of objects that have been inspected exceeds the available length of the history pane, the history pane employs scrolling icons (up and down arrows) in the scroll bar region. You may need to scroll lines or pages to find the appropriate object previously inspected.

The history pane also maintains a cache that allows quick redisplay of previously displayed objects.

NOTE: Merely reinspecting an object does not reflect any changes in its state. The Refresh command in the command menu pane clears everything from the cache and retrieves the current contents of each object in the history pane. Clicking middle while the mouse blinker is in the line region deletes an object from the cache as well as from the history pane.

**Command Menu
Pane**

16.5 The command menu pane appears towards the lower left portion of the Flavor Inspector frame (see Figure 16-1). The menu contains a set of commands that you can select with the mouse (see the name column of Table 16-1).

Introduction

17.1 Peek is a window-oriented utility that displays a continually updating system status of the following items:

- Processes
- Areas
- Windows
- Network
- Servers
- File System
- Counters
- Function Histogram

You select the status to be displayed from one of the above listed modes. When the Peek window is displayed, select a mode (or the Host Status command) by pressing the key with the first letter of the mode/command you want (P for Processes, N for Network, and so on). One exception is the Function Histogram mode, which is invoked with M. (M stands for metering; F is for the File System mode.) Alternatively, you can select a Peek menu item with the mouse.

You can enter Peek by using the `peek` function or by pressing SYSTEM P.

`peek &optional character`

Function

To access several modes of system status, call the function `peek`. You enter a specific mode by typing the single key naming the mode you want. You can specify an initial mode with the argument, *character*. The *character* is the first letter of the mode name except for the Function Histogram mode, which is M. (M stands for metering; F is for the File System mode.)

Example: (`peek "M"`)

Rather than attempting to enter Peek by calling the `peek` function, it is usually easier to press SYSTEM P on the keyboard.

When you press CTRL-HELP (or select the Help menu item) after entering Peek, the following window (Figure 17-1) appears on your screen:

Figure 17-1 Example of the Peek Window

```

                                PEEK HELP

PEEK is a utility composed of MODES and COMMANDS.

MODES contain status information about various components of the Explorer
system. They are listed at the bottom of the screen and are briefly described
below. A MODE can be displayed by highlighting it with the mouse cursor and
then clicking Mouse-Left or by entering its key assignment.

  * WINDOWS--Displays the hierarchy of window inferiors, depicting which
              are exposed. Key assignment: w

  * AREAS--Displays status of areas, including how much memory is allocated
            and used. Key assignment: a

  * PROCESSES--Lists status of every process. Key assignment: p

  * FILE STATUS--Displays status of file protocol connections to remote file
                 systems. Key assignment: f

  * SERVERS--Lists all servers, who they are serving, and their status.
             Key assignment: s

  * NETWORK--Displays network connection information. Key assignment: n

  * COUNTERS--Displays the value of all microcode meters. Key assignment: c

  * FUNCTION HISTOGRAM--Displays most frequently called functions in one or
                       more processes. Key assignment: h

COMMANDS are actions that can be performed. They are listed at the bottom
of the screen in the right-hand corner and are briefly described below.

  * DOCUMENTATION--Displays this message. Key assignment: d

  * SET TIMEOUT--Reset time between updates of the screen in units of 1/60ths of
                 a second. Key assignment: t

  * HOST STATUS--Displays network statistics of all known hosts. Key assignment: h

  * EXIT--Exits the Peek utility. Key assignment: <ESC> or q

Press the space bar to remove this message.
    
```

Peek

Modes			Commands	
Windows	Processes	Servers	Counters	Documentation
Areas	FILE Status	Network	Function Histogram	Host Status
Print out some brief general documentation about Peek.			SET TIMEOUT	Exit
Key assignments: CTRL-HELP or META-HELP				

As shown in Figure 17-1, the Peek windows include the following:

- Modes menu window
- Commands menu window
- Viewing window

Mode and Command Menu Windows

17.2 The Modes and Commands menu windows appear at the bottom of the Peek window (see Figure 17-1). The Modes menu window lists ways to examine the status of different aspects of the Explorer system. The Commands menu window lists some operations you can perform. You can select one of the Peek modes or commands either with the mouse or by entering the appropriate key sequence as shown in the right-hand column of Table 17-1, which follows:

Table 17-1

Peek Modes and Commands

Name	Keystroke
------	-----------

Modes:

Processes	P
Counters	C
Areas	A
File Status	F
Windows	W
Servers	S
Network	N
Function Histogram	M

Commands:

Host Status	H
Set Timeout	T
Exit	END
Documentation	CTRL-HELP

Viewing Window

17.3 The viewing window displays status information about whichever item you select from the Modes or Commands menu.

Most of the displays are dynamic; that is, they have periodic updates. You can set the time interval between updates by using the Set Timeout (T) command. After you press T, a pop-up window appears, allowing you to update the time interval in 60ths of a second.

Some of the items in the displays are mouse-sensitive. These items and the operations that can be performed on them vary from one display to another.

The Peek window has scrolling capabilities for use when the status display exceeds the available display area.

The Peek window continues to update its display as long as it is exposed. For this reason, a Peek window can be used to examine what is being done in other windows in real time.

Processes 17.3.1 Selecting the Processes item from the Modes menu window displays the process name, status, priority, quantum, and idle time of each process. Clicking on a process name displays a pop-up menu of useful actions to perform on this process, as follows:

You can perform the following operations on a process:

- **Debugger** — Invokes the debugger to examine the selected process. This is sometimes the only way of entering a process to debug it.
- **Arrest** — Arrests the selected process; undone by Un-Arrest.
- **Un-Arrest** — Unarrests the selected process; Un-Arrest is the complement of Arrest.
- **Flush** — Forces the process to wait forever. You are asked for confirmation. A process cannot `:flush` itself. Flushing a process is different from stopping it in that the process is still active, and if it is reset or preset, it starts running again.

Note that a flushed process is reset by the window system if it is a process controlled by `process-mixin` and the window is selected or exposed. Also, the flushed process is given a run reason if it does not have one.

- **Reset** — Resets the selected process. You are asked for confirmation.
- **Reset & Enable** — Resets and enables the selected process. You are asked for confirmation.
- **Kill** — Kills the selected process. You are asked for confirmation.
- **Describe** — Describes the selected process; equivalent to `(describe process-object)`.

For more information on the `describe` function, refer to Section 25, Miscellaneous Debugging Functions.

- **Priority** — Changes priority of the selected process.
- **Inspect** — Invokes the Inspector and inspects this process; equivalent to `(inspect process)`.

Figure 17-2 Example of the Peek Processes Screen

Process Name	State	Priority	Quantum	%	Idle
Flavor Inspector 2	Keyboard	0.	60/60.	0.0%	22 min
Inspect Frame 2	Keyboard	-1.	60/60.	0.0%	27 min
Dormant FTP Connection GC	Sleep	0.	60/60.	0.0%	28 min
Vt100 Frame 1-Typein	Keyboard	0.	60/60.	0.0%	2 hr
Vt100 Frame 1-Typeout	Never-open	0.	60/60.	0.0%	2 hr
File Server	Chaosnet Input	0.	60/60.	0.0%	2 min
Glossary Frame 1	Keyboard	-1.	60/60.	0.0%	19 hr
Profile Frame 1	Keyboard	-1.	60/60.	0.0%	20 hr
Print Daemon	Queue Empty	-1.	60/60.	0.0%	20 hr
Peek Frame 1	Run	0.	35/60.	15.0%	
Znacs Frame 1	Keyboard	-1.	60/60.	6.0%	18 sec
Mail Daemon	Mailer Sleep	-5.	60/60.	0.0%	6 min
TCP Background	Background Task	15.	60/60.	0.0%	2 sec
FTP Server	UDP Input	-5.	60/60.	0.0%	22 hr
IP Packet Fragment timer	Sleep	-15.	60/60.	0.0%	25 sec
ICMP Listener	ICMP Listen	15.	60/60.	0.0%	1 min
Suggestions Frame 1	Arrest	0.	60/60.	0.0%	20 hr
Pop-Up Keystrokes	Pop Up Keystrokes	0.	10/10.	0.0%	20 hr
Hardware Monitor	Hardware Event Wait	-50.	60/60.	0.0%	22 hr
Tn-Update	Sleep	-5.	60/60.	0.0%	1 min
GC Daemon	GC Daemon	0.	60/60.	0.0%	22 hr
GC-PROCESS	Stop	0.	0/60.	0.0%	forever
Dormant FILE connection GC	Qfile gc sleep	-2.	60/60.	0.0%	17 min
NUBUS Receiver, #HF0	Wait NuEther Input	25.	60/60.	7.0%	
Chaos RUT transmitter	Stop	30.	0/60.	0.0%	forever
Chaos Background	Background Task	25.	60/60.	0.0%	7 sec
Screen Manager Background	Screen Manage	0.	60/60.	0.0%	1 min
Mouse	MOUSE	30.	60/60.	3.0%	2 sec
Keyboard	terminal-	30.	60/60.	0.0%	
Initial Process	Keyboard	-1.	60/60.	0.0%	1 hr
Page-Background	Sleep	-100.	60/60.	0.0%	17 min
Clock Function List					
UPDATE-FUNCTION-HISTOGRAM					
BLINKER-CLOCK					
Peek Processes					
Modes					
Windows	FILE status	Servers	Counters	Commands	
Areas		Network	Function Histogram	Documentation	Host Status
List status of every process -- why waiting, how much run recently.				Set Timeout	Exit
Key assignments: p					

Counters 17.3.2 When you select the Counters item from the Modes menu window, Peek lists the statistics of all the microcode meters. No items are mouse-sensitive. Several of the counters provide statistics on virtual memory, the paging system, garbage collection, and system bus errors. Also, there are several miscellaneous counters and values. Not all counters are of general interest.

For detailed information on these counters, refer to the *Explorer System Software Design Notes*.

Figure 17-3 Example of Counters Screen

%COUNT-FIRST-LEVEL-MAP-RELOADS	5906945
%COUNT-SECOND-LEVEL-MAP-RELOADS	239490009
%COUNT-PDL-BUFFER-READ-FAULTS	57572642
%COUNT-PDL-BUFFER-WRITE-FAULTS	3161475
%COUNT-PDL-BUFFER-MEMORY-FAULTS	105195467
%COUNT-META-BITS-MAP-RELOADS	0
%COUNT-CONS-WORK	121871584
%COUNT-SCAVENGER-WORK	0
%TVU-CLOCK-RATE	67
%LOWEST-DIRECT-VIRTUAL-ADDRESS	33553400
%COUNT-FINDCORE-STEPS	1229161
%COUNT-FINDCORE-EMERGENCIES	0
%COUNT-FINDCORE-CLEAR-PAGES	4582
%PAGE-TABLE-SEARCH-COUNT	0
%PHT-SEARCH-DEPTH	12
%COUNT-CLEAN-PAGE-REQUESTS	3033
%COUNT-CLEAN-PAGE-REQUEST-FAILED	0
%COUNT-DISK-PAGE-READS	17525
%COUNT-FRESH-PAGES	15141
%COUNT-DISK-PAGE-READ-RESUBMISSIONS	0
%COUNT-DISK-PAGE-WRITE-OPERATIONS	3022
%COUNT-DISK-PAGE-WRITES	19260
%COUNT-DISK-PAGE-WRITE-APPENDS	15438
%COUNT-DISK-PAGE-WRITE-WRITS	520
%COUNT-DISK-PAGE-WRITE-BUSYS	2430
%ABORTED-SWAPOUTS	2
%COUNT-SWAPOUT-PAGE-COUNT-REACHED	164
%SWAPOUT-SIZED-BY-RDB-OR-PAGE-COUNT	164
%DISK-WAIT-TIME	620579994
%TOTAL-PAGE-FAULT-TIME	0
%COUNT-SB-FROM-SWAPPER	0
%SWAPOUT-SEQUENCE-BREAKS-IN-PROGRES	0
%LEAST-USED-PAGE	14012
%MOST-RECENTLY-REFERENCED-PAGE	5712
%PAGE-HASH-TABLE-ADDRESS	4095754240
%PHT-INDEX-SIZE	14
%PHT-INDEX-LIMIT	131072
%PHYSICAL-PAGE-DATA-ADDRESS	4095737856
%PHYSICAL-PAGE-DATA-END	16384
%COUNT-NUBUS-GACBL-RETRIES	41164
%COUNT-NUBUS-PARITY-RETRIES	0
%METER-WAIT-TIME	0
%TGC-COUNTER	0
%SLOTS-I-OWN	4294967295
%IO-SPACE-VIRTUAL-ADDRESS	33422336
%CRASH-RECORD-PHYSICAL-ADDRESS	4126905248

Statistic Counters

Modes	Processes	Servers	Counters	Commands
Windows	FILE Status	Network	Function Histogram	Documentation Host Status
Areas				Set Timeout Exit

Display the values of all the microcode meters.
Key assignments: c or %

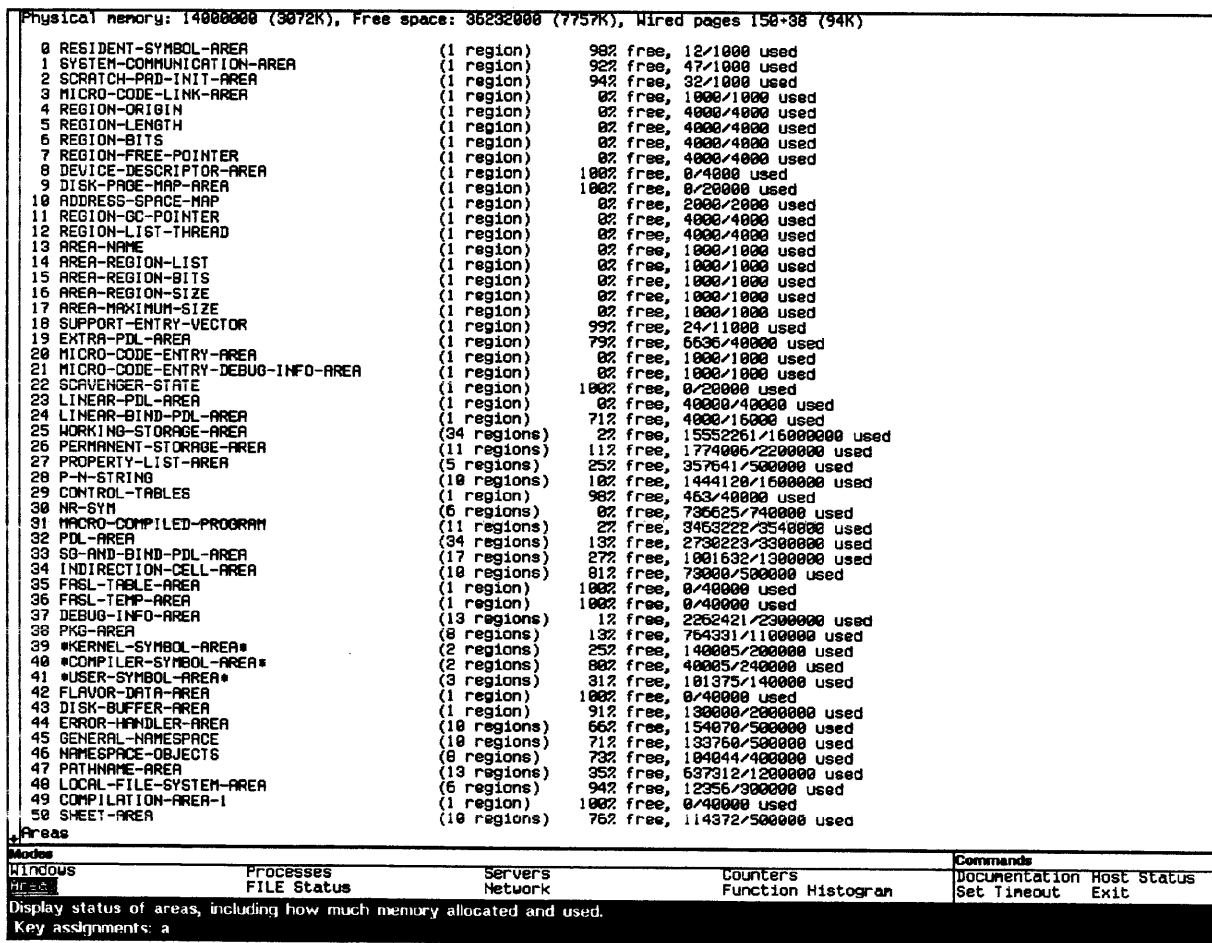
Areas 17.3.3 Selecting the Areas item from the Modes menu window displays for each area the area name, number of regions, and allocation. Clicking on an area name displays information about all the regions associated with the particular area.

An *area* provides a way of grouping related objects. For example, the working objects of Zmacs are kept together in the ZWEI area.

In the display, the areas that precede WORKING-STORAGE-AREA are special system areas (composed of one region each). All the other areas are for user objects. Most user objects are created by default in the WORKING-STORAGE-AREA. The system creates some special user objects elsewhere, such as flavor system data objects in the flavor area, pathnames in the pathname area, and sheets in the sheet area.

Refer to the Storage Management section of the *Explorer Lisp Reference* for details on areas, properties of areas and regions, and the describe-area and describe-region functions. The describe-area and describe-region functions are also discussed briefly in Section 25, Miscellaneous Debugging Functions, of this *Explorer Tools and Utilities* manual.

Figure 17-4 Example of the Areas Screen



File Status 17.3.4 When you select the File Status item from the Modes menu window, Peek lists all the hosts that have remote file connections to/from this host and additionally shows any connections (host-units) associated with those hosts. This list of hosts is not a complete list of hosts that are accessible on the network.

Clicking on a host invokes a menu of operations to perform on that host, including the following:

- **Reset** — Resets all connections associated with this host.
- **Host Status One** — Performs the equivalent of `(net:host-status selected-host)` to show if this host is responding over the network.
- **Insert Host Status** — Puts a static display of this host's status on the display.
- **Remove Host Status** — Removes the static display of this host's status from the display.
- **Describe** — Describes the selected host; equivalent to `(describe host-object)`.
- **Inspect** — Invokes the Inspector and inspects this host; equivalent to `(inspect host-object)`.

Clicking on a host-unit (connection) invokes a similar menu of operations to perform, but only includes operations Reset, Describe, and Inspect.

Figure 17-5 Example of File Status Screen

```

LISPM Host NEWTON.t1-7
LISPM Host Lima.t1-7
Host unit #<HOST-UNIT 27124400>, control connection in OPEN-STATE
UNIX-UCB Host Janus.t1-7
Host unit #<FTP-CONTROL-CONNECTION 50645395>, control connection in ESTABLISHED state
LISPM Host ICARUS.t1-7
LMFS Host SYMA.t1-7
VMS4 Host RUSOME.t1-7
LISPM Host MR-X.t1-7
Host unit #<HOST-UNIT 30851536>, control connection in NONEXISTANT-STATE
LISPM Host L-YOUNG.t1-7
UNIX-UCB Host Jd-young.t1-7
LISPM Host Romeo.t1-7
LISPM Host dsq.t1-7
Host unit #<HOST-UNIT 24734575>, control connection in OPEN-STATE
LISPM Host DLS.t1-7
LISPM Host Quebec.t1-7
LISPM Host Brigham-Young.t1-7
LISPM Host DSO.BOOT
LISPM Host DLS.BOOT
LISPM Host QUEBEC.BOOT
LISPM Host BRIGHAM-YOUNG.BOOT
LISPM Host ROMEO.BOOT

```

File System Status

Modes		Processes		Servers		Counters		Commands	
Windows		FILE STAT		Network		Function Histogram		Documentation	Host Status
Areas								Set Timeout	Exit

Display status of FILE protocol connections to remote file systems.
Key assignments: f

Windows 17.3.5 Selecting the Windows item from the Modes menu window lists the hierarchy of windows in the system. Clicking on a window name pops up a menu of useful actions to be performed on this window, as follows:

- **Deexpose** — If the window is exposed, Deexpose makes it not visible by exposing the next window in the window hierarchy.
- **Expose** — Puts this window on the top of the window hierarchy and makes it visible, but not ready for input.
- **Select** — Exposes this window and allows it to accept input.
- **Deselect** — If the window is currently selected, Deselect makes the window unable to accept any input.
- **Deactivate** — Removes this window from the window hierarchy but does not delete it. The Select command on the System Menu will not list this window (until it is exposed again).
- **Kill** — Deletes this window.
- **Bury** — If the window is exposed, Bury puts the window on the bottom of the window hierarchy and selects the next window in the hierarchy.
- **Describe** — Describes the selected window; equivalent to `(describe window-object)`.
- **Inspect** — Invokes the Inspector and inspects this window; equivalent to `(inspect window-object)`.

For details on these items, refer to the *Explorer Window System Reference*.

Figure 17-6 Example of the Peek Windows Screen

```

Screen Main Screen
Peek Frame 1
  Basic Peek 1
  Dynamic Highlighting Command Menu Pane 1
  Dynamic Highlighting Command Menu Pane 2
Flavor Inspector 2
  Inspect Window 8
  Inspect Window 7
  Inspect Window With Timeout 4
  Inspector Interaction Pane 4
  Inspector Menu Pane 4
  Inspect History Window 4
Lisp Listener 1
Suggestions Off Frame 1
  Suggestions Off Pane 1
Znacs Frame 1
  Mode Line Window 3
  Typein Window 4
  Zwei Mini Buffer 6
  Znacs Window Pane 1
Inspect Frame 2
  Inspector Menu Pane 3
  Inspect Window 6
  Inspect Window 5
  Inspect Window With Timeout 3
  Inspect History Window 3
  Inspector Interaction Pane 3
Vt100 Frame 1
  Vt100 Screen 1
  Led 1
  Vt100 Telnet Menu 1
Glossary Frame 1
  Gloss Command Menu Pane 1
  Kbd Pane 1
  Glossary Text Pane 1
  Entry Menu Pane 1
  Alph Menu Pane 1
Profile Frame 1
  Profile Cuv Pane 1
  Profile Menu Pane 2
  Profile Menu Pane 1
Traveling Search Window 7
Fake Minibuffer Window 7
Traveling Search Window 6
Fake Minibuffer Window 6
Traveling Search Window 5
Fake Minibuffer Window 5
Traveling Search Window 4
Fake Minibuffer Window 4
Traveling Search Window 3
Fake Minibuffer Window 3
Pop Up Keystrokes Window 1
Traveling Search Window 2
Window Hierarchy
Modes
Window
  Processes
  Servers
  Counters
  Commands
  FILE Status
  Network
  Function Histogram
  Documentation Host Status
  Set Timeout Exit
Display the hierarchy of window inferiors, saying which are exposed.
Key assignments: w
  
```

Servers 17.3.6 Selecting the Servers item from the Modes menu window displays a list of all the servers running on this host that are serving other hosts. Some servers that might be seen are File Servers, Telnet Servers, Eval Servers, and Namespace Servers. The display includes the server's contact name, host, process, and connection. Refer to the *Explorer Networking Reference* for more information on these objects.

Clicking on a host, process, or connection invokes a menu of operations to perform on that object. The menu contains appropriate operations depending on which type of object you select. You can perform the following operations on a host:

- **Reset** — Resets all connections associated with this host.
- **Host Status** — Performs the equivalent of `(net:host-status)` to show the network status of all hosts on the display.
- **Describe** — Describes the selected host; equivalent to `(describe host-object)`.
- **Inspect** — Invokes the Inspector and inspects this host; equivalent to `(inspect host-object)`.

Refer to paragraph 17.3.1, Processes, for information on the operations you can perform on a process.

You can perform the following operations on a connection:

- **Describe** — Describes the selected server object; equivalent to `(describe server-object)`.
- **Inspect** — Invokes the Inspector and inspects this server object; equivalent to `(inspect server-object)`.

Figure 17-7 Example of Peek Servers Screen

Active Servers Contact Name	Host	Process / State Connection
FILE	ICARUS.ti-7	#<SYS:PROCESS File Server 25121614> Chaosnet Input #<CHAROS Connection 6640220>
User: PRINTER	Server Tag: 02321	
File Server Data 01582	Data Conn Cmd, sibling 11581, OUTPUT, cmd: (Idle)	
File Server Data 11581	Data Conn Cmd, sibling 01582, INPUT, cmd: (Idle)	
File Server Data 01576	Data Conn Cmd, sibling 11575, OUTPUT, cmd: (Idle)	
File Server Data 11575	Data Conn Cmd, sibling 01576, INPUT, cmd: (Idle)	
TELNET	NIL	#<SYS:PROCESS Telnet Server 25123236> TCP Input #<NET::GENERIC-PEEK-BS-SERVER 55361034>
FILE	Lina.ti-7	#<SYS:PROCESS File Server 6757735> Chaosnet Input #<CHAROS Connection 6640060>
User: Mailer	Server Tag: 02296	
FILE	Lina.ti-7	#<SYS:PROCESS File Server 5143635> Chaosnet Input #<CHAROS Connection 1015730>
User: Mailer	Server Tag: 02275	

Active Servers			
Modes	Processes	Network	Commands
Windows	FILE Status	Counters	Documentation Host Status
Areas	Function Histogram	Set Timeout	Exit
R2: System Menu			

Network 17.3.7 When you select the Network item from the Modes menu window, a menu of protocols appears. Several protocols may be listed depending on the software on your system. This paragraph only discusses the Chaos and Ethernet protocols. If you have purchased TCP/IP, for example, the protocols TCP, UDP, IP, and ICMP will also be listed. For information on using Peek on other protocols, refer to the appropriate related manual.

After selecting the Chaos protocol, Peek describes the state of the connection, contact name, host, local and foreign windows, packets received and sent, and any interesting meters. The packets received and sent are both mouse-sensitive, and clicking on them causes Peek to show all of the packets on the receive/send list. The Network item displays only those hosts already contacted since booting.

The connection name and host are also mouse-sensitive items. On host items, you can perform the following operations:

- **Reset** — Resets all connections associated with this host.
- **Host Status One** — Performs the equivalent of `(net:host-status 'host)` to show the network status of the selected host.
- **Insert Host Status** — Inserts a static Host Status for the selected host into the display.
- **Remove Host Status** — Removes the static Host Status for the selected host.
- **Describe** — Describes the selected host; equivalent to `(describe host-object)`.
- **Inspect** — Invokes the Inspector and inspects this process; equivalent to `(inspect process)`.

You can perform the following operations on a connection name.

- **Close** — Closes this connection.
- **Probe** — Probes this connection.
- **Status** — Sends Host Status on this connection.
- **Retransmit** — Retransmits any packets still pending on this connection.
- **Send LOS** — Sends an LOS packet to close the other side's connection.
- **Remove** — Arbitrarily removes this connection as an active connection.
- **Describe** — Describes this connection.
- **Inspect** — Inspects this connection.

If you select the Ethernet protocol, Peek provides information on Ethernet meters. For more information on Ethernet and Chaos, refer to the *Explorer Networking Reference*.

Figure 17-8 Example of Peek Network Screen

```

Chaosnet connections at 05/06/87 13:04:00
Connection to MSG-DISPATCHER at host ARMSTRONG (4005),
OPEN-STATE, local idx 33444, foreign idx 100562
Windows: local 13, foreign 13, (13 available)
Received: pkt 4 (time 27773670), read pkt 4, ack pkt 4, 0 queued
Sent: pkt 3, ack for pkt 3, 0 queued

Connection to 01066 from host dsg (2703),
OPEN-STATE, local idx 31434, foreign idx 100214
Windows: local 13, foreign 13, (13 available)
Received: pkt 0 (time 27772216), read pkt 0, ack pkt 0, 0 queued
Sent: pkt 1, ack for pkt 1, 0 queued

Connection to FILE 1 at host dsg (2703),
OPEN-STATE, local idx 31233, foreign idx 100411
Windows: local 5, foreign 13, (13 available)
Received: pkt 25 (time 27771732), read pkt 25, ack pkt 25, 0 queued
Sent: pkt 24, ack for pkt 24, 0 queued

Connection to 00777 at host BAZ (1506),
OPEN-STATE, local idx 30430, foreign idx 62445
Windows: local 13, foreign 13, (13 available)
Received: pkt 10 (time 27775230), read pkt 10, ack pkt 10, 0 queued
Sent: pkt 0, ack for pkt 0, 0 queued

Connection to FILE from host BAZ (1506),
OPEN-STATE, local idx 30026, foreign idx 62244
Windows: local 13, foreign 5, (5 available)
Received: pkt 4 (time 27766175), read pkt 4, ack pkt 4, 0 queued
Sent: pkt 5, ack for pkt 5, 0 queued

Connection to MSG-DISPATCHER at host ARMSTRONG (4005),
CLS-RECEIVED-STATE, local idx 27022, foreign idx 176624
Windows: local 13, foreign 13, (13 available)
Received: pkt 10 (time 24543071), read pkt 10, ack pkt 10, 0 queued
Sent: pkt 3, ack for pkt 3, 0 queued

Connection to 00987 from host dsg (2703),
OPEN-STATE, local idx 26016, foreign idx 133376
Windows: local 13, foreign 13, (13 available)
Received: pkt 3 (time 27766577), read pkt 3, ack pkt 3, 0 queued
Sent: pkt 4, ack for pkt 4, 0 queued

Connection to FILE 1 at host dsg (2703),
CLS-RECEIVED-STATE, local idx 25615, foreign idx 133175
Windows: local 5, foreign 13, (13 available)
Received: pkt 23 (time 25471135), read pkt 0, ack pkt 23, 19 queued
Sent: pkt 22, ack for pkt 22, 0 queued

Type Slot Subnet #-In #-Out Aborted Lost FCS-Error Timeout Too-Big
NUBUS F0 0 73962 3733 0 23 0 0
Network

Modes
Windows Processes Servers Counters Commands
Areas FILE Status [F] [S] Function Histogram Documentation Host Status
Set Timeout Exit

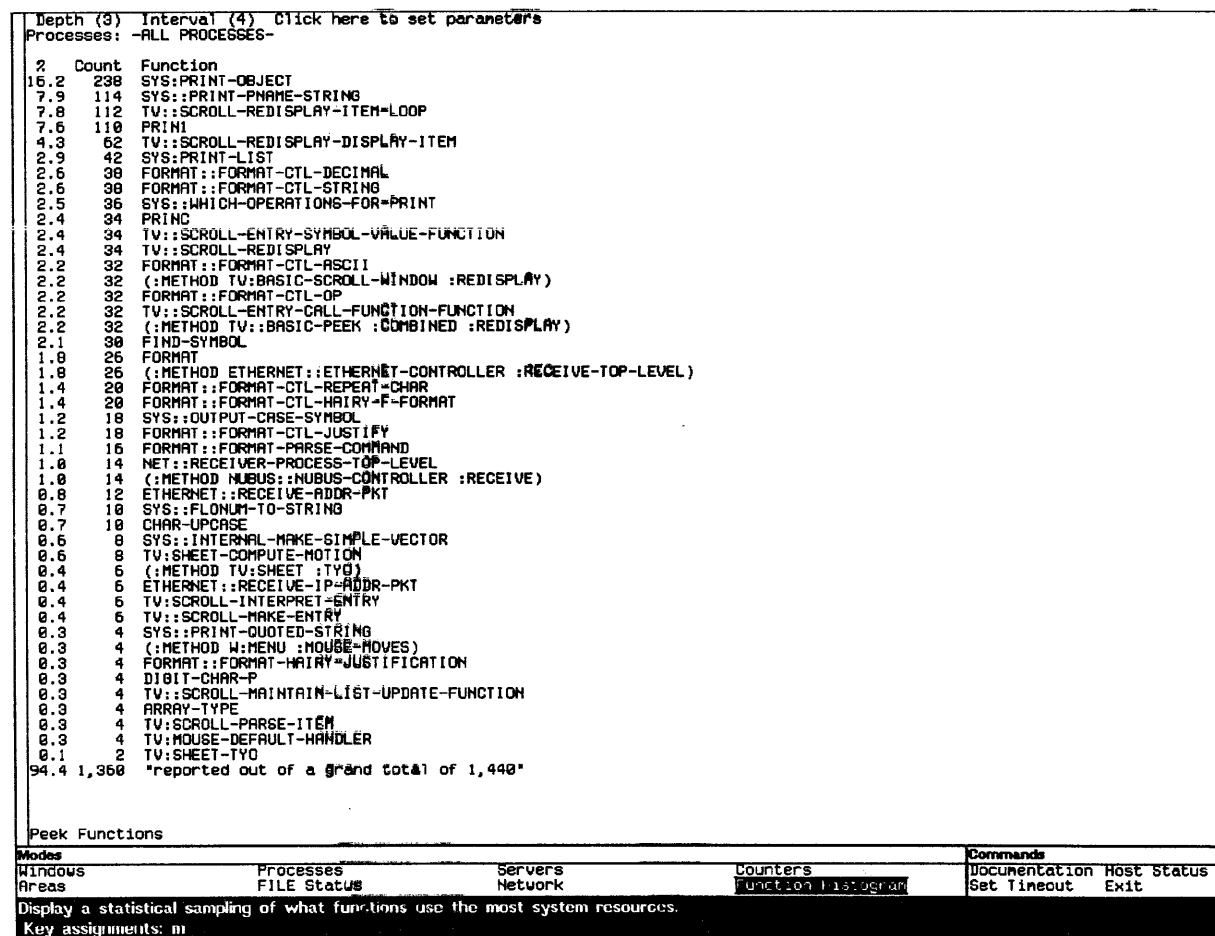
Display a statistical sampling of what functions use the most system resources.
Key assignments: m
    
```

Function Histogram 17.3.8 When you select the Function Histogram item from the Modes menu window, Peek provides a continuously-updating statistical sampling of which functions use the most system resources. Using the mouse to select options, you can control which processes are sampled, how many functions are recorded, how many are displayed, and the sampling frequency.

You can also select the Function Histogram mode by pressing M, which stands for metering. (The F keystroke is for the File Status mode.)

Every time the scheduler runs, a count is incremented (in a hash table) for the top *n* functions on the stack for the process that just ran, where *n* defaults to 3. The scheduler normally runs once every 67 tv-clock interrupts (about once a second). The scheduler is speeded up to once every 4 interrupts (about 16 times a second) to gather more data.

Figure 17-9 Example of Peek Function Histogram Screen



If you click on [click here to set parameters](#), the Function Histogram Parameters menu appears. The following list explains the prompts:

- Histogram — Allows you to reset, stop, or continue the histogram:
 - Reset — Clears the histogram data that has been collected so far and reactivates the sampling process for all processes by default. If the functions display does not continue showing the updated histogram, click on the Function Histogram item from the Modes menu window again to restart the display.
 - Stop — Stops collecting the histogram data.
 - Continue — Continues collecting the histogram data.

NOTE: If you do not specify Stop before you exit Peek, the function histogram will still be running.

- Processes — Allows you to select which processes to sample. The default is `-ALL PROCESSES-`.
- Histogram Depth — The number of stack frames to go down. In other words, if the value is n , the count is incremented for the top n functions on the stack when the sample is taken. The default is 3.
- Histogram Interval — The time between samples in 60ths of a second. The default is 4.
- Entries Displayed — The number of functions to display. The default is 45.

Host Status 17.3.9 When you select the Host Status item from the Commands menu window, Peek calls the `net:host-status` function, which then displays the status of each host contacted since booting. Refer to Section 33, Miscellaneous Debugging Functions, for details on `net:host-status`.

Figure 17-10 Example of Peek Host Status Screen

ADDR	HOST	STATUS
2053	"ARMSTRONG.ti-7"	is NOT responding on medium CHAOS.
1107337736	"ARMSTRONG.ti-7"	is responding on medium IP.
2053	"ARMSTRONG.ti-dallas"	is responding on medium CHAOS.
1107337736	"ARMSTRONG.ti-dallas"	is responding on medium IP.
1397	"AUSOME.ti-7"	is responding on medium CHAOS.
1090550437	"AUSOME.ti-7"	is NOT responding on medium IP.
838	"BAZ.ti-7"	is NOT responding on medium CHAOS.
1090559878	"BAZ.ti-7"	is responding on medium IP.
838	"BAZ.ti-dallas"	is responding on medium CHAOS.
1090559878	"BAZ.ti-dallas"	is responding on medium IP.
1464	"Brigham-Young.ti-7"	is responding on medium CHAOS.
1090550504	"Brigham-Young.ti-7"	is responding on medium IP.
842	"DLS.ti-7"	is responding on medium CHAOS.
1090559882	"DLS.ti-7"	is responding on medium IP.
842	"DLS.ti-dallas"	is responding on medium CHAOS.
1090559882	"DLS.ti-dallas"	is responding on medium IP.
1475	"dsg.ti-7"	is responding on medium CHAOS.
1090560515	"dsg.ti-7"	is responding on medium IP.
1090560048	"HOME.ti-7"	is responding on medium IP.
1353	"ICARUS.ti-7"	is NOT responding on medium CHAOS.
1090560393	"ICARUS.ti-7"	is NOT responding on medium IP.
1090519141	"Janus.ti-7"	is responding on medium IP.
1124073473	"jd-young.ti-7"	is NOT responding on medium IP.
1395	"L-YOUNG.ti-7"	is responding on medium CHAOS.
1090560435	"L-YOUNG.ti-7"	is responding on medium IP.
1477	"Lima.ti-7"	is NOT responding on medium CHAOS.
1090560517	"Lima.ti-7"	is responding on medium IP.
1394	"MR-X.ti-7"	is responding on medium CHAOS.
1090560434	"MR-X.ti-7"	is responding on medium IP.

Peek Processes

Windows	Areas	Commands
FILE Status	Servers Network	Documentation Host Status Set Timeout EXIT

Print Network statistics of all known hosts.
Key assignments: h

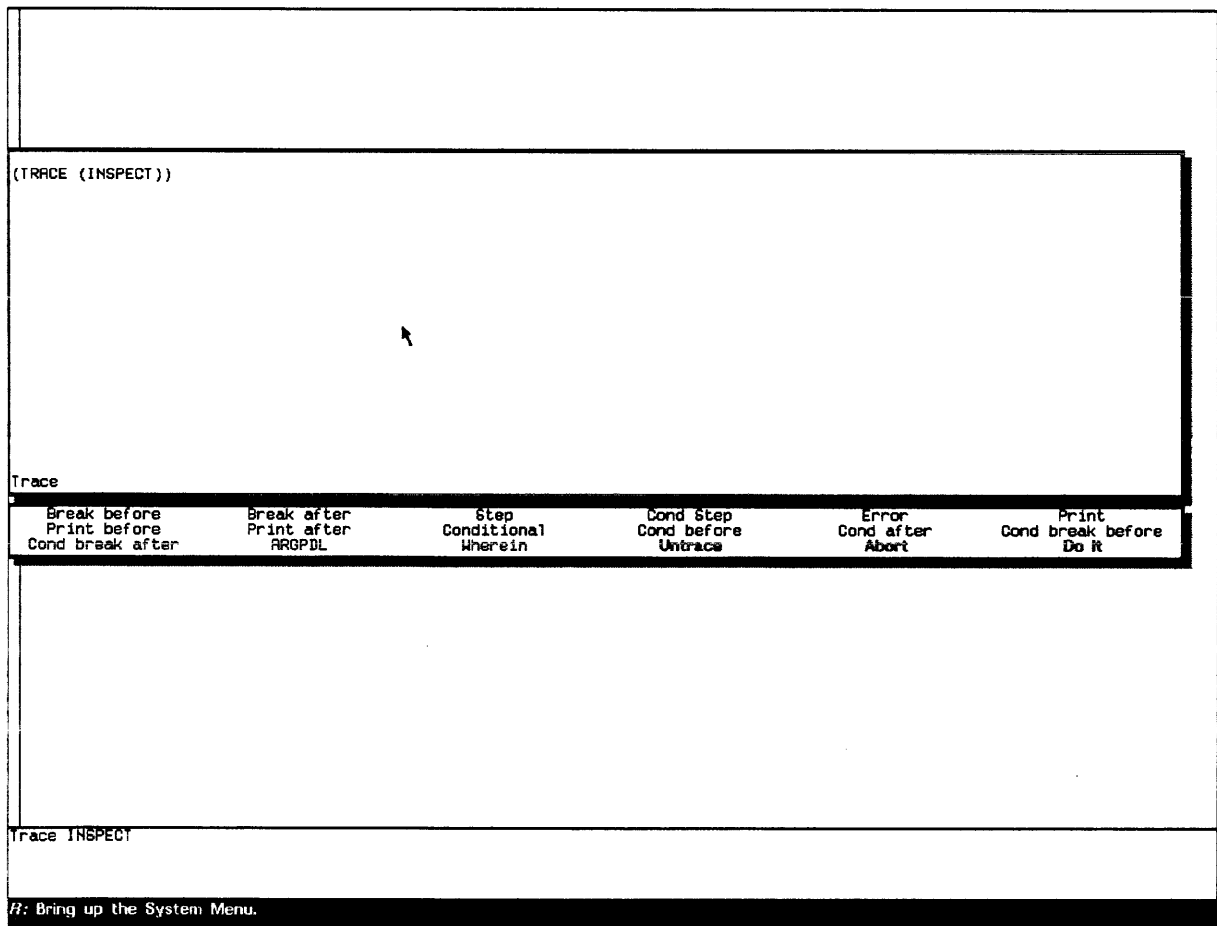
18

TRACE

The trace facility allows the user to *trace* certain functions (and macros, but not special forms). When a function is traced, certain special actions are taken when it is called and when it returns. The default tracing action is to print a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and value(s).

You can invoke the trace facility through the `trace` and `untrace` special forms, whose syntax is described in the following paragraphs. Alternatively, you can use the trace system by clicking on Trace in the System menu, clicking on the Trace option in the Flavor Inspector, or by entering the META-X Trace command in Zmacs. This procedure allows you to select the trace options from a menu instead of having to remember the syntax. An example of the syntax follows Figure 18-1. Figure 18-1 shows how the trace menu can appear.

Figure 18-1 Trace Menu



trace spec-1 spec-2 . . .

Macro

Each *spec* of the **trace** form can take any of the following forms:

- Arguments:**
- a symbol* — This is a function name, with no options. The function is traced in the default way, printing a message each time it is called and each time it returns. You can also trace macros, but not special forms.
 - a list* — (**function-name option-1 option-2 ...**) The *function-name* is a symbol and the *options* control how it is to be traced. The various options are listed below. Some options take arguments, which should be given immediately following the option name.
 - a list* — (**:function function-spec option-1 option-2 ...**) This is like the previous form except that *function-spec* need not be a symbol (see *function-spec* in the *Explorer Lisp Reference*). This form exists because if *function-name* were a list in the previous form, it would instead be interpreted as the following form.
 - a list* — (**((function-1 function-2 ...) option-1 option-2 ...)**) All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

You can use the following **trace** options:

- Options:**
- :break predicate** — This option causes a breakpoint to be entered after printing the entry trace information but before applying the traced function to its arguments, if and only if *predicate* evaluates to non-**nil**. During the breakpoint, the **arglist** symbol is bound to a list of the arguments of the function.
 - :exitbreak predicate** — This option is exactly like **break** except that the breakpoint is entered after the function has been executed and the exit trace information has been printed but before control returns. During the breakpoint, the **arglist** symbol is bound to a list of the arguments of the function, and the **values** symbol is bound to a list of the values that the function is returning.
 - :error** — This option causes the debugger to be invoked when the function is entered. Use **RESUME** to continue execution of the function. If this option is specified, no trace output is printed, other than the error message printed by the error handler. This option is obsolete because **breakon** is more convenient.
 - :step** — This option causes a function to be single-stepped whenever that function is called. For more information, refer to Section 19, Stepper.
 - :stepcond predicate** — This option causes the function to be single-stepped only if *predicate* evaluates to non-**nil**.
 - :entrycond predicate** — This option causes trace information to be printed on function entry only if *predicate* evaluates to non-**nil**.
 - :exitcond predicate** — This option causes trace information to be printed on function exit only if *predicate* evaluates to non-**nil**.
 - :cond predicate** — This option specifies both **:exitcond** and **:entrycond** together.

:wherein *function* — This option causes the function to be traced only when called, directly or indirectly, from the specified function. You can give several trace specifications to **trace**, all specifying the same function but with different **wherein** options, so that the function is traced in different ways when called from different functions.

This option is different from the form **advise-within**, which only affects the function being advised when it is called directly from the other function. The **trace :wherein** option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, and so on.

:argpdl *pdl* — This option specifies *pdl*, a symbol for a push-down list, whose value is initially set to **nil** by **trace**. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments is consed onto *pdl* when the function is entered and removed (via **cdr**) when the function is exited.

The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own PDL, or one PDL may serve several functions.

:entryprint *form* — The *form* is evaluated, and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values appear preceded by ****.

:exitprint *form* — The *form* is evaluated, and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values appear preceded by ****.

:print *form* — The *form* is evaluated, and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values appear preceded by ****.

:entry *list* — This option specifies a list of arbitrary forms whose values are to be printed along with the usual entry trace. The list of resulting values, when printed, is preceded by **** to separate it from the other information.

:exit *list* — This option is similar to **:entry** but specifies expressions whose values are printed with the exit trace. Again, the list of values printed is preceded by ****.

:arg :value :both nil — These options specify which of the usual trace print-outs should be enabled. If **:arg** is specified, then on function entry, the name of the function and the values of its arguments are printed. If **:value** is specified, then on function exit, the returned value(s) of the function are printed. If **:both** is specified, both options are printed. If **nil** is specified, neither is printed. If none of these four options is specified, the default is to **:both**. If any further *options* appear after one of these, they are not treated as options. Rather, they are considered to be arbitrary forms whose values are to be printed on entry to and/or exit from the function, along with the normal trace information. The values printed are preceded by a **//** and follow any values specified by **:entry** or **:exit**. Note that because these options *swallow* all following options, any one of these options, if one is given, should be the last option specified.

Variables You Can Use With the Options:

arglist variable — If the **arglist** variable is used in any of the expressions given for the **:cond**, **:break**, **:entry**, or **:exit** options (or after the **:arg**, **:value**, **:both**, or **nil** options) when these expressions are evaluated, the value of **arglist** is bound to a list of the arguments given to the traced function. Therefore, the following example causes a break in **:foo** if, and only if, the first argument to **foo** is **:nil**.

```
(trace (foo :break (null (car arglist))))
```

If the **:break** or **:error** option is used, the **arglist** variable is valid inside the break loop. If you execute **setq** on **arglist**, the arguments seen by the function change. The **arglist** variable should perhaps have a colon, but this colon can be omitted because this is the name of a system function and is therefore global.

values variable — Similarly, the **values** variable is a list of the resulting values of the traced function. For obvious reasons, this variable should be used only with the **:exit** option. If the **:exitbreak** option is used, the **values** and **arglist** variables are valid inside the break loop. If you execute **setq** on **values**, the values returned by the function will change. The **values** variable should perhaps have a colon, but this colon can be omitted because the symbol **values** is the name of a system function and is therefore global.

The trace specifications can be *factored*, as explained above. For example:

```
(trace ((foo bar) :break (bad-p arglist) :value))
```

The preceding example is equivalent to the following:

```
(trace (foo :break (bad-p arglist) :value)
       (bar :break (bad-p arglist) :value))
```

Because a list as a function name is interpreted as a list of functions, nonatomic function names (see the *Explorer Lisp Reference*) are specified as follows:

```
(trace (:function (:method flavor :message) :break t))
```

The **trace** macro returns as its value a list of names of all functions it traced. If called with no arguments, simply as **(trace)**, it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, **trace** calls **untrace** before setting up the new trace.

Tracing is implemented with encapsulation (see the *Explorer Lisp Reference*), so if the function is redefined (that is, with **defun** or by loading it from an object file) the tracing is transferred from the old definition to the new definition.

Examples: The following example shows how to define a conditional step. First, define a predicate that returns `t` if the first value of `arglist` is an instance. (A predicate is simply a function that returns `t` or `nil` based on some criteria.) This predicate will be used to test the arguments passed into the traced function (although you could use it to test any criteria you want):

```
(defun bad-p (arglist)
  (instancep (first arglist)))
```

Next, define a test function `foo` to take an optional set of arguments and return a list of them:

```
(defun foo (&optional a b c &rest d)
  (list a b c d))
```

Now, turn on trace for `foo`, and conditionally step `foo` if the function `bad-p` returns a non-`nil` value. The `arglist` argument will be bound to a list of all the arguments specified for the function `foo`:

```
(trace (foo :stepcond (bad-p arglist)))
```

Calling `foo` with an argument of 10 does not invoke the Stepper because `foo`'s `arglist` of 10 causes `bad-p` to return `nil`:

```
(foo 10)
(1 ENTER FOO: 10)
(1 EXIT FOO: (10 NIL NIL NIL))
(10 NIL NIL NIL)
```

In the next case, again `foo` is not stepped because `foo`'s `arglist` of (1 2 3) causes `bad-p` to return `nil`:

```
(foo 1 2 3)
(1 ENTER FOO: 1 2 3)
(1 EXIT FOO: (1 2 3 NIL))
(1 2 3 NIL)
```

Calling `foo` with an instance for the first argument invokes the Stepper because `bad-p` returns `t`:

```
(foo w:selected-window)
(1 ENTER FOO: #<ZMACS-WINDOW-PANE Zmacs Window Pane 1 4740102 exposed>)
←(BLOCK FOO (LIST A B C D))
←(LIST A B C D)
  ← A → #<ZMACS-WINDOW-PANE Zmacs Window Pane 1 4740102 exposed>
  ← B → NIL
  ← C → NIL
  ← D → NIL
←(LIST A B C D) → (#<ZMACS-WINDOW-PANE Zmacs Window Pane 1 4740102
  exposed> NIL NIL NIL)
←(BLOCK FOO (LIST A B C D)) → (#<ZMACS-WINDOW-PANE Zmacs Window
  Pane 1 4740102 exposed> NIL NIL NIL)
(1 EXIT FOO: (#<ZMACS-WINDOW-PANE Zmacs Window Pane 1 4740102 exposed>
  NIL NIL NIL))
(#<ZMACS-WINDOW-PANE Zmacs Window Pane 1 4740102 exposed> NIL NIL NIL)
```

trace-output

Variable

Tracing output is printed to the stream that is the value of ***trace-output***. The default value of this variable is ***terminal-io***.

untrace

Macro

The **untrace** macro is used to cancel the effects of **trace** and restore functions to their normal, untraced state. The **untrace** macro takes multiple specifications, that is, (**untrace** *foo quux fuphoo*). Calling **untrace** with no arguments untraces all functions currently being traced.

trace-compile-flag

Variable

If the value of **trace-compile-flag** is non-**nil**, the functions created by **trace** are compiled, allowing you to trace special forms such as **cond** without interfering with the execution of the tracing functions. The default value of this flag is **nil**.

You can also cause the tracing of a particular function to be compiled by calling **compile-encapsulations**. Setting **compile-encapsulations-flag** to non-**nil** does what **trace-compile-flag** does and causes all kinds of encapsulations to be compiled. For more information about the **compile-encapsulations** function, see the Compiler Operations section in the *Explorer Lisp Reference*.

The Stepper facility allows you to follow every step of the evaluation of a form and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. If your program is behaving strangely, then the Stepper can help you determine the cause of the problem.

There are two ways to enter the Stepper. One is by use of the `step` macro.

`step form`

Macro

This macro evaluates *form* with single-stepping. It returns the value of *form*. For example, if you have a function named `foo` with typical arguments such as `t` and `3`, you type the following:

```
(step (foo t 3))
```

The form `(foo t 3)` is then evaluated with single-stepping.

The other way to get into the Stepper is to use the `:step` option of `trace` (refer to Section 18, Trace). If a function is traced with the `:step` option, then whenever this function is called, it is single-stepped.

Note that any function to be stepped must be interpreted; that is, it must be a lambda expression. Compiled code cannot be stepped by the Stepper.

The Stepper uses the following symbols:

Symbol	Description
--------	-------------

- | | |
|---|---|
| → | When evaluation is proceeding with single-stepping, before any form is evaluated, the form is (partially) printed, preceded by a forward arrow (→) character. |
| ↔ | When a macro is expanded, the expansion is printed out using a double arrow (↔) character. |
| ← | When a form returns a value, the form and the values are printed out preceded by a backward arrow (←) character. |
| ∧ | When a form returns more than one value, an and-sign (∧) character is printed between the values. |
| λ | When the Stepper has evaluated the arguments to a form and is about to apply the function, it prints a lambda symbol (λ) because entering the lambda form is the next step in evaluating the function. |

Since the forms being stepped may be very long, the Stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, the Stepper indents each form proportionally to its level of recursion.

After the Stepper prints anything associated with these forms, it waits for a command from the user. Several commands tell the Stepper how to proceed or to look at what is happening. The commands are described in Table 19-1. Also note that Suggestions menus are available for the Stepper commands.

The Stepper also provides an automatic stepping feature, which is controlled by the following variable and functions:

sys:*step-auto* Variable

When this variable is **nil** (the default), automatic stepping is disabled. Stepping is then controlled by the keystroke commands.

When this variable is **t**, the Stepper continuously single steps through the function until completion, without waiting for a Stepper command. This feature is useful when you want to turn on dribble and capture all the step information in a file to be reviewed at a later time.

When this variable is **:no-print**, the Stepper automatically steps through the function but does not print anything.

sys:step-auto-on Function

This function turns on the automatic stepping feature (with printing).

sys:step-auto-off Function

This function turns off automatic stepping.

Table 19-1 Stepper Commands

Key Sequence	Explanation of Command
CTRL-N (Next)	Steps to the item to be evaluated. The Stepper continues until the next item to print out, and it accepts another command.
SPACE	Advances to the next item at this level. In other words, this command continues to evaluate at this level but does not step anything at lower levels. This is a good way to skip over parts of the evaluation that do not interest you.
CTRL-A (Args)	Skips over the evaluation of the arguments of this form but pauses in the Stepper before calling the function that is the car of the form.
CTRL-U (Up)	Continues evaluating until the Stepper goes up one level. This resembles the space command, except that it skips over anything on the current level as well as on lower levels.
END (exit)	Exits; finishes evaluating without any more stepping.
CTRL-T (Type)	Retypes the current form in full (without truncation).
CTRL-G (Grind)	Grinds (that is, pretty-prints) the current form.
CTRL-E (Editor)	Switches windows to the editor.
CTRL-B (Breakpoint)	<p>Activates a breakpoint (that is, a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:</p> <ul style="list-style-type: none"> ■ *step-form* — The current form ■ *step-values* — The list of returned values ■ *step-value* — The first returned value <p>If you change the values of these variables, you affect execution.</p>
CTRL-L	Clears the screen and redisplay the last 10 pending forms (forms that are being evaluated).
META-L	Resembles CTRL-L but does not clear the screen.
META-CTRL-L	Resembles CTRL-L but redisplay all pending forms.
HELP	Pops up the Universal Command Loop (UCL) Help menu.

The evaluator evaluates a form recursively until it has evaluated all the necessary components of that form, and then it returns the result. You may want more control over this evaluation process. For example, you may want to use a different evaluator entirely, or more commonly, you may want an interface so that you can observe or modify the forms on which the evaluator is working. The Stepper is implemented using the evalhook feature.

evalhook

Variable

If this variable is bound to a function (or a symbol that has a function definition), the evaluator calls this function to perform the evaluation *instead* of calling the `eval` function to perform the evaluation. The function to which this variable is bound should be defined with two arguments: the first for the form and an optional second argument for the environment. (Refer to the `eval` function in the *Explorer Lisp Reference* for information on environments.) It is the responsibility of this function to handle the evaluation of the form. Typically, this function calls `evalhook` on the form after observing and/or modifying it.

applyhook

Variable

If this variable is bound to a function (or a symbol that has a function definition), the evaluator calls this function to perform the apply operation instead of calling the `apply` function. (Refer to the *Explorer Lisp Reference* for information on the `apply` function.) The function should be defined with three arguments: the first is the function to be applied, the second is the list of arguments to be applied, and the third is an optional argument for the environment. It is the responsibility of this function to handle the apply operation. Typically, this function calls `applyhook` on these arguments after observing and/or modifying them.

evalhook *form evalhook-function applyhook-function*
&optional *environment*

Function

This function allows easy use of the ***evalhook*** and ***applyhook*** features by binding these variables to hook functions and by calling `eval`.

applyhook *function args evalhook-function applyhook-function*
&optional *environment*

Function

This function is the standard interface for applying the function bound to the ***evalhook*** and ***applyhook*** variables and then applying the function to the *args*.

Examples:

In the following simple example, the function defines an ***evalhook*** function that prints the forms on which the evaluator is operating. Note that after printing the form, `catch-eval` recursively calls `evalhook` on this form to evaluate it, with our `evalhook` still in place. This makes `evalhook` perform all the evaluations that would occur without the hook.

```
(defun catch-eval (form &optional env)
  (format t "-% CATCH-EVAL for form = -s" form)
  (evalhook form `catch-eval nil env))
```

The following function defines an ***applyhook*** function that prints the arguments on which **apply** is about to operate. As in the preceding example, after printing this information, the function calls **applyhook** to actually perform the apply operation with the evalhook and applyhook still in place. This function is invoked if the evaluator sees an **apply**.

```
(defun catch-apply (fcn args &optional env)
  (format t "-% CATCH-APPLY function = -s-% -12Twith args = -s"
    fcn args)
  (applyhook fcn args 'catch-eval 'catch-apply env))
```

The following macro makes it more convenient to enter a test form:

```
(defmacro show-eval (form)
  `(evalhook ',form 'catch-eval 'catch-apply))
```

The following example shows the evalhook and applyhook features:

```
(show-eval (apply '+ 2 '(3 4 5)))
```

The following appears on the screen:

```
CATCH-APPLY function = #<DTP-FUNCTION FORMAT 2504374>
  with args = (T "-% CATCH-EVAL for form = -s" (QUOTE +))
CATCH-EVAL for form = (QUOTE +)
CATCH-APPLY function = #<DTP-FUNCTION EVALHOOK 2467873>
  with args = ((QUOTE +) CATCH-EVAL NIL (NIL NIL))
CATCH-APPLY function = #<DTP-FUNCTION FORMAT 2504374>
  with args = (T "-% CATCH-EVAL for form = -s" 2)
CATCH-EVAL for form = 2
CATCH-APPLY function = #<DTP-FUNCTION EVALHOOK 2467873>
  with args = (2 CATCH-EVAL NIL (NIL NIL))
CATCH-APPLY function = #<DTP-FUNCTION FORMAT 2504374>
  with args = (T "-% CATCH-EVAL for form = -s" (QUOTE (3 4 5)))
CATCH-EVAL for form = (QUOTE (3 4 5))
CATCH-APPLY function = #<DTP-FUNCTION EVALHOOK 2467873>
  with args = ((QUOTE (3 4 5)) CATCH-EVAL NIL (NIL NIL))
CATCH-EVAL for form = (BLOCK CATCH-APPLY (FORMAT T "-% CATCH-APPLY
  function = -s-% -12Twith args = -s" FCN ARGS)
  (APPLYHOOK FCN ARGS (QUOTE CATCH-EVAL) (QUOTE CATCH-APPLY) ENV))
CATCH-EVAL for form = (FORMAT T "-% CATCH-APPLY function = -s-%
  -12Twith args = -s" FCN ARGS)
CATCH-EVAL for form = T
CATCH-EVAL for form = "-% CATCH-APPLY function = -s-% -12Twith
  args = -s"
CATCH-EVAL for form = FCN
CATCH-EVAL for form = ARGS
CATCH-APPLY function = #<DTP-FUNCTION APPLY 2503323>
  with args = (+ 2 (3 4 5))
CATCH-EVAL for form = (APPLYHOOK FCN ARGS (QUOTE CATCH-EVAL)
  (QUOTE CATCH-APPLY) ENV)
CATCH-EVAL for form = FCN
CATCH-EVAL for form = ARGS
CATCH-EVAL for form = (QUOTE CATCH-EVAL)
CATCH-EVAL for form = (QUOTE CATCH-APPLY)
CATCH-EVAL for form = ENV
14
```

**Advising
a Function**

21.1 To advise a function is to tell it to do something in addition to its actual definition. Advising is performed by means of the `advise` function. This extra action is called a piece of advice, and it can be performed before, after, or around the definition itself. The advice and the definition are independent in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function. It gives you information about when and how the function is called and about when and with what value it returns. Advising is intended for semipermanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing a change to a function in a way that is easy to retract. In this case, you call `advise` from the console. It can also be used for customizing a function that is part of a program written by someone else. In this case, you include a call to `advise` in one of your source files or your login initialization file (see Section 3, Login Initialization File), rather than modifying the other person's source code.

Advising is implemented with encapsulation (refer to the *Explorer Lisp Reference*), so if the function is redefined (for example, with `defun` or by loading it from an object file), the advice will be transferred from the old definition to the new definition.

advise

Macro

A function is advised by the `advise` macro, which has the following syntax:

(advise function class name position form1 form2 ...)

None of the arguments are evaluated. The *function* argument is the function on which to put the advice. It is usually a symbol, but any function spec is allowed (see *function-spec* in the *Explorer Lisp Reference* manual). The *forms* are the advice; they are evaluated when the function is called. The *class* should be one of `:before`, `:after`, or `:around`; it indicates when to execute the advice (before, after, or around the execution of the definition of the function). The meaning of `:around` advice is explained in the following paragraphs.

The *name* argument is used to keep track of multiple pieces of advice on the same function. The *name* is an arbitrary symbol that is remembered as the name of this particular piece of advice. If you have no name in mind, use `nil`; in this case, the piece of advice is considered *anonymous*. A given function and class can have any number of pieces of anonymous advice. However, that function or that class can have only one piece of named advice for any one name. If you try to define a second piece of named advice with the same name, the first is replaced. Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program is usually named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it is not executed twice.

The *position* argument indicates where to put this piece of advice in relation to others of the same class already present on the same function. If *position* is `nil`, the new advice is placed in the default position: it usually is placed at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it belongs in the same place as the old piece of advice.

The *position* argument is a numerical index pointing to an existing piece of advice before which the new piece of advice is to be inserted. Zero places the new advice at the beginning; a very large number places it at the end. However, *position* can be the name of an existing piece of advice of the same class on the same function; the new advice is inserted before the one specified by *position*.

Example:

```
(advise factorial :before negative-arg-check
  nil (print "Entering factorial."))
```

This code modifies the factorial function so that it first prints "Entering Factorial." when it is called.

unadvise

Macro

The following code removes pieces of advice:

```
(unadvise function class position)
```

None of its arguments are evaluated. The *function* and *class* arguments have the same meaning as they do in the `advise` function. The *position* argument specifies which piece of advice to remove. It can be the numeric index (zero means the first one), or it can be the name of the piece of advice.

The `unadvise` macro can remove more than one piece of advice if some of its arguments are missing. If *position* is missing or is `nil`, then all advice of the specified class on the specified function is removed. If *class* is missing or is `nil`, then all advice on the specified function is removed. The following code removes all advice on all functions, because *function* is not specified.

```
(unadvise)
```

You can see what advice a function has by invoking the `pprint-def` function. This function pretty-prints the advice on the function as forms that are calls to `advise`. (These forms are in addition to the definition of the function.) For example, `(pprint-def factorial)` prints the following:

```
(ADVISE FACTORIAL :BEFORE NEGATIVE-ARG-CHECK 0 (PRINT "Entering
  factorial 1."))
(DEFUN FACTORIAL (N)
  ...)
```

To investigate the advice structure with a program, you must work with the encapsulation mechanism's primitives (see the *Explorer Lisp Reference*).

To cause the advice to be compiled, call `compile-encapsulations` or set `compile-encapsulations-flag` to `non-nil`. For information about the `compile-encapsulations` function, see the Compiler Operations section in the *Explorer Lisp Reference*.

`sys:advised-functions`

Variable

This variable is a list of all functions that have been advised.

Designing the Advice

21.2 For advice to interact usefully with the definition and intended purpose of the function, it must be able to interface to the data flow and to control flow through the function. The Explorer system provides conventions for doing this.

The list of the arguments to the function can be found in the `arglist` variable.

```
(advise factorial :before negative-arg-check
  nil (if (minusp (first arglist))
    (ferror nil "factorial of negative argument")))
```

Any `:before` advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, copy the whole list first with the following:

```
(setf arglist (copy-list arglist))
```

After the function's definition has been executed, the list of the values it returned can be found in the `values` variables. Any `:after` advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a `prog`, so any piece of advice can exit the entire function with `return`. The arguments of the `return` are returned as the values of the function. No further advice is executed. If a piece of `:before` advice performs this operation, then the function's definition is not even called.

:around Advice

21.3 A piece of **:before** or **:after** advice is executed entirely before or entirely after the definition of the function. In contrast, **:around** advice is wrapped around the definition; that is, the call to the original definition of the function is executed at a specified place inside the piece of **:around** advice. You specify where by putting the **:do-it** symbol in the desired place.

For example, `(+ 5 :do-it)` as a piece of **:around** advice adds 5 to the value returned by the function. This result could also be produced by the following code as **:after** advice:

```
(setq values (list (+ 5 (car values))))
```

When there is more than one piece of **:around** advice, the pieces are nested, and the outer piece of advice is called first. The second piece is substituted for **:do-it** in the first one. The third one is substituted for **:do-it** in the second one. The original definition is substituted for **:do-it** in the last piece of advice.

The **:around** advice can access **arglist**, but **values** is not set until the outermost **:around** advice returns. At that time, it is set to the value returned by the **:around** advice. It is reasonable for the advice to receive the values of the **:do-it**, (that is, with **multiple-value-list**) and process them before returning them (that is, with **values-list**).

The **:around** advice can **return** from the **prog** at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain **:do-it**. Containing two instances of **:do-it** may be useful under peculiar circumstances. If you are careless, the original definition may be called twice, but something like the following always works reasonably:

```
(if (foo)
    (+ 5 :do-it)
    (* 2 :do-it))
```

Advising One Function Within Another

21.4 It is possible to advise the function `foo` only for when it is called directly from a specific other function `bar`. You perform this procedure by advising the function specifier `(:within bar foo)`. This specifier works by finding all occurrences of `foo` in the definition of `bar` and replacing them with `altered-foo-within-bar`. This procedure works even if `bar`'s definition is compiled code. The symbol `altered-foo-within-bar` starts off with the symbol `foo` as its definition; then the symbol `altered-foo-within-bar`, rather than `foo` itself, is advised. The system remembers that `foo` has been replaced inside `bar`, so if you change the definition of `bar`, or advise it, then the replacement is propagated to the new definition or to the advice. If you remove all the advice on `(:within bar foo)` so that its definition becomes the symbol `foo` again, then the replacement is canceled and everything returns to its original state.

The `(pprint-def bar)` function prints `foo` where it originally appeared, rather than `altered-foo-within-bar` so the replacement is not seen. Instead, `pprint-def` prints out calls to `advise` to describe all the advice that has been put on `foo` or anything else within `bar`.

An alternate way of putting on this sort of advice is to use **advise-within**.

advise-within *within-function function-to-advise class name position* Macro
&body forms...

This macro advises *function-to-advise* only when called directly from the *within-function*.

The other arguments mean the same thing as with **advise**. None of them are evaluated.

To remove advice from (**:within** bar foo), you can use **unadvise** on that function specifier. Alternatively, you can use **unadvise-within**.

unadvise-within *within-function function-to-advise class position* Macro

This macro removes advice that has been placed on the following:

(**:within** *within-function function-to-advise*)

The *class* and *position* arguments are interpreted the same as for **unadvise**. For example, if these two arguments are omitted, then all advice placed on *function-to-advise* within *within-function* is removed. Additionally, if *function-to-advise* is omitted, all advice on any function within *within-function* is removed. If there are no arguments, then all advice on one function within another is removed. Other pieces of advice that have been placed on one function and not limited to **:within** another are not removed.

The (**unadvise**) macro removes absolutely all advice, including advice for one function within another.

BREAKON



The `breakon` function allows you to request that the debugger be entered whenever a certain function is called. When the function is called, you can evaluate lexically scoped variables and see who is calling the function and with what arguments.

`breakon` *function-spec* &optional *condition-form*

Function

This function encapsulates the definition of *function-spec* (see *function-spec* in the *Explorer Lisp Reference*) so that a trap-on-call occurs when it is called. The trap-on-call enters the debugger. A trap-on-exit occurs when the stack frame is exited.

If *condition-form* is non-`nil`, its value should be a form to be evaluated each time *function-spec* is called. The trap occurs only if *condition-form* evaluates to non-`nil`. Omitting the *condition-form* is equivalent to supplying `t`. If `breakon` is called more than once for the same *function-spec* with different *condition-forms*, the trap occurs if any one of the conditions is true.

Using `breakon` with *condition-form* is useful for causing the trap to occur only in a certain stack group. This situation sometimes allows debugging of call functions that are being used frequently in background processes:

```
(breakon 'foo `(eq current-stack-group 'current-stack-group))
```

If you wish to trap on calls to `foo` when called from the execution of `bar`, you can use `(sys:function-active-p 'bar)` as the condition. If you want to trap only calls made directly from `bar`, use the following rather than a conditional `breakon`:

```
(breakon '(:within bar foo))
```

Another useful form of conditional `breakon` allows you to control trapping from the keyboard:

```
(breakon 'foo '(w:key-state :mode-lock))
```

The trap occurs only when the `MODE-LOCK` key is depressed. (Normally, this key is used to make the arrow keys move the mouse cursor one pixel at a time.) With this technique, for example, you can successfully trap on functions used by the debugger.

`unbreakon` *function-spec* &optional *conditional-form*

Function

This function removes the `breakon` set on *function-spec*. If *conditional-form* is specified, this function removes only that condition. Breakons with other conditions are not removed.

With no arguments, `unbreakon` removes all breakons from all functions.

eh:*breakon-functions*

Variable

This variable represents a list of all function specs on which breakons currently exist.

To cause the encapsulation that implements the breakon to be compiled, call **compile-encapsulations** or set **compile-encapsulations-flag** to non-nil (see the Compiler Operations section in the *Explorer Lisp Reference*). This procedure may eliminate some of the problems that occur if you break on a function such as **prog** that is used by the evaluator. (A conditional to trap only in one stack group can also help in this situation.)

The name MAR, from the similar device on the ITS PDP™-10s, is an acronym for Memory Address Register. On the Explorer system, the MAR facility is used to signal a MAR break condition when a particular word or words in memory are referenced.

The MAR can be a useful debugging tool if you want to know when a particular memory location (a program variable, for example) is read or written. You can set the MAR to signal a MAR break condition when a particular process accesses the MAR locations, or when any program in the system accesses it. When a MAR break is signaled after the MAR is set, you can use the debugger to see the calling sequence that led to the memory reference, and you are provided with various options on how to proceed from the break.

The MAR checking is performed by the Explorer memory management hardware, so the speed of general execution is not significantly slowed down when the MAR is enabled. However, the speed of accessing pages of memory containing the locations being checked is slowed down somewhat, because every reference involves a microcode trap.

Two properties are associated with the MAR:

- **MAR locations** — These locations are the range of addresses for which the MAR is set. You specify the MAR locations as a start location and a number of words when the MAR is enabled with the **set-mar** function. Note that the MAR location settings are pervasive; that is, there is only one set of MAR locations active in the system at any given time. The virtual memory system detects an access to these MAR locations by any software in the system.
- **MAR mode** — This mode specifies the kind of access that triggers a MAR break. The following modes are possible:
 - **nil** — Access to MAR locations does nothing.
 - **:read** — Any read attempt to a MAR location produces a MAR break.
 - **:write** — Any write attempt to a MAR location produces a MAR break.
 - **t** — Any kind of access to a MAR location produces a MAR break.

Note that whereas the MAR locations are a system-wide setting, MAR mode is a property associated with each stack group in the system. By default, the MAR mode of all stack groups is **nil** (that is, access to any MAR location does nothing). When you enable the MAR using **set-mar**, by default you only specify the MAR mode for the current stack group.

Also, when a stack group is reset, its MAR mode is reset to the default of `nil`. As a result, the MAR mode of a stack group can be reset sometimes when you do not expect it to be. For example, suppose you set the MAR mode in the initial Lisp Listener and later have an error that invokes the debugger. When you abort from the error, the MAR mode is reset to `nil`. The system-wide MAR locations, however, are still active.

Furthermore, because the MAR locations are memory addresses, unpredictable behavior results if the objects at these addresses are moved by garbage collection (GC). For this reason, the GC process (if active) is arrested while the MAR is enabled for any stack group.

The following are the functions that control the MAR:

set-mar *location mar-mode* &optional (*n-words* 1) Function
(*stack-group* **current-stack-group**)

The **set-mar** function clears any previous *mar-mode* of *stack-group* and sets the MAR locations to *n-words* starting at *location*. This function also arrests the GC process if it is active.

location — The *location* argument can be any object. Often, it is a locative pointer to a cell, probably created with the **locf** special form.

n-words — The *n-words* argument defaults to 1.

mar-mode — The *mar-mode* argument indicates under which conditions to trap. The **:read** keyword means that reading the location should cause an error, **:write** means that writing the location should, and **t** means that both should.

stack-group — Specifies the stack group for which you want to enable MAR. The default is **current-stack-group**. To enable MAR for all stack groups, specify **:all**.

To set the MAR to detect a **setq** or binding of the variable *foo*, use the following:

```
(set-mar (value-cell-location 'foo) :write)
```

clear-mar &optional (*globally* `nil`) Function

This function turns off the MAR by clearing the MAR locations and setting the MAR mode to `nil`. By default, the MAR mode is reset for the current stack group only. If you specify a non-`nil` value for *globally*, the MAR mode is reset for all stack groups for which it is on. Warm booting the machine clears the MAR globally.

If the GC process was arrested by **set-mar**, the **clear-mar** function unarrests it as long as there are no more stack groups for which the MAR is set. Since **(clear-mar t)** clears all MAR settings, you should always use this form at the end of a MAR session to ensure that GC can be reactivated.

mar-mode &optional (*sg* **current-stack-group**) Function

The **mar-mode** function returns a symbol indicating the current state of the MAR for the stack group *sg*. The default value of *sg* is **current-stack-group**. It returns one of the following: `nil`, **:read**, **:write**, or **t**.

sys:mar-break *condition*

Condition

This is the condition, not an error, signaled by a MAR break.

The condition instance supports these methods:

- **:object** — The object, one of whose words was being referenced
- **:offset** — The offset within the object of the word being referenced
- **:value** — The value read or to be written
- **:direction** — Either **:read** or **:write**

The **:no-action** proceed type simply proceeds, continuing with the interrupted program as if the MAR had not been set. If the trap was due to writing, the **:proceed-no-write** proceed type is also provided and causes the program to proceed, but it does not store the value in the memory location.

Most, but not all, write operations first perform a read. The **setq** and **rplaca** functions are examples. Thus, if the MAR is in **:read** mode, it catches writes as well as reads; however, these functions trap during the reading phase, and consequently, the data to be written is not displayed. Also, setting the MAR to **t** mode causes most writes to trap twice, first for a read and then again for a write. So when the MAR indicates that it trapped because of a read, a read has been performed at the hardware level, which may not look like a read in your program.

Proceeding from a MAR break allows the memory reference that produced a condition to take place and continues the program with the MAR still effective. When proceeding from a write, you have the choice of whether to allow the write to take place or to inhibit it, leaving the location with its old contents.

Introduction

24.1 The Explorer crash reporting and analysis utilities allow you to analyze all system shutdowns, both normal and abnormal, by providing detailed descriptions of each shutdown. The following topics are discussed in this section:

- Crash reporting — Provides an overview of how crash reporting works.
- Preparing non-volatile RAM (NVRAM) — Talks about preparing NVRAM for the proper recording of crash information.
- Crash analyzer functions — Discusses the functions that allow you to analyze the shutdown records.
- Shutdown record analysis format — Describes the format of the crash analyzer's output.

NOTE: The remaining topics provide a complete list of crash descriptions currently in existence. These crash descriptions can be roughly divided into two categories: those that indicate the hardware may be faulty and those that indicate a probable problem with system software.

- Hardware crash descriptions and troubleshooting — Talks about hardware crash descriptions and how to troubleshoot them.
- The force crash keychord — Tells how to force a crash by using a keychord.
- Software crash descriptions — Discusses the crash descriptions that may indicate a problem with the system software. You should note that hardware problems can lead to erratic crashes in this software category also, but the pattern of crashes is not likely to be firm.

Crash Reporting

24.2 When the system detects an irrecoverable or “can't happen” error, it halts the processor after recording some information about the error in a crash table stored in NVRAM on the System Interface Board (SIB). After the next successful system startup, you can use the crash analyzer to display the information stored in the crash table.

The crash table area of NVRAM is partitioned into individual shutdown records, each one containing information about one boot session. The records for the last few system shutdowns are kept in a circular buffer so that an unsuccessful attempt to restart the system does not lose the original crash data. Currently, the crash table area has room for 12 shutdown records. Older records of abnormal shutdowns (crash records) are stored in the system log. The system log is discussed in the *Explorer Input/Output Reference*.

Each time the system is started, a record is allocated from the buffer and initialized with some information about the boot, such as the load device and system version. This allocation is performed by the microcode as early in the boot as possible so that useful information can be recorded if the machine crashes during the boot process itself. When the system halts, information about the reason for the halt is also stored in the shutdown record by the microcode.

Furthermore, the system boot process can determine if the last shutdown was abnormal (crash). If the shutdown was abnormal, the following message is provided at the end of the print herald and the shutdown is logged in the system log (if present):

```
"Last system shutdown was abnormal. To view crash record, use
(report-last-shutdown)."
```

Preparing NVRAM

24.3 For the proper recording of crash information to take place, you must first initialize the structure of NVRAM by using the `sys:setup-nvram` function. Only then can valid shutdown records be maintained. Since NVRAM is non-volatile memory, you should need to run the `sys:setup-nvram` function only when the system is first installed, after service maintenance has been performed on the SIB, or after the installation of new microcode that may require a different NVRAM format.

For more information on `sys:setup-nvram`, refer to the *Introduction to the Explorer System*.

Crash Analyzer Functions

24.4 The crash analyzer consists of a number of Lisp routines that report the contents of shutdown records to the user in a readable format. The two user-callable functions to invoke the crash analyzer are `report-last-shutdown` and `report-all-shutdowns`.

`report-last-shutdown &key :stream :pathname :abnormal-only` [c] Function

Reports the results of analyzing the shutdown record from the previous boot. If `:abnormal-only` is `t`, the shutdown record is reported only if it represents a crash (versus a normal Lisp shutdown or boot). The `:abnormal-only` keyword defaults to `nil`.

Usually, the analysis is written to the stream indicated by the `:stream` keyword (which can be `nil` to return a string). The `:stream` keyword defaults to `*standard-output*`. If `:pathname` is non-`nil`, however, the shutdown record is written to a file instead.

- If `:pathname` is `:default`, the default crash file pathname (the value of `sys:*default-crash-file-pathname*`) is used.
- Otherwise, `:pathname` must be parsable into a pathname, with `sys:*default-crash-file-pathname*` used as the default in the parsing.

Shutdown records written to a file are marked internally as logged (see `report-all-shutdowns`).

The `report-last-shutdown` function always reports some information about the last shutdown unless there is no shutdown record for that shutdown. In this case, `report-last-shutdown` just returns `nil`. Otherwise, the report for the shutdown can be one of the following:

- A normal tabular description of the information stored in the shutdown record.
- A warning message indicating that the processor type or shutdown record format revision stored in NVRAM does not match what the analyzer expects to see. This can indicate either that NVRAM has not been properly initialized (using `sys:setup-nvram`) or that it was initialized with an earlier shutdown record format.
- A message indicating that the shutdown record format does not look reasonable. This may indicate that NVRAM is not functioning properly or that its contents have somehow been overwritten or otherwise corrupted.

`report-all-shutdowns &key :stream :pathname :abnormal-only` `[c]` Function
`:unlogged-only`

Reports the results of analyzing all currently recorded shutdown records. If `:abnormal-only` is `t`, the shutdown record is reported only if it represents a crash (versus a normal Lisp shutdown or boot). The `:abnormal-only` keyword defaults to `nil`.

Usually, the analysis is written to the stream indicated by the `:stream` keyword (which can be `nil` to return a string). The `:stream` keyword defaults to `*standard-output*`. If `:pathname` is non-`nil`, however, the shutdown record is written to a file instead.

- If `:pathname` is `:default`, the default crash file pathname (the value of `sys:*default-crash-file-pathname*`) is used.
- Otherwise `:pathname` must be parsable into a pathname, with `sys:*default-crash-file-pathname*` used as the default in the parsing.

If `:pathname` is non-`nil` and `:unlogged-only` is `t`, only records that have not previously been logged are written to the log file. (Shutdown records are marked internally as logged after being written to a log file either by this function or by `report-last-shutdown`.)

The `report-all-shutdowns` function produces a display consisting of one of the following:

- A normal tabular listing of shutdown record information for each shutdown record currently in the crash table. Unused shutdown records are suppressed in this listing, as is the current shutdown record (the one for this boot). If a shutdown record does not look as if it has reasonable values in it, this fact is reported in the listing.
- A warning message indicating that the processor type or shutdown record format revision stored in NVRAM does not match what the analyzer expects to see. This can indicate either that NVRAM has not been properly initialized (using `sys:setup-nvram`) or that it was initialized with an earlier shutdown record format.

Shutdown Record Analysis Format

24.5 Figure 24-1 shows an example of the crash analyzer's output after the following is entered in a Lisp Listener:

```
(report-all-shutdowns)
```

Figure 24-1 Sample Output From report-all-shutdowns

SHUTDOWN RECORD AT OFFSET #x+600

```
Load Band:          LOD1, Version 3.0, on Unit 1 (currently called D1)
Microcode Band:     MCR3, Version 207., on Unit 1 (currently called D1)
Boot Type:          Warm Boot
Progress Into Boot: Time Initialized
Boot Time:          4/30/87 21:05
Shutdown Time:      5/01/87 11:17
Shutdown Reason:    System Boot
```

SHUTDOWN RECORD AT OFFSET #x+500

```
Load Band:          LOD1, Version 3.0, on Unit 4 (currently called D4)
Microcode Band:     MCR2, Version 207., on Unit 4 (currently called D4)
Progress Into Boot: Time Initialized
Boot Time:          4/30/87 01:58
Shutdown Time:      4/30/87 20:52
Shutdown Reason:    Microcode Halt
Ucode Halt Address: #x+1F52
Shutdown Description: Crash called by Force-Crash Keychord
FEF Running at Crash: #<DTP-FUNCTION MY-INFINITE-LOOP-WITHOUT-INTERRUPTS 24735324>
PC Within Function: 13.
Register Values:
M-1:      #x+6A      <DTP-TRAP #x+6A>
M-2:      #x+1D      <DTP-TRAP #x+1D>
M-T:      #x+D6A8B   <DTP-TRAP #x+D6A8B>
MD:       #x+FFFFFF05 <DTP-ONES-TRAP #x+1FFFFFF05 CDR-NEXT>
VMA:      #x+F5FC0000 <DTP-FEF-HEADER #x+1FC0000 CDR-NEXT>
UPC-1:    #x+380     <DTP-TRAP #x+380>
UPC-2:    #x+1F6     <DTP-TRAP #x+1F6>
M-FEF:    #x+1853BAD4 <DTP-FUNCTION #x+53BAD4>
LC:       #x+A775B6  <DTP-TRAP #x+A775B6>
```

SHUTDOWN RECORD AT OFFSET #x+400

```
Load Band:          LOD1, Version 1.10, on Unit 4 (currently called D4)
Microcode Band:     MCR2, Version 207., on Unit 4 (currently called D4)
Progress Into Boot: Time Initialized
Boot Time:          4/30/87 08:52
Shutdown Time:      4/30/87 12:52
Shutdown Reason:    Lisp Halt
Lisp Crash Code:    0.
Lisp Halt Reason:   Normal Shutdown By Shutdown
Lisp Object:        The symbol NIL
```

Figure 24-1 Sample Output From report-all-shutdowns (Continued)

SHUTDOWN RECORD AT OFFSET #x+F00

```

Load Band:                LOD9, Version 3.0, on Unit 1 (currently called D1)
Microcode Band:          MCR1, Version 207., on Unit 1 (currently called D1)
Progress Into Boot:      Time Initialized
Boot Time:               4/30/87  11:36
Shutdown Time:          4/30/87  00:26
Shutdown Reason:        Microcode Halt
Ucode Halt Address:     #x+3BD7
Shutdown Description:    Disk error on swap: device error:
                        Uncorrectable data error
                        Status Word = #x+80005200
                        Command Word: #x+1280001, Logical Unit: 1.
                        Block Address: 55931., Transfer Count: 2048.
FEF Running at Crash:   #<DTP-FUNCTION INTERN 2502360>
PC Within Function:     92.
Register Values:
M-1:                    #x+12800001 <DTP-INSTANCE #x+800001>
M-2:                    #x+800 <DTP-TRAP #x+800>
M-T:                    #x+DA7B <DTP-TRAP #x+DA7B>
MD:                    #x+60005200 <DTP-STACK-GROUP #x+5200 CDR-ERROR>
VMA:                    #x+A002897 <DTP-FIX #x+2897>
UPC-1:                 #x+3BB2 <DTP-TRAP #x+3BB2>
UPC-2:                 #x+3849 <DTP-TRAP #x+3849>
FEF:                   #x+180A84F0 <DTP-FUNCTION #x+A84F0>
LC:                    #x+150A3D <DTP-TRAP #x+150A3D>

```

All shutdown record reports include header information describing the load devices, boot time, progress, and shutdown reason.

The load information fields are initialized during shutdown record allocation by the microcode. They reflect the load information saved by the boot process.

If the `Boot Type: Warm Boot` item is present, the boot session represented in the record was initiated with a warm-boot keychord. This is important to note if the record also indicates that the shutdown was abnormal. Therefore, the record may only be indicating that a secondary crash occurred when warm booting from a primary crash. In this case, the shutdown record of interest is the one from the previous boot session showing the primary crash.

The `Progress Into Boot` item indicates how far into the boot process the system progressed before halting. Currently supported values for this field are as follows:

- `Initial-Value`: Shutdown record has not yet been allocated.
- `Allocated`: Shutdown record allocation by the microcode was successful.
- `Starting-Lisp`: The microcode transferred control to Lisp.
- `System-Initializations`: The *system* initialization list was being run.
- `cold-Initializations`: The *cold* initialization list was being run.

- **Warm-Initializations:** The *warm* initialization list was being run.
- **Time-Initialized:** The time base has been initialized, and the current time has been written to the shutdown record. If the reported value is *Time-Initialized*, the initial Lisp environment was probably reached before the halt occurred.

The *Boot Time* is written to the shutdown record by a function on the *warm* initialization list after the system time base has been initialized. The *shutdown time* is updated about once every five minutes from Lisp so that it reflects the approximate time when the machine halts.

The *Shutdown Reason* item indicates the type of the last shutdown. This field is initialized to the *system Boot* state by the shutdown record allocation microcode, then updated during the abnormal termination process (called *ILLOP*) to reflect the kind of crash. Possible values for the kind of halt are as follows:

- **System Boot.** The last shutdown was caused by a cold boot sequence or by a warm boot sequence during normal operation (that is, not a warm boot initiated after the machine crashed). Note that since the shutdown record allocation routine is called from the warm boot, the system can be warm booted after an abnormal shutdown and still retain all the shutdown record information from that shutdown. However, if the machine was in a hung state when warm or cold booted, *ILLOP* processing is not done, and the shutdown is reported in this category.
- **Lisp Halt.** The last halt was called by Lisp through the *sys:%crash* function. In this case, a Lisp crash code (which is one of the arguments to *sys:%crash*) is reported. The Lisp crash code of 0 indicates a normal system shutdown called from Lisp. This code is seen when the system was halted by a user-initiated call to either *sys:shutdown* or *sys:system-shutdown*. Other valid crash codes are contained in the variable *sys:lisp-crash-code-alist*. The second argument to *sys:%crash* (an object) is reported under the *Lisp object* item in the case of a Lisp halt. If this object is a symbol that exists in the current Lisp environment, its name is reported.
- **Microcode Halt.** The last crash was called from the microcode when it detected an unrecoverable error condition. In this case, *ILLOP* stores the micro-pc address from which *ILLOP* was called in the *Ucode Halt Address* field. Later, the crash analyzer looks up this micro-pc in the crash table database to provide the user with a text description of the microcode crash reason under the *Shutdown Description* item.

This crash table database consists of the table of crash codes and crash descriptions produced by the micro assembler. The table is kept in the file *SYS:UBIN;microcode-type.CRASH#nnn*, where *nnn* is the microcode revision number. The *microcode-type* is a microcode type name such as *EXP1-UCODE* or *EXP2-UCODE*. (The list of all valid microcode types can be found in *sys:*processor-ucode-name-alist**.) The crash analyzer loads the appropriate version of this file into memory when a microcode crash is being reported.

Note that a few microcode crashes do not have a description in the crash database. This fact is reported by the crash analyzer with a shutdown Reason Of Halt description not found in crash table. The crash micro-pc for such crashes is valid, however, and can be used to determine the path taken to ILLOP. See the following paragraphs for more information on interpreting crash descriptions.

The FEF Running at Crash item indicates the Lisp function running at the time the machine halted, if the shutdown was abnormal. You can use this information, along with the pc Within Function field and the disassembler, to narrow down the portion of code executing when the machine crashed. If the running function stored by the microcode does not exist in the current Lisp environment, this fact is indicated in the report.

NOTE: A small chance exists that the function information reported is not valid if the system version level (both major and minor) of the system booted does not match the level of the system the crash occurred under or if garbage collection (GC) has moved objects a great deal. This is quite rare, however, and the function information should generally be regarded as reliable. The crash analyzer warns you of this condition.

The Register Values item contains the values of the indicated processor registers when ILLOP was called. This information is generally useful only to Texas Instruments analysts investigating the crash.

Hardware Crash Descriptions and Troubleshooting

24.6 A complete list of crash descriptions currently in existence is detailed in the paragraphs that follow. These crash descriptions can be roughly divided into two categories: those that indicate the hardware may be faulty and those that indicate a possible problem with the system software or a corrupted data structure. This numbered paragraph lists the hardware crash descriptions, with notes on each crash category and some problem troubleshooting hints. The following topics are discussed:

- NuBus crashes
- Processor fault crashes
- Power fail crash
- Mass storage subsystem crashes
- Troubleshooting NUPI device and controller error crashes
- Troubleshooting NUPI special event crashes

After these hardware crash descriptions, paragraph 24.7 talks about forcing a crash using the META-CTRL-META-CTRL-C keychord. Then Table 24-3 in paragraph 24.8 lists all the remaining software crash descriptor texts. Remember that hardware problems can lead to erratic crashes in this software category also, but the pattern of crashes in this case is not likely to be firm.

NuBus Crashes 24.6.1 The NuBus error crash descriptions are as follows:

```
System bus error : PARITY limit exceeded
System bus error : Possible other memory parity
System bus error : GACBL limit exceeded
System bus error : Nubus error or timeout
System bus error : NO-MEM or WEIRD
NuBus error in NuBus error handler
```

Most often these NuBus error crash descriptions indicate a problem referencing an address on one of the boards on the NuBus (most frequently, memory boards). Note that these crashes correspond to the error messages Lisp delivers when a NuBus error is encountered during calls to the `sys:%nubus-read` or `sys:%nubus-write` functions. The difference here is that the error was encountered when the microcode was performing the memory reference. The microcode first retries the reference and then crashes only if the retries were unsuccessful.

The `PARITY limit exceeded` crash occurs when bad memory parity was found on the bus interface memory board (usually in slot 4), while `Possible other memory parity` occurs when a parity error is found on another board. Usually, this other board is the local memory board in slot 3, but it can also be from other boards (see next numbered paragraph). The `GACBL limit exceeded` crash occurs when the go-away-come-back-later wait limit has expired. The `Nubus error or timeout` error can occur when a nonexistent memory address is being referenced, while the `NO-MEM` or `WEIRD` error specifically indicates that a “hole” in a board’s physical memory was being referenced by microcode routines.

Each of these crash texts is followed by three lines of error location information of the form:

```
Physical Address: <hex address>
LVL1 Control:    <hex value>
LVL2 Control:    <hex value>
```

The physical address given can often be quite useful in diagnosing the faulty hardware, since the second digit of the hex address is the board’s NuBus slot number. For example, the address `#x+F31C500` is found on the board in slot 3 (usually a memory board).

**Processor
Fault Crashes**
24.6.2 The processor fault error crash descriptions are as follows:

```
I-Mem parity error
Boot request interrupt
Power fail interrupt
Wild transfer to zero
```

```
Illegal page fault at <hex address>--from <micro-pc of caller>
Page fault in physical memory reference.
Page fault in physical memory store.
```

These crashes are generally caused by processor board failures. The first four indicate a problem with the processor’s interrupt sensing logic. The others occur when a page fault condition was detected on a physical memory reference of some sort. The saved VMA register usually contains the physical address. These page fault crashes probably indicate that the processor’s

memory mapping hardware is faulty, although repeatable cases of these crashes may also indicate system software problems.

Power Fail Crash 24.6.3 The power fail error crash description is as follows:

Power failure

This crash description is usually recorded anytime that power to the machine is interrupted while the system is running. While such power interruptions are generally caused by external events, they may on occasion indicate a problem with the Explorer power supply.

Mass Storage Subsystem Crashes 24.6.4 The crash codes specific to the mass storage subsystem are as follows:

Disk error on swap: *<type>* error:
<Description of controller or device error>
 Status Word = *<Actual RQB status word, in hex>*
 Command Word: *<RQB command word, in hex>*, Logical Unit: *<unit>*
 Block-Address: *<RQB block address>*, Transfer Count: *<number of bytes>*

Cold-Disk error: *<type>* error:
<Description of controller or device error>
 Status Word = *<Actual RQB status word, in hex>*
 Command Word: *<RQB command word, in hex>*, Logical Unit: *<unit>*
 Block-Address: *<RQB block address>*, Transfer Count: *<number of bytes>*

NUPI Special Event Signaled:
<Special event type description>

Unknown NUPI Special Event;
 NUPI error status request timed out

Unknown NUPI Special Event;
 NUPI error status request got an error:
<Error description>

Time out on Disk Swap Out

The crash codes indicate probable failure of one of the subsystem's components, such as the NUPI board, a disk drive or tape drive mass storage unit (MSU), or cabling problems. The following list describes these crash codes:

Disk error on swap: *<type>* error:
 Cold-Disk error: *<type>* error:
 These crash codes indicate that some sort of error condition was returned by the NUPI on a request issued by the system microcode. (In contrast, NUPI errors occurring on requests initiated from Lisp simply cause the error handler to be invoked, or otherwise deliver an explanatory message without crashing the system.)

Disk error on swap: *<type>* error: indicates that the NUPI error was received on a page swap-in or swap-out operation. Cold-Disk error: *<type>* error: indicates that the error occurred during the early stages of the boot process.

For both of these crashes, the *<type>* field is *controller* if the error was an error condition for the NUPI board itself or *device* if the error was from another formatter or device on the SCSI bus. The next line of the analysis output gives a brief English description of the error code, followed by a line giving the actual contents of the erroring disk RQB (request block) status word. Examples of these errors include device errors such as uncorrectable data error, seek error, invalid command, and write protected; and controller errors such as invalid command parameter, NuBus bus error, and SCSI bus parity error. A list of all NUPI error codes is shown in Table 24-1.

The last shutdown record in Figure 24-1 shows a *disk-error-on-swap* crash that was caused by an uncorrectable data error. This disk error crash is the most frequent disk error crash; it usually indicates a bad spot on the disk. Note that the crash record also contains information about the logical unit, the disk block number being accessed, and the transfer length in bytes. You can use this information to determine if the bad spot is *hard* (that is, an error occurs on any access) or *soft* (the error occurs only intermittently). Consider the following function:

```
(defun test-disk-spot (unit address nbytes
  &optional (times-to-test 1))
  (sys:with-rqb (rqb (sys:get-disk-rqb (ceiling nbytes 1024.)))
    (dotimes (i times-to-test)
      (sys:disk-read rqb unit address))))
```

You can call this function with the *unit*, *address*, and *nbytes* reported in the crash record. If this function produces a Lisp disk error when called the first time, the data error is probably a hard one. If the function does not produce an error when called the first time, perhaps calling this function with a large *times-to-test* argument will eventually cause an error to be signaled, indicating the data error is soft.

If you cross-check the disk block address with the display from (*print-disk-label unit*), you can determine the partition to which the bad block belongs and take appropriate measures to correct the problem.

NUPI Special Event Signaled: *<Special event type description>*

This crash code occurs when the NUPI signals one of its special event error conditions, such as overtemperature, multiple commands issued to the same device, NuBus error encountered reading the command block, and so forth. These special event types are enumerated in Table 24-2. The *<Special event type description>* describes which of the special events was signaled. If the event was a formatter overtemperature, the formatter number is also included in the analyzer's output.

Unknown NUPI Special Event; NUPI error status request timed out

Unknown NUPI Special Event; NUPI error status request got an error

These crashes occur when the status request to determine the special event type itself received an error. The type of error should be described in the report.

Time out on Disk Swap Out

This crash occurs when the microcode routine's timeout period expired during a page-fault swap operation. The timeout period for page-fault swap requests is approximately 30 seconds. This can indicate a NUPI or formatter problem.

For more detailed information about NUPI operations, consult the *Explorer NuBus Peripheral Interface General Description (NUPI board)* manual.

**Troubleshooting
NUPI Device and
Controller Error
Crashes**

24.6.5 When the crash analysis points to the mass storage subsystem, you should consult the NUPI device or controller error, if given, for further troubleshooting information. Table 24-1 lists all the NUPI error codes, along with their English descriptions. Note that these troubleshooting hints can also be applied when these errors are generated by Lisp code.

Table 24-1

NUPI Device and Controller Error Codes

Error of form: #x+60xx0000 (NUPI Controller Error)

2x	Self test error (<i>x</i> can be any value)
3x	Self test error (<i>x</i> can be any value)
61	NuBus time out
62	NuBus bus error
63*	SCSI bus error
64	Formatter busy
65	Rate error
66	Bus error trap occurred
81	Command aborted
82	Invalid command
83	Invalid parameter
84	SCSI command completed w/o transfer
8C	Attempted a Read and Hold Command with one pending
8D	Attempted a Write & Swap of Buffer to NuBus command without a valid buffer reserved by Read & Hold
A1*	Illegal interrupt
A2*	SCSI Function Complete Without Cause
A3*	Timeout on NCR 5385 Data Register Full Wait
A4*	SCSI Invalid Command Interrupt
A5	Software error trap occurred
A6	Hardware error trap occurred
A7	Queue overflow occurred
A8	Address error trap occurred
A9	Illegal instruction error trap occurred
AA	NuBus DMA locked up; may need HW reset

NOTES:

* This error code indicates a cabling or MSU problem.

Errors that are not manifestly a cabling or MSU problem can be caused by a variety of different failures. Always check the cabling and make sure the connectors have no bent pins. For NUPI controller errors, then check the NUPI itself. For formatter or device errors, check the MSU.

Table 24-1

NUPI Device and Controller Error Codes (Continued)

 Error of form: #x+6000xx00 (Device or Formatter Error)

2x	Self test error (x can be any value)
3x	Self test error (x can be any value)
41	No selected unit
42	Media not loaded
43	Write protected
44	Mass storage enclosure power reset
45	Media change
46	Temperature fault
47	Invalid media type
4A	Tape EOM (end of medium)
50	SCSI Bus hung, needs hardware reset
51	SCSI Device won't reconnect
52	SCSI Device won't complete
53	Status: Non-extended error code is zero
54	Device has multiple block descriptors
55	Device has undefined block length
61	SCSI bus parity error
62	Drive not ready
63	Rate error
64	Invalid SCSI interrupt: selected
65	Device Offline
66	Invalid SCSI testability interrupt
67	Invalid SCSI disconnect
68	Invalid mode for SCSI status
69	Invalid mode for SCSI command byte request
6A	Sequence error: SCSI completion address
6B	Sequence error: SCSI requested data
6C	Sequence error: SCSI DMA start-stop address
70	Unknown MSG Received from Formatter
71	Invalid mode on SCSI message in
72	Excess SCSI status
73	Excess SCSI command bytes requested
74	Expected SCSI restore message but not received
75	Reconnected to unit not waiting for it
76	Expected SCSI cmd complete msg; did not receive it
77	Illegal SCSI message for reconnected state
78	Reselected without valid SCSI ID
79	Invalid mode on SCSI message out
7A	Invalid mode on SCSI data transfer
81	Command aborted
82	Invalid command
83	Illegal command sequence or Invalid parameter
84	Illegal block address
85	Volume overflow
8A	Formatter failed to connect

Table 24-1

NUPI Device and Controller Error Codes (Continued)

Error of form: #x+6000xx00 (Device or Formatter Error, continued)

8E*	Unknown error code returned from formatter
8F	Formatter busy
A1*	Missing index signal
A2*	No seek complete
A3*	Write fault
A4*	Track 0 not found
A5*	Multiple units selected
A6*†	Seek error
A7	Formatter hardware error
C1*†	ID error
C2*†	Uncorrectable data error
C3*†	ID address mark not found
C4*†	Data address mark not found
C5*†	Sector address not found
C6*†	Bad block found
C7*	Format error
C8*	Correctable data check
C9*	Interleave error
CA*	Media error

NOTES:

* This error code indicates a cabling or MSU problem.

† This error code could indicate a soft error on the media. The disk should be checked for soft errors.

Errors that are not manifestly a cabling or MSU problem can be caused by a variety of different failures. Always check the cabling and make sure the connectors have no bent pins. For NUPI controller errors, then check the NUPI itself. For formatter or device errors, check the MSU.

**Troubleshooting
NUPI Special
Event Crashes**

24.6.6 Table 24-2 lists all NUPI special event codes along with possible causes and troubleshooting hints.

Table 24-2

NUPI Special Event Codes

Code #x+0002: Diagnosis/Action:	Overtemperature on formatter <formatter number>. A formatter has signaled an overtemperature condition. Ensure that the appropriate formatter is properly cooled and that airflow around the MSU is not restricted.
Code #x+0200: Diagnosis/Action:	Hardware Error. This is due to a hardware problem in the NUPI's processor. Have the NUPI checked.
Code #x+0001:	Unrecoverable NuBus error encountered while fetching command block.
Code #x+0040: Code #x+0080: Diagnosis/Action:	Command aborted with no command block updates. Illegal interrupt encountered on NuBus access. These conditions can be caused by faulty NUPI bus interface hardware or a bad memory board. The NUPI should be checked first.
Code #x+0400: Diagnosis/Action:	Invalid SCSI operation attempted. This may indicate a formatter or cabling problem. Ensure that the formatter cables are secure and properly connected. Have the formatter checked.
Code #x+0004: Code #x+0008:	Illegal access to NUPI control register. Multiple commands issued to one device, formatter, or to NUPI.
Code #x+0010: Code #x+0020: Code #x+0100: Diagnosis/Action:	Illegal command found in command block. Invalid Special Event Type Found. Invalid Special Event address. These conditions are usually caused by faulty software, although the NUPI can also be the cause.

The Force Crash Keychord

24.7 If the machine gets into a “hang” state where Lisp is unresponsive and the system clock is no longer being updated, you can press the keychord META-CTRL-META-CTRL-C to force the microcode into immediate ILLOP processing. This sort of state often indicates that Lisp or the microcode is in a hard run, with interrupts disabled. (Therefore, keystrokes including the ABORT key sequences are disabled, although the mouse may still be tracking.) The advantage of the force crash keychord is that after rebooting, the shutdown record should give some indication of the function that was hung in the FEF *Running at Crash* field. It is best to try to warm boot after forcing a crash with this keychord, although that may not always work.

A forced crash is indicated in the shutdown Description with a shutdown description of Crash called by Force-Crash Keychord. The second shutdown record in Figure 24-1 is an example of a forced crash. The shutdown record indicates that the offending function may have been MY-INFINITE-LOOP-WITHOUT-INTERRUPTS. The source code can then be consulted to see if it contains code inside a **without-interrupts** form.

NOTE: The screen should invert when the force crash sequence is used. If it does not, the “hang” you are observing may actually be a crash caused by the processor in which ILLOP processing was bypassed. Check to see if any of the small yellow LEDs on the side of the processor board are illuminated (they are visible through the cutout in the metal chassis door). If lights 1 and 7 are lit (counting from the top), this is a PROM-generated #x+82 crash, indicating that the writable control store on the processor is faulty.

Software Crash Descriptions

24.8 Although most software crash descriptions indicate unexpected error conditions, the following two crash types can be considered normal if the machine’s resources are being stretched to their limits:

Out of swap space
Virtual Memory Overflow while traps disabled

The first crash type, *out of swap space*, indicates that you have run out of paging store, which is generally caused by a program that generates large numbers of data objects that cannot be reclaimed by garbage collection (GC). This crash type is usually preceded by a number of swap-space-low warnings from the GC daemon (unless you have disabled these warnings).

The second crash type, *Virtual Memory Overflow while traps disabled*, may also occur under similar circumstances even if you have a lot of paging store. This crash type is usually preceded by a number of address-space-low warnings from the GC daemon. However, the system is often able to invoke the debugger before crashing with a virtual memory overflow. You then have a chance to save (quickly) any buffers before the imminent crash. For information about the paging, virtual memory, and garbage collection systems, refer to the *Explorer Lisp Reference*.

Table 24-3 provides a list of the remaining software crash descriptions, which generally indicate a problem with system software or a corrupted data structure. You should note that hardware problems can lead to erratic crashes in this software category also, but the pattern of crashes is not likely to be firm.

Table 24-3 Software Crash Descriptions

a-qcstkg not stack-group at sg-load-static-state
A-QCSTKG not stack-group at SGLV.
Argument-pointer not fixnum during virtual-memory unwind in %throw.
Attempt to allocate past end of Device-Descriptor region
Attempt to overpop binding frame
Bad Abbreviated Jump to CALL-TO-BUS-ERROR
Bad Abbreviated Jump to CALL-TO-UNUSED
Bad header address in routine SINFS-BFWD
Bad PROM entry address.
Bad Representation Type at address *<hex number>*
Band not format 2000, no valid DPMT
Binding cell on PDL not locative
Body-Forward without Header-Forward: *<hex number>*
Boot request interrupt
Call-info word not fixnum while setting epph bit in LEXICAL-CLOSURE-FOUND-THIS-FRAME.
Couldn't find partition in disk label
Crash called by Force-Crash Keychord
Data illegal type: *<hex number>* with pointer *<hex number>* read from *<hex number>*
Data illegal type: *<hex number>* with pointer *<hex number>* read from OLDSPACE
Data type in FEF was not a EVCP during Bind instruction
Data type not array header when trying to get array header.
Disk RQB not wired for Swap
Disk RQB not Wired on Swap
Disk write RQB not mapped at Append-Additional-Pages-To-Disk-Write
DISK-error on swapout. Complete bit not set in status word, but set in info word.
DTP-FUNCTION *<hex number>* doesn't point at DTP-FEF-HEADER.
Error handler stack group already active
EVCP encountered on Transporting header. Address: *<hex number>*
Free Cluster Count Went Negative
FULL NODE in stack list where CDR cell not in PDL Buffer.
Header forward doesn't point at a header, points instead at *<hex number>*
in Find-Structure-Header(Structure-HFWD)
Illegal data type for a structure header in Structure-Info
Illegal data type found in internal storage -- from *<hex number>*
Illegal data-type in throw-tag for %throw-n: *<hex number>*
Illegal data-type in throw-tag for %throw: *<hex number>*
Illegal header type for extended number in EQL.
Illegal header type for extended number.
Illegal header type in Find-Structure-Header; header word = *<hex number>*
Illegal interrupt (type = *<decimal number>*)
Illegal number-type as arg in arith-any--fix.
Illegal or unused combination of destination and return-type.
Illegal page fault at *<hex number>* -- from *<hex number>*
Illegal region representation type in Find-Structure-Leader
Illegal stack group state at SG-RETURN, state *<state description>*
Illegal stack group state: *<state description>*
Illegal stack-group state at SG-RESUME, state *<state description>*
Illegal type in numeric arg.
Illegal use of One-Q-Forward at *<hex number>*
Impossible error in BIGNUM-RIGHT-JUST.
Inconsistent function state
Inconsistent function state in %throw-n.
Inconsistent function state in %throw.
Inconsistent swap device status, should be Unassigned

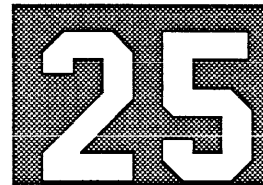
Table 24-3 Software Crash Descriptions (Continued)

Index to unbind to isn't a locative
 Info Word shows DONE, but done bit not set in status word on page read
 Invalid address (not in region) on get-map-bits
 Invalid disk address for virtual address
 Invalid Header-Type, <hex number>, in Structure-Info
 Invalid level 2 map status
 Invalid level 2 map swap status (in PHT)
 Invalid level 2 map swap status in PHT (PHT2 word in MD)
 Invalid operation type in internal arithmetic dispatch.
 Invalid page device (not assigned)
 Invalid Page Hash Table swap status
 Invalid Page Hash Table swap status on Findcore
 Invalid structure: header not found in region
 Invalid virtual address (not in region) Page-In-Get-Map-Bits
 Keyboard initialize not supported
 L1-Map-Entry-Invalid for wired meter buffer
 Level 1 map invalid at level 2 reload
 Local-pointer not fixnum during PDL unwind in %throw-n.
 Local-pointer not fixnum during PDL unwind in %throw.
 Local-pointer not fixnum during virtual-memory unwind in %throw-n.
 Local-pointer not fixnum during virtual-memory unwind in %throw.
 Local-Pointer not Fixnum in LEX-CLOSURE-SCAN-PDL-FRAMES.
 LRU Linked list contains invalid PPD Index (address in VMA)
 Map level 1 invalid at level 2 handler
 Micro stack not empty on stack group enter
 Needfetch not set after loading location-counter.
 No Physical Memory available for page swap at Get-Free-Memory-Page
 Not DTP-HEADER after transport-header.
 Number of thrown values must be a fixnum in %throw-n.
 Number of values to return was not a fixnum.
 Number-of-results not T, NIL, or a number in %throw-n.
 Number-of-results not T, NIL, or a number in %throw.
 NuPI interrupt handler, inconsistent state (no request)
 Overran Physical Page Data Table in Delete-Physical-page (virtual address in VMA)
 Overran RQB scatter list - address in VMA
 Page already has page hash table entry at Page-In-Make-Known. Memory manager is broken.
 Page fault in physical memory reference.
 Page fault in physical memory store.
 Page fault in physical-array reference
 Page fault in physical-array store
 Page Hash Table entry invalid at Mark-Dirty-Page
 PDL overpopped
 Power fail interrupt
 Ran off bottom of stack in LEX-CLOSURE-SCAN-PDL-FRAMES.
 Recursive Trap
 Recursive use of Transporter or transporter save routine
 Region-GC-pointer has gone past the end of region <hex number>
 Returning a value to a D-MICRO call that doesn't want any.
 RQB not wired down in NuPI start IO request
 Sequence break not allowed
 Short interval timer interrupt
 Single Float not pointing to DTP-Header\header-type-flonum.
 TRANS-OLD but pointer is not to OLDSPACE: <hex number>
 Trap before error handler enabled (debug flags)
 Trap while sequence breaks disabled
 Trap while traps disabled (M-Flags)
 Type <type> not header after TRANSPORT-HEADER in Find-Structure-Header

Table 24-3 Software Crash Descriptions (Continued)

Unexpected data type in Find-Structure-Header: *<data type>*
Unimplemented stack-object return situation.
Unused call-type in FEF header.
While restoring upc stack: return address data type not fixnum.
Wild transfer to zero
XMINUSP-T of fixnum.
XPLUSP-T of fixnum.

MISCELLANEOUS DEBUGGING FUNCTIONS



Introduction

25.1 The miscellaneous debugging functions and special forms provide the means to discover the complex relationships between various Lisp forms and, to some degree, the workings of the Explorer system. The following groups of debugging functions are discussed in this section:

- Describe functions — Provide interesting information about objects such as a symbol, **defstruct**, flavor, system, package, partition, area, region, and so on.
- Apropos functions — Search for symbols that contain a certain substring.
- Who-calls functions — Find the functions that call other functions.
- Property list functions — Return the property list of a symbol or a pathname.
- Print functions — Provide numerous options to print objects and function definitions.
- Dribble file functions — Control the dribble file, which records all the terminal interaction performed during a session.
- Environment functions and variables — Provide information about your environment, such as the Lisp implementation you are using, machine type and version, software type and version, and the documentation for a symbol.

You can use most of these functions to great advantage in conjunction with either the Inspector or Zmacs.

Also, Zmacs provides numerous useful commands that you can use for debugging. The following lists a few of them. Refer to the *Explorer Zmacs Editor Reference* for details.

- Dired commands — The directory editor (Dired) presents a listing of a directory and provides a wide range of commands to deal with the files in the directory, such as editing, viewing, comparing, copying, printing, loading, and compiling.
- Caller commands — These commands list and edit functions that call other functions. The List Callers command is similar to the **who-calls** function, which is described in paragraph 25.4, Who-Calls Functions.
- Flavor commands — These commands list and edit flavors and methods, including component flavors, dependent flavors, direct dependent flavors, and all methods used for a specified operation on a flavor.
- Edit Source commands — These commands edit the definition of a function, **META-** being the most notable command.

- Possibility commands — These commands allow you to edit multiple definitions of a function.
- List and Edit Changed Definition commands — These commands perform operations on the changed definitions in a buffer. Zmacs keeps track of which definitions have been modified.
- Source Compare commands — These commands allow you to compare two files or buffers.

Describe Functions 25.2 The describe functions include **describe** (which describes an object) and several more specialized functions that describe the following: **defstruct**, **flavor**, **system**, **package**, **partition**, **area**, and **region**:

describe *x*

[c] Function

The **describe** function displays the interesting information about an object *x*. This function can handle arrays, closures, FEFs, symbols, lists, characters, locatives, stack groups, packages, floating-point numbers, fixnums, bignums, complex numbers, select methods, named structures, and **defstructs**. It prints the attributes of any of these objects. Additionally, this function can describe an item located within another; such recursive descriptions are indented appropriately. For instance, **describe** on a symbol tells you about the symbol's value, its function definition, and each of its properties. Calling **describe** on a floating-point number shows you its internal representation in a way that is useful for tracking down round-off errors and the like. Executing **describe** on an array tells you about the array's type, dimension, and length, but the content of each array element is not shown. (The Inspector shows array contents.)

If *x* is a named structure, **describe** handles it specially. To understand this, read the discussion of named structures in the *Explorer Lisp Reference*. First, **describe** retrieves the named structure symbol and determines whether the function handles the **:describe** operation. If the operation is handled, **describe** applies the function to two arguments: the symbol **:describe** and the named structure instance itself. Otherwise, **describe** examines the named structure symbol's property list for information that might have been left by **defstruct**; this information indicates the symbolic names for the entries in the structure, and **describe** uses these names to print each field's name and contents.

Example: (describe 'foo)

The following appears on the screen:

```
Symbol FOO is in USER package.
It has no value, definition, or properties.
```

describe-defstruct *instance* &optional *name*

Function

This function takes an *instance* of a structure named by *name* and displays a description of the instance, including the values of each of its slots. The *name* argument is optional only if *instance* is an instance of a named structure; if it is unnamed, the function returns an error message if *name* is not provided. The reason for this error message is that **describe-defstruct** must have some way to know which structure *instance* is an instance of. If *instance* is a named structure, then the structure name is already embedded within *instance*, and

`describe-defstruct` knows where to find it; therefore, *name* is optional only in this case.

This function prints out the information of a structure in the way shown in the following example; however, you can define your own function to print the information of a named structure in a format of your own choosing.

Suppose an instance of a structure called `player` had been constructed as follows. (This example assumes the `defstruct` named `player` has already been defined.)

```
(setf sir-slam (make-player :name "Darryl Dawkins"
                           :team "New Jersey Nets"
                           :position "C"
                           :games 1
                           :points 12
                           :rebounds 13
                           :assists 2
                           :pts-per-game 12.0))
```

Then, a call to `describe-defstruct` would produce the following:

```
(describe-defstruct sir-slam)
=> #<PLAYER 87655644>

name:          "Darryl Dawkins"
team:          "New Jersey Nets"
position:      "C"
games:         1
points:        12
rebounds:      13
assists:       2
pts-per-game: 12.0

#S(player :name "Darryl Dawkins" :team "New Jersey Nets" :position
"C" :games 1 :points 12 :rebounds 13 :assists 2 :pts-per-game
12.0)
```

`describe-flavor` *flavor-name*

Function

The `describe-flavor` function prints out descriptive information about a flavor including component flavors, instance variables, noninherited methods, hash table information, and much more.

You can also invoke the `describe-flavor` function with the Zmacs META-X Describe Flavor command.

Example: `(describe-flavor 'w:essential-window)`

The following appears on the screen:

```
Flavor TV:ESSENTIAL-WINDOW directly depends on flavors: TV:SHEET
and is directly depended on by TV:BASIC-FRAME, TV:PANE-MIXIN,
TV:BASIC-SCROLL-WINDOW, TV:MINIMUM-WINDOW
Not counting inherited methods, the methods for TV:ESSENTIAL-WINDOW are:
: PANE-SIZE
: AWAIT-EXPOSURE
: ALIAS-FOR-INFERIORS
.
.
: MOUSE-SELECT, :BEFORE :MOUSE-SELECT, :WRAPPER :MOUSE-SELECT
Keywords in the :INIT message handled by this flavor: :EDGES-FROM,
:MINIMUM-WIDTH, :MINIMUM-HEIGHT, :ACTIVATE-P, :EXPOSE-P
Defined in package TV
Properties:
  SYS:UNMAPPED-INSTANCE-VARIABLES: (TV:SCREEN-ARRAY TV:LOCATIONS-PER-LINE
  TV:OLD-SCREEN-ARRAY ... TV:BACKGROUND-COLOR)
  :DOCUMENTATION: ...
Flavor TV:ESSENTIAL-WINDOW does not yet have a method hash table
NIL
```

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

The **describe-system** function prints information about the system named *system-name*.

:show-files — When this keyword is set to **t** (the default), this function displays the history of each file in the system. When this keyword is set to **nil**, the function does not display the history. When this keyword is set to **:ask**, the user is queried on whether to display the history or not.

:show-transformations — When this keyword is set to **t** (the default), this function displays the transformations that **make-system** should execute. When this keyword is set to **nil**, this function does not display the history. When this keyword is set to **:ask**, the user is queried about whether to display the history.

The *Explorer Lisp Reference* discusses **make-system** and **defsystem** in detail.

Example: (describe-system 'debug-tools)

A display such as the following appears:

```
System Debug Tools is defined in file SYS: DEBUG-TOOLS; DEFSYSTEM.#
newton: DEBUG-TOOLS; DEFSYSTEM.LISP#5 was created 3/03/87 11:36:45

Debug Tools is patchable, Experimental, 7.0 is loaded;
  a typical patch file is SYS: PATCH-DEBUG-TOOLS; PATCH-7-0.LISP#>
Do you want to see the patches for Debug Tools? (Y or N) No.
Compilation and loading of files in this system:
```



```

SYS: DEBUG-TOOLS; STEP.XLD#> is loaded
newton: DEBUG-TOOLS; STEP.XLD#6 was created 3/03/87 11:37:19
SYS: DEBUG-TOOLS; PEEK.XLD#> is loaded
newton: DEBUG-TOOLS; PEEK.XLD#8 was created 3/03/87 11:37:20
SYS: DEBUG-TOOLS; INSPECT.XLD#> is loaded
newton: DEBUG-TOOLS; INSPECT.XLD#8 was created 3/03/87 11:37:20
SYS: DEBUG-TOOLS; FLAVOR-INSPECTOR.XLD#> is loaded
newton: DEBUG-TOOLS; FLAVOR-INSPECTOR.XLD#9 was created 3/03/87 11:37:21
SYS: DEBUG-TOOLS; WINDOW-DEBUG.XLD#> is loaded
newton: DEBUG-TOOLS; WINDOW-DEBUG.XLD#9 was created 3/03/87 11:37:21
SYS: DEBUG-TOOLS; TRACE-WINDOW.XLD#> is loaded
newton: DEBUG-TOOLS; TRACE-WINDOW.XLD#2 was created 3/03/87 11:37:21
Make Debug Tools patchable
Transformations required to MAKE-SYSTEM now:

Need to Increment Debug Tools patch version
"Debug Tools"

```

describe-package *package* Function

The **describe-package** function prints all available information about *package*, except for all the symbols interned in it. The *package* argument can be a package object or the name of one.

Example: (describe-package 'user)

A display such as the following appears:

```

Package USER.
  347. symbols out of 2000. Hash modulus = 2503.
Packages which are used by this one:
  LISP
  TICL
  ZLC

#<Package USER -77577665>

```

To view all symbols interned in a package, use the following form:

(mapatoms #'print *package*)

sys:describe-partition *part* &optional *unit* Function

The **describe-partition** function displays information about the partition named *part* on the disk specified in *unit*. If the disk contains more than one partition with the same name, the first occurrence is selected.

If *part* is a LOD partition, this function displays (in addition to other information) the microcode version required by the band, the size of the data in the partition, and the highest virtual address used. The size informs you of the amount of space required to copy this partition; the highest virtual address indicates the size of the PAGE partition needed to run *part*.

Arguments: *part* — Specifies the name of the partition to describe.

unit — Specifies the logical unit that contains the partition. The *unit* argument can be a logical unit number, such as 1; a pack name, such as "D1"; a host name, such as "C9"; a host name and unit number, such as "C9:1", or a host name and a pack name, such as "C9:D1". If a host name is specified without a unit number, the default unit for that host is used. The default value is `sys:*default-disk-unit*`.

Example: (sys:describe-partition 1)

A display such as the following appears:

```
Partition LOD1 starts at 37241 and is 35000 blocks long.
It is in non-compressed format, data length 35000 blocks.
Goes with microcode version 0.
It is a (Load Band) of CPU type (Explorer)
Default band: T
NIL
```

If you execute the Lisp function (sys:describe-partition "mcr1"), the display is similar to the following:

```
Partition MCR1 starts at 1781 and is 160 blocks long.
It is in non-compressed format, data length 160 blocks.
Contains microcode version 315.
It is a (Microcode) of CPU type (Explorer)
Default band: T
NIL
```

describe-area <i>area</i> &key :base :verbose	Function
describe-region <i>region</i> &key :base	Function

These functions provide information on the current state of memory allocation for a particular *area* or *region*. If :base is supplied, the virtual addresses that are printed are formatted in that base; the default is 10.

If :verbose is true (the default) for **describe-area**, the **describe-region** function is called, with its default arguments, for each region in the specified area.

Examples:

```
(make-area :name 'a-dynamic-area) ; Make an area.
(make-array 100 :area a-dynamic-area) ; Something in structure region.
(make-list 100 :area a-dynamic-area) ; Something in list region.
(describe-area a-dynamic-area))
```

A display such as the following appears:

```
Area 71: A-DYNAMIC-AREA
There are now 131072 words assigned, 201 used. The area is growable.
Region size 65536
Default cons generation = 3
It currently has 2 regions.
251: 22331392 Origin, 65536 Length, 100 Used, 0 GC, Type LIST NEW, Gen 3
250: 22265856 Origin, 65536 Length, 101 Used, 0 GC, Type STRUC NEW, Gen 3
```

In this example, a new area is made and then something is put in each type of region. This display is done in base 10, and the default is to print information on each region. Note that the region numbers are 250 and 251. For each region, the following definitions are used:

- Origin — The virtual address of the origin of this region.
- Length — The allocation of this region in words.
- Used — The number of words used in this region.
- GC — A scavenger pointer; only meaningful when garbage collection is active in this region.
- Type — The type of data in this region; the type is either LIST or STRUCTURE.

- Space — The current space type, which is one of the following:
 - NEW — A new object is being created in this region.
 - OLD — Old space is being collected.
 - COPY — Objects are being copied from this region.
 - STATIC — This region contains very old objects.
 - FIXED — Similar to STATIC; for system use only.
- Gen — The age of objects in this region; this value is 0, 1, 2, or 3, where 3 is the oldest.
- *flags* — The following informative flags can appear at the end of this line:
 - READ-ONLY — This region is read-only.
 - MAR — The MAR points to something in this region.

Example: (describe-region 251. :base 8.)

A display such as the following appears:

```
251: #0125140000 Origin, #o200000 Length, #0144 Used, #o0 GC, Type LIST NEW, Gen 3
```

The information for this region is the same as that in the preceding **describe-area** example except that the information for **describe-region** is printed in base 8.

Apropos Functions 25.3 The apropos functions, which include **apropos**, **aproposf**, **aproposb**, **apropos-flavor**, **apropos-method**, **apropos-resource**, **apropos-list**, and **sub-apropos**, find symbols for you.

apropos *substring* &optional *package* [c] Function
apropos *substring* &key :package :inheritors (:inherited t) Function
 :predicate :boundp :fboundp :dont-print

Tries to find all symbols in the entire system whose print names contain *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 is bound to a value, that information is included, along with the names of the arguments (if any) to the function. (You can also invoke the **apropos** function with the Zmacs META-X Function Apropos command.)

If **:predicate** is non-**nil**, its value should be a function; only symbols for which the function returns non-**nil** are mentioned.

The **apropos** function looks for symbols in *package* and for symbols in packages used by the specified *package* (unless **:inherited** is **nil**). If **:inheritors** is **t**, the packages that use the specified *package* are searched as well. The *package* argument defaults to **nil**, meaning that all packages are searched.

NOTE: When you specify a *package* in **apropos**, it finds the same symbols that **do-symbols** does. (Refer to the *Explorer Lisp Reference* for information on **do-symbols**.)

If **:boundp** is non-**nil**, only bound symbols are mentioned. If **:fboundp** is non-**nil**, only symbols with function definitions are mentioned.

Finally, **apropos** prints a listing of all the symbols it finds. If **:dont-print** is non-**nil**, a list of the symbols found is returned, and the printing of additional information (form, type, bound or not, and so on) is suppressed.

Example: (apropos "copy-file" 'fs)

A display such as the following appears:

```

FS:*COPY-FILE-KNOWN-TEXT-TYPES*      Bound (:LISP :TEXT :MIDAS...), plist
FS:*COPY-FILE-KNOWN-BINARY-TYPES*    Bound (:XLD :QFASL :PRESS...), plist
FS:FS-COPY-FILE                       function (PATHNAME-OR-STREAM NEW-NAME &KEY ...),
                                       plist
FS:PRIMITIVE-COPY-FILE                function (INPUT-PLIST-OR-PATHNAME MAPPED-PATHNAME
                                       OUTPUT-SPEC ...), plist
FS:*COPY-FILE-KNOWN-SHORT-BINARY-TYPES* Bound (:ARCHIVE :OBJECT), plist
FS:COPY-FILE                           function (FILES TO &REST ...), plist
HARD-COPY-FILE
COPY-FILE                             function (PATHNAME-OR-STREAM NEW-NAME &KEY ...),
                                       plist

```

apropos-list *substring* &optional *package* [c] Function
apropos-list *substring* &key *:package* *:inheritors* (*:inherited* *t*) Function
:predicate *:boundp* *:fboundp* (*:dont-print* *t*)

Identical to calling the **apropos** function with **:dont-print** *t*. However, instead of printing symbol names, it returns a list of all the symbols it finds that match *substring*. For a description of the keywords, refer to the **apropos** function.

Example: (apropos-list "copy-file" 'fs)

A display such as the following appears:

```

(COPY-FILE HARDCOPY-FILE FS:COPY-FILES
FS:*COPY-FILE-KNOWN-SHORT-BINARY-TYPES* FS:PRIMITIVE-COPY-FILE
FS:FS-COPY-FILE FS:*COPY-FILE-KNOWN-BINARY-TYPES*
FS:*COPY-FILE-KNOWN-TEXT-TYPES*)

```

aproposf *substring* &key *:package* *:inheritors* (*:inherited* *t*) Function
:predicate *:boundp* (*:fboundp* *t*) *:dont-print*

Calls the **apropos** function, searching for functions. The **:fboundp** option defaults to *t*. This function is exactly equivalent to calling **apropos** with **:fboundp** set to *t*. For a description of the other keywords, refer to the **apropos** function.

Example: (aproposf "copy-file" 'fs)

A display such as the following appears:

```
FS:FS-COPY-FILE          function (PATHNAME-OR-STREAM NEW-NAME &KEY ...), plist
FS:PRIMITIVE-COPY-FILE  function (INPUT-PLIST-OR-PATHNAME MAPPED-PATHNAME OUTPUT-SPEC ...),
                        plist
FS:COPY-FILES           function (FILES TO &REST ...), plist
COPY-FILE               function (PATHNAME-OR-STREAM NEW-NAME &KEY ...), plist
T
```

aproposb *substring* &key :package :inheritors (:inherited t) Function
 :predicate (:boundp t) :fboundp :dont-print

This function calls the **apropos** function, searching for bound variables. The **:boundp** option defaults to **t**. This function is exactly equivalent to calling **apropos** with **:boundp** set to **t**. For a description of the other keywords, refer to the **apropos** function.

Example: (aproposb "copy-file" 'fs)

A display such as the following appears:

```
FS:*COPY-FILE-KNOWN-TEXT-TYPES*      Bound (:LISP :TEXT :MIDAS...), plist
FS:*COPY-FILE-KNOWN-BINARY-TYPES*    Bound (:XLD :QFASL :PRESS...), plist
FS:*COPY-FILE-KNOWN-SHORT-BINARY-TYPES* Bound (:ARCHIVE :OBJECT), plist
T
```

apropos-flavor *substring* &key :predicate :dont-print Function

Finds all flavors whose names contain *substring*. If **:predicate** is non-**nil**, its value is the function to be called for each flavor symbol in the package; only symbols for which the predicate returns non-**nil** are displayed or returned.

The **apropos-flavor** function prints a listing of the symbols it finds. If **:dont-print** is non-**nil**, a list of the symbols found is returned, and the printing of additional information about the symbols, such as documentation, is suppressed.

Example: (apropos-flavor "font")

A display such as the following appears:

```
FONT          This is the basic font for use in the graphic window system.
ZWEI:INTERVAL-STREAM-WITH-FONTS
PRINTER:TI855-FONT-MIXIN
PRINTER:TI880-FONT-MIXIN
T
```

apropos-method *substring flavor-instance* &key :predicate :dont-print Function

Finds all methods of a flavor whose names contain *substring*. The *flavor-instance* must be the instance of a flavor. If **:predicate** is non-**nil**, its value is the function to be called for each method name in the package; only symbols for which the predicate returns non-**nil** are displayed or returned. The **:package** keyword defaults to the **GLOBAL** package.

The **apropos-method** function prints a listing of the symbols it finds. If **:dont-print** is non-**nil**, a list of the symbols found is returned, and the printing of additional information about the symbols, such as documentation, is suppressed.

Example: (apropos-method "doc" w:selected-window)

A display such as the following appears:

```
(ZWEI:ZWEI-WITHOUT-TYPEOUT :COMBINED :WHO-LINE-DOCUMENTATION-STRING)
(.DAEMON-CALLER-ARGS.)
(ZWEI:ZWEI :SET-MOUSE-DOCUMENTATION-STRING) (.NEWVALUE.)
(ZWEI:ZWEI :MOUSE-DOCUMENTATION-STRING-LOCATION) ()
(ZWEI:ZWEI :MOUSE-DOCUMENTATION-STRING) ()
T
```

apropos-resource *substring* &key :predicate :dont-print Function

Finds all resources whose names contain *substring*. If :predicate is non-nil, its value is the function to be called for each resource name in the package; only symbols for which the predicate returns non-nil are displayed or returned.

The **apropos-resource** function prints a listing of the symbols it finds. If :dont-print is non-nil, a list of the symbols found is returned, and the printing of additional information about the symbols, such as documentation, is suppressed.

Example: (apropos-resource "window")

A display such as the following appears:

```
GLOSS:GLOSS-POP-UP-KEYSTROKES-WINDOW plist
SUGG:SUGGESTIONS-POP-UP-NOTIFICATION-WINDOW plist
SUGG:POP-UP-KEYSTROKES-WINDOW          plist, flavor
TELNET:VT100-WINDOWS                   Bound NIL, plist
.
.
.
TV:POP-UP-FINGER-WINDOW                 plist
T
```

sub-apropos *substring starting-list* &key :predicate :dont-print Function

Finds all symbols in *starting-list* whose names contain *substring* and that satisfy :predicate. The *starting-list* argument must be a list of symbols. If :predicate is nil, the substring is the only condition. The symbols are printed if :dont-print is nil. In any case, a list of the symbols found is returned.

This function is most useful when applied to the value of *, after **apropos** has returned a long list.

Example: (sub-apropos "abc" '(abcdef def bcd abc defabc))

A display such as the following appears:

```
ABCDEF
ABC
DEFABC
T
```

Who-Calls Functions

25.4 The who-calls functions include **who-calls**, **what-files-call**, and **where-is**. The **who-calls** function tells you which functions call a given function, the **what-files-call** function tells you what files contain the calling functions, and the **where-is** function tells you which packages contain a certain symbol.

who-calls *x* &optional *package* Function

With the **who-calls** function, *x* must be a symbol or a list of symbols. This function tries to find all of the functions in the Lisp world that call *x* as a function, use *x* as a variable, or use *x* as a constant. (This function does not find forms that use constants containing *x*, such as a list, one of whose elements is *x*; it only finds *x* if *x* itself is used as a constant.) This function searches all of the function cells of all of the symbols in *package* and *package*'s descendants. The *package* argument defaults to nil, meaning that all packages are checked.

If **who-calls** finds an interpreted function definition, it states whether *x* appears anywhere in the interpreted code. More information is provided about compiled code. The **who-calls** function can find macros expanded in the compilation of the code because they are recorded in the caller's debugging information association list, even though they are not actually referred to by the compiled code.

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

The Zmacs command META-X List Callers is similar to **who-calls**.

The **:unbound-function** symbol is treated specially by **who-calls**. The following form searches the compiled code for any calls through a symbol that is not currently defined as a function:

```
(who-calls `:unbound-function)
```

Such a form is useful for finding errors such as functions you misspelled or forgot to write.

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

Example:

```
(who-calls 'copy-file)
```

A display such as the following appears:

```
COPY-DIRECTORY calls COPY-FILE as a function.
ZWEI:COM-TEACH-ZMACS calls COPY-FILE as a function.
ZWEI:COPY-FILE-1 calls COPY-FILE as a function.
ZWEI:RENAME-FILE-ACROSS-DIRECTORIES calls COPY-FILE as a function.
ZWEI:COM-DIRED-COPY-AUX calls COPY-FILE as a function.
MT:MT-WRITE-FILES calls COPY-FILE as a function.
MT:BACKUP-1 calls COPY-FILE as a function.
MT:RESTORE-1 calls COPY-FILE as a function.
SYS:COPY-SYSTEM calls COPY-FILE as a function.
NAME:CREATIVE-APPEND calls COPY-FILE as a function.
(NAME:CREATIVE-APPEND SYS:COPY-SYSTEM MT:RESTORE-1 MT:BACKUP-1
MT:MT-WRITE-FILES ZWEI:COM-DIRED-COPY-AUX
ZWEI:RENAME-FILE-ACROSS-DIRECTORIES ZWEI:COPY-FILE-1
ZWEI:COM-TEACH-ZMACS COPY-DIRECTORY)
```

what-files-call *x* &optional *package* (*inheritors* *t*) (*inherited* *t*) Function

This function is similar to **who-calls** but returns a list of the pathnames of all the files containing functions that **who-calls** would have printed. This function is useful if you need to recompile and/or edit all of these files.

The **what-files-call** function finds all files in *package* that use *x*, which must be a symbol or list of symbols. The *package* argument defaults to `nil`, meaning that all packages are checked. The packages that inherit from *package* are processed also, unless *inheritors* is `nil`. In addition, the packages from which *package* inherits are processed, unless *inherited* is `nil`. (Other packages that merely inherit from the same ones are *not* processed.) The files are printed and a list of them is returned. The symbol `:unbound-function` is special: (`what-files-call` `:unbound-function`) finds all functions that are used but not currently defined.

Example: `(what-files-call 'copy-file)`

A display such as the following appears:

```
(#CFS:LOGICAL-PATHNAME "SYS: NAMESPACE; UTILITY.U#U"␣
#CFS:LOGICAL-PATHNAME "SYS: MAKSYS; MAKSYS.U#U"␣
#CFS:LOGICAL-PATHNAME "SYS: STREAMER-TAPE; MTAUX.U#U"␣
#CFS:LOGICAL-PATHNAME "SYS: ZWEI; Dired.U#U"␣
#CFS:LOGICAL-PATHNAME "SYS: ZWEI; FILES.U#U"␣
#CFS:LOGICAL-PATHNAME "SYS: ZWEI; COMP.U#U"␣
#CFS:LOGICAL-PATHNAME "SYS: FILE; DIRECTORY-SUPPORT.U#U"␣ )
```

where-is *pname* &optional *package* Function

The **where-is** function prints the names of all packages that contain a symbol with the print name *pname*. If *pname* is a string, note that character case matters. (Most symbols are created by the Lisp Reader with uppercase print names.)

Unless *package* is specified, **where-is** assumes you want to search all packages. If you specify *package*, the function searches only *package* and those packages inheriting from the specified package. The **where-is** function returns a list of all the symbols it finds.

Example: `(where-is "COPY-FILE")`

A display such as the following appears:

```
TICL: COPY-FILE is accessible from packages CHAOS, COMPILER, EH,
ETHERNET, FED, FILE-SYSTEM, FONTS, FORMAT, GLOSS, IMAGEN,
IMAGEN-FONTS, INSTALLER, IP, LISP, MAIL, MATH, METER, MT, NAME,
NET, NET-CONFIG, NEW-USER, NSE, NUBUS, PRINTER, PROFILE,
SRCCOM, SUGG, SYSLOG, SYSTEM, TELNET, TICL, TIME, TV, UCL,
USER, W, ZLC, ZWEI
```

find-all-symbols *pname* &optional *package* Function

This function is similar to the **where-is** function except that **find-all-symbols** only searches the packages where the symbol is defined. The **where-is** function searches the packages where the symbol is defined *and* all the other packages that inherit from the specified package.

Example: Suppose the symbol `foo` is defined in several packages:

```
(setq user:foo nil w:foo nil zwei:foo nil)
```


Then the following form is entered:

```
(find-all-symbols "FOO")
```

A display such as the following appears:

```
(FOO FS::FOO :FOO W::FOO ZWEI::FOO TV::FOO SYS::FOO COMPILER::FOO)
```

To illustrate how `find-all-symbols` differs from `where-is`, the following form is entered:

```
(where-is "FOO")
```

A display such as the following appears:

```
USER:FOO is accessible from package USER
W:FOO is accessible from package W
ZWEI:FOO is accessible from package ZWEI
TV:FOO is accessible from package TV
FILE-SYSTEM:FOO is accessible from package FILE-SYSTEM
SYSTEM:FOO is accessible from package SYSTEM
COMPILER:FOO is accessible from package COMPILER
KEYWORD:FOO is accessible from package KEYWORD
(:FOO COMPILER::FOO SYS::FOO FS::FOO TV::FOO ZWEI::FOO W::FOO FOO)
```

Property List Functions

25.5 The property list functions include `symbol-plist` and `fs:file-properties`.

`symbol-plist` *symbol*

[c] Function

The `symbol-plist` function returns the association list that represents the property list of *symbol*. Note that this is not the property list itself; that is, you cannot perform a `get` on it.

`fs:file-properties` *pathname* &optional (*signal-error-p* *t*)

Function

This function returns a list of property-value pairs for *pathname*. The car of the returned list is the truename of the file, and the cdr is an alternating list of properties and values.

If an error occurs while the file is being accessed or the properties are being changed, the *signal-error-p* argument controls what is done. If *signal-error-p* is `nil`, a condition object describing the error is returned; if *signal-error-p* is true, a Lisp error is signaled. If no error occurs, `fs:file-properties` returns *t*.

Print Functions

25.6 The print functions provide numerous options to print objects and function definitions. These functions include `print`, `prin1`, `pprint`, `princ`, and `pprint-def`, among others. These functions are described here along with some of the print variables that control how certain kinds of objects print. See the Output Functions section in the *Explorer Input/Output Reference* for details on the printing operation, such as printing with escaping (discussed briefly in the following paragraphs) and printing different object types.

You can set a number of print variables before calling the printer to control how certain kinds of objects print. The following descriptions explain some of the print variables that are used to control the `print` function. The *Explorer Input/Output Reference* lists several other variables in addition to these.

print-escape [c] Variable

This variable controls the escape feature during printing. (*Printing with escaping* prints special characters with a quoting symbol so that the Lisp Reader can read an object back in.) If this variable is `nil`, then escaping is not performed during printing. If it is non-`nil`, then escaping is performed during printing, and printed expressions can be read back in. The default value of this variable is `t`.

For example, suppose the Lisp Reader encounters the following token: `|1.5|`. This token is converted to the *symbol* (not the number) `1.5`. If this symbol is printed without escaping, then only `1.5` is printed, and if this symbol is read back in, it is not interpreted as a symbol but as a number. Therefore, to be interpreted as a symbol, the symbol must be printed with the escape characters `|1.5|`.

print-pretty [c] Variable

This variable controls the way a list expression is printed. If the value of this variable is non-`nil`, the expression is printed with indentation, making the expression more readable. If the variable's value is `nil`, then a list expression is printed without indentation. The default value of this variable is `nil`.

print-base [c] Variable

This variable specifies the radix that is used to print integers and ratios; it has no effect on floating-point numbers, which are always printed in decimal notation. The default value of ***print-base*** is 10. Radices larger than 10 use alphabetic characters for digits larger than 9.

print-radix [c] Variable

If non-`nil`, this variable specifies that integers and ratios be printed with a prefix or suffix (which is a lowercase letter) indicating the radix used. For example:

```
; Current radix is 24 and integer 19 and 24 (decimal) are printed as:
#24rJ
#24r10
```

The radix specifiers printed for bases 2, 8, and 16 are `#b`, `#o`, and `#x`, respectively. For integers in base 10, a trailing decimal point is printed instead of using a leading radix specifier, but for ratios in base 10, the leading radix specifier is used. The default value of ***print-radix*** is `nil`.

The following are several of the print functions:

prin1 <i>object</i> &optional <i>output-stream</i>	[c] Function
print <i>object</i> &optional <i>output-stream</i>	[c] Function
pprint <i>object</i> &optional <i>output-stream</i>	[c] Function
princ <i>object</i> &optional <i>output-stream</i>	[c] Function

These functions take an optional argument called *output-stream*, which is where to send the output. If unsupplied or *nil*, the argument defaults to the value of **standard-output**. If it is *t*, the value of **terminal-io** is used. Note that **terminal-io** should never be explicitly passed as a stream argument that could cause a locked window situation. The following list describes the differences between these functions:

prin1 — Using escape characters, the **prin1** function prints the printed representation of *object* to *output-stream* with escaping, if needed. The value returned by **prin1** is *object*.

print — The **print** function is similar to **prin1**, but its output (*object*) is preceded by a **Newline** character and is followed by a space (see the **terpri** function in the *Explorer Input/Output Reference*).

pprint — The **pprint** function is similar to **print** but omits the space after *object*, which is printed when the **print-pretty** flag is set to *t*. This function produces readable indented output but returns no values.

princ — The **princ** function is similar to **prin1** but performs no escaping. Thus, the printed representation of a symbol is the same as its print name, and the printed representation of a string omits the preceding and trailing double quotation marks. Generally, output from **princ** is intended to be used as messages to users, and output from **prin1** is intended to be read back into the system as code. The value returned by **princ** is *object*.

Examples: Suppose the following code is entered:

```
(setq list `(.tv:selected-window "string with \"double quotes\""))
```

Entering `(prin1 list)` prints the objects with quoting characters so that the Lisp Reader can read the objects back in, as follows:

```
(prin1 list)(#<ZMACS-WINDOW-PANE Zmacs Window Pane 1 14613713 exposed>
"string with \"double quotes\"")
```

Entering `(print list)` inserts a **Newline** character before printing with quoting characters, as follows:

```
(print list)
(#<ZMACS-WINDOW-PANE Zmacs Window Pane 1 14613713 exposed> "string with
\"double quotes\"")
```

Entering `(princ list)` prints without quoting characters so that the output is more readable, as follows:

```
(princ list)(Edit: *Buffer-1* string with "double quotes")
```

pprint-def *function-spec...*

Special Form

The **pprint-def** special form prints the definitions of one or more functions, with indentation to make the code more readable. Certain other pretty-printing transformations are performed: The **quote** special form is represented with the `'` character. Displacing macros are printed as the original code rather than the result of macro expansion. The code resulting from the backquote (`'`) Reader macro is represented in terms of `'`.

The arguments to **pprint-def** are the function specifications whose definitions are to be printed. The **pprint-def** special form normally is used with a form (such as `foo`) to print the definition of `foo`.

If any of the arguments to **pprint-def** are bound, **pprint-def** also pretty prints the value of that symbol as well as the function definition on that symbol. Definitions are printed as **defun** special forms, and values are printed as **setq** special forms.

Example:

```
(setq foo "something")

(defun foo ()
  (format t "this is fun"))
```

Entering `(pprint-def foo)` displays a `setq` form for the value of `foo` and a `defun` form for the function definition, as follows:

```
(SETQ FOO ' "something")
(DEFUN FOO NIL
  (FORMAT T "this is fun"))
```

If a function is compiled, **pprint-def** indicates this fact and attempts to find the previous interpreted definition by looking on an associated property list (see **uncompile** in the *Explorer Lisp Reference*). However, **pprint-def** can only do so if the function's interpreted definition was once in force. If the definition of the function was simply loaded from an `xld` file, **pprint-def** cannot find the interpreted definition, and therefore cannot return any useful information.

With no arguments, **pprint-def** assumes the same arguments as when it was last called.

Dribble File Functions

25.7 An Explorer dribble file records all terminal interaction performed at a session; that is, it contains all keyboard type-in and most system typeout. To open a dribble file, you enter a form such as the following in a Lisp Listener:

```
(dribble "lm:misc;dribble.text")
Entering Dribble Read-Eval-Print Loop.
Type (DRIBBLE) to exit.
```

The following example shows how the dribble file might appear after the `copy-directory` and the `dribble-end` functions have been executed:

```

> (copy-directory "lima:fonts;" "romeo:fonts;")
Romeo: FONTS; COURIER.XFASL#1 → [Different file exists at target]
Romeo: FONTS; GWIN-MOUSE.XFASL#2 → [Already Exists]
Copied Lima: FONTS; ILGREEK10.XFASL#3 to Romeo: FONTS; ILGREEK10.XFASL#3
Byte size 16, Characters NIL
Copied Lima: FONTS; ILMATHA10.XFASL#1 to Romeo: FONTS; ILMATHA10.XFASL#1
Byte size 16, Characters NIL
Copied Lima: FONTS; ILMATHE10.XFASL#1 to Romeo: FONTS; ILMATHE10.XFASL#1
Byte size 16, Characters NIL
Copied Lima: FONTS; ILOLD.XFASL#1 to Romeo: FONTS; ILOLD.XFASL#1
Byte size 16, Characters NIL
Copied Lima: FONTS; ILSYMBOLS10.XFASL#4 to Romeo: FONTS; ILSYMBOLS10.XFASL#4

Byte size 16, Characters NIL
(#CFS:LISPM-PATHNAME "Lima.ti-7: FONTS; ILGREEK10.XFASL#3"␣)
#CFS:LISPM-PATHNAME "Lima.ti-7: FONTS; ILMATHA10.XFASL#1"␣)
#CFS:LISPM-PATHNAME "Lima.ti-7: FONTS; ILMATHE10.XFASL#1"␣)
#CFS:LISPM-PATHNAME "Lima.ti-7: FONTS; ILOLD.XFASL#1"␣)
#CFS:LISPM-PATHNAME "Lima.ti-7: FONTS; ILSYMBOLS10.XFASL#4"␣)

> (dribble-end)

```

When you enter the **dribble-end** function to close the dribble file, a message such as the following appears:

```

Closing dribble file, Romeo: MISC; DRIBBLE.TEXT#1.
NIL

```

The following functions are used to control the dribble file.

dribble *&optional filename* [c] Function

The **dribble** function opens *filename* as a *dribble file*. The function rebinds ***standard-input*** and ***standard-output*** so that all of the terminal interaction is directed to the file as well as to the terminal. Also see **dribble-all**, which follows shortly.

Calling **dribble** again without supplying an argument closes the dribble file.

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

The **dribble-start** function opens *filename* as a *dribble file*. The function rebinds ***standard-input*** and ***standard-output*** so that all of the terminal interaction is directed to the file as well as to the terminal. If *editor-p* is non-nil, then instead of opening *filename* on the host specified by the file name, **dribble-start** writes into a Zmacs buffer whose name is *filename*, creating it if it does not exist. Also see **dribble-all**, which follows shortly.

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

The **dribble-all** function is like **dribble-start** except that all input and output goes to the dribble file, including break loops, queries, warnings, and sessions in the debugger. This function works by binding ***terminal-io*** instead of ***standard-output*** and ***standard-input***. If *editor-p* is non-nil, then instead of opening *filename* to the host specified by the filename, **dribble-start** writes into a Zmacs buffer whose name is *filename*, creating it if it does not exist.

dribble-end Function

This function closes the *dribble-file* opened by **dribble**, **dribble-start**, or **dribble-all**, and then resets the I/O streams.

NOTE: The Zmacs META-X Ztop Mode command activates a major mode in Zmacs that is similar to a dribble file. This mode simulates a real-eval-print loop within a Zmacs buffer, allowing you to have a Lisp Listener within an edit buffer. You can then save the buffer to a file.

Environment Functions and Variables

25.8 The environment functions and variables provide information about your environment, such as the Lisp implementation you are using, machine type and version, software type and version, and the documentation for a symbol.

documentation name *doc-type*

documentation name &optional *doc-type*

[c] Function
Function

This function returns the documentation string of *name* in the context of *doc-type*. Common Lisp requires that both arguments be symbols; however, on the Explorer system only the print name matters, so keywords and strings are acceptable. When *doc-type* is **function**, then *name* can be any function spec. A function spec that is not a symbol does not conform to Common Lisp specifications but is allowed as an Explorer extension. For the Explorer extension version of **documentation**, *doc-type* defaults to **'function**.

Documentation strings for any *doc-type* can be added and updated by using the **setf** form. For example:

```
(setf (documentation symbol-name doc-type-symbol)
      "<new documentation string>")
```

The following documentation types are supported on the Explorer system:

variable — Documentation of the *name* argument as a special variable. Such documentation is recorded automatically by **defvar**, **defconstant**, and **defparameter**. This type is defined by Common Lisp.

type — Documentation of *name* as a type for the function **typep**. Recorded automatically when a documentation string is given in a **deftype** form. This documentation type is defined by Common Lisp.

structure — Documentation of *name* as a **defstruct** type. Recorded automatically by **defstruct** for *name*. This documentation type is defined by Common Lisp.

setf — Documentation about what it means to call a **setf** form on an object that starts with *name*. Recorded when there is a documentation string in a **defsetf** of *name*. This documentation type is defined by Common Lisp.

flavor — Documentation of the flavor specified by *name*. This documentation was installed in the flavor by the **:documentation** option in a **defflavor** for *name*.

resource — Documentation of the resource specified by *name*. This documentation is installed when there is a documentation string in a **defresource** of *name* (see the *Explorer Lisp Reference* for information on **defresource**).

signal — Documentation for *name* as a signal name. This documentation is installed when there is a documentation string in a **defsignal** or **defsignal-explicit** for *name* (see the *Explorer Lisp Reference* for information on **defsignal** and **defsignal-explicit**).

In Zmacs, CTRL-SHIFT-D invokes the **documentation** function on the function where the keyboard cursor is located; CTRL-SHIFT-V invokes the **documentation** on the variable where the keyboard cursor is located.

features [c] Variable

The value of this variable is a list of atoms that describes the software and hardware features of a Lisp implementation. On the Explorer system, by default, this list is the following:

```
(:ti :explorer :common-lisp :ieee-floating-point :lispm :flavors :defstruct
:loop :chaos :sort :fasload :string :newio :trace :grindef)
```

Most important is the symbol **:lispm**. This indicates that the program is executing on a Lisp machine; **:explorer** indicates the type of hardware; **:chaos** indicates that the Chaosnet protocol is available; and **:common-lisp** indicates that Common Lisp is supported.

The **#+** and **#-** Reader-macro constructs check for the presence of an element in this list. Thus, **#+lispm** when read by a Lisp machine causes the next expression to be significant, because **:lispm** is present in the features list (see the Input Functions section in the *Explorer Input/Output Reference* for more details on Reader macros).

user-name [c] Function

This function returns a string that identifies the user of the current Common Lisp environment. On the Explorer system, this string is the name specified by the most recent call to the **login** function. If **user-name** cannot determine the user, it returns **nil**.

lisp-implementation-type [c] Function

This function returns a string that states what kind of Lisp implementation you are using.

lisp-implementation-version [c] Function

This function returns a string that states the version numbers of the Lisp implementation.

machine-type [c] Function

This function returns a string that states what kind of hardware is used.

machine-version [c] Function

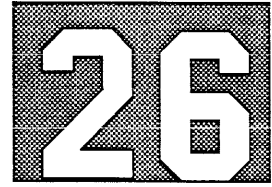
This function returns a string that describes the kind of hardware and microcode version.

machine-instance [c] Function

This function returns a string that states the name of the particular machine.

- software-type** [c] Function
This function returns a string that describes the type of operating system software the Lisp environment is working with.
- software-version** [c] Function
This function returns a string that describes the version number of the operating system software in use.
- short-site-name** [c] Function
This function returns a string that briefly states the name of the site where you are. A *site* is an institution that has a group of Lisp machines.
- long-site-name** [c] Function
This function returns a string that completely states the name of the site where you are.
- identity object** [c] Function
This function is a general identity operation. It returns *object*. Often, this function is useful as an argument to a function that requires a dummy functional argument.
- user-id** Variable
The value of this variable is a string that indicates who is logged in to this local host.
- sys:local-host-name** Variable
The value of this variable is a string that contains the host name for the local host. This value comes from the host definition in the namespace.
- sys:local-pretty-host-name** Variable
The value of this variable is a string that contains a pretty-printed version of the host name. This value comes from the host definition in the namespace.
- sys:local-finger-location** Variable
The value of this variable is a string that indicates where this host is located. This value comes from the host definition in the namespace.
- sys:local-host** Variable
The value of this variable is the host object for the local host.
- sys:associated-machine** Variable
The value of this variable is the host object of the default machine on which to log in from this Lisp machine.

LISP LISTENER AND BREAK



On the Explorer, a user normally interacts with Lisp through a top-level read-eval-print loop. This loop is the highest level of control and consists of an infinite loop that reads an expression, evaluates it, and prints the results. This is exactly what happens in the Lisp Listener window and the break window. The following variables and functions are used and maintained by the top-level loop.

sys:lisp-top-level Function

This is the first function called in the initial Lisp environment. It calls `lisp-reinitialize`, clears the screen, and calls `sys:lisp-top-level1`.

sys:lisp-reinitialize Function

This function performs a wide variety of operations, such as resetting the values of various global constants and initializing the error system.

sys: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 print escaping) to `*standard-output*`, and repeats indefinitely. If several values are returned by the form, all of them are printed. Also, the values of the variables `*`, `+`, `-`, `/`, `//`, `///`, `++`, `**`, `+++`, and `***` are maintained.

break *&optional format-string &rest format-args* [c] Function

The function `break` is used to enter a breakpoint loop, which is similar to a Lisp top-level loop. The arguments *format-string* and *format-args* are passed to the function `format` to print a *message* of the following form:

Breakpoint *message* Resume to continue, Abort to quit.

A difference between a break loop and the top level loop is that when reading a form, the `break` function checks for the following special cases:

- If the ABORT key is pressed, control is returned to the previous break or error handler or to the top level if there is no previous break or error handler.
- If the RESUME key is pressed, the function `break` returns nil.
- If the form (`return form`) is typed in the `break` function, the *form* is evaluated and the result is returned without calling the function `return`.

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

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 can execute **push** on elements in this list (adding to the front of it) but should not replace the list wholesale because several of the variable bindings on this list are essential to the operation of **break**.

- [c] Variable

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

+ [c] 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.

++ [c] Variable

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

+++ [c] Variable

The variable +++ holds the previous value of ++.

* [c] Variable

While a form is being evaluated by a read-eval-print loop, * is set to the result printed the last time through the loop. If there were several values printed (because of a multiple-value return), * is bound to the first value.

** [c] Variable

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

*** [c] Variable

The variable *** holds the previous value of **.

/ [c] Variable

// [c] Variable

/// [c] Variable

The / variable is similar to * but / contains all the results (instead of just the first result) of the last iteration of the read-eval-print loop for a particular form. These results are stored as a list. Every time / is updated, its previous value is stored in //, whose previous value is stored in ///.



PERFORMANCE TOOLS

Overview

27.1 The performance tools discussed in this section are the metering system, the timing macros, and the function histogram utility. The difference between these three performance tools is a matter of focus on the problem you are trying to solve:

- If you need very detailed, low-level information for a well-defined section of the code (or for a relatively small processing task), then metering is a good candidate.
- If you want to know more global information about the execution characteristics of an arbitrarily large (or small) amount of processing, then the timing macros are what you want.
- If you are only interested in function-calling statistics, the function histogram utility is the best choice because of less overhead.

The data collected by these performance tools is as accurate as possible because system overhead is measured and normalized.

Metering Overview

27.1.1 The metering system allows you to perform a detailed analysis of your program's execution. It is primarily a performance tool for evaluating the execution efficiency of a section of code. It can tell you which function (or path of functions) executes most frequently, uses the most real time or CPU time, conses the most words, or causes the most page faults. This information is important for determining where to start to improve the execution characteristics of the code.

In addition, you can use metering to analyze the program control flow either to determine if it can be cleaned up (made more efficient or less convoluted) or to understand what is unexpected or wrong about how the program is passing control between various components. The tree-related analysis routines of the metering system (particularly the interactive Call Tree Inspector) are very important for the latter analysis.

Because of both the time and disk space considerations for running metering, it is not feasible to meter a large or long-running piece of code. Metering is useful when you have a relatively small problem to analyze.

Timing Macros Overview

27.1.2 The timing macros, in contrast to metering, are efficient in time and disk space usage, and they can be wrapped around any arbitrarily large piece (or pieces) of code. They provide a more global picture of the performance of the code. Primarily, you use the timing macros to see how much time a particular piece of code is taking. You can also use the timing macros to focus on the section of a system that is using the most resources, which you can then meter in order to obtain a more detailed look at the code.

One of the timing macros provides a label feature that you can use to obtain a finer resolution on which code paths in a problem are taking more time, disk, or memory. You wrap this macro around numerous branches of the expected execution tree and use the label feature to identify each branch in the results.

Function Histogram Overview

27.1.3 The function histogram utility provides a statistical sampling of function calling over a certain time span. The only information collected is the number of times a specified function was called. The function histogram utility does not collect information on timing, consing, page faults, and function calling sequences. To collect information such as this, you must use metering and/or the timing macros. If you are only interested in function-calling statistics, the function histogram utility is the best choice because of less overhead. By using the Peek interface to the function histogram utility, you receive a real-time, function-calling analysis, which is not possible with metering.

Metering

27.2 The metering system allows you to determine which parts of your program use the most resources. When you run your program with metering, every function call and return is recorded, together with the time at which it took place. Page faults and words consed are also recorded. Afterward, the metering system analyzes the records and tells you how much time was spent executing within each function. The records are stored in a meter partition on disk.

Metering is designed to give extremely low-level, detailed information about the execution of a piece of code. It uses a lot of system resources in order to do this. As a rule of thumb, you should allocate 350 disk blocks per second of normal execution time when setting up a meter partition, but for code that is function-calling intensive, metering may require as much as 600 blocks per normal execution second.

Normal execution time differs from the execution time that includes metering. Metering a piece of code slows down its real time execution by a very large factor. Note that this slowdown does not adversely affect the real time and CPU time that metering reports for the functions. (These times are as accurate as possible.) The slowdown in time occurs while metering is gathering information and writing it out to the disk. The metering system does not count this time into the reported times for the functions. However, for a program that is function-calling intensive, it may take as much as 45 times longer to run the program with metering than to run the program without metering. For a simple program with relatively few function calls, such as `(meter (print 'foo))`, it only takes about two times longer to meter the print.

Because of both the time and disk space considerations for running metering, it is not feasible to meter a large or long-running piece of code. Metering is useful when you have a relatively small problem to analyze.

Do not confuse meters with metering. Metering provides accurate, detailed performance information about a relatively small unit of code. Meters are several counters that the microcode updates. You can see these in Peek by clicking on Counters in the Modes menu. The timing macros, which are discussed later in this section, read these counters and are designed to do so in the most accurate way possible.

The following topics are covered in the discussion of metering:

- Setting up a metering partition — Tells how to set up a meter partition, which is required for metering.
- Controlling metered data — Tells how to control what kind of metered data is recorded.
- Evaluating forms with metering — Tells how to use the `meter` macro to evaluate forms with metering.
- Analyzing the metered data — Tells how to analyze the metered data. One utility for analyzing metered data is the Call Tree Inspector, which is an interactive utility that allows you to view and inspect the call tree of a function.
- Resuming garbage collection — Tells how to turn garbage collection back on; it is turned off when you turn on metering.
- Customized metering sessions — For almost any metering session, the `meter` macro does everything you want. However, other options are available that allow you to customize your metering sessions.
- Metering examples — Provides several metering examples.

The following list describes a typical metering session (after you have set up a meter partition):

1. Evaluate your form with metering, for example:

```
(meter (inspect 'foo))
```

2. Execute the `meter-analyze` function to analyze the metered data, and choose `TREE-INSPECT` for the Output Type.

Specifying `TREE-INSPECT` invokes the Call Tree Inspector.

3. Use the Call Tree Inspector to view and inspect the call tree of the metered function.

Figure 27-1 shows one of the displays you may see in the Call Tree Inspector when examining the metered data.

Figure 27-1 Sample Call Tree Inspector Display

Level	function-name	Real t	Run t	faults	Inclusive	Called	real	Called	run	Called	faults
[0]	Sys:*Eval	55	55	0	4,996,950	4,996,950	0	4,996,950	4,996,950	0	0
[1]	Sys:*Eval	55	55	0	4,996,950	4,996,950	0	4,996,950	4,996,950	0	0
[1]	Lisp:Progn	155	155	0	165	165	0	165	165	0	0
[2]	Sys:*Eval	217	217	0	4,996,895	4,996,895	0	4,996,895	4,996,895	0	0
[3]	Lisp:Setq	42	42	0	4,996,678	4,996,678	0	4,996,678	4,996,678	0	0
[3]	Sys:*Eval	12	12	0	12	12	0	12	12	0	0
[4]	Sys:*Eval	291	34,078	0	4,996,624	4,996,624	0	4,996,624	4,996,624	0	0
[4]	Lisp:Inspect	54	54	0	54	54	0	54	54	0	0
[5]	Tv:Find-Or-Create-Window	909	909	0	4,996,279	4,996,279	0	4,996,279	4,996,279	0	0
[5]	(:method Tv:Inspect-Frame :Inspect-Object)	938	938	0	4,378,855	4,378,855	0	4,378,855	4,378,855	0	0
[5]		252	252	0	616,515	616,515	0	616,515	616,515	0	0

Level	function-name	Real t	Run t	faults	Inclusive	Called	real	Called	run	Called	faults
[4]	INSPECT	909	909	0	12,649,059	4,995,370	0	4,995,370	4,995,370	0	204

Call the Inspector to inspect OBJECT. Selects an Inspector window.
 The Inspector runs in its own process, so your special variable bindings will not be visible.
 If you want to see special variable bindings, use INSPECT*.
 If you type END in the inspector, the value of will be returned from the function INSPECT.
 Call-tree inspector

L: Expand call, L2: Expand all, M: Display call, M2: edit, R: Contract call
 The FND ABORT Control-V and Meta-V keys work as you would expect. "P" prints the call tree.

**Setting Up a
Meter Partition**

27.2.1 Before you run metering, you need to set up a meter partition. Use the `sys:edit-disk-label` function to add a meter partition. Here are some general rules of thumb:

CAUTION: You should not attempt to edit a disk label unless you are skilled at manipulating field-level components. If you write an incorrect disk label over the current disk label, you may not be able to access the contents of the disk again. If you are working with a disk label that includes information you do not want to lose, you should print a copy of the current disk label. Knowing the current location of the partitions on the disk may enable you to recover the disk contents should an error occur. Refer to the Explorer Input/Output Reference for details on editing a disk label.

- You can name the meter partition anything you want (METR is normally used), but the attribute must say `Meter Band`.
- If you have a spare load partition, you can convert it to a meter partition. Note that you can no longer use the load partition.
- Make the partition at least 10,000 blocks long. However, metering may take much less space than this. As long as the partition size is sufficient for the task, metering provides accurate results.
- You can have multiple meter partitions. Later, when you analyze the metered data, you can specify which meter partition you want to use.
- You will need about 350 blocks for every second of program run time. However, this guideline varies depending on whether the code is compute intensive or function-calling intensive. If your code is compute intensive, metering requires fewer blocks. If your code is function-calling intensive, metering may require as many as 600 blocks per normal execution second.

Controlling Metered Data 27.2.2 The `sys:%meter-micro-enables` variable allows you to control what kind of metered data is recorded. When you meter a form with `meter`, you can accept the default value (`#o14`) for this variable or you can specify one of the other values described here.

`sys:%meter-micro-enables`

Variable

This fixnum variable enables the recording of metered data. Each bit controls the recording of one kind of event:

- `#o1` — This bit enables recording of page faults: page in (PAGI) and page out (PAGO). Expert users should also see the notes on page faults later in this numbered paragraph.

NOTE: Page faults (bit `#o1`) and words consed (bit `#o2`) share the same column of meter reports. You must remember if you had bit `#o2` set during the metering session because the report will indicate words consed instead of page faults.

- `#o2` — This bit substitutes words consed for page faults in meter reports.

If you disable the recording of page faults (by setting bit `#o2`), you can still obtain some indication of the paging in a routine. The difference in time between real time and run time is the disk wait time for paging operations.

- `#o4` — This bit enables recording of function entry (CALL), function exit (RET), stack group switching (UNWIND), and recursive function entry (RECL). The RECL type is recorded for tail recursive calls. (Remember that the compiler eliminates tail recursion when possible, unless instructed otherwise.)
- `#o10` — This bit enables recording of stack group switching (NEW Stack Group).

The value of the `sys:%meter-micro-enables` variable is normally zero, which turns off all recording.

The value that you specify can be cumulative. Some example values are as follows:

- `#o14` — Controls a *typical* metering session; records function entry/exit, stack group switching, and recursive function entry.
- `#o4` — Enables the recording of function entry/exit, stack group switching, and recursive function entry.
- `#o16` — Collects standard metered data but substitutes consing information for page fault information.

If you do not set the `sys:%meter-micro-enables` variable when you use the `meter` macro, the default of `#o14` is used.

Notes for Expert Users on Page Faults The page fault information produced by setting bit #01 is in addition to the page fault information you receive for functions when collecting the default information generated for the default event types. By enabling page fault as an event, you receive more detailed information about individual page faults. This causes metering to write out one of its 7-word blocks whenever a page in (PAGI) or page out (PAGO) occurs.

This information, when examined using the LIST-EVENTS analyzer (described later), gives extremely detailed information about every page fault that occurred during the execution of your test. However, it is so detailed that it is virtually unusable unless you write a sophisticated post-processor to sort out what you were actually trying to find out.

The page fault information that appears with the standard reports is collected regardless of whether you enable page faults as an event.

Evaluating Forms With Metering

27.2.3 To evaluate a form with metering, you use the following **meter** macro:

meter &body *body*

Macro

This macro executes *body* with metering enabled in the current stack group only. This is typically what you want because including measurements from other stack groups would require extra disk space to collect data you do not want and your reports would be longer in the analysis phase, including one report for each stack group.

- The meter partition is reset (that is, the previous metered data is cleared).
- All interrupts are turned off except for keyboard interrupts. This keeps processing in this process but allows you to use CTRL-ABORT to abort the metering.
- In the *body*, you can supply an optional value for **sys:%meter-micro-enables** and only one top-level form. If the first form in *body* is a fixnum, it is used as the value for **sys:%meter-micro-enables**. The default is #014 (record function entry/exit, stack group switching, and recursive function entry). After metering, the variable is reset to 0.

Example: (meter #016 (inspect 'foo))

NOTE: If you cold boot after you execute the **meter** macro, you will not be able to analyze the data.

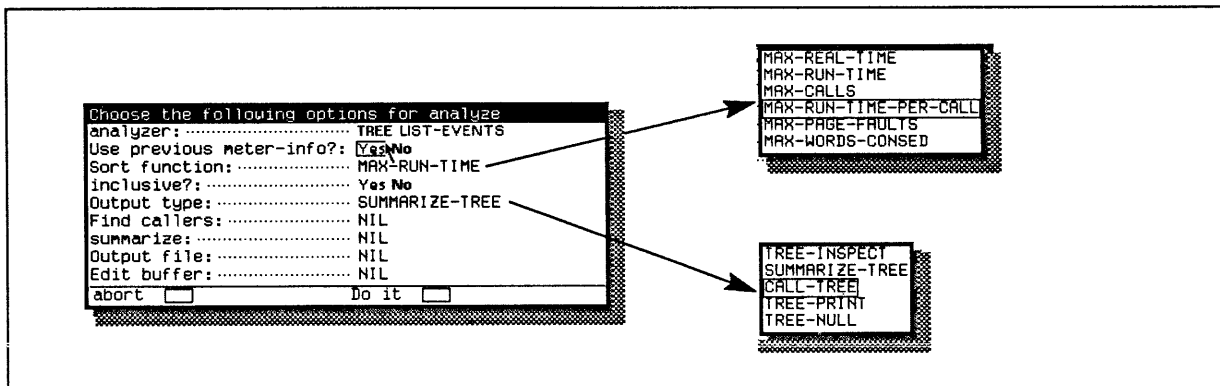
For almost any metering session, the **meter** macro does exactly what you want. However, if you want more options than what **meter** provides, refer to paragraph 27.2.6, Customized Metering Sessions.

Analyzing the Metered Data

27.2.4 The `meter-analyze` function provides a menu-based interface that allows you to specify how you want the metered data displayed. Several options are available. For example, the `TREE-INSPECT` option of Output Type invokes the Call Tree Inspector, which is an interactive utility that allows you to view and inspect the call tree of a function.

Figure 27-2 shows the menu that appears when you execute the `meter-analyze` function. If you have multiple meter partitions, you are prompted to specify the one you want.

Figure 27-2 Meter-Analyze Menu



Most of the operations on this menu are obvious. The simplest way to learn to use them is to experiment with the menu. If you need details on an option, refer to the following paragraphs.

Some of these options are mutually exclusive. If you use options that do not work together, you can reinvoke the menu, specify `yes` for Use previous meter-info?, and try again. The metered data will still be intact. The following list details which options do not work together:

- **LIST-EVENTS Analyzer:**
 - You should take the defaults for all prompts except Use Previous Meter Info?, Output File, and Edit Buffer, which you can set. However, note that Output File and Edit Buffer are mutually exclusive.
 - Sort Function has no meaning here.
 - Output Type must be set to `SUMMARIZE-TREE`.
 - Find Callers and Summarize should be `nil`.
- Sort Function or Inclusive? — The Output Type must be set to `SUMMARIZE-TREE`.
- Find Callers or Summarize — Output Type must be set to `SUMMARIZE-TREE`, and they cannot be used with `LIST-EVENTS`.
- Output File and Edit Buffer — These are mutually exclusive. Also, the Output Type cannot be set to `TREE-INSPECT`.

The following paragraphs discuss these options in detail. Following these details are examples of using the Call Tree Inspector.

Analyzer **27.2.4.1** The Analyzer option offers a choice between two different types of analyzers, TREE and LIST-EVENTS:

TREE — The default analyzer TREE prints out the amount of run time and real time spent executing each function that was called. The run time is the CPU time only, excluding the disk wait time. The real time includes time spent waiting and time spent writing metered data to disk. The following shows an example of the tree analyzer output.

```
Stack Group: MAIN-STACK-GROUP

functions          # calls      Run t   Avg Run t   consing or faults   Real t
SI::EVAL1          5           390     78          0                   505
SETQ                2           147     73          0                   236
UNWIND-PROTECT     1           130     130         0                   168
PROGN               1            54     54          0                    92
SI::INTERPRETER-FSYMEVAL  1            31     31          0                    50
SI::INTERPRETER-SET  2             0      0          0                     0

total              12          752     62          0                   1,051
```

The headings for the columns mean the following:

- # calls — The number of times a function is called.
- Run t — The CPU time (in microseconds) for the function.
- Avg Run t — Average amount of time (in microseconds) per call to a function instead of the total of all calls to that function.
- consing or faults — The number of words consed, or the number of page faults that occurred. This depends on the value of bit #02 of `sys:%meter-micro-enables` when metering was done. If set, consing information is substituted for page fault information.
- Real t — The wall clock time (in microseconds) for the function.

LIST-EVENTS — The LIST-EVENTS analyzer prints out one line about each event recorded, as shown in the following example:

```
Real t   Run t   consing   Function name   stack   function
         or   or faults   name           depth  event    or address
0         0         0 SETQ           365 RET  SI::INTERPRETER-SET
95        31         0 SI::EVAL1      336 RET  SETQ
129       46         0 PROGN         323 RET  SI::EVAL1
181       79         0 PROGN         323 CALL SI::EVAL1
259      138         0 PROGN         323 RET  SI::EVAL1
298      158         0 SI::EVAL1     294 RET  PROGN
333      173         0 UNWIND-PROTECT 272 RET  SI::EVAL1
.
.
.
```

The headings Real t, Run t, and consing or faults are the same as for TREE. The other headings mean the following:

- Function name — The name of the function being called.
- stack depth — The depth of the stack.
- event — The event types in this column depend on the value of `sys:%meter-micro-enables`. The types can include the following:
 - Page in (PAGI) and page out (PAGO) — These two types are collected when bit 1 is set.
 - Function entry (CALL), function exit (RET), function unwind (UNWIND), and recursive function entry (RECL)— These four types are collected when bit 4 is set.
 - Stack switch (NEW Stack Group) — This type is collected when bit #o10 is set.

If an event occurs that is not one of these, the entry BAD EVENT may appear in this column.

- function or address — The value of this column is based on the value of the event type in the previous column, as follows:

Event Type	Function or Address
CALL, RET, or RECL	Function name being called, function name being returned to, or tail recursive call
UNWIND	No entry
PAGI or PAGO	Virtual memory address of page event
NEW Stack Group	Name of new stack group
BAD EVENT	No entry

Use Previous Meter Info?

27.2.4.2 Analyzing the metered data involves creating a large intermediate database. You can specify whether to reuse this database when you reexecute `meter-analyze`:

- If you are metering different code from the previous time you ran the analyzer, you should specify `no`.
- If you want to look at the analyzer data in a different way on the same metered code, you should specify `yes`. You might want to use different sort functions, for example, such as `MAX-RUN-TIME` the first time you run the analyzer and then `MAX-REAL-TIME` on the next run.

Sort Function

27.2.4.3 The sort functions allow you to sort the tree analyzer display according to one of the columns in the display. The numbers are sorted from largest (max) to smallest.

- `MAX-PAGE-FAULTS` — Sorts by the number of page faults.
- `MAX-CALLS` — Sorts by the number of times a function is called.

Figure 27-2

Meter-Analyze Menu (This figure is repeated for your convenience.)

```

Choose the following options for analyze
analyzer:..... TREE LIST-EVENTS
Use previous meter-info?: YesNo
Sort function:..... MAX-RUN-TIME
inclusive?:..... Yes No
Output type:..... SUMMARIZE-TREE
Find callers:..... NIL
summarize:..... NIL
Output file:..... NIL
Edit buffer:..... NIL
abort  Do it 

```

- MAX-RUN-TIME — Sorts by the run time for the function. This is the default sort function.
- MAX-REAL-TIME — Sorts by the wall clock time for the function.
- MAX-RUN-TIME-PER-CALL — Sorts by the average amount of time per call to a function instead of the total of all calls to that function.
- MAX-WORDS-CONSED — Sorts by the number of words consed.

Inclusive? **27.2.4.4** If you specify `Yes` for `Inclusive?`, the times for each function include the time spent in executing all functions called from the function and any other functions they called. (Basically, it includes the function plus all nodes of the call tree below that function.) If a function is called recursively, the time spent in the inner call(s) is counted twice (or more).

Output Type **27.2.4.5** The tree analyzer provides several different output formats:

TREE-INSPECT — Invokes the Call Tree Inspector, which is an interactive utility that allows you to view and inspect the call tree.

NOTE: Because the Call Tree Inspector provides a large number of interactive operations, it is described later in this section after the other options on the menu. See paragraph 27.2.4.10, Call Tree Inspector.

SUMMARIZE-TREE — Prints summary information for each function.

CALL-TREE — Displays a line for each call, indenting by level.

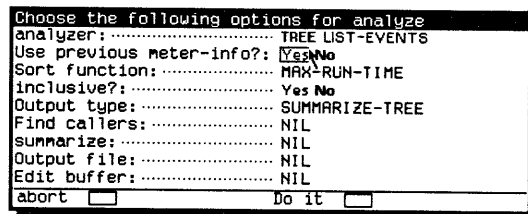
TREE-PRINT — Displays a line for each call.

TREE-NIL — Does nothing (used purely for side effect). You might use **TREE-NIL**, for example, with the Find Callers option because you do not want tree output. You want only caller information on the data collected by metering.

Find Callers **27.2.4.6** If you specify a function specification or a list of function specifications here, a list of who called the specified functions, and how often, is printed.

Figure 27-2

Meter-Analyze Menu (This figure is repeated for your convenience.)



Summarize 27.2.4.7 If you specify a function specification or a list of function specifications here, only the statistics for those functions are printed.

Output File 27.2.4.8 You can specify a file to which the analysis is written, instead of the terminal.

Edit Buffer 27.2.4.9 You can specify an editor buffer to which the analysis is written, instead of the terminal.

Call Tree Inspector 27.2.4.10 The Call Tree Inspector, which is invoked when you select TREE-INSPECT for the Output Type, is an interactive utility that allows you to view and inspect the call tree. The Call Tree Inspector displays a full screen containing three windows. The top window displays function names, their calling level, and optional timing information that is selected with the items in the middle window. The bottom window is used to display information about the functions in the top window. Figure 27-3 shows a sample Call Tree Inspector display that illustrates the use of these windows.

When a function is called several times at the same level, it is displayed once with * *n* after it, where *n* is the number of times it was called.

Figures 27-4 through 27-10 illustrate the operations you can perform in the Call Tree Inspector window. The following list summarizes these operations.

Top Window You can click on functions in the top window with the following buttons:

- L — Clicking left once (L) on a function expands the next level of functions in the tree below the level clicked on. Clicking L on a function with the * *n* format displays the *n*th repetition of that function. For example, clicking L on si:foo-bar * 5 displays:

```

si:foo-bar * 4
si:foo-bar           ⇐Data for the fifth call

```

- L2 — Clicking left twice (L2) on a function expands all levels of the tree below the level clicked on. If there are functions below this function that are called more than once, they are expanded with the * *n* format. Clicking L2 on a function with the * *n* format expands all of the calls at that level.

Figure 27-3 Call Tree Inspector Windows

		More above		
Run time	Real time	Page faults	Inclusive Run time	Inclusive Page faults
[9] Tv:Sheet-Free-Temporary-Locks			27	27
[9] (:Method Tv:Essential-Window :Alias-For-Selected-Windows)			250	250
[9] Tici:Get-Handler-For			122	122
[9] (:Method Tv:Inspect-Frame :Combined :Select)			393	32,550
[10] (:Method Tv:Window :Combined :Select)			842	842
[11] (:Method Tv:Basic-Frame :Inferior-Select)			23	23
[11] (:Method Tv:Stream-Mixin :Before :Select)			40	40
[11] (:Method Tv:Select-Mixin :Before :Select)			955	955
[11] (:Method Tv:Select-Mixin :Select)			107	107
[11] (:Method Tv:Select-Mixin :After :Select)			96	96
[11] Tv:Screen-Manage-Delaying-Screen-Management-Internal			44	44
[8] Tv:Screen-Manage-Delaying-Screen-Management-Internal			64	64
[9] Tv:Screen-Manage-Dequeue			212	212
[10] Tv:Screen-Manage-Dequeue-Entry * 5			452	452
[11] Tv:Sheet-Can-Get-Lock-Internal			39	39
[11] Sys>Delete-List-Eq			141	141
[11] Tv:Sheet-Get-Lock			96	96
[12] Tv:Sheet-Can-Get-Lock-Internal			39	39
[12] Tv:Sheet-Get-Lock-Internal			67	67
[11] (:Method Tv:Sheet :Combined :Screen-Manage)			211	211
[12] Tv:Sheet-Get-Lock			75	75
[13] Tv:Sheet-Get-Lock-Internal			61	61
[12] (:Method Tv:Sheet :Screen-Manage)			169	169
[13] (:Method Tv:Sheet :Order-Inferiors)			76	76
[14] Lisp:Stable-Sort			35	35
[13] (:Method Tv:Sheet :Screen-Manage-Autoexpose-Inferiors)			84	84
[14] Tv:Screen-Manage-Autoexpose-Inferiors			132	132
[15] Tv:Sheet-Get-Lock			81	81
[15] Tv:Sheet-Get-Lock-Internal			60	60
[15] Lisp:Nreverse			22	22
[15] Tv:Sheet-Release-Lock			94	94
[13] Tv:Screen-Manage-Sheet			549	549
[14] Tv:Canonicalize-Rectangle-Set			32	32
[14] Sys:Nconc			29	29
[14] Tv:Screen-Manage-Flush-Knowledge			26	26
[14] Tv:Sheet-Get-Lock			76	76
[15] Tv:Sheet-Get-Lock-Internal			61	61
[14] Tv:Screen-Manage-Sheet-Final			107	107
[15] (:Method Tv:Sheet :Screen-Manage-Uncovered-Area)			63	63
[16] Sys>Delete-List-Eq			73	73
[14] Tv:Sheet-Release-Lock			94	94
[12] Tv:Sheet-Release-Lock			94	94
[11] Tv:Sheet-Release-Lock			103	103
[10] Tv:Screen-Manage-Dequeue-Entry			543	543
			13,197	13,197
		More below		
Run time	Real time	Page faults	Inclusive Run time	Inclusive Page faults
The Inspector runs in its own process, so your special variable bindings will not be visible.				
If you want to see special variable bindings, use INSPECT*.				
If you type END in the inspector, the value of will be returned from the function INSPECT.				
Level function-name	Real t	Run t	faults	Called real
[5]TV::FIND-OR-CREATE-WINDOW	938	938	0	18,689,922
Expansion stopped by user.				4,977,917
Call-tree inspector				177
L: Expand call, L2: Expand all, M: Display call, M2: edit, R: Contract call				
The END ADONT Control-V and Meta-V keys work as you would expect. "P" prints the call tree.				

NOTE: You can cancel the expansion of the tree at any level by pressing any key on the keyboard. This is helpful if you accidentally click L2 on a function that is taking too long to expand, for example, a function that is called * 999 times.

- **M** — Clicking middle once (M) displays detailed metering data for the function in the bottom window (not just what you selected for the top window). It also displays the documentation for the function (like doing CTRL-SHIFT-D on a function name).

The six columns of statistics are actually two groups of three columns each. Real T, Run T, and Faults are columns whose data was collected while in this function. Called Real, Called Run, and Called Faults are columns whose data was collected while in functions called by this function.

- M2 — Clicking middle twice (M2) edits the function clicked on (that is, performs a META-. on the function). A copy of the source for the function is placed in an edit buffer. You can return to the Call Tree Inspector by pressing SYSTEM L.
- R — Clicking right once (R) collapses all levels of the tree below the function clicked on.

NOTE: Although you can press any key to stop the expansion of the call tree, you cannot press any key to stop the collapsing.

Available Keyboard Commands The following keyboard commands are available.

- END or ABORT — Exits the program
- CTRL-V — Scrolls the top window down one screen
- META-V — Scrolls the top window up one screen

Middle Window When you select items from the middle window, the information (timing or page fault) is displayed in the top window. These items mean the following:

- Real Time — The wall clock time (in microseconds) for the function.
- Run Time — The CPU time (in microseconds) for the function.
- Page Faults — The number of page faults that occurred.
- Inclusive Run Time — The CPU time for this function plus all the functions that this function calls (that is, the total time required to execute this branch of the tree).
- Inclusive Page Faults — The total number of page faults for this function plus all the functions that this function calls (that is, the total number of page faults taken while executing this branch of the tree). If bit #02 of `sys:%meter-micro-enables` was set during the metering session, this column shows words consed instead of page faults.

The numbers for the selected items appear on the right in the order in which they appear in the middle window. You can click any button to select an item; repeated clicks on an item toggle whether or not it is selected.

*Call Tree Inspector
Examples*

27.2.4.11 The following figures illustrate the operations you can perform in the Call Tree Inspector window. You can follow the steps to get similar results, or you can simply look at the examples. The specific steps are followed by explanations of general features.

1. First, you need to meter a function. Enter the following form in a Lisp Listener:

```
(meter (inspect 'foo))
```

2. Press the END key to exit the Inspector window.

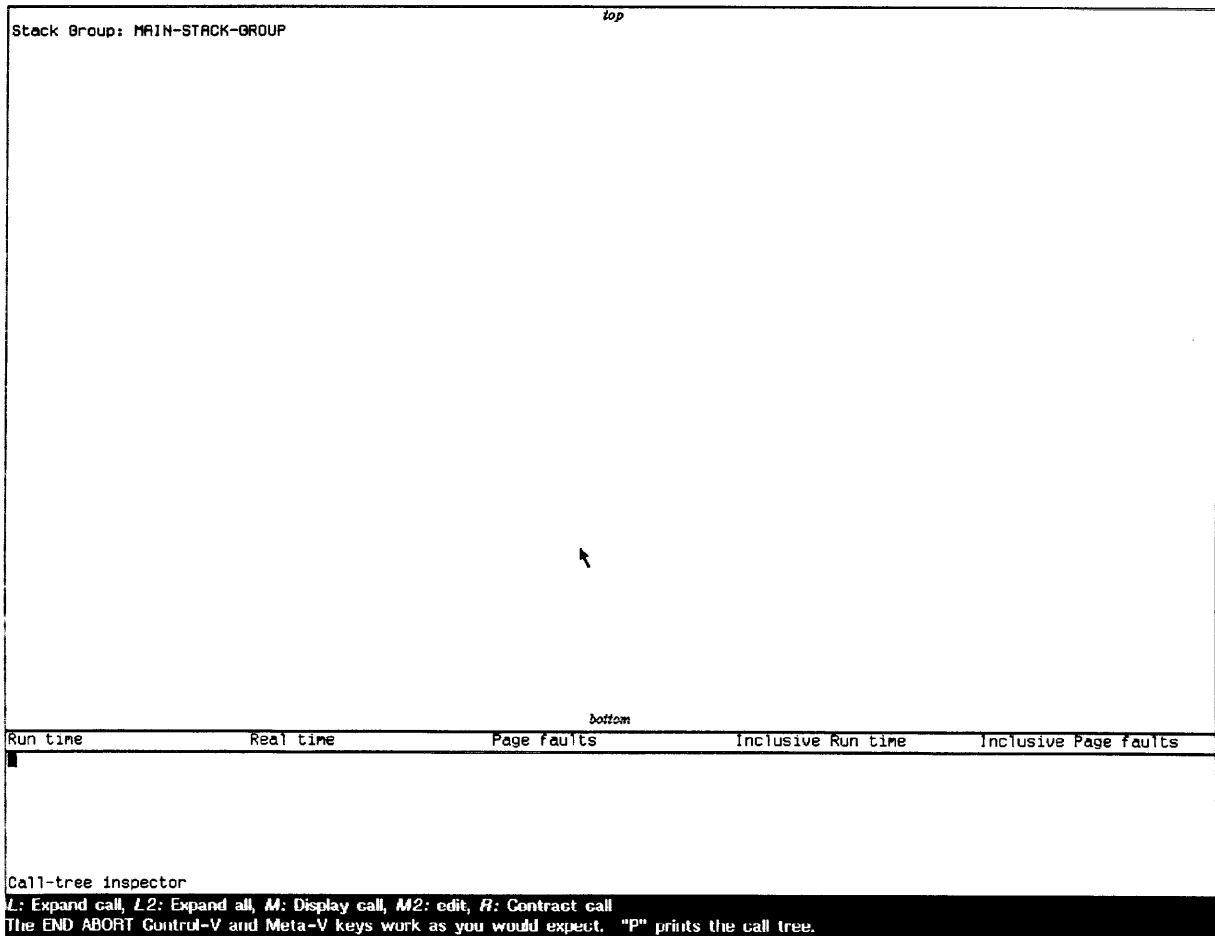
3. Enter the following form and choose TREE-INSPECT for the Output Type:

(meter-analyze)

Figure 27-4 shows the screen that first appears.

NOTE: You may not be able to exactly reproduce the function calls in these examples because there may have been modifications to the system since these examples were generated. Basically, the intention of these examples is to give you an idea of what behavior to expect from the Call Tree Inspector.

Figure 27-4 Call Tree Inspector — Initial Screen



Click L on Stack Group: MAIN-STACK-GROUP and some of the items that follow to produce a screen similar to that shown in Figure 27-5. Clicking L on a function expands the next level of functions in the tree below the level clicked on.

Figure 27-5 Call Tree Inspector — Clicking L

The screenshot shows the Call Tree Inspector interface. At the top, it displays the stack group as 'MAIN-STACK-GROUP' with a 'top' button. Below this, a list of stack frames is shown: '[0] Sys:Eval', '[1] Sys:Eval', and '[1] Lisp:Progn'. The 'Lisp:Progn' entry is selected, and a mouse cursor is pointing at it. The main area of the window is empty, indicating that the call tree has not yet been expanded. At the bottom of the window, there is a table with five columns: 'Run time', 'Real time', 'Page faults', 'Inclusive Run time', and 'Inclusive Page faults'. Below the table, the text 'Call-tree inspector' is displayed, followed by a legend: 'L: Expand call, L2: Expand all, M: Display call, M2: edit, R: Contract call'. A final line of text states: 'The END ABORT Control-V and Meta-V keys work as you would expect. "P" prints the call tree.'

Figure 27-6 shows how the screen looks after selecting Run Time, Real Time, and Inclusive Run Time from the items in the middle window. Click any button to select an item; repeated clicks on an item toggle whether or not it is selected.

The numbers for the selected items appear on the right in the order in which they appear in the middle window. In this figure, the numbers correspond to the items Run Time, Real Time, and Inclusive Run Time, respectively.

Figure 27-6 Call Tree Inspector – Middle Window Items

		<i>top</i>	
Stack Group:	MAIN-STACK-GROUP		
[0]	Sys:*Eval	55	55 4,996,950
[1]	Sys:*Eval	55	55 4,996,950
[1]	Sys:*Eval	165	165 165
[1]	Lisp:Progn	217	217 4,996,895
		<i>bottom</i>	
<input type="checkbox"/>	Run time	<input type="checkbox"/>	Real time
<input type="checkbox"/>	Page faults	<input type="checkbox"/>	Inclusive Run time
<input type="checkbox"/>		<input type="checkbox"/>	Inclusive Page faults
Call-tree inspector			
Display Run-time used by a function and the functions called by that function			

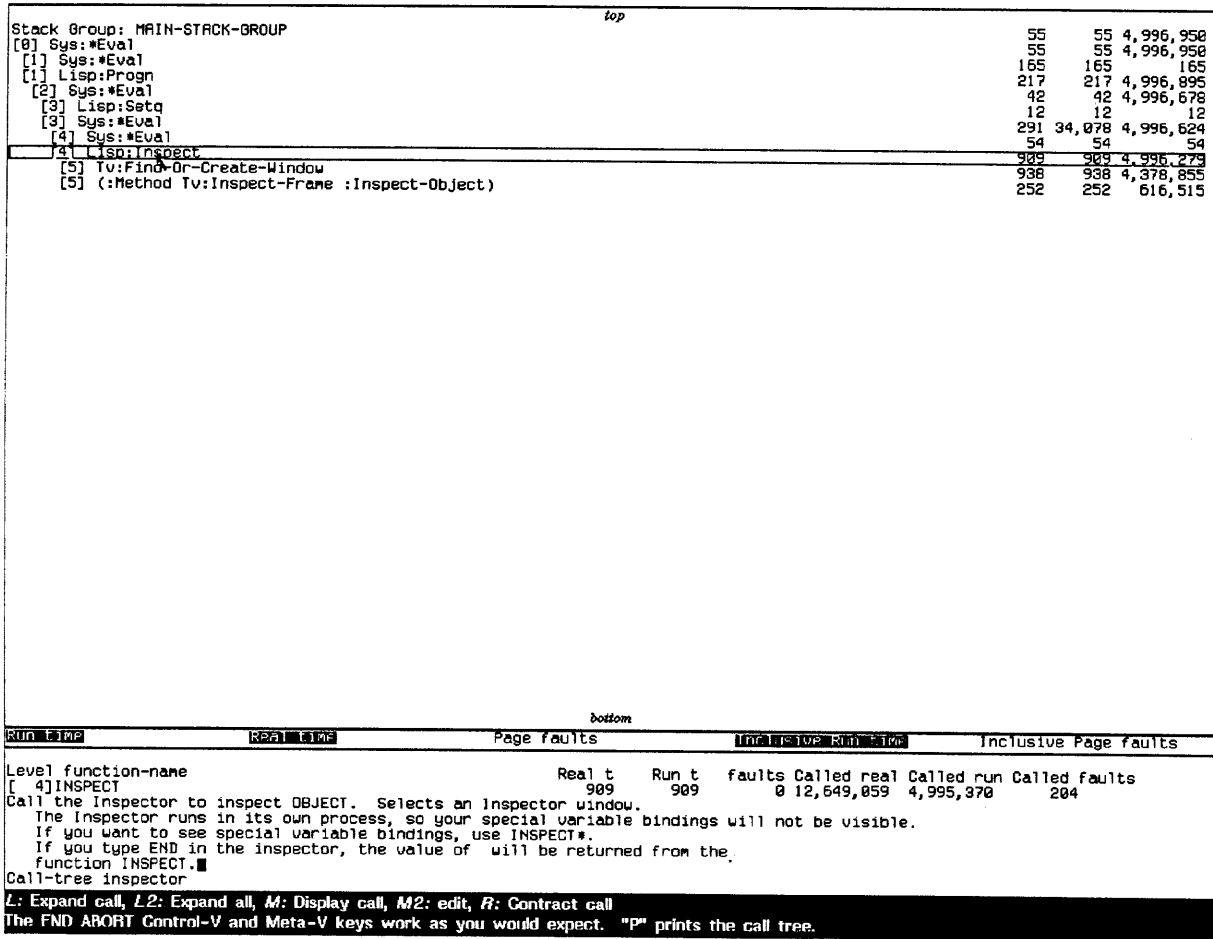
To produce a screen similar to that shown in Figure 27-7, perform the following steps:

1. Click L on some of the items until you find the function Lisp:Inspect.
2. Click M on Lisp:Inspect.

Clicking M displays detailed metering data for the function in the bottom window (not just what you selected for the top window) and the documentation for the function (like doing CTRL-SHIFT-D on a function name).

3. Click L on Lisp:Inspect.

Figure 27-7 Call Tree Inspector – Clicking L and M



To produce a screen similar to that shown in Figure 27-8, first find the function `Tv:Find-Or-Create-Window` and then perform the following steps:

1. Click M on `Tv:Find-Or-Create-Window`.
2. Click L2 on `Tv:Find-Or-Create-Window`.

Clicking L2 on a function expands all levels of the tree below the level clicked on. If functions below this function are called more than once, they are expanded with the * *n* format.

3. Cancel the expansion of the screen at any time by pressing any key on the keyboard.

There are several hundred function calls in this example, and it is not necessary to view them all. When the call tree extends beyond the bottom of the window, you can use CTRL-V and META-V to scroll. (However, for these examples, it is not necessary to scroll.)

NOTE: If you click R to retract the tree, you cannot cancel it by pressing any key.

Figure 27-8 Call Tree Inspector — Clicking L2

Run time	Real time	Page faults	Inclusive Run time	Inclusive Page faults		
Stack Group: MAIN-STACK-GROUP top						
[8] Sys:*Eval	55	55	4,996,950	950		
[1] Sys:*Eval	55	55	4,996,950	165		
[1] Lisp:Progn	165	165	165	217		
[2] Sys:*Eval	217	217	4,996,895	42		
[8] Lisp:Setq	42	42	4,996,676	12		
[3] Sys:*Eval	12	12	12	291		
[4] Sys:*Eval	291	34,078	4,996,624	54		
[4] Lisp:Inspect	54	54	54	909		
[5] Tv:Find-Or-Create-Window	909	909	4,996,279	938		
[6] Tv:Find-Window-Of-Flavor	938	938	4,378,859	1,659		
[6] Tv:Make-Window	1,659	1,659	9,832	284		
[7] (:Method Tv:Basic-Frame :Combined :Mouse-Select)	284	284	3,511,555	106		
[7] (:Method Tv:Window :Combined :Mouse-Select)	106	106	856,530	521		
[8] (:Method Tv:Basic-Frame :Inferior-Select)	521	521	856,424	24		
[8] (:Method Tv:Essential-Window :Before :Mouse-Select)	24	24	24	641		
[8] (:Method Tv:Select-Mixin :Mouse-Select)	641	641	2,579	425		
[9] (:Method Tv:Window :Combined :Activate)	425	425	826,033	66		
[9] Tv:Sheet-Within-Sheet-P	66	66	1,662	87		
[9] Tv:Sheet-Free-Temporary-Locks	87	87	169	27		
[9] (:Method Tv:Essential-Window :Alias-For-Selected-Windows)	27	27	27	250		
[9] Tvl:Get-Handler-For	250	250	400	122		
[9] (:Method Tv:Inspect-Frame :Combined :Select)	122	122	526	393		
[10] (:Method Tv:Window :Combined :Select)	393	32,550	822,824	842		
[11] (:Method Tv:Basic-Frame :Inferior-Select)	842	842	822,431	23		
[11] (:Method Tv:Stream-Mixin :Before :Select)	23	23	23	40		
[11] (:Method Tv:Select-Mixin :Before :Select)	40	40	60	955		
[11] (:Method Tv:Select-Mixin :Select)	955	955	815,845	107		
[11] (:Method Tv:Select-Mixin :After :Select)	107	107	2,366	96		
[11] Tv:Screen-Manage-Delaying-Screen-Management-Internal	96	96	3,251	44		
[8] Tv:Screen-Manage-Delaying-Screen-Management-Internal	44	44	44	64		
[9] Tv:Screen-Manage-Dequeue	64	64	27,267	212		
[10] Tv:Screen-Manage-Dequeue-Entry * 6	212	212	27,203	452		
[11] Tv:Sheet-Can-Get-Lock-Internal	452	452	26,991	39		
[11] Sys:Delete-List-Eq	39	39	39	141		
[11] Tv:Sheet-Get-Lock	141	141	141	96		
[12] Tv:Sheet-Can-Get-Lock-Internal	96	96	202	39		
[12] Tv:Sheet-Get-Lock-Internal	39	39	39	67		
[11] (:Method Tv:Sheet :Combined :Screen-Manage)	67	67	67	211		
[12] Tv:Sheet-Get-Lock	211	211	2,304	75		
[13] Tv:Sheet-Get-Lock-Internal	75	75	136	61		
[12] (:Method Tv:Sheet :Screen-Manage)	61	61	61	159		
[13] (:Method Tv:Sheet :Order-Inferiors)	159	159	1,663	76		
[14] Lisp:Stable-Sort	76	76	111	35		
	35	35	35			
More below						
The inspector runs in its own process, so your special variable bindings will not be visible. If you want to see special variable bindings, use INSPECT*. If you type END in the inspector, the value of nil will be returned from the function INSPECT.						
Level function-name	Real t	Run t	faults	Called real	Called run	Called faults
[5]TV::FIND-OR-CREATE-WINDOW	938	938	0	10,689,922	4,377,917	177
Expansion stopped by user.█						
Call-tree inspector						
L: Expand call, L2: Expand all, M: Display call, M2: edit, R: Contract call						
The LND ABORT Control-V and Meta-V keys work as you would expect. "P" prints the call tree.						

These remaining figures show the behavior of the tree when you click on functions that have been executed multiple times at a given level of the tree. Find the call Tv:Screen-Manage-Dequeue-Entry * n.

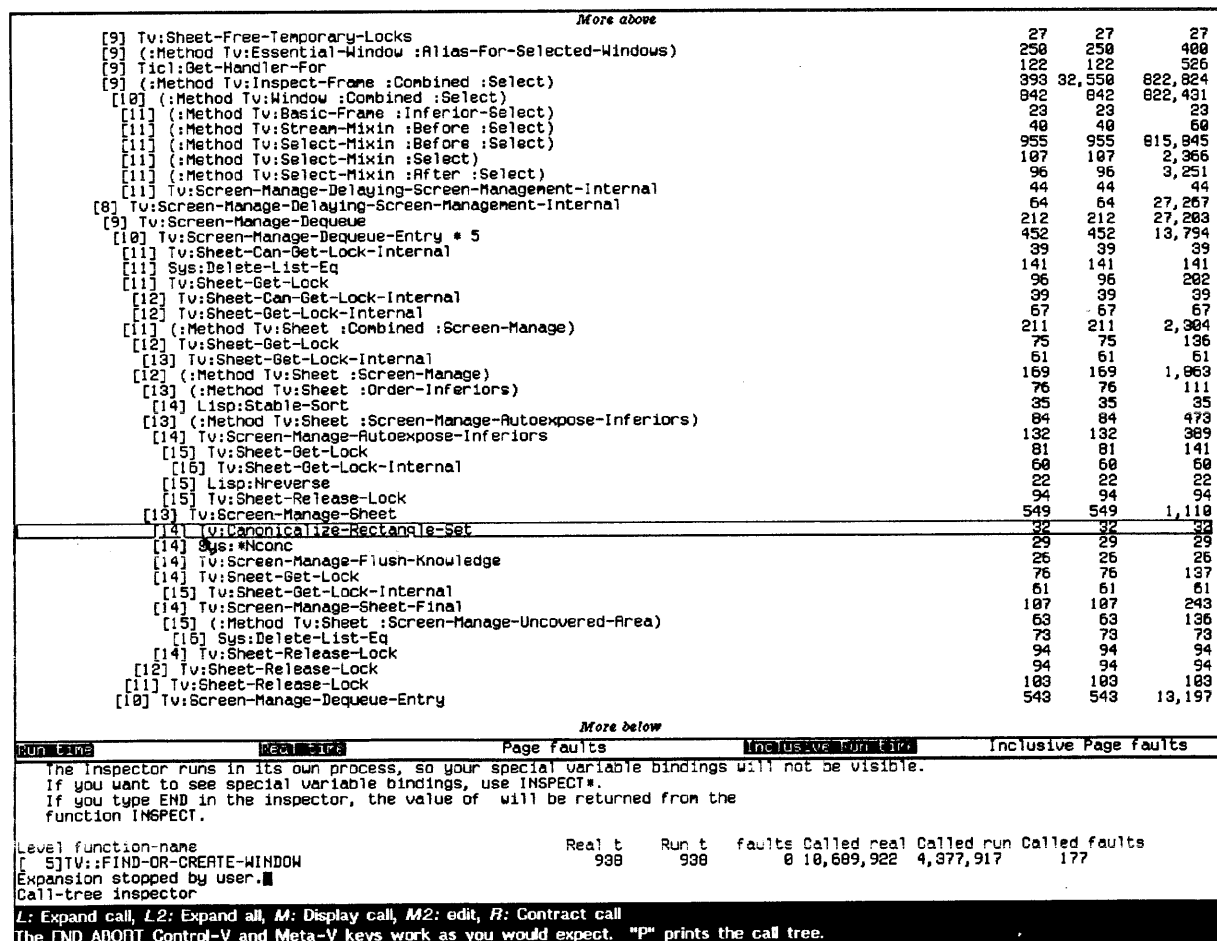
The data displayed for Tv:Screen-Manage-Dequeue-Entry * n is for the first execution of the function only. It is not inclusive for all executions at that level of the tree.

Click L on Tv:Screen-Manage-Dequeue-Entry * n to produce a screen similar to that shown in Figure 27-9.

The newly displayed version of Tv:Screen-Manage-Dequeue-Entry is for the last execution of the function. The screen now displays the data for the first and last executions.

Also notice that the documentation for Tv::Find-Or-Create-Window in the bottom window is still there. Until you click M on another function for its documentation, the previous documentation remains.

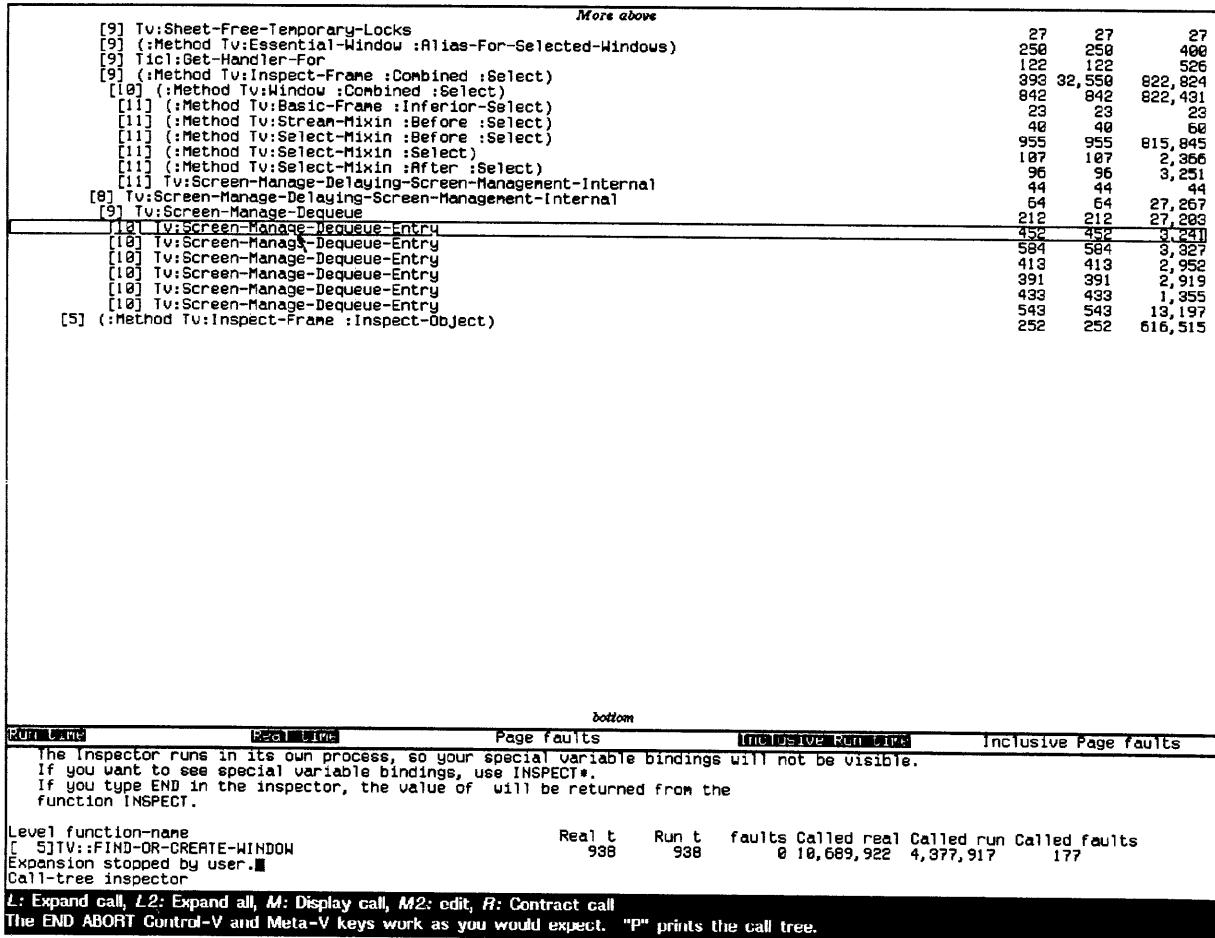
Figure 27-9 Call Tree Inspector – Clicking L on a * n Function



To produce a screen similar to that shown in Figure 27-10, click L2 on Tv:Screen-Manage-Dequeue-Entry * n.

Clicking L2 on a function with the * n format expands all of the calls at that level.

Figure 27-10 Call Tree Inspector – Clicking L2 on a * n Function



Press the END key to exit the Call Tree Inspector window, or experiment with some of the operations.

Resuming Garbage Collection

27.2.5 Because metering makes a record of pointers to Lisp objects in a disk partition that is not part of the Lisp address space, the garbage collection (GC) process must be arrested while metering is executing. Metering temporarily changes the meaning of addresses in the virtual memory. In other words, GC would fail if it tried to collect these Lisp objects because they reside outside the virtual address space managed by GC. In addition, there are pointers in the meter partition to objects in memory. If GC were allowed to move these objects, it would not be possible to analyze the collected data. GC must be off for the duration of the data collection and analysis process.

You can unarrest the GC process after collecting the metering data by using the `meter:resume-gc-process` function.

meter:resume-gc-process

Function

Allows garbage collection (GC) to continue (if it is on) by unarresting it. The GC process is unarrested with a `:revoke-arrest-reason` code of `metering`. This resumes the GC process in its previous state: with incremental GC on if a previous (`gc-on`) had been executed, or with incremental GC off. In either case, once GC is unarrested, you can perform any type of GC by using the normal GC functions.

Customized Metering Sessions

27.2.6 If you want more options than what the `meter` macro provides, you can customize your metering sessions by using the options discussed in paragraph 27.2.6.1, Evaluating Forms. The Examples paragraph (27.2.6.2) presents several different examples of these options.

The Another Meter Analysis Function paragraph (27.2.6.3) discusses the `meter:analyze` function, which also analyzes metered data. It is superseded by the menu-based `meter-analyze` function, but it is still supported. Many of the options are the same as the `meter-analyze` options.

Evaluating Forms

27.2.6.1 When you evaluate forms with metering, you have your choice of one of the following options:

- The `meter` macro, which is most commonly used.
- The `meter:test` function.
- The `meter:run` function, which is used in conjunction with the `meter:enable` and `meter:disable` functions. (Unlike `meter` and `meter:test`, the `meter:run` function does not automatically enable and disable stack groups for you.)
- As an alternative to using `meter`, `meter:test`, or `meter:run`, you can enable metering for a stack group with the `meter:enable` function, and then bind `sys:%meter-micro-enables` to a legal value.

For almost any metering session, the `meter` macro does exactly what you want. However, if you want to customize your metering sessions, you can use one of the other options.

NOTE: If you cold boot after you evaluate a form with metering, you will not be able to analyze the data.

meter:reset Function

This function resets metering and clears all metered data currently in the meter partition.

meter:test *form* &optional (*enables* #o14) Function

The **meter:test** function clears the metered data, enables metering for the current stack group only, and evaluates *form* with **sys:%meter-micro-enables** bound to *enables*. The default value of *enables* is #o14 (record function entry/exit, stack group switching, and recursive function entry).

The **meter:test** function calls the **meter** macro. It works similarly to the **meter** macro except that it is not as convenient to enter `(meter:test '(bar))` as it is to enter `(meter (bar))`.

meter:run *forms* Function

Meters one or more *forms* at a time:

- The **meter:run** function resets the meter partition.
- You must enable the stack groups first with **meter:enable** and then disable them afterwards with **meter:disable**.
- The **meter:run** function records metered data for all functions executed in any of the enabled stack groups.
- You cannot specify a different value for **sys:%meter-micro-enables**; the default of #o14 is used, but this is typically what you want.
- In order to allow multiple stack groups to be metered, interrupts are not turned off.

meter:enable &rest *items* Function

This function enables metering in the stack groups specified by *items*. Each item in *items* can be one of the following:

- A stack group.
- A process (specifying the process's stack group).

You can obtain the process object by using the **find-process** function, which finds a process whose name contains a specified string. For example, the following form enables metering for the mouse process:

```
(meter:enable (find-process "mouse"))
```

If more than one mouse process exists, a menu of mouse processes appears that allows you to select the object passed to **meter:enable**. Also, calling the **find-process** function with no argument invokes a menu of processes.

- A window (specifying the window-process's stack group).
- The value `t`, which enables metering in all stack groups. However, this requires a large amount of disk space to hold the data.

Each stack group has a flag that controls whether events are recorded while running in this stack group. When you use the `meter:enable` function to enable metering in your stack groups, this flag is set. The variable `meter:metered-objects` contains a list of all the items for which you enabled metering.

Once you have enabled metering in your stack groups, you can turn metering on and off by setting bits in the `sys:%meter-micro-enables` variable.

The Examples paragraph (27.2.6.2) contains an example of using the `meter:run` function in conjunction with `meter:enable` and `meter:disable`. There is also an example of metering without using `meter`, `meter:test`, or `meter:run`. The example uses `meter:enable` and sets the `sys:%meter-micro-enables` variable.

meter:metered-objects Variable

This variable is a list of all the *items* that you have enabled with `meter:enable` and have not yet disabled.

meter:disable &rest items Function

This function disables metering in the stack groups specified by *items*. The arguments allowed are the same as for `meter:enable`. Specifying `(meter:disable t)` turns off `(meter:enable t)` but does not disable stack groups enabled individually. Specifying `(meter:disable)` with no arguments disables all stack groups no matter how you specified to enable them.

Examples **27.2.6.2** The examples presented in this discussion illustrate the following:

- A comparison of `meter`, `meter:test`, and `meter:run`
- Metering several forms at once with `meter:run`
- Metering without using `meter`, `meter:test`, and `meter:run`
- Metering only one stack group

Comparison of `meter`, `meter:test`, and `meter:run` Suppose the following function is defined:

```
(defun bar ()
  (make-list 100))
```

Now meter the function with `meter:test` and then execute `meter-analyze` to analyze the results:

```
(meter:test '(bar))
(meter-analyze)
```

The following is returned:

Stack Group: MAIN-STACK-GROUP

functions	# calls	Run t	Avg Run t	consing or faults	Real t
SYS:*EVAL	1	156	156	0	156
EVAL	1	110	110	0	110
BAR	1	106	106	0	106
METER:TEST	1	0	0	0	0

Now meter the function with `meter:run` and then execute `meter-analyze` to analyze the results. (Notice that `meter:enable` enables the current stack group only and `meter:disable` turns it off.)

```
(meter:reset)
(meter:enable sys:%current-stack-group)
(meter:run '(bar))
(meter:disable)
(meter-analyze)
```

The following is returned:

Stack Group: MAIN-STACK-GROUP

functions	# calls	Run t	Avg Run t	consing or faults	Real t
SYS:*EVAL	1	161	161	0	161
EVAL	1	111	111	0	111
BAR	1	106	106	0	106
METER:RUN	1	0	0	0	0
total	4	378	94	0	378

Now meter the function with `meter` and then execute `meter-analyze` to analyze the results:

```
(meter (bar))
(meter-analyze)
```

The following is returned:

Stack Group: MAIN-STACK-GROUP

functions	# calls	Run t	Avg Run t	consing or faults	Real t
SYS:*EVAL	5	593	118	0	593
BAR	1	119	119	0	119
SETQ	3	87	29	0	87
PROGN	1	13	13	0	13
total	10	812	81	0	812

Metering Several Forms at Once With `meter:run` This example shows how you can use `meter:run` to meter several forms at the same time.

```
(meter:reset)
(meter:enable t)
(meter:run '(prin1 11) '(prin1 12) '(prin1 13))
(meter:disable t)
(meter-analyze)
```

Metering Without meter, meter:test, or meter:run You can also do metering without using `meter`, `meter:test`, or `meter:run`. To do this, perform the following steps:

1. Enable metering for one or more stack groups.
2. Bind `sys:%meter-micro-enables` to some legal value.

Data is collected if these two conditions are true and a process with metering enabled is active. A typical example is as follows:

```
(meter:reset)
(meter:enable sys:current-process)
(setf sys:%meter-micro-enables #o14)
(inspect 'foo)
(setf sys:%meter-micro-enables 0.)
(meter-analyze)
```

Notice that the setting of `sys:%meter-micro-enables` controls when metering starts and stops. The setting of this variable provides a finer granularity of control than using `meter:enable` to turn on metering and `meter:disable` to turn off metering.

Metering Only One Stack Group This example illustrates why you usually only want to meter one stack group.

Suppose you are metering the Zmacs process and currently executing a Find File command. Suppose also that during the Find File command the mail daemon wakes up and receives some new mail. Metering data is not collected for the mail daemon, and the real time excludes the time Zmacs was sleeping while the mail daemon was running. The following lists shows how you could do this:

1. You can use `meter:enable` to enable metering for Zmacs as follows:


```
(meter:enable (find-process "zmacs"))
```
2. Set `sys:%meter-micro-enables` to turn on metering while in the Lisp Listener.
3. Switch to Zmacs to actually run the tests (Find File and so on). Only the Zmacs stack group information is recorded.

Another Meter Analysis Function 27.2.6.3 The `meter:analyze` function also analyzes metered data. The `meter-analyze` function (discussed in paragraph 27.2.4) is a menu interface to this `meter:analyze` function. Many of the `meter:analyze` options are the same as the `meter-analyze` options.

`meter:analyze &key :analyzer :stream :file :buffer :return :info` Function
`:find-callers :stack-group :sort-function :summarize`
`:inclusive :output`

The `meter:analyze` function analyzes the data recorded by metering.

Keywords: The following keywords allow you to specify the particular type of monitoring activity:

`:analyzer` — This keyword specifies a kind of analysis. Its default value is `:tree`. Another useful alternative is `:list-events`. These are explained in more detail below.

At most, only one of the following three keywords should be specified.

`:stream` — The `stream` keyword specifies a stream on which to print the analysis. The default is to print on `*standard-output*`.

`:file` — The `file` keyword specifies a file to which the analysis is written.

`:buffer` — The `buffer` keyword specifies an editor buffer to which the analysis is written.

Analyzing the metered data involves creating a large intermediate database. Normally, this database is created afresh each time `meter:analyze` is called. If you specify a non-nil value for the argument of `:return`, the intermediate data structure is returned by `meter:analyze`. This data structure can be passed as the `:info` argument on another call to `meter:analyze`. This process can save time. But you can only do this if you use the same *analyzer* each time because different analyzers use different temporary data structures.

The default analyzer `:tree` prints out the amount of run time and real time spent executing each function that was called. The run time is the CPU time and not disk wait time. The real time includes time spent waiting and time spent writing metered data to disk. The `:tree` keyword handles these additional keyword arguments to `meter:analyze`:

Keywords: `:find-callers` — The argument for this keyword is a function specification or a list of function specifications. A list of who called the specified functions and how often is printed.

`:stack-group` — The argument is a stack group or a list of them; only the activities in those stack groups are printed.

`:sort-function` — The argument is the name of a suitable sorting function that is used to sort the items for the various functions that were called. The sorting functions provided include `meter:max-page-faults`, `meter:max-calls`, `meter:max-run-time` (the default), `meter:max-real-time`, `meter:max-run-time-per-call`, and `meter:max-words-consed`.

`:summarize` — The argument is a function specification or a list of function specifications; only the statistics for those functions are printed.

`:inclusive` — If the argument to `:inclusive` is non-nil, the times for each function include the time spent in executing all functions called from the function and any other functions they called. If a function is called recursively, the time spent in the inner call(s) is counted twice (or more).

:output — The name of an alternative function to use on the meter tree. The default value is **meter:summarize-tree**. Other functions you can use are **meter:tree-print**, **meter:call-tree**, **meter:tree-inspect**, and **meter:tree-null**. Refer to paragraph 27.2.4.5, Output Type, for information on the output format.

The **:list-events** analyzer prints out one line about each event recorded. For example:

Real t	Run t	consing	Function name	stack	function
	or	faults		depth	or address
0	0	0	SETQ	365 RET	SI::INTERPRETER-SET
95	31	0	SI::EVAL1	336 RET	SETQ
129	46	0	PROGN	323 RET	SI::EVAL1
181	79	0	PROGN	323 CALL	SI::EVAL1
259	138	0	PROGN	323 RET	SI::EVAL1
298	158	0	SI::EVAL1	294 RET	PROGN
333	173	0	UNWIND-PROTECT	272 RET	SI::EVAL1
.
.

Refer to paragraph 27.2.4.1, Analyzer, for a description of the column headings.

Timing Macros

27.3 The timing macros, **timeit** and **time**, provide timing and resource information for the forms executed in the macro body. This timing information includes total wallclock (real) time, total CPU time, total disk wait time, consing count, paging time, and page fault count.

Both the **time** and **timeit** macros can be used at top level around the form of interest, and they can also be embedded around key sections of the executing code. You will obtain more accurate results by compiling the macro and the form being timed within a test function.

- The **timeit** macro is the more flexible of the two because it has more options. The **timeit** macro returns the result of the executed form as well as any of the timing and resource information listed above, in the order that you specify.

You can have timing information saved for later reporting. The times can be returned instead of (or in addition to) the results of the last form in the macro's body.

If you embed the macro around key sections of the executing code, the **:label** feature makes it very easy to track which piece of the code generated any resulting set of timings.

The **timeit** macro also runs with interrupts disabled, which generally helps when taking measurements. Running without interrupts provides a more accurate timing for a given piece of work. The scheduler is prevented from waking up every 1/60th of a second to look for higher priority work to schedule or to let another process of equal priority have equal chance at a timeslice. Thus, disabling interrupts is a benefit when you use **timeit** to time something that relies on no other processes. However, if multiple processes are involved in the work you are trying to measure, you cannot use this benefit. (The **timeit** macro provides an option to run with interrupts enabled.)

You can also define new meters to be used with **timeit**.

- The **time** macro prints the real time, paging time, page fault count, and consing information. It returns the results of the last form in its body.

You cannot save the performance information printed by **time** as you can with **timeit**. Also, **time** does not have the **:label** feature, which makes its results more difficult to interpret if it is wrapped around several pieces of the executing code.

However, the **time** macro contains one feature not found in **timeit**: printing consing information by area.

The following paragraphs describe the timing macros.

`timeit options &body body`

Macro

Executes *body* and prints or returns the execution time. The *options* is a list of one or more of the following option keywords:

<code>:label <i>string</i></code>	<code>:print <i>stream</i></code>	<code>:repeat <i>count</i></code>
<code>:min</code>	<code>:interrupts</code>	<code>:time</code>
<code>:cpu</code>	<code>:paging</code>	<code>:disk</code>
<code>:faults</code>	<code>:cons</code>	<code>:number-cons</code>
<code>:units <i>time-units</i></code>	<code>:microseconds</code>	<code>:ms</code>
<code>:milliseconds</code>	<code>:us</code>	<code>:seconds</code>
<code>:sec</code>	<code>:s</code>	<code>:collect <i>variable</i></code>
<code>:values <i>keyword(s)</i></code>	<code>:header <i>t-or-nil</i></code>	<code>:separator <i>string</i></code>
<code>:report <i>function</i></code>		

NOTE: You can set default options in the `time:*timeit-defaults*` variable (described later). The order in which performance items are printed, returned, or collected corresponds to the order in which you specify them in the `time:*timeit-defaults*` variable or in the `timeit` call. The default items in `time:*timeit-defaults*` precede those specified in the `timeit` call. The `:label` option does not follow this ordering; it is always first.

The following list describes each of these options in detail:

`:label string` — Label to print.

`:print stream-to-print-on` — Defaults to `*trace-output*`.

`:repeat repeat-count` — Executes the form that `timeit` is wrapped around *repeat-count* times. If you do not specify `:min` with `:repeat`, `timeit` prints the *average* for each performance data you specified, which may include CPU time, paging time, consing count, real time, disk wait time, number consing count, and page faults.

This averaging also applies to performance values returned by `timeit` by using the `:values` option and those stored in a variable by using the `:collect` option.

`:min` — If you use `:min` with `:repeat`, `timeit` prints or returns the *minimum* of the *repeat-count* values instead of the *average*.

`:interrupts` — If present, allows interrupts. (The default is to run with interrupts disabled.) If you measure with interrupts disabled, your timings will be more consistent and repeatable. If you enable interrupts, the scheduler runs about every 1/60th of a second to look for higher priority work to schedule. The scheduler also lets equal priority work run if your process has met or exceeded its quantum. This will randomly bias your timings based on the network and background activity present at that time on your system. However, if your form causes other processes to execute and you are interested in the total amount of time to complete all the work you requested, you need to allow interrupts so that the scheduler can schedule the other processes and let them run.

`:time` — If present, prints the total real time.

- :cpu** — If present, prints the time excluding the paging time.
- :paging** — If present, prints the total paging time.
- :disk** — If present, prints the disk wait time.
- :faults** — If present, prints the number of page faults.
- :cons** — If present, prints the number of words consed in all areas, excluding the Extra PDL Area.
- :number-cons** — If present, prints the amount of consing done in the Extra PDL Area (which is also known as the number consing area). This area is the area into which all bignums and flonums are initially consed.
- :units *time-unit*** — Time units to report in. *time-unit* can be any one of the following:
 - :seconds, :s, or :sec** — Time units in seconds.
 - :microseconds or :us** — Time units in microseconds.
 - :milliseconds or :ms** — Time units in milliseconds.
 - nil** — Allows **timeit** to choose the most appropriate time unit.

NOTE: The *default* unit of time depends on whether **timeit** is printing or returning the collected values. The returned time values are in microseconds by default, unless otherwise specified. Printed times, unless specified, default to the most convenient unit of zero, microseconds, milliseconds, or seconds. This is also true in cases where the times are printed and returned simultaneously. If you specify the unit of time instead of accepting the default, the unit you specify applies to both printed and returned values.

- :microseconds or :ms** — Short for **:units :microseconds** or **:units :ms**.
- :milliseconds or :us** — Short for **:units :milliseconds** or **:units :us**.
- :seconds, :s, or :sec** — Short for **:units :seconds**, **:units :s**, or **:units :sec**.
- :collect *variable*** — Collects timings into *variable*. You should use the **time:timeit-report** function to print these timings. (The **time:timeit-report** function is described later in this section.)
- :values *keyword(s)*** — Values to return. *keyword(s)* can be one of the following. To specify multiple keywords, put them in a list.
 - :value** — Returns the value of the body of **timeit**.
 - :label** — Returns the value of the **:label string** keyword.
 - :time** — Returns the value of the real time.
 - :cpu** — Returns the value of the CPU time.

:disk — Returns the value of the disk wait time.

:paging — Returns the value of the total paging time.

:faults — Returns the value of the number of page faults.

:cons — Returns the value of the number of words consed, excluding the Extra PDL Area.

:number-cons — Returns the value of the amount of consing done in the Extra PDL Area.

NOTE: If you want to define a new meter with the **time:define-meter** function (described later), you can also specify it as a **:values** option.

The values for the **:values** keywords are reported in the units specified in the **:units** option (the default is microseconds).

The default of the **:values** option is **:value**.

:header t-or-nil — When **t**, prints two lines, headers on the first line, values on the second line. When **nil**, does not print any headers for the values. When **:side** (the default), prints headers beside values.

:separator string — String printed between values. The default is “, ” when **:header** is **:side**. Otherwise, the default is “ ”.

:report function — The report function to use. The default is **time:timeit-print**.

Because **timeit** compiles different code depending on its options, most keyword values are not evaluated. The parameters whose arguments are evaluated are as follows:

:repeat :print :separator :header :correct :label :units

You can also specify parameters as (*keyword value*). This allows you to turn off features selected in **time:*timeit-defaults***. For example, **(:cpu nil)** or **(:cpu)** turns off CPU time reporting. (The description of **time:*timeit-defaults*** below contains an example of this.)

If the first option for the **timeit** macro is a string, it is taken as the label. Output is printed when there is a label, when you specify the **:print** option, or when there is no **:values** option.

NOTE: Several examples of the **timeit** macro appear after the discussion of the **time:*timeit-defaults*** variable.

time:*timeit-defaults* (:cpu) Variable

Default options for the **timeit** macro. You can set this variable to the list of arguments you use repeatedly. The **timeit** macro appends the arguments you pass to it with those listed in this variable. The default is (:cpu).

Example 1:

```
(setf time:*timeit-defaults* `(:time :cpu :disk :cons))
(timeit () (* 32 5))
```

The following is printed on the screen:

```
Real Time: 1.55 ms, CPU: 1.55 ms, Disk: 0.0, Consing 0 words
160
```

Example 2: The next example shows how you can turn off features selected in the **time:*timeit-defaults*** variable when you use **timeit**:

```
(timeit ((:cpu nil)) (* 32 5))
```

The following is printed on the screen:

```
Real Time: 1.55 ms, Disk: 0.0, Consing 0 words
160
```

timeit *Examples:* The following examples illustrate several of the **timeit** options:

Example 1: In the following example, **timeit** *prints* the CPU time the function takes to execute. This is because the default value of **time:*timeit-defaults*** is equal to (:cpu):

```
(timeit () (* 32 5))
```

The following is printed on the screen:

```
CPU: 1.6 ms
160
```

NOTE: If you try these examples on the Explorer system, the values printed or returned may vary slightly.

Example 2: With **:values** and **:time** in the next example, the total real time (in milliseconds) that the function takes to execute is *returned*, and the variable **clock** is set to this time:

```
(setf clock (timeit (:values :time) (* 32 5)))
```

The variable **clock** is set to 1609.

Example 3: With **:seconds** in the next example, the total real time returned is in seconds:

```
(setf clock1 (timeit (:values :time :seconds) (* 32 5)))
```

The variable **clock1** is set to 0.001579.

Example 4: With `:repeat 8` in the next example, the function is executed 8 times, and the value returned is the total real time divided by 8. The variable `clock1` is set to this time:

```
(setf clock2 (timeit (:values :time :repeat 8)
  (* 32 5)))
```

The variable `clock2` is set to 6604.316666.

Example 5: When `:min` is used in the next example with `:repeat`, the minimum time for the 8 executions is returned:

```
(setf clock3 (timeit (:values :time :repeat 8 :min)
  (* 32 5)))
```

The variable `clock3` is set to 1568.

Example 6: In the next example with `:values (:value :cpu :cons)`, three values are returned: the value of the body of `timeit`, the CPU time, and the number of words consed:

```
(multiple-value-setq (value cpu-time words-consed)
  (timeit (:values (:value :cpu :cons)) (* 32 5)))
```

The variable `value` is set to 160, `cpu-time` is set to 1627, and `words-consed` is set to 0.

Example 7: The following function illustrates the use of the `:label` feature. When you run `(math)`, three different segments of code are timed, and the labels identify each segment:

```
(defun math ()
  (timeit (:label "Multiply") (* 4 3))
  (timeit ("Add") (+ 4 3))
  (timeit ("Divide") (floor 40 8)))
```

When you enter `(math)`, the following is printed (times will vary greatly):

```
Multiply, CPU: 2.0 us
Add, CPU: 3.0 us
Divide, CPU: 1.0 us
5
```

Example 8: To obtain greater accuracy and prettier output, the following version uses the `:repeat` and `:header` options:

```
(defun math2 (n)
  (timeit (:label "Multiply" :repeat n :header t) (* 4 3))
  (timeit ("Add" :repeat n :header nil) (+ 4 3))
  (timeit ("Divide" :repeat n :header nil) (floor 40 8)))
```

When you enter `(math2 10000)`, the following is printed (times may vary slightly).

NOTE: This example is *much* more accurate than the previous example, which does not use the `:repeat` option.

```
Name           CPU
Multiply       1.48 us
Add            1.48 us
Divide        1.48 us
5
```

time:timeit-report *timings* &key **:label** **:plot** (**:stream** **terminal-io**) Function
 Reports timings collected with the `timeit :collect` option.

timings — The timings list produced by `timeit`.

:units — Specifies a default time unit to use when *timings* does not specify one.

:label — Allows you to restrict the report to those timings whose labels are string-equal to this parameter.

:plot — When **:plot** is the name of one of the meters collected in *timings* (for example, **:time**, **:cpu**, **:disk**, and **:cons**), this value is displayed graphically.

:stream — The stream on which to report the timings. The default is **terminal-io**.

In the following example, timing information is collected in the special variable *timings*:

```
(defun print-test (object n)
  (declare (special timings))
  (dotimes (i n)
    (timeit ("Print" :cons :collect timings)
      (print object 'sys:null-stream))
    (let ((*print-pretty* t))
      (timeit ("Print Pretty" :cons :collect timings)
        (print object 'sys:null-stream)))
    (timeit ("Format" :cons :collect timings)
      (format 'sys:null-stream "-s" object))))

(setq timings nil)
(print-test '(a b (c (d) e) (f g) . h) 3)
(time:timeit-report timings)
```

The following output is produced:

Name	CPU	Consing
Print	18.2 ms	0 words
Print Pretty	35.9 ms	76 words
Format	19.0 ms	0 words
Print	17.7 ms	0 words
Print Pretty	35.2 ms	76 words
Format	18.7 ms	0 words
Print	17.7 ms	0 words
Print Pretty	35.2 ms	76 words
Format	18.8 ms	0 words
NIL		

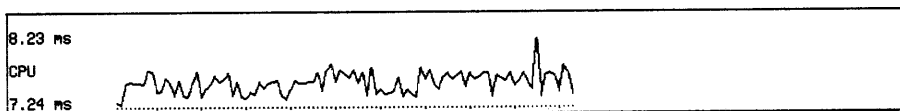
Entering `(time:timeit-report timings :label 'format)` produces the following output:

Name	CPU	Consing
Format	19.0 ms	0 words
Format	18.7 ms	0 words
Format	18.8 ms	0 words
NIL		

The next example shows the use of `:plot`:

```
(print-test '(a b (c (d) e) (f g) . h) 100)
(time:timeit-report timings :label 'format :plot :cpu)
```

The following output is produced:



The horizontal (x) axis is the data items. In this example, there are 100 data items being reported. The graph plots the CPU time for each of the 100 data items. The call to the `print-test` function causes its `dotimes` to be executed 100 times. Each iteration makes three calls to `timeit`, each with a different label: `print`, `print pretty`, and `format`. Each of the three calls generates one data item that is collected into the variable `timings` for a total of 300 data items.

The call to the report function `(time:timeit-report timings :label 'format :plot :cpu)` plots the CPU time for all of the data items with the label `format`, which results in 100 CPU times plotted. If `:label` is omitted, 300 data items are plotted, making the graph difficult to read.

time:define-meter *name &body options* Macro

Defines a performance meter *name* with optional parameters, as follows:

:form — A form that returns the meter value.

:difference — A function that subtracts two meter values. The default is `-`.

:header — A string used for printing the header value.

:format — A **format** string used for reporting the value.

:report — A function to apply to the value before printing it (with **:format**).

To define new meters, it is necessary to have a high-level understanding of the way **timeit** collects meter values for a given body of code. The following sequence of events explains what **timeit** does. It is very simplistic because **timeit** more intelligently computes timing meters with greater accuracy.

```
record start-time
execute body of code
record stop-time
compute the difference between stop-time and start-time
```

When defining a meter, you can basically do whatever you want to suit your application, but the most common use is to read the system microcode counters. To see the variety and extent of them, invoke the Peek utility (SYSTEM P) and click on Counters. These counters are documented in the *Explorer System Software Design Notes*.

Suppose you want to know the total number of read and write references to the PDL buffer that were virtual memory references that trapped during the execution of a critical piece of code. You can define a meter called `:pdl-faults`, as follows:

```
(time:define-meter :pdl-faults
  :header "PDL-faults"
  :format " -%d "
  :form si:(+ (read-meter '%count-pdl-buffer-read-faults)
              (read-meter '%count-pdl-buffer-write-faults)))
```

Next, in order to print your new meter, you can execute your code within a **timeit** form similar to the following:

```
(timeit (:pdl-faults :disk)
  (copy-file "your-file" "my-file"))
```

The following is printed on the screen:

```
CPU: 687. ms, PDL-faults: 6303, Disk: 93.7 ms
NIL
```

In this example, the defaults are accepted for the **:difference** and **:report** options because a simple integer counter is being collected. When you use meters that record timings, both of these options are usually specified. The **:difference** option is usually specified as the function **time:microsecond-time-difference** (described later) to compensate for clock wraparound. The **:report** option is usually specified as **time:pretty-time** to print the value in the most appropriate unit of time.

The following shows an example of a meter definition that uses these two options:

```
(time:define-meter :paging
  :header "Paging"
  :format "-7a"
  :difference time:microsecond-time-difference
  :report time:pretty-time
  :form sys:(read-meter '%total-page-fault-time))
```

time & optional *form describe-consing-p*

[c] Macro

When you specify *form*, this macro returns the value of the evaluated form while printing to ***trace-output*** the real time, paging time, number of page faults, and the consing count. If you specify *describe-consing-p*, **time** prints the consing information divided into areas instead of printing only the total consing count.

The first **time** example below times a form without using *describe-consing-p*. The second example times the same form but uses *describe-consing-p*.

Example 1: (time (inspect 'foo))

The following is printed on the screen:

```
Evaluation of (INSPECT 'FOO) took 1.819595 Seconds of elapsed time,
including 0.055314 seconds of paging time for 4 faults, Consed 827
words.
NIL
```

Example 2: (time (inspect 'foo) t)

The following is printed on the screen:

```
Evaluation of (INSPECT 'FOO) took 0.479713 Seconds of elapsed time,
including 0.000226 seconds of paging time for 1 faults, Consed 615
words.
42 words consed into EXTRA-PDL-AREA
588 words consed into WORKING-STORAGE-AREA
5 words consed into SHEET-AREA
NIL
```

If you use **time** without specifying *form*, the macro returns time in 60ths of a second. However, this usage of the macro is not a performance related tool. For more information on this usage, refer to the Dates and Times section in the *Explorer Lisp Reference*.

The **time** and **timeit** macros are maintained to use all of the correct meters and counters on the system and to normalize out the timing function overhead from the results. The most efficient methods are used to gather time while running the timing analysis tests. If for some reason you decide not to use these macros, you can use **time:microsecond-time** and **time:fixnum-microsecond-time** to read the real time microsecond clock both before and after the form(s) of interest execute. You will then need to use **time:microsecond-time-difference** in order to properly account for time, given the properties of the microsecond clock. You will also need to handle all normalizations.

If you want information on reading system counters, refer to the discussion of the `read-meter` function in the *Explorer System Software Design Notes*. However, please note that the `time` and `timeit` macros will always be updated to correctly assimilate the counter data so that the various measures of work will be accurately accounted for. For example, calculating page faults requires particular combinations of the system counters in order to generate the desired information. It is not intuitively obvious what meters to use.

time:microsecond-time

Function

Returns the current value of the microsecond clock (a bignum). Only differences in clock values are meaningful. There are 32 bits of data, so the value wraps around about every 72 minutes.

This function is useful for performance timings. However, it actually wraps every 4,294,967,296 microseconds and can cause problems when this happens. Therefore, you need to use `time:microsecond-time-difference` when subtracting two values returned by this function, to ensure accurate timings.

Example:

The following example uses `time:microsecond-time` to return the value of the microsecond clock both before and after the `hostat` function is executed. The value of `end-time2` (also obtained by using `time:microsecond-time`) is later used by `time:microsecond-time-difference` to normalize the amount of time that it takes to read the `end-time`. Depending on how short your test is, this may be very important to do.

```
(setf start-time (time:microsecond-time))
(hostat 'hotel)
(setf end-time (time:microsecond-time))
(setf end-time2 (time:microsecond-time))
(setf run-time (time:microsecond-time-difference end-time
  (+ start-time (time:microsecond-time-difference
    end-time2 end-time))))
```

time:microsecond-time-difference end start

Function

Returns the difference between *end* and *start* microsecond times. This function handles microsecond clock wraparound. Refer to `time:microsecond-time` for a discussion of this wraparound and for an example of using `time:microsecond-time-difference`.

time:fixnum-microsecond-time

Function

Returns the current value of the microsecond clock as two fixnums.

The first fixnum returned is the least significant 23 bits of the 32-bit microsecond counter. The second fixnum returned is the most significant 9 bits of the 32-bit microsecond counter. This function is a little more difficult to utilize than `time:microsecond-time`.

Calculating a time difference using values returned by this function can be very difficult. This is because the lower fixnum wraps around every 8.4 seconds and the upper fixnum wraps around every 72 minutes. If you know that the event being timed is less than 8.4 seconds, then `time:fixnum-microsecond-time` is straightforward. Otherwise, the algorithm for determining what has happened between measurements becomes complicated.

Function Histogram

27.4 The function histogram utility provides you with a statistical sampling of function calling either continuously or over a discrete time interval. You can select which stack groups to monitor, how deep a calling sequence to monitor, and how often to sample the stack frame. This statistical sampling can be started, stopped, reported, and saved to a data file using functions available in the METER package. Using the Peek interface to the function histogram utility, you can view a continuously-updating report of function calling with the same controls on the depth and the frequency of the sampling that you have when using the histogram functions.

Every time the scheduler runs, a count is incremented (in a hash table) for the top n functions on the stack for the process that just ran, where n defaults to 3. The scheduler normally runs once every 67 tv-clock interrupts (about once a second). The scheduler is speeded up to once every 4 interrupts (about 16 times a second) to gather more data.

Peek Function Histogram

27.4.1 When you select the Functions item from the Modes menu window, Peek provides a continuously-updating statistical sampling of which functions use the most system resources. (See Figure 27-11.) Using the mouse to select options, you can control which processes are sampled, how many functions are recorded, how many are displayed, and the sampling frequency.

You can also select the Functions mode by pressing M, which stands for metering. (The F keystroke is for the File Status mode.)

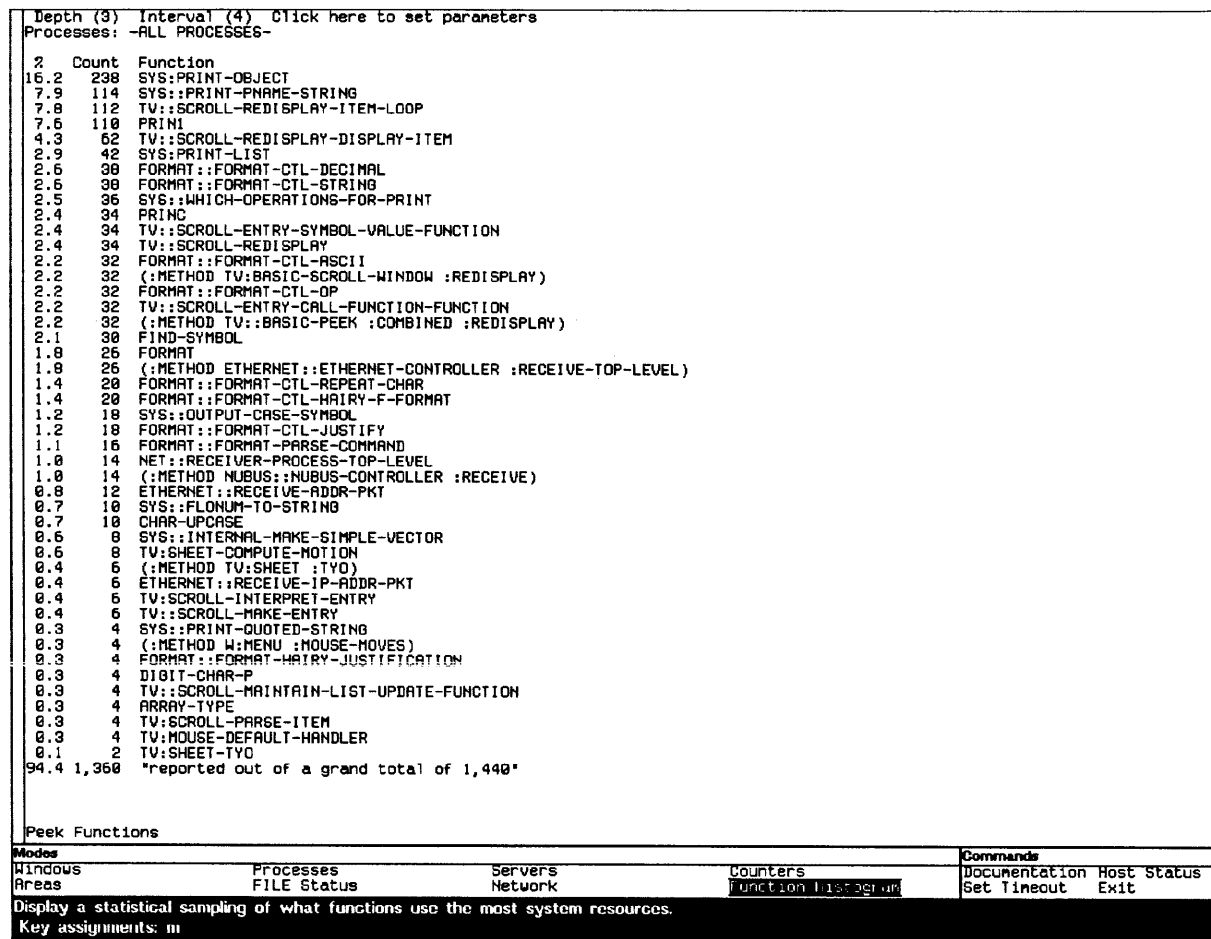
If you click on [click here to set parameters](#), the Function Histogram Parameters menu appears. The following list explains the prompts:

- Histogram — Allows you to reset, stop, or continue the histogram:
 - Reset — Clears the histogram data that has been collected so far and reactivates the sampling process for all processes by default. If the functions display does not continue showing the updated histogram, click on the Function Histogram item from the Modes menu window again to restart the display.
 - Stop — Stops collecting the histogram data.
 - Continue — Continues collecting the histogram data.

NOTE: If you do not specify Stop before you exit Peek, the function histogram will still be running.

- Processes — Allows you to select which processes to sample. The default is `-ALL PROCESSES-`.
- Histogram Depth — The number of stack frames to go down. In other words, if the value is n , the count is incremented for the top n functions on the stack when the sample is taken. The default is 3.

Figure 27-11 Example of Peek Function Histogram Screen



- Histogram Interval — The time between samples in 60ths of a second. The default is 4.
- Entries Displayed — The number of functions to display. The default is 45.

Function Histogram Functions

27.4.2 The functions described in this paragraph also allow you to use the function histogram utility to obtain a statistical sampling of which functions use the most system resources. Using these histogram functions, you can start, stop, and report the histogram, and you can save and restore the histogram from disk.

The function histogram data is collected only for those processes whose stack group has metering enabled. Refer to the **meter:enable** and **meter:disable** functions discussed in paragraph 27.2.6, Customized Metering Sessions.

The following sequence shows a typical use of the function histogram utility. (The functions mentioned are discussed later in detail.)

1. Reset the meter partition with `(meter:reset)`.
2. Enable the current stack group with `(meter:enable sys:current-stack-group)`.
3. Start the function histogram utility with `(meter:start-histogram)`.
4. Execute the code you want to sample, for example, `(make-list 100)`.
5. Stop the function histogram utility with `(meter:stop-histogram 'name)`. The name is an arbitrary name that you specify.
6. View the information with `(meter:report-histogram 'name)`. The name is the name you supplied when you executed `meter:stop-histogram`.

A display similar to the following appears:

```

Depth (3)  Interval (4)
8.3%      18          SYS:PRINT-OBJECT
5.6%      12          FS:READ-DIRECTORY-ENTRY
4.6%      10          FS:LMFS-READ-DIRECTORY
.
.
.
0.9%      2           SYS:INTERAL-READ-CHAR
79.6.%    172         reported out of a grand total of 216

```

In the first line, the `Depth` is the number of stack frames to go down, and the `Interval` is the time between samples in 60ths of a second. The first column in the second line shows the percentage of the total function calls recorded. (That is, this percentage is the number of function calls recorded for the function divided by the number of total recorded function calls.) The second column is a count of the number of times the function was called. The third column lists the function that was called.

The following paragraphs describe the variables and functions for the function histogram utility.

meter:*function-histogram-depth* 3 Variable

The number of stack frames to go down. In other words, if the value is n , the count is incremented for the top n functions on the stack when the sample is taken. The default is 3.

meter:*function-histogram-interval* 4 Variable

The time between samples, in 60ths of a second. The default is 4.

meter:*function-histogram-number* 45 Variable

The number of functions to display when reporting. The default is 45.

meter:start-histogram &key (:interval meter:*function-histogram-interval*) Function
(:depth meter:*function-histogram-depth*) :pathname :name

Starts collecting function histogram information. A count is incremented every :interval 60ths of a second for the names of the top :depth functions on the control stack of the current process.

:interval — The time between samples, in 60ths of a second. The default is the value of the **meter:*function-histogram-interval*** variable.

:depth — The number of stack frames to go down. In other words, if the value is *n*, the count is incremented for the top *n* functions on the stack when the sample is taken. The default is the value of the variable **meter:*function-histogram-depth***.

:pathname — An optional pathname used to pre-load a histogram saved with the **meter:save-histogram** function. This **:pathname** keyword performs the same operation as the **meter:restore-histogram** function.

:name — The name of a previous histogram to use.

By using **:name** and **:pathname**, you can continue adding newly-acquired histogram data to an existing histogram. If you do not specify **:name** or **:pathname**, a new histogram is created.

meter:modify-histogram &key :interval :depth Function

Changes the **:interval** or **:depth** for the current histogram by updating the **meter:*function-histogram-interval*** and **meter:*function-histogram-depth*** global variables, respectively.

:interval — The time between samples, in 60ths of a second. The default is the value of the **meter:*function-histogram-interval*** variable.

:depth — The number of stack frames to go down. In other words, if the value is *n*, the count is incremented for the top *n* functions on the stack when the sample is taken. The default is the value of the variable **meter:*function-histogram-depth***.

meter:stop-histogram &optional *name* *pathname* Function

Stops collecting function histogram information, and calls it *name*. If you do not specify *name*, the histogram hash table is deleted. Optionally, you can save the histogram in a file by specifying *pathname*.

meter:report-histogram &optional *name* Function
(*number* meter:*function-histogram-number*) *threshold*

Displays a histogram-like report for the histogram *name*. The *threshold* and *number* arguments are the two constraints that determine how many functions are included in the report.

threshold — Limits the report to functions that were called more than *threshold* times. The initial default of threshold is 0; however, the value you supply becomes the new default. Each histogram known by the system has its own default threshold.

Note that large *threshold* numbers greatly increase the speed of this **meter:report-histogram** function.

number — Limits the report to a total of *number* functions, all of which must be above the *threshold*. No recorded function is reported unless it was called more than *threshold* times. (The default for *number* is the value of the `meter:*function-histogram-number*` variable.)

name — The name of the histogram to report. The default is the currently running histogram. If specified, *name* must be a known histogram that was created with the `meter:stop-histogram` function or `meter:restore-histogram` function.

`meter:save-histogram` *pathname* &optional *name* Function

Saves the histogram *name* into *pathname*. The default for the *name* argument is the currently running histogram.

`meter:restore-histogram` *pathname* &optional *name* Function

Restores a histogram from *pathname* into *name*. The *name* defaults to the currently running histogram. You can combine the data in two or more histogram data files by restoring them to the same *name*.

Introduction

28.1 The Telnet window allows you to use the Explorer screen as a terminal to another host. When you are in a Telnet window, characters typed on the keyboard are passed to the remote host's Telnet server.

You can enter Telnet in any of the following ways:

- Press SYSTEM T.
- Type (telnet) in a Lisp Listener.
- Click on the Telnet item in the main System menu.

The following describes the `telnet` function.

<code>telnet</code> & optional <i>path mode</i>	Function
---	----------

This function makes a Telnet connection to a host specified by *path*, which can be either the name of a valid host or `nil`.

For those sites that have hosts serving as gateways (bridges) between Chaosnet and Arpanet subnetworks, *path* can also be a string that contains information about how to get to a valid host. The following are example formats:

gateway-name ESCAPE *internet-host-name*

internet-host-name / *socket-number*

gateway-name ESCAPE *internet-host-name* / *socket-number*

In the previous examples, ESCAPE means to press the ESCAPE key at that point.

The default for *mode* is `t`. If *mode* is set to `t` and *path* is specified, a non-connected Telnet window is selected and a connection to *path* is made. If *mode* is set to `t` and *path* is `nil`, a connected Telnet window is selected, if there is one available; if there is not one available, a connection is made to the host named by the variable `telnet:telnet-default-path`.

When *mode* is `nil`, a Telnet window is selected. If *mode* is set to `nil` and *path* is specified, a new Telnet window is selected and a connection to *path* is attempted. If *mode* is set to `nil` and *path* is `nil`, a connection is made to the host whose name is the current value of the variable `telnet:telnet-default-path`.

Once you have established a Telnet connection, you can transmit key combinations, or key sequences, for generating all 128 USASCII codes. For more information, refer to J. Pastel and J. Reynolds' *Telnet Protocol Specification*, RFC 854, USC/Information Sciences Institute, May 1983.

Entering a Telnet Window

28.2 When you enter a Telnet window, you are prompted for a name that specifies the host to which you want to connect. If a connection is already established when you enter the window, Telnet continues to use that connection and you are not prompted for a host name.

The Telnet General Help window shows the various ways of identifying this host. One way is to specify the host as a string that the remote host recognizes as its host name or alias.

You can also specify a string formed from the name of the host followed by a slash and the *connect name* (for example, "HOST/TELNET"). The connect name is used by a server to recognize a service and must be formed of upper-case ASCII letters, numbers, and/or punctuation marks. After a connection is established, the connect name is discarded. Telnet's connect name is "TELNET". The *Explorer Networking Reference* describes connect names in detail.

For those sites that have hosts serving as gateways (bridges) between Chaosnet and ARPANET subnetworks, the host name can be substituted by a string that contains information about how to get to a valid host. The following are example formats:

gateway-name ESCAPE internet-host-name

internet-host-name / socket-number

gateway-name ESCAPE internet-host-name / socket-number

Once you are connected to a host, most of the keys on the terminal keyboard lose their normal function and their characters pass to the remote host. Particular keys are affected as follows:

- The SYSTEM and TERM keys retain their normal function (and are therefore not forwarded).
- The ABORT, BREAK, and RESUME keys retain their normal function and they are not forwarded to the remote host.
- The CLEAR INPUT key sends the Erase Line (EL) Telnet command to the remote host.
- The STATUS key sends the Are You There (AYT) Telnet command to the remote host.
- The NETWORK key is the first key of a two-key sequence that sends a Telnet command to the remote host.
- The END key exposes the previously selected window and leaves the Telnet connection open.

Because Telnet is implemented with the Universal Command Loop (UCL), Telnet command descriptions and online help are available from the HELP key.

Telnet Commands 28.3 To enter a Telnet command, you use a two-key sequence, where NETWORK is the first key.

Table 28-1

Telnet Commands	
Keystroke	Description
NETWORK A	Send the Abort Output (AO) Telnet command to the remote host.
NETWORK CLEAR INPUT	Send the Erase Line (EL) Telnet command to the remote host.
NETWORK D	Disconnect from the current remote host and ask for the name of another remote host to which you want to connect.
NETWORK END	Expose the previously selected window and leave the Telnet connection open.
NETWORK HELP	Describes the Telnet commands. Also, you can obtain this information by pressing HELP and clicking on Command Display.
NETWORK M	Toggle the <i>*more*</i> processing variable on the Telnet window.
NETWORK O	Toggle between Insert mode (the default mode) and Overwrite mode.
NETWORK P	Send the Interrupt Process (IP) command to the remote host.
NETWORK Q	Expose the previously selected window and disconnect from the remote host.
NETWORK STATUS	Send the Are You There (AYT) Telnet command to the remote host.

Telnet Server

28.4 The default Telnet server on the Explorer system is a Chaosnet server that has the contact name **telnet**. If you have TCP/IP on your system, the Telnet server also responds to a port number, *n*, on TCP. The Telnet server accepts data and Telnet commands from a remote host's terminal, using standard Telnet protocol. For systems with the Explorer TCP/IP software loaded, an IP-based Telnet server also exists.

When the connection is first established, Telnet sends the remote host the print herald of this machine. Telnet then uses the read-eval-print loop to set up a communications loop, reading characters from the remote host until a complete Lisp expression arrives. Telnet then evaluates the Lisp expression and returns the resulting value(s) to the remote host.

Use the following function on your system to enable or disable the Telnet server.

telnet:telnet-server-on (*mode* :notify)

Function

This function enables and disables the Telnet server. *mode* can take on the following values: **t** for on; **nil** for off; **:notify** (the default) for on, with the condition that the user is notified when a connection is made; and **:not-logged-in** for on, if no one is logged in.

NOTE: Note that this function affects what happens when a remote host attempts to use Telnet on your system. It does not affect a remote host on which you may later want to use Telnet.

VT100 EMULATOR



The VT100 emulator runs as an application on top of Telnet. With the VT100 emulator, you can use the Explorer monitor and keyboard as a VT100 terminal. The VT100 emulator is resident on the Explorer system and has the following features:

- All alphanumeric keys and cursor-control keys transmit the proper VT100 codes.
- All function keys transmit VT100 codes, except the BREAK key and its variants.
- The keys on the Explorer keypad transmit VT100 auxiliary keypad codes.
- Most of the control commands used by a VT100 terminal are emulated on the Explorer.
- There are no Explorer keys that correspond to the VT100 SETUP and NO SCROLL keys.
- CTRL-H corresponds to the VT100 BACKSPACE key.

The VT100 emulator is implemented using the UCL; command descriptions and online help are available from the HELP key.

For a description of the VT100 escape and control sequences, refer to the *VT100 User's Guide*, published by Digital Equipment Corporation.

To enter an Explorer window that emulates the VT100 terminal, press SYSTEM V. The VT100 emulator frame has an automatic-scroll window and a light-emitting diode (LED) window. If a connection has not already been established before you enter the VT100 frame, you are prompted in the automatic-scroll window for the name of a host to which you want to connect. The LED window displays four LEDs that simulate the LEDs on a VT100 keyboard. The LEDs are turned on and off by the appropriate key sequences.

In addition to a default character font, the font map for a VT100 window includes the following:

- Graphics font
- Top and bottom fonts
- Double-wide font

The graphics font matches the VT100 special graphics character set. It is selected after you enter the proper VT100 command sequence. The VT100 emulator uses the top and bottom fonts when you enter the double-height, top-bottom command sequence. It uses the double-wide font after you enter the double-width command sequence.

The following VT100 control sequences are currently not implemented on the Explorer system:

- Underscore on
- Bold on (used with graphics font)
- Invoke confidence test
- Cursor Key mode
- ANSI/VT52 mode
- Scrolling mode
- Origin mode
- Autorepeat
- Interlace

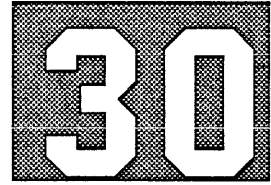
The Explorer system builds the VT100 emulator frame on top of Telnet. Just as in the Telnet window, most of the keys pressed on the keyboard are passed to the remote host. Particular keys are affected as follows:

- The **SYSTEM** and **TERM** keys retain their normal function (and are therefore not forwarded).
- The **ABORT**, **BREAK**, and **RESUME** keys retain their normal function and are not forwarded to the remote host.
- The **CLEAR INPUT** key sends the Telnet **EL** command to the remote host.
- The **STATUS** key sends the Telnet **AYT** command to the remote host.
- **NETWORK** is the first key of a two-key sequence that sends a Telnet command to the remote host. Most of the Telnet commands are also valid in the VT100 emulation frame. Refer to Section 28, Telnet, for a description of these commands.

Table 29-1 describes the VT100 commands displayed in the command menu. Note that these commands require a two-key sequence, where **NETWORK** is the first key.

Table 29-1 VT100 Commands

Name	Keystroke	Description
Answerback	NETWORK B	Sends the Answerback message string in *vt100-answerback-message*
80/132 Columns	NETWORK C	VT100 Setup-A 80/132 Columns
Set Lines	NETWORK L	Sets the number of lines for the VT100 screen and reconfigure screen
Reset	NETWORK R	VT100 Setup-A Reset
VT100 Switch	NETWORK S	Enables/Disables VT100 escape sequence processing
Truncate	NETWORK T	Toggles truncating of VT100 screen pane
Reverse Video	NETWORK V	Toggles between standard video and reverse video for the VT100 screen



Introduction

30.1 Converse is an interactive message editor that displays all the messages that you have sent or received. You can enter Converse in any of the following ways:

- Press SYSTEM C.
- Type the form (qsend) in a Lisp Listener.
- Click on the Converse item in the System menu.

On the screen, Converse groups into a *conversation* all the messages that you send to or receive from a particular user. Conversations from different users are separated by thick dividing lines. Do not delete these dividing lines because this will cause messages to be lost. Messages are grouped according to user name. Messages are put into separate conversations in the following cases:

- The same user name is employed on different hosts
- Different user names are employed

Within a conversation, the name of the other party to the conversation appears at the top of a chronologically ordered group of messages. Converse creates a new conversation when you begin communicating with someone for whom there is no existing conversation.

When Converse asks you to enter a user name with the `to:` prompt, use the following syntax conventions:

- When you want to deal with one user, specify the form *user@host*, where *user* is a valid user name and *host* is a valid host name.
- When you want to deal with multiple users, specify the form *user@host, user-1@host-1, user-2@host-2, ... user-n@host-n*. A separate copy of your message will go into the conversation for each of the recipients.
- If you just enter *user* instead of *user@host*, Converse will try to find an Explorer system that *user* is logged in to and forward the message to that host. If you give the variable `zwei:*converse-extra-hosts-to-check*` a list of hosts to check, Converse tries to determine where *user* is logged in among these hosts. If *user* is not logged in under any of the hosts in that list, Converse gives you a menu of hosts from which to choose in order to continue your search for *user*.
- You can omit the user name and specify only the host as *@host*. Whoever is logged in on that host receives your message.

To send a message, perform the following:

1. Move the cursor to the right of the `to:` prompt and type the user name.
2. Press the RETURN key and enter your message.
3. Press the END key or use the CTRL-END sequence in order to transmit.

Zmacs Editor Commands With Converse

30.2 Converse allows you to use most of the Zmacs editor commands to edit your messages and move them around to different conversations. Table 30-1 lists the additional Converse commands that are available.

Table 30-1 Converse Commands

Key Sequence	Explanation of Command
END	Send the current message without exiting from Converse.
CTRL-END	Send the current message and exit from Converse.
ABORT	Eliminate the current Converse window.
CTRL-M	Mail the current message instead of sending it with Converse.
META-{	Move to the previous To: line.
META-}	Move to the next To: line.
META-X Delete Conversation	Delete the current conversation.
META-X Write Buffer	Write all of the conversations into a file.
META-X Write Conversation	Write only the current conversation into a file.
META-X Append Conversation	Append the current conversation to the end of a file.
META-X Regenerate Buffer	Rebuild the buffer structure. This command is useful if you edit in or across the thick dividing lines that separate conversations, which damages the buffer structure. Some error messages may suggest that you execute this command before retrying an operation. Note that this command deletes anything you have inserted in the buffer but have not yet sent.
META-X Gag Converse	Toggle the value of the <code>zwei:*converse-gagged*</code> variable. If set to <code>t</code> , the variable tells Converse to reject incoming messages; if set to <code>nil</code> , it tells Converse to accept incoming messages.

Converse Functions 30.3 You can use the following Converse functions to send or reply to a message:

qsends-off &optional *gag-message* Function

If the value of *gag-message* is set to **t**, which is the default, Converse rejects all incoming messages; if it is set to **nil**, Converse accepts all incoming messages. This command is useful to specify whether you want to be interrupted with any interactive messages. *gag-message* can be a string that is automatically forwarded as a reply to the sender of a message.

qsends-on Function

This function specifies that all incoming messages are to be accepted. This function is the complement to the **qsends-off** function.

qsend &optional *destination message mail-p wait-p* Function

This function sends *message*, which should be a string, to the user name(s) specified in *destination*. If *message* is empty, you are prompted for its contents. *destination* can be one of the following:

- When you want to deal with one user, specify *user@host*, where *user* is a valid user name and *host* is a valid host name.
- When you want to deal with multiple users, specify a list in the form (*user@host user-1@host-1 user-2@host-2 ... user-n@host-n*).
- If you just enter *user* instead of *user@host*, Converse will try to find an Explorer system that *user* is logged in to and forward the message to that host. If you give the variable **zwei:*converse-extra-hosts-to-check*** a list of hosts to check, Converse tries to determine where *user* is logged in among these hosts. If *user* is not logged in under any of the hosts in that list, Converse gives you a menu of hosts from which to choose in order to continue your search for *user*.
- You can omit the user name and specify only the host as *@host*. Whoever is logged in on that host receives your message.

If *destination* is empty, the user is put into the Converse window and asked to specify the destination.

If *mail-p* is set to **nil**, which is the default value, the message is sent interactively; if it is set to **t**, the message is mailed instead.

If *wait-p* is set to **nil**, the **qsend** function immediately returns **nil** as its value and sends the message in background mode. If *wait-p* is set to **t**, the **qsend** function monitors the status of the message it sends and returns a list of the recipients that received the message. The default value for *wait-p* is set to the value of the **zwei:*converse-wait-p*** variable.

```
(qsend '(john@zebra jane@giraffe) "The ark is ready")
```

zwei:reply & optional *message destination mail-p wait-p* Function

This function sends *message*, a string, to the last user who sent you a message. If *message* is empty, the command prompts you for the contents of the message.

Because the default value of *destination* is set to the value of **zwei:*last-converse-sender***, the message goes to the host from which the last user sent a message, unless you specify another host as *destination*. If you specify *destination*, provide a user name.

If *mail-p* is set to **nil**, which is the default value, the message is sent interactively; if it is set to **t**, the message is mailed instead.

If the value of *wait-p* is **nil**, the **zwei:reply** function returns the recipient of the message. If *wait-p* is set to **t**, the **zwei:reply** function monitors the status of the message it sends and returns a list of the recipients that received the message. The default value for *wait-p* is set to the value of the **zwei:*converse-wait-p*** variable.

User Options With Converse

30.4 You can set the following user options in your login initialization file:

zwei:*converse-receive-mode* Variable

This variable controls what occurs when you receive a new interactive message. The variable can take on one of five values:

:auto indicates that the Converse window is automatically entered when a message arrives.

:notify indicates that, whenever a message arrives, you are to be informed about both its arrival and its origin.

:notify-with-message is similar to **:notify**, except that you are given the sender's message as well as the sender's name. This is the default value for the variable.

:pop-up indicates that the receipt of a message results in the appearance of a pop-up window on the screen. The window gives you the option to reply to the message, to enter the Converse window, or to do nothing at all. This window notifies you in the same way as **:notify-with-message**.

:simple is the same as **:pop-up**.

zwei:*converse-append-p* Variable

If this variable is set to **t**, a new message is appended to the end of the sender's conversation. If this variable is set to **nil**, which is the default value, a new message is added to the beginning of the sender's conversation.

zwei:*converse-beep-count* Variable

This variable indicates the number of times the Explorer system will beep when a message arrives. The default value is two.

zwei:*converse-extra-hosts-to-check* Variable

This variable indicates which hosts are checked when someone types *user* instead of *user@host* when asked for a user name. The hosts are checked to see if *user* is logged in to any of those hosts. To specify a group of hosts, enter a list of valid host names. This variable defaults to *nil*, which means that all hosts are checked.

zwei:*converse-end-exits* Variable

If the value of this variable is *t*, the following will happen after you type a message:

- If you press END, the message is sent and you exit Converse.
- If you press ABORT, the message is not sent and you exit Converse.
- If you press CTRL-END, the message is sent and you remain in Converse.

If the value of this variable is *nil*, which is the default value, the following will happen after you type a message:

- If you press END, the message is sent and you remain in Converse.
- If you press ABORT, the message is not sent and you exit Converse.
- If you press CTRL-END, the message is sent and you exit Converse.

zwei:*converse-gagged* Variable

If the value of this variable is not *nil*, Converse will reject all incoming messages. In this case, **zwei:*converse-gagged*** should be a string that contains the reason why you are not receiving messages.

If the value of this variable is *nil*, which is the default value, you will receive all incoming messages.

Although this variable is available for your convenience, the **qsends-on** and **qsends-off** functions are recommended instead.

zwei:*converse-wait-p* Variable

If this variable is set to *t*, which is the default value, Converse waits to determine the status of any message that you send. If set to *nil*, Converse does not monitor the status of messages.

Introduction

31.1 The Explorer mail system is an electronic mail system that allows the exchange of mail messages between users on a computer network. The system is composed of the mail reader, which allows you to receive messages from other network users, and the mailer, which sends messages over the network.

This section describes the Explorer mail system and explains how to access and use the system. The following specific topics are discussed:

- Getting started in the mail reader — Describes the basics of using the mail reader. If you are interested only in the basics, you should need only to work through this part of the section. You can then use the paragraph on mail reader commands for reference when you want details on a certain command.
- Mail reader commands — Describes each of the mail reader commands in detail. The commands are divided into groups, depending on the operations they perform, such as reading mail, sending mail, selecting and viewing messages, and printing messages.
- Mailer — Describes the mailer system, which provides the facilities for transmitting and receiving mail messages over the network. These facilities include items such as network mail protocols and functions to parse and interpret mail addresses. You can configure the mailer for adaptation to a variety of mail environments.
- Customizing the mail system — Describes how to customize the mail system.

Mail Reader — Getting Started

31.2 The Explorer mail reader is a Zmacs-based utility that allows you to manage special files containing mail messages. The overall design of the mail reader is based on the directory editor (Dired) and can be accessed and used in much the same way as Dired. Thus, familiarity with Zmacs and Dired should aid you in using the mail reader. The *Explorer Zmacs Editor Reference* provides detailed information about Zmacs and Dired.

The mail reader provides commands that allow you to do the following:

- Receive new messages
- View, organize, and manipulate messages
- Compose messages to send to other users or groups of users

**Entering and Exiting
the Mail Reader**

31.2.1 The Explorer system provides several ways to enter the mail reader, the easiest of which are the following:

- Press SYSTEM M.
- Select the Mail item from the System menu.

When reading mail, the mail reader accesses a file (usually in your directory) that contains a collection of messages called a *message sequence*. The two types of buffers associated with every message sequence are the summary buffer and the message buffer.

NOTE: If you are using the mail reader for the first time, a message sequence will probably not be visible because you have not yet sent or received mail messages.

The *summary buffer* (Figure 31-1), which is the default viewing mode, contains one-line descriptions of all messages in the message sequence. This buffer is marked with indicators as you read your mail. For example, before you view a message, it is marked with an asterisk (*). After viewing the message, the asterisk is removed from the summary buffer.

The *message buffer* (Figure 31-2) displays the text of one message at a time. You can use the message buffer to display a message that you select when using the summary buffer. You can also use it to move around in the message sequence; however, you can still display only one message at a time.

These buffers are very closely related and are used strictly for displaying a message sequence. When you save or modify either of these buffers, you are actually saving or modifying the message sequence. With few exceptions, all mail commands operate identically whether the summary buffer or the message buffer is displayed. Thus, a summary/message buffer pair are often referred to simply as a mail buffer.

You can change the default viewing mode by changing the value of the variable `mail:*user-mail-reading-mode*` (described in paragraph 31.5.4, Mail Variables). You can have the message buffer displayed first when you enter the mail reader instead of the summary buffer, or you can have both buffers displayed simultaneously in two-window mode.

You can also enter the mail utility from a Zmacs buffer by executing the META-X Read Mail command. This command activates the default viewing mode. When you enter this command, you are prompted in the minibuffer for the name of the mail file you wish to access. Press RETURN to accept the default mail file, or enter a different file pathname if you want to access another mail file.

Your default mail file is the value stored in the variable `mail:*user-default-mail-file*`. This value is a pathname derived from the host to which you logged in, your user name (that is, your login directory), and the filename BABYL.TEXT. Paragraph 31.5.4, Mail Variables, explains how to change your default mail file.

The keystrokes Q and END allow you to exit the mail reader or return to the default viewing mode. When exiting, you are asked to save your changes (if necessary). The keystroke ABORT exits the mail reader without asking you to save your changes.

Figure 31-1 Typical Summary Buffer

Attrs.	Msg#	Chars	From	Subject	Date	Keywords
A	1	1366	Sierra@MADRE	Mountain View Homes	3 Jun	{PROPERTY}
	2	433	Bandy@BANDERA	Branding Irons	3 Jun	
A	3	659	Matilda@MATAHARI	Re: Mountain View Homes	7 Jul	{PROPERTY}
	4	365	Big-Tex@LONESTAR	South 40 Fence Line	9 Jul	
	5	1120	Bubba@LONESTAR	Cattle Drive	12 Jul	
	6	1863	Sierra@MADRE	Prices of Mountain View Homes	15 Jul	{PROPERTY}
DA	7	1147	Matilda@MATAHARI	Closing Costs	21 Jul	{PROPERTY}
D	8	496	Albatross@CASABLANCA	Rendezvous	25 Jul	
	9	2523	Rardvark@ARMADILLO	Ant Hills	28 Jul	
A	10	4153	Sierra@MADRE	Congratulations from Mountain	30 Jul	{PROPERTY}
	11	1134	Enterprise@INDUSTRY	Widget Production	31 Jul	
	12	1412	Eyebean@FISHBOWL	Seafood Salad	31 Jul	
	13	2009	Alpha@OMEGA	Beginning to End	1 Aug	
H	14	1477	Matilda@MATAHARI	Mountain View Rock Slide	3 Aug	{PROPERTY}
	15	956	Bob@11@CARMA	Monkey Business	6 Aug	
	16	1421	Bumby@GOLF	Nine Under Par	10 Aug	
	17	2717	Hortense@HYDRA	Hilda's New Hat	15 Aug	
	18	3128	Horatio@HORNBLOWER	Seventy-six Trombones	21 Aug	
	19	1715	Maverick@MULEHEAD	Wild Horse Round Up	26 Aug	
	20	2318	Ledger@DOMAIN	Black Magic	3 Sep	
	21	2124	Atom@OPPENHEIMER	The Big Blast	3 Sep	
	22	1922	Sierra@MADRE	Insurance Exclusions	5 Sep	{PROPERTY}
	23	3854	Zorke1@SNORKEL	Scuba Diving	6 Sep	
	24	4718	F-Lee@MATAHARI	Matilda's Home Insurance	6 Sep	{PROPERTY}
*	25	5430	Romeo@JULIETT	Father's Ladder	7 Sep	

ZRACS (Read-Mail) BABYL.TEXT#> MATILDA; Lima: 6 (55) (Message 14/25 Answered Filed {PROPERTY}) *

/: Select message, /?: Move to point, M: Message Menu, M?: Save/Kill/Yank, B: Mail Menu, B?: System Menu

Figure 31-2 Typical Message Buffer

Date: 8-Aug-86 13:38:09
From: Matilda@MATAHARI
Subject: Mountain View Rock Slide
To: matilda@Lina

Dear Sierra,

You were absolutely right about Mountain View Homes. I moved into my new home just two days ago, and I must tell you that I never saw a more beautiful place in my entire life. The view is magnificent. The beauty and grandeur of the valley that stretches for miles below cannot be described by mortal words. And the sunset is nothing short of breathtaking. I remember thinking: "Mountain View is an absolute utopia! I'll remain here forever."

But there is just one slight problem -- one little item you forgot to mention. Last night, just as I was preparing to retire for the evening, I heard a noise outside that, at first, sounded like the roar of thunder. Then the earth began to tremble as boulders the size of elephants tumbled down the mountainside. Although I had never before seen a rock slide, there was not a doubt in my military mind that this was, indeed, just that.

The rocks continued to tumble and slide down the mountainside for, what seemed like, hours. When the turmoil finally subsided, I climbed out of the shambles that had, just yesterday, been my beautiful home in Mountain View. I was lucky to escape with my life!

Please send me the necessary forms to file a claim against my home owner's insurance.

Sincerely,
Matilda

ZHAC6 (Read-Mail) BABYL.TERT#> MATILDA; Romeo: (57) (Message 14/25 Answered Filed (PROPERTY))

L: Select message, L2: Move to point, M: Message Menu, M2: Save/Kill/Yank, R: Mail Menu, R2: System Menu

Getting Help 31.2.2 The mail reader provides several ways of getting help. The following ways are identical to the ways used in Zmacs.

- **HELP M** — Probably the most useful way, you can press **HELP M** at any time in any of the mail buffers to display a list of available mail commands.
- **HELP A** — Finds all commands whose names contain a specified string.
- **HELP D** — Describes a command that you specify by name.
- **HELP C** — Describes a command that you specify by pressing its keystroke.

Suggestions menus provide another form of help that is available for the mail reader. You can turn on these menus by selecting the Suggestions item from the System menu and then clicking on the Do It item in the menu that appears.

The mail reader provides two menus of commands that you can access at any time in any of the mail buffers. Both of these menus provide help in the mouse documentation window when you position the mouse cursor over a command name.

- The Mail Command menu, which you invoke by clicking right anywhere in the buffers, provides general mail commands.
- The Message Command menu, which you invoke by clicking middle when the mouse cursor is on a specific message, provides commands for that message.

Mail Displays 31.2.3 As mentioned previously, the mail reader provides two separate buffers that allow you to view and manipulate mail messages: the summary buffer (Figure 31-1) and the message buffer (Figure 31-2). Note that you can also display these buffers simultaneously in a two-window mode.

The summary buffer provides a listing of all your messages and lists the following information about these messages:

- **Attrs. (attributes)** — This column shows important characteristics of designated mail messages, for example:
 - **D** — Indicates that you have marked the message for deletion
 - **P** — Indicates that you have marked the message for printing
 - **A** — Indicates that you have answered the message
 - ***** — Indicates that you have not yet viewed the message
 - **!** — Indicates that you have marked the message with the reminder attribute
- **Msg# (message number)** — This column shows the sequential numbers of all messages in the current mail buffer.

- Chars (characters) — This column shows the number of characters in each listed message.
- From — This column shows the network user who originated each of the listed messages. For example, message number 1 was originated by user Sierra at host MADRE.
- Subject — This column shows the subject assigned by the originator of each message.
- Date — This column shows the date on which each message was sent.
- Keywords — This column shows keywords assigned by the owner of the current mail file. Keywords can be reminders to the owner or words associated with more than one message to show that several messages are related. For example, all messages concerning Mountain View Homes are related by the keyword `PROPERTY`.

The message buffer holds all the text of your messages. Only the selected message can be viewed in the message buffer; mail commands that select a different message in the message buffer display only the newly selected message. You can use standard Zmacs commands to scroll, position the cursor, search, yank text from the kill history, and so on. All Zmacs commands are limited to the text of the selected message.

Both the summary and message buffers display the following information in the Zmacs mode line:

- ZMACS — Tells you that you are in Zmacs.
- (Read-Mail) — Tells you that the current Zmacs major mode is Read-Mail mode.
- `BABYL.TEXT#> directory-name;host: (file-version)` — The name of the mail buffer you are reading. In the summary buffer, the character S is between *host* and *file-version*.
- `(Message selected-message-number/total-number-of-messages attributes {keywords})` — Lists the message number of the selected message, the total number of messages in the mail buffer, message attributes (if any) such as Unseen, and message keywords (if any).
- Up or down arrows — Indicate that there is more of the buffer above or below what is displayed on the screen.
- * — Indicates that the mail buffer has modifications that have not been saved.

**Executing
Mail Commands**

31.2.4 The Explorer mail system provides a versatile set of commands that allows you to receive messages from or send messages to other network users, as well as commands for creating new messages or for editing and manipulating existing messages. The Explorer system allows you to execute these commands in the following ways:

- Press a key such as V for View Message.
- Select a command such as View from the Message Command menu.
- Press META-X and type the name of the command. For example, to view a specified message, press META-X, type `view Message` in the mini-buffer, and press RETURN.

Each command description in this section mentions whether the command is on a menu or a keystroke and how to execute the command via META-X. You can execute all mail commands via META-X.

The mail commands can be divided into two groups: general commands that affect all messages and specific commands that affect one message. When you execute a command to operate on a specific message in the summary buffer, the keyboard cursor must be on the line listing the message; the message with the keyboard cursor on its line is referred to as the *selected message*. (When you click the middle mouse button to invoke the Message Command menu on a specific message, the keyboard cursor is first moved to the message highlighted with the mouse cursor box, thus making it the selected message.) In the message buffer, the selected message is the one being displayed, thus the location of the keyboard cursor is not important. Nor is it important when you execute a general command (such as those on the Mail Command menu).

Many of the commands can accept numeric arguments (in the same way that Dired does), regardless of whether you use keystrokes or menus to execute the commands. These numeric arguments modify the action of a command in some way, such as causing the command to operate on multiple messages as specified by the numeric argument.

Note that most commands with numeric arguments consider *consecutive* messages. For example, if you press 5 D in the summary buffer, the line containing the keyboard cursor and the next four messages are marked with a D.

You can also enter negative arguments. For example, if you press - 5 D, the line containing the keyboard cursor and the previous four messages are marked for deletion.

Basic Mail Commands 31.2.5 The following basic mail commands are essential to a new user of the mail reader. You can execute these commands by their keystrokes, from the mail menus, and via META-X. (These commands are described in more detail in later paragraphs.) Also, most Zmacs commands work the same way in the mail reader as they do in Zmacs.

NOTE: If you are using the mail reader for the first time, you may want to send yourself a message so that you can experiment with these commands. Refer to the commands for sending mail in the following paragraphs.

■ **Commands for viewing mail:**

- **G — Get New Mail.** This command reads new messages into the mail buffer. These are messages that have arrived at any of your inbox files since you entered the mail utility.

An *inbox* is a file of new messages written by a mail delivery system. When the mail reader creates your default mail file, it is assigned an inbox automatically. The name of this file is MAIL.TEXT, with the same host and directory name (login directory) as your mail file. Paragraph 31.3.1, Mail Files and Inboxes, describes how to change, add, and delete inboxes.

- **O — Other Mail Buffer.** This command allows you to toggle between displaying the summary buffer and the message buffer.
- **V — View Message.** This command allows you to view the selected message.
- **N — Next Message.** This command allows you to view the next undeleted message (that is, the message is not marked with a D).
- **P — Previous Message.** This command allows you to view the previous undeleted message.

■ **Commands for manipulating messages:**

- **D — Delete Message.** This command allows you to mark the selected message for deletion. The selected message is marked with the attribute D. The marked message is not removed until you invoke the Execute (X) command.
- **CTRL-SHIFT-P — Print Message.** This command allows you to mark the selected message for printing on printers. The selected message is marked with the attribute P. The marked message is not printed until you invoke the Execute (X) command.
- **U — Undelete Message.** This command allows you to remove the deletion mark or the printing mark from a message that you have previously marked for deletion.
- **C — Copy Message to Mail File.** This command allows you to copy a message into another mail file.

- X — Execute. After you mark messages for deletion or printing, this command allows you to perform the operations. This command displays the messages you marked and asks you to confirm the operations.
- Commands for sending mail:
 - M — Mail. This command creates a separate Zmacs buffer, called a *mail template buffer*, that allows you to compose messages to send to other users. The following shows a typical message, ready for delivery:

```
From: Charlie@Bar
To: Noah@Ark
CC: Sam@Tavern,
    Jim@Inn
Subject: Clear Skies
```

We respectfully turn down your invitation. There's not a sign of rain in the skies.

Because the mail template buffer is a Zmacs buffer, all the Zmacs editing commands are available. To create a new mail message, you simply enter the required information in the header fields. Use the Zmacs cursor movement commands to position the cursor at the end of each line and then type the proper information. Next, skip one line, and type the message text. Also, note that pressing HELP M in this buffer displays help on using the template buffer.

A typical mail address consists of the user name and the host name in the form *user@host* (Noah@Ark for example).

Each header field begins in column 0, ends with a colon (:), and is followed by the header contents. The From: header specifies who sent the message and provides an address to which replies can be sent. The From: header is normally generated by the system although you can edit it. The To: header contains the addresses of the primary recipients of the message, and the CC: header contains the addresses of other interested parties.

Multiple addresses in one header must be separated by commas. To continue a list of addresses that exceeds the length of a single line, move to the next line by pressing RETURN and enter spaces or tabs so that the next address does not begin in column 0.

To send the message, press the END key. Press the ABORT key to abort the message.

Note that you can also execute the Mail command from anywhere within Zmacs by pressing CTRL-X M.

- R — Reply. This command creates a mail template buffer (similar to the template buffer for the Mail command) that allows you to reply to the selected message.
- F — Forward. This command creates a mail template buffer (similar to the template buffer for the Mail command) that allows you to forward the selected message to a specified user.

- Other basic commands:
 - B — List Mail Buffers. This command lists all currently active mail buffers and mail template buffers in a mouse-selectable list having a format similar to that for the Zmacs command List Buffers. Note that you can also execute this command anywhere from within Zmacs by pressing CTRL-X CTRL-M.
 - CTRL-X CTRL-S — Save Mail File. This command saves your mail file, including all changes made during the current session.
 - CTRL-X CTRL-R — Revert Mail Buffer. When you execute this command, your most recently saved mail file and any associated inboxes are read. Thus, any new mail is displayed, and any changes that you made during the current session—but did not save—are lost. Use this command rather than G if you want to undo some changes you already made to the mail buffer.
 - Q or END — Exit Mail Reader. These commands allow you to quit your current mail session or return to the default viewing mode. If you have made changes to the mail buffer without saving them, you are prompted to save your changes.
 - ABORT — Abort Mail Reader. You are not prompted to save your changes.

**Mail Reader
Commands**

31.3 The mailer reader commands are divided into the following groups, according to the operation they perform:

- **Mail files and inboxes** — Discusses mail file formats, inbox locations, and the commands that perform operations on mail files and inboxes, such as saving your mail file, getting new mail, and creating new inboxes.
- **Mail buffers and windows** — Describes the commands that perform operations on mail buffers and windows, such as listing your mail buffers, moving between the summary and message buffers, displaying the summary and message buffers in two-window mode, and creating and displaying *filters* (which contain related groups of messages).
- **Mail messages** — Provides an overview of message attributes and message headers. The commands that perform operations on mail messages are divided into the following groups:
 - **Selecting and viewing messages** — Describes the commands that allow you to select and view messages.
 - **Sending mail** — Describes in detail how to use message template buffers and mail addresses; then describes the commands that allow you to send mail.
 - **Deleting and expunging messages** — Describes the commands that delete, undelete, and expunge messages.
 - **Editing messages** — Describes the command that edits messages.
 - **Printing messages** — Describes the command that prints messages.
 - **Message keywords** — Describes the commands that allow you to create, change, and delete message keywords.
 - **Miscellaneous commands** — Discusses the commands that allow you to sort messages, mark messages for command application, perform marked operations, revert your mail buffer, change message attributes, toggle the reminder attribute, and reformat message headers.
- **Command summary** — Presents a table listing all the mail reader commands.

**Mail Files
and Inboxes**

31.3.1 A mail file consists of saved messages organized in a format understood by the mail reader. As mentioned previously, an inbox is a file of new messages written by a mail delivery system. When the mail reader creates your default mail file, it is assigned an inbox automatically; you can assign more than one inbox to a mail file. One task of the mail reader is to read newly arrived mail from designated inboxes into a mail file, then delete the inboxes from which the mail was read. Generally, a mail file should not be used as an inbox, nor should an inbox be used as a mail file.

Mail File Formats 31.3.1.1 The Explorer mail reader supports the following mail file formats:

- **BABYL** — This is the preferred format that supports the full range of features of the Explorer mail reader. This format is also compatible with the following mail readers:
 - Rmail provided by GNU EMACS on UNIX®-based systems
 - BABYL provided by TECO EMACS on TOPS-20™ systems
 - Zmail provided by Symbolics™ and LMI
- **UNIX** — This format is used by most mail utilities on Unix-based machines.
- **TOPS** — This format is used by the MM mail reader on TOPS-20 systems. It is a read-only format to which the Explorer system cannot write.

When the mail reader reads a file, it automatically determines the mail file format. When writing or copying a message to an existing mail file, the format of the existing mail file is used; if the mail file is new, you are prompted for the format to use when creating the file. If the mail reader encounters a file of unknown format, it signals an error condition.

Once a mail file is read, you can convert it to either a UNIX format or a BABYL format. However, since the TOPS format is a read-only format, the Explorer system cannot convert any file to this format. Also, if a BABYL format file is converted to a UNIX format, all message keywords, attributes, and inbox pathnames are lost once the file is written, then read back in.

Inbox Locations 31.3.1.2 Inbox files contain newly arrived mail and are written by mail delivery processes on various systems. The mail reader can read inboxes written by Lisp machines, or UNIX and TOPS systems. Inboxes are typically written to a predetermined disk file according to the user's login name. Some examples of inbox file pathnames are as follows:

- Lisp machines — HOST:USERNAME;MAIL.TEXT
- UNIX systems — HOST:/usr/spool/mail/username or
HOST:/usr/username/mbox
- TOPS systems — HOST:<USERNAME>MAIL.TEXT

UNIX is a registered trademark of AT&T.

TOPS-20 is a trademark of Digital Equipment Corporation.

Symbolics is a trademark of Symbolics, Inc.

31.3.1.3 The following mail commands allow you to perform operations on mail files and inboxes.

Read Mail	Command
------------------	----------------

Keystroke: I or META-X Read Mail

Inputs a specified mail file into a mail file buffer. This command activates the default viewing mode. When you execute the Read Mail command, you are prompted in the minibuffer for the pathname of the mail file you wish to read. If the file you specify does not exist, you are asked if you wish to create it. If you answer yes, you must then specify the format of the new mail file. If the specified file exists, but its format is one that the mail reader cannot recognize, an error condition is signaled in the minibuffer.

The following commands are similar to the Read Mail command:

- **META-X Mail Summary** — This command reads a specified mail file into a mail file buffer and activates the mail summary buffer.
- **META-X Mail Buffer** — This command reads a specified mail file into a mail file buffer and activates the message buffer, making one of the messages visible for reading.

Two additional commands (**META-CTRL-M** and **SYSTEM M**) allow you to go directly to the default mail file specified by the variable **mail:*user-default-mail-file***. **META-CTRL-M** is valid at anytime within Zmacs, whereas **SYSTEM M** can be invoked from anywhere within the Explorer system.

Get New Mail	Command
---------------------	----------------

Keystroke: G or META-X Get New Mail

Mail menu item: Get New Mail

The mail reader executes this command automatically when you first enter the mail reader. However, you can invoke the command at anytime during a given session to determine if you have received new mail. The command checks all inboxes associated with the current mail buffer to determine if they contain any new mail messages. If new messages are found, they are added to the end of the current mail buffer and marked as unseen. The mail reader then selects the first of the new messages for viewing.

Execution of the Get New Mail command saves all new mail messages in a temporary backup file and deletes the original inbox. The backup file exists until you save your mail file, at which time it is deleted. This feature prevents loss of new mail in case a mail buffer containing the new mail is lost before it can be saved. The name of the backup file is typically the same as the original inbox except that the file type is **_ZMAIL_TEMP**.

A numeric argument with this command allows you to read inboxes other than those associated with the current buffer. You are prompted in the minibuffer for the name of the inbox you wish to read. The mail reader then reads the specified file and adds new messages to the mail buffer. In this

case, the automatic backup operation is not performed and the file is not deleted.

Save Mail File Command

Keystroke: CTRL-X CTRL-S or META-X Save Mail File

Mail menu item: Save Mail File

Writes the contents of the current mail file buffer back to the file from which it was read. Successful execution of the Save Mail File command deletes all inbox backup files and excessive versions of the mail file. The number of file versions kept is determined by the variable **mail:*mail-file-versions-kept***. The default is 6.

Note that mail template buffers and filter buffers are not saved. (Filter buffers are discussed in paragraph 31.3.2, Mail Buffers and Windows.)

A numeric argument with this command causes the save operation to be performed in a low-priority, background process.

Write Mail File Command

Keystroke: W, CTRL-X CTRL-W, or META-X Write Mail File

The Write Mail File command is identical to the Save Mail File command except that it allows you to specify the pathname of the file into which you wish to write the contents of the current mail buffer.

Copy Message to Mail File Command

Keystroke: C or META-X Copy Message to Mail File

Mail menu item: Copy

Copies the current message to a specified destination mail file. A message can be copied to any mail file having a format recognized by the Explorer mail system (except TOPS). For example, you can copy a message from a BABYL mail file to a UNIX mail file.

If the specified destination mail file exists, the message is appended to the file in the proper format. If the specified destination mail file does not exist, you are asked in the minibuffer if you wish to create it. If you answer yes, you are asked to specify the format of the new mail file. If the specified file exists but its format is one that the mail reader cannot recognize, an error condition is signaled in the minibuffer.

If the specified destination mail file is already in memory (in a mail buffer), this command appends the copied message to the buffer and reminds you to save the buffer in order not to lose the appended message.

A numeric argument *n* with this command copies *n* messages to the specified destination mail file.

Copy Message to Text File

Command

Keystroke: CTRL-C or META-X Copy Message to Text File

Copies the current message to a specified text file. If the specified file exists, the copied message is appended to it; if the file does not exist, you are asked in the minibuffer if you wish to create it.

The mail reader cannot read a text-format file back in as a mail file. However, since this type of file contains separating page characters between the messages, it is useful for printing copies of a group of messages.

A numeric argument n with this command copies n messages to the specified destination file.

Change Inboxes

Command

Keystroke: META-X Change Inboxes

Mail menu item: Change Inboxes

Allows you to add new inboxes or to delete existing inboxes. If the current mail file has at least one inbox associated with it, the command displays a choice menu of all existing inboxes.

- To delete an existing inbox, click on its name so that it is no longer highlighted; then select the Do It item. This deletes any inboxes associated with the current mail file that are not highlighted.
- To create a new inbox, select the New Inbox item. You are prompted in the minibuffer for the name of the inbox you wish to create.

If the current mail file has no inboxes, the Change Inboxes command simply prompts for the name of the inbox without displaying the choice menu.

Note that the inboxes associated with a mail file can only be saved to mail files in BABYL format. UNIX and TOPS mail files do not provide this facility. If your default mail file is in UNIX format, you can associate an inbox with it by setting the variable `mail:*unix-inbox-pathname*`.

Change File Options

Command

Keystroke: META-X Change File Options

Mail menu item: Change File Options

This command is valid for BABYL files only. The command displays a menu of the following variables whose values can be changed or set by the user:

- **Owner** — A string that contains the user ID of the owner of the current mail file. When any user other than the one specified reads the mail file, the associated inboxes are not read and no changes can be saved. A value of **nil** (the system default value) allows other users to check the inboxes and save changes.
- **Append New Mail** — If set to **Append** (the system default value), adds new messages to the end of the mail file; a value of **Prepend** adds new messages to the beginning of the file.
- **Reformat Headers** — If set to **Yes** (the system default value), automatically reformats message headers; a value of **No** leaves the headers in their original format.

You may want to remove headers that are of no interest to you. Most messages contain a number of headers that provide trace information about where the message originated and the network path it traveled during delivery. When the headers are reformatted, these trace headers are hidden from the display and the remaining headers are listed in a consistent order.

Set Mail File Format

Command

Keystroke: META-X Set Mail File Format

Allows you to change the format of the current mail file. Any mail file format can be converted to either BABYL or UNIX format. However, the TOPS format is a read-only format; therefore, no other format can be converted to TOPS.

NOTE: Converting a BABYL file to UNIX format causes all keywords and attributes to be lost when the file is written.

**Mail Buffers
and Windows**

31.3.2 As mentioned previously, a mail buffer is a special Zmacs buffer that performs operations on a collection of messages called a message sequence. A message sequence contains messages such as those from a mail file or those obtained by filtering a mail file. *Filtering* is a method of selecting a group of messages having a certain specified characteristic, such as a common date.

The two types of buffers associated with every mail sequence are the summary buffer and the message buffer. With few exceptions, all mail commands operate identically whether in the summary buffer or the message buffer. Thus, a summary/message buffer pair are often referred to simply as a mail buffer.

Mail buffers can be generated by either reading a mail file or by filtering an existing message sequence. When it is necessary to distinguish between these two means of mail buffer generation, the buffer is referred to as a *mail file buffer* or a *mail filter buffer*. In addition to creating mail filter buffers by filtering a message sequence, you can change the viewing mode of the summary buffer to *filtered mode*. In this mode, all of the filters you have created are displayed as one-line entries. You can expand these entries to show the message sequences in the filter by using the Expand Filter (S) command as you would use the S command in Dired.

Typically, the summary buffer or the message buffer occupies the full screen, and the normal viewing mode is to switch back and forth between these two buffers. However, an alternate two-window mode is available, in which both buffers can be displayed simultaneously in two separate windows.

The following mail commands allow you to perform operations involving mail buffers and windows.

Other Mail Buffer	Command
--------------------------	----------------

Keystroke: O or META-X Other Mail Buffer

Mail menu item: Other Mail Buffer

Toggles between displaying the summary buffer and the message buffer.

Exit Mail Reader	Command
-------------------------	----------------

Keystroke: Q, END, or META-X Exit Mail Reader

Returns to the default viewing mode or to the current mail file buffer, or exits the mail reader completely. In effect, repeatedly invoking this command *unwinds* the current state of the mail reader until you exit completely, as illustrated in the following examples:

- If you are currently viewing the message buffer and your default viewing mode is the summary buffer, this command returns you to the summary buffer.
- If you are currently in a filter buffer in the default viewing mode, this command returns you to the mail file buffer.
- If you are already in the default viewing mode in a mail file buffer, this command first asks you about messages marked for deletion, printing, and command application (described later). Then the command asks you whether to save the mail file. Finally, the command exits the mail reader by selecting the previous Zmacs buffer on the Zmacs buffer history.

List Mail Buffers

Command

Keystroke: B, CTRL-X CTRL-M, or META-X List Mail Buffers

Lists all currently active mail buffers and mail template buffers in a mouse-selectable list having a format similar to that for the Zmacs command List Buffers. The sequence of the list is as follows:

1. Each mail file buffer, followed by any associated filter buffers
2. Any unsent mail template buffers
3. Any sent mail template buffers

Figure 31-3 shows a sample display. You can select any item from the list by positioning the mouse cursor on the item and clicking the left mouse button. Clicking the right mouse button displays a list of available buffer commands (Save, Unmodify, Kill, Write, and Select).

Although the main Zmacs buffer list includes all currently active mail file buffers, the filter buffers and sent mail template buffers are removed from the list to reduce clutter. Thus, the List Mail Buffers command is the only command that allows you to operate on these items.

An additional keystroke command (CTRL-X CTRL-M, which can be invoked from anywhere in Zmacs) operates exactly the same way as the keystroke B.

Toggle Mail Window Configuration

Command

Keystroke: CTRL-W or META-X Toggle Mail Window Configuration

Toggles between the one-window and two-window configurations. In two-window mode, the summary buffer and the message buffer are simultaneously displayed in two separate windows. The buffers are synchronized so that when you select a message in one buffer's window, the other window is updated to reflect the newly selected message. The synchronization occurs whether you execute the message selection command in the window of the summary buffer or the window of the message buffer.

When you press CTRL-W to return to one-window mode, other windows are deleted and the currently selected buffer (summary or message) occupies the full screen.

Figure 31-3 List Mail Buffers Display

```

Mail Buffers:                               File Version:  Length:  Major Mode:
BABYL.TEXT#> RALPH; Romeo:                   (49)      38 Messages  (Read-Mail)
* BABYL.TEXT#> MARK; Romeo:                  (363)     355 Messages (Read-Mail)
* RMAIL.TEXT#> BUEHRING; Spud:                (590)     257 Messages (Read-Mail)
  (Keyword: ACTION) RMAIL.TEXT#> BUEHRING; *
  (Answered) RMAIL.TEXT#> BUEHRING; Spud:
  (Search: property) RMAIL.TEXT#> BUEHRING; *
                                         1 Message
                                         36 Messages
                                         4 Messages

Unsent Mail Templates:
* *Reply-To-Sender-3*                        17 Lines

Sent Mail Templates:
#Mail-7# "clear skies"                       6 Lines
#Mail-6# "Project Canceled"                  6 Lines

# means buffer modified.  * means name truncated.

```

```

ZMACS (Common-Lisp) *Buffer-1*
Control-X Control-M

```

```

#: Bring up the System Menu.

```

Filtered Mode

Command

Keystroke: META-X Filtered Mode

Toggles between filter summary mode and basic summary mode for the current mail buffer. The current summary is redisplayed in the new mode.

To make filter summary mode your default summary mode, set the variable `mail:*mail-summary-mode*` to `:filtered` in your login initialization file. The initial value of `mail:*mail-summary-mode*` is `:basic`.

The filter summary initially contains one line for each mail filter defined in your message file. No summary lines for the messages are initially displayed.

- To expand a filter, use the Expand Filter command (S).
- To create new filters and save them in your mail file, use the Filter Messages command (=).
- To delete filters from the display and from your mail file, use the Delete Filters command (META-X).

These commands are described in detail in the following paragraphs.

Expand Filter

Command

Keystroke: S

Expands or contracts the filter on the line where the keyboard cursor resides. When you execute this command on a filter that is not currently expanded, the command displays summary lines for all messages that meet the criteria of the filter. These summary lines are displayed directly beneath the filter line, and the filter line is displayed in a heavier font.

When a filter is already expanded, this command removes the filter's summary lines from the display, and the filter line is displayed in a regular font.

This command only works in filtered summary mode.

Filter Messages

Command

Keystroke: = or META-X Filter Messages

Mail menu item: Filter Messages

Creates a new filter, which tests a message sequence for a specified set of criteria.

- If you execute this command in the regular summary mode, the messages that pass the test are copied into a separate buffer called a *filter buffer*.
- If you execute this command in filter summary mode, you are asked if you want to save the filter in your mail file. If you answer N, the filter is created and placed in the filter summary, but it does not appear the next time you read your mail file. If you answer Y, you are prompted for a name to assign to the filter (a default is provided) and the filter appears in the filter summary each time you read your mail file.

Note that this command also differs depending on whether you invoke it with a keystroke or from the Mail Command menu.

- When you use the = key to invoke this command, you are prompted in the minibuffer for a filter name. (The ESCAPE key provides command completion, and CTRL-? provides a list of available possibilities.)
- Invoking the command from the Mail Command menu displays a menu of mouse-selectable filters. You can filter all messages by clicking left on this menu, or you filter the current message sequence only by clicking right on the filter name.

The mail reader provides the following set of default filters:

- Unseen — Messages that have not yet been viewed
- Recent — Messages that arrived after the last save or expunge operation

- Deleted — Messages that have been marked for deletion
- Answered — Messages that have been replied to
- Reminder — Messages that have been marked with the reminder attribute.
- Marked — Messages that have been marked for command application
- Filed — Messages that have been copied to another file
- Keywords — Messages that contain one or more specified keywords
- Search — Messages that contain a specified string
- Subject — Messages that contain a specified string in the Subject: field
- From — Messages that contain a specified string in the From: field
- To/CC — Messages that contain a specified string in the To: field or the CC: Field
- Header — Messages that contain a specified string in a specified header field
- After — Messages that were sent after a specified date
- Before — Messages that were sent before a specified date
- On — Messages that were sent on a specified date

Some of the filters can be inverted by prefixing the filter name with *not*. For example, if you press = and type *not unseen* in the minibuffer, this command copies all messages that *have* been viewed into a filter buffer. When you invoke the command from the Mail menu, the menu provides a mouse-selectable *not* item; to select an inverted filter, first select *not*, then select the desired filter.

Delete Filters

Command

Keystroke: META-X Delete Filters

Deletes the filters that you specify from a menu of all filters. The deleted filters are removed from the display and the mail file.

This command only works in filtered summary mode.

Mail Messages 31.3.3 Each Explorer mail message consists of a group of headers and the message text. These messages may also be marked with attributes and keywords as displayed in the message summary and the Zmacs mode line.

Headers The headers contain all administrative information such as date of origin, subject of the message, and so on. Some common header fields are described in the following list:

- **Date:** — This field is generated by a mail delivery system to mark the date and time that the message was originally sent.
- **From:** — This field specifies who sent the message and provides an address to which replies can be sent. The **From:** field is normally a system-generated field. However, when creating a message, you can override this system-generated field by substituting your own **From:** field to the header. This is desirable when you are logged in to a host other than the one on which your mail file exists.
- **To:** — This field identifies the primary addressees of a message. You must specify one or more primary recipients in every message you send. All other recipients (such as **CC** recipients) are optional.
- **CC:** — This field (carbon copy) specifies recipients of the message other than the primary addressees specified in the **To:** field.
- **Subject:** — This field contains the subject of your message. No structure is imposed on the contents of this field; it can contain any text.
- **Reply-To:** — This field specifies the mail address recipients should use when replying to the message. The **Reply** command uses this field, when present, for replies, rather than using the **From:** field.
- **In-Reply-To:** — This field is often present in a **Reply** message. This field provides a reference to the original message to which you are replying.
- **Message-ID:** — This field is generated by a mail delivery system to uniquely identify one message among all the messages sent from a particular host. This field is usually not displayed after automatic header reformatting.
- **Sender:** — This system-generated field identifies the actual login name of the message's sender; it is present only if the sender's login name differs from the name in the **From:** field. When you create your own **From:** field, the system automatically generates the sender field. This field is usually not displayed after automatic header reformatting.

Paragraph 31.3.3.2, **Sending Mail**, describes how to add headers to mail messages.

For more information on all the possible headers, refer to the following specification:

RFC822 — Standard for the Format of ARPA Internet Text Messages

This specification is available from the following address:

ARPANET Network Information Center
SRI International
Room EJ221
333 Ravenswood Avenue
Menlo Park, CA 94025

Attributes An Explorer mail message may be marked with one of the following attributes. *Attributes* are flags that can be assigned to messages to indicate operations that have been performed on the messages or operations that are pending. (The commands mentioned below are described in different parts of the section according to the operation they perform.)

- Unseen (*) — The text of the message has not been displayed since it was read from an inbox. This attribute appears when you execute the Get New Mail (G) command, and it finds new messages.

NOTE: If you do not save your mail buffer, attributes that you added or removed from messages are not saved. For example, viewing a message removes the * but if you do not save the mail buffer, the * will be there when you read your mail file again.

- Reminder (!) — The message has been marked to indicate that it is an important message. You executed the Toggle Reminder Message (!) command on the message.
- Answered (A) — The message has been replied to. You executed the Reply (R) command on the message.

The following attributes appear when you mark a message for an operation to be performed. You can use the Undelete (U) command to remove these attributes. These operations are not performed until you invoke the Execute (X) command.

- Deleted (D) — The message has been marked for deletion and will be removed from the mail file during the next expunge operation. That is, the user has executed the Delete Message (D) command.
- Marked (@) — The message has been marked for command application (that is, an operation such as forwarding all marked messages to another user). The user has executed the Mail Mark for Apply (A) command.
- Print (P) — The message has been marked for printing. The user has executed the Print Message (CTRL-SHIFT-P) command.

The following attributes appear in the mode line, instead of the summary buffer:

- **Recent** — The message was read from an inbox after the mail file was last saved or expunged. This attribute appears when you execute the Get New Mail (G) command. It is removed when you execute the Save Mail File (CTRL-X CTRL-S) or when you expunge messages by using the Execute (X) command.
- **Filed** — The message has been copied to another file. You executed the Copy Message to Mail File (C) command.

Furthermore, the Change Message Attributes command (described in paragraph 31.3.3.7, Miscellaneous Mail Commands) allows you to assign attributes to the selected message.

Keywords You can assign keywords to your messages as reminders or words associated with more than one message to show that several messages are related. For example, you might assign the keyword `PROPERTY` to all your messages concerning Mountain View Homes.

Selecting and Viewing Messages

31.3.3.1 The *selected message* is the message that all commands operate on by default. If the summary buffer is visible, the keyboard cursor resides on the summary line of the selected message. (Remember that when you click the middle mouse button to invoke the Message Command menu on a specific message, the keyboard cursor is first moved to the message highlighted with the mouse cursor box, thus making it the selected message.) If the message buffer is visible, the text of the selected message is displayed in the buffer. The following commands provide ways of selecting messages.

Next Undeleted Message Command

Keystroke: N or META-X Next Undeleted Message

Selects the next undeleted message. A numeric argument *n* skips forward *n* deleted messages and then selects the next undeleted message.

Previous Undeleted Message Command

Keystroke: P or META-X Previous Undeleted Message

Selects the previous undeleted message. A numeric argument *n* skips backward *n* deleted messages and then selects the previous undeleted message.

Next Message Command

Keystroke: CTRL-N or META-X Next Message

Selects the next message, regardless of whether it has been marked for deletion. A numeric argument *n* skips forward *n* messages and then selects the next message.

Previous Message Command

Keystroke: CTRL-P or META-X Previous Message

Selects the previous message, regardless of whether it has been marked for deletion. A numeric argument *n* skips backward *n* messages and then selects the previous message.

Next Unseen Command

Keystroke: META-X Next Unseen

Selects the next unseen message. A numeric argument *n* skips forward *n* messages and then selects the next unseen message.

Previous Unseen Command

Keystroke: META-X Previous Unseen

Selects the previous unseen message. A numeric argument *n* skips backward *n* messages and then selects the previous unseen message.

Next Message With Keyword Command

Keystroke: SUPER-N or META-X Next Message With Keyword

Prompts for keywords, then selects the next message having the specified keywords. With a numeric argument *n*, skips forward *n* messages, then selects the next message having the specified keywords.

Previous Message With Keyword Command

Keystroke: SUPER-P or META-X Previous Message With Keyword

Prompts for keywords, then selects the previous message having the specified keywords. With a numeric argument *n*, skips backward *n* messages, then selects the previous message having the specified keywords.

First Message Command

Keystroke: < or META-X First Message

Selects the first message in the mail buffer.

Last Message Command

Keystroke: > or META-X Last Message

Selects the last message in the mail buffer.

Jump to Message Command

Keystroke: J or META-X Jump to Message

Selects a message that you specify. If you execute this command in the message buffer, you are prompted in the minibuffer for the message number. If you execute this command in the summary buffer, this command displays the selected message. A numeric argument *n* with this command selects message number *n* without prompting.

Mail Incremental Search Command

Keystroke: META-S or META-X Mail Incremental Search

Searches all messages (from the current cursor position forward) for a specified string in the current mail buffer. You are prompted in the minibuffer to specify the string for which to search. The mail reader displays the first message in which the specified string is found. Pressing the ABORT key returns you to the position from which the search began. ESCAPE exits the search at the current position and selects the current message. CTRL-R reverses the search operation in progress; that is, it performs a backward search for the specified string.

Mail Reverse Incremental Search Command

Keystroke: META-R or META-X Mail Reverse Incremental Search

Works like the Mail Incremental Search command (META-S) except that this command searches backward from the current cursor position. Also, CTRL-S reverses the search operation in progress; that is, it performs a forward search for the specified string.

Next Message Screen Command

Keystroke: space bar or META-X Next Message Screen

If only the message buffer is visible or you are in two-window mode, scrolls the message text in the message buffer forward to the next screen. If only the summary buffer is visible, advances the keyboard cursor to the next summary line.

Previous Message Screen Command

Keystroke: RUBOUT or META-X Previous Message Screen

If only the message buffer is visible or you are in two-window mode, scrolls the message text in the message buffer backward to the previous screen. If only the summary buffer is visible, moves the keyboard cursor backward to the previous summary line.

View Message

Command

Keystroke: V or META-X View Message

Displays the current message in a separate window; you cannot edit the message. While viewing, you can press the following keystrokes to scroll:

- CTRL-V, CTRL-↓, or the space bar to scroll forward a screen of text
- META-V or CTRL-↑ to scroll backward a screen of text
- CTRL-N or ↓ to scroll forward one line
- CTRL-P or ↑ to scroll backward one line

To exit, press END.

Sending Mail

31.3.3.2 The Explorer mail system provides four different types of mail delivery. The Mail command allows you to send new messages to other network users. The Reply command allows you to respond to messages from other users. The Forward command allows you to add comments to a message and forward the message to other users. The Resend command allows you to resend a message unchanged to other users.

Message Template Buffers Each of the commands for sending mail creates a Zmacs buffer, called a *template buffer*, that allows you to edit the message headers and text and to create a complete mail message, ready for delivery to other network users. Each template buffer provides a choice of either sending (the END key) or aborting (the ABORT key) the message. The mail template buffer for the Mail command contains the following header fields, arranged as shown:

```
From: user@host
To:
CC:
Subject:
```

Because this template buffer is a Zmacs buffer, all the Zmacs editing commands are available. To create a new mail message, you simply enter the required information in the header fields. Use the Zmacs cursor movement commands to position the cursor at the end of each line and then type the proper information. You can insert other header fields among the ones provided, skip one line after the headers, and type the message text. A typical message, ready for delivery, appears as follows:

```
From: Charlie@Bar
To: Noah@Ark
CC: Sam@Tavern,
    Jim@Inn
FCC: LM: CHARLIE; NOAH.TEXT
Subject: Clear Skies
```

```
We respectfully turn down your invitation. There's not a sign
of rain in the skies.
```

Each header field begins in column 0, ends with a colon (:), and is followed by the header contents. The From: header specifies who sent the message and provides an address to which replies can be sent. The From: header is normally generated by the system although you can edit it. The To: header contains the addresses of the primary recipients of the message, and the CC: header contains the addresses of other interested parties. The FCC: header (described later) illustrates how you can add headers to this template buffer.

Multiple addresses in one header must be separated by commas. To continue a list of addresses that exceeds the length of a single line, move to the next line by pressing RETURN and enter spaces or tabs so that the next address does not begin in column 0. Paragraph 31.3.3, Mail Messages, provides detailed information about the message headers.

You can add the following useful headers to any mail template buffer:

- **BCC:** — This header (blind carbon copy) specifies the recipients who should receive a copy of the message but should not appear in the final headers of the delivered message. That is, the recipients receive a copy of the message, but the BCC header is deleted before the message is sent.
- **FCC:** — This header (file carbon copy) specifies one pathname to which a copy of the message should be appended. You can specify multiple FCC: fields to append a copy to multiple files.

When you are ready to send a completed message, press the END key to begin the delivery process. This causes the mailer to perform the following steps:

1. Process the message.
2. Verify all recipient addresses to whatever extent possible.
3. Attempt to transmit the message over the network.
4. Display the delivery status of each recipient address.

An address can be accepted for delivery, rejected, or (if for some reason the destination host is not responding) queued for later delivery. If the mailer rejects an address, it displays the reason for rejection. A rejection can be caused by a syntax error, an unknown host, or an unknown user name.

The Forward, Reply, and Resend commands provide template buffers that are similar to the template buffer of the Mail command and are used in the same manner. Also, the Mail Template command provides a menu of these template buffers.

Mail Addresses A typical mail address consists of the user name and the host name in the form *user@host*. However, the exact format can vary depending on the site network configuration and the types of networks accessible at a particular site. Since many people prefer to receive mail on a particular machine, the best way to determine the proper mailing address is to ask the recipient before sending him/her mail the first time. However, when replying to a message received from another network user, the Reply command automatically supplies the proper mailing address.

Refer to paragraph 31.4, *Mailer*, for more information on mail addresses. Also discussed in the *Mailer* paragraph are mailing lists, which allow you to use a single mail address to send mail to multiple recipients.

Return Address One task of the mailer is to create a *From:* field in your message during the sending process. This *From:* field contains the address to which recipients of your message can send replies. The default *From:* field consists of the user and host names (*login-id@login-host*) used when logging in. In this case, you must have a directory that matches your *login-ID* on the file system of the *login-host* and your default mail file must have an inbox named *login-host:login-id;MAIL.TEXT*. (Refer to paragraph 31.3.1, *Mail Files and Inboxes*.)

To have replies sent to a different host and directory, you must specify a different *From:* field. You can do this in any one of the following ways:

- Insert a *From:* field in the mail template buffer prior to sending the message.
- Set the variable `mail:*user-mail-address*` to the string desired for the *From:* field. You can do this in either of the following ways:
 - Use the Profile utility to set the value of `mail:*user-mail-address*`.
 - Enter the function `login-setq` (whose argument is the desired value of the *From:* field) in your *login-init* file, for example:

```
(login-setq mail:*user-mail-address* "in")
```

- Use the namespace editor to create a user object containing a `:mail-address` attribute. Set the value of the attribute to the string desired for the *From:* field. Refer to Section 32, *Namespace Utilities*, for information on the namespace editor.

Commands Although each of the commands for sending mail serves its own unique purpose, they all follow the same procedure for composing and sending a message. All of the following commands send mail across the network.

Mail	Command
-------------	----------------

Keystroke: M, CTRL-X M, or META-X Mail

Allows you to create and send a new mail message. The keystroke sequence CTRL-X M invokes the mail command from anywhere within Zmacs.

META-CTRL-Y inserts a copy of the message that is currently selected in the mail reader into the current mail template buffer.

Reply Command

Keystroke: R or META-X Reply

Allows you to reply to the sender of the current message. When you press the R key, the mail reader automatically fills the To: field with the address of the user who sent the current message. The Subject: field is initialized with the prefix Re: (regarding), followed by the subject of the current message. The keystroke sequence META-CTRL-Y, inserts a copy of the message to which you are replying.

The following numeric arguments, used with this command, provide the indicated features:

- 1 — Creates a reply to all recipients of the current message.
- 2 — Generates the reply in the two-window mode. The top window contains the message being replied to; the bottom window displays the reply template buffer.
- 3 — This automatically performs the equivalent of the keystroke sequence META-CTRL-Y, inserting a copy of the message to which you are replying in the reply template buffer.

Reply to Sender Command

Keystroke: META-X Reply to Sender

Works like the Reply command except that this command allows you to reply only to the sender of the current message.

Reply to All Command

Keystroke: META-X Reply to All

Works like the Reply command except that this command allows you to reply only to all recipients of the current message.

Forward Command

Keystroke: F or META-X Forward

Forwards the current message to specified network users. The Forward command inserts the sender and subject of the current message in the Subject: field of the forward template buffer; the text contains a copy of the current message.

This command allows you to add comments to the message. (The Resend command, which is similar to this command, allows you to resend the message unchanged.) Any reply to a forwarded message is delivered to the person who forwards the message.

Resend Command

Keystroke: META-X Resend

Resends the current message to specified network users. This command is similar to the Forward command except that it resends the message unchanged. Also, any reply to a resent message is delivered to the original sender and not to the person who resends the message.

Mail Template Command

Keystroke: META-X Mail Template

Message menu item: Mail Template

Provides a menu of message template buffers for Mail, Forward, Reply to Sender, Reply to All, and Resend.

Yank Message Command

Keystroke: META-CTRL-Y

Inserts the text of the current message into the mail template buffer at the keyboard cursor position. In a Reply template buffer, this command inserts the contents of the message to which you are replying. The message text is indented three spaces, and the headers are reformatted to include only the Date:, From:, and Subject: fields.

If you supply a numeric argument, the message is inserted with no indentation or header reformatting.

*Deleting and
Expunging Messages*

31.3.3.3 Removing messages from a mail file requires two distinctly different operations called deleting and expunging. As in Dired, *deleting* merely marks the message for deletion whereas *expunging* actually removes the message. This feature greatly reduces chances of accidentally removing messages from a file, and it allows you the freedom to change your mind after deciding to delete a message.

As a further precaution against accidentally removing a message from a mail file, the expunge operation does not affect the mail file until a save operation is performed on the file. Thus, if after performing an expunge operation on a given buffer, you decide you want to keep some of the deleted messages, you can simply execute the Revert Mail Buffer (CTRL-X CTRL-R) command to read the original mail file back in, thereby discarding your changes. (Note that other changes you make, such as keyword and attribute changes, are also discarded.)

Delete Message Command

Keystroke: D or META-X Delete Message

Marks the current message for deletion and moves the keyboard cursor to the next undeleted message. With a numeric argument *n*, marks *n* messages for deletion.

Delete Message Backward Command

Keystroke: CTRL-D or META-X Delete Message Backward

Marks the current message for deletion and moves the keyboard cursor to the previous undeleted message. With a numeric argument *n*, marks *n* messages for deletion.

Undelete Message Command

Keystroke: U or META-X Undelete Message

Unmarks the deletion (D), print (P), and/or marked (@) attribute from the next or previous message. With a numeric argument *n*, this command removes the attribute(s) from *n* messages.

Expunge Mail Buffer Command

Keystroke: X, CTRL-X CTRL-E, or META-X Expunge Mail Buffer

Mail menu item: Expunge Messages

This command differs depending on how you invoke it. If you invoke the command from the Mail Command menu, by pressing CTRL-X CTRL-E, or by pressing META-X, the messages that you marked for deletion are expunged.

The X key actually invokes the Execute command, which allows you to perform expunge, print, or command application operations on the messages that you marked. A list of the messages that you marked for deletion, printing, or command application is displayed. You then specify whether to perform the operations. If you are expunging messages, remember the removals become permanent only when you perform a save operation on the current mail file.

Editing Messages 31.3.3.4 The mail system allows you to edit existing mail messages with the following command.

Edit Message Command

Keystroke: E or META-X Edit Message

Message menu item: Edit

Allows you to edit the text of a message. This command places the current message in a separate buffer in which all standard Zmacs commands are available for editing the message contents. After editing the message, you can either press END to place the edited message back in the mail buffer or press ABORT to discard all changes, thus leaving the message unmodified.

NOTE: While editing a message, be sure to leave the blank line that separates the message headers from the text.

Printing Messages 31.3.3.5 The mail system allows you to print individual messages or groups of messages on printers by using the following command:

Print Message Command

Keystroke: CTRL-SHIFT-P or META-X Print Message

Marks the current message for printing. With a numeric argument *n*, marks *n* messages for printing. (You can use the Undelete (U) command to remove the P attribute.) To actually print the messages, use the Execute (X) command.

When you print a group of messages, a header page is printed first with a summary line for each message.

Message Keywords **31.3.3.6** *Message keywords* are user-defined labels that group messages into common categories and allow you to filter a message sequence according to the labels. The following commands allow you to assign new keywords or to change or delete existing keywords.

Change Message Keywords Command

Keystroke: K or META-X Change Message Keywords

Message menu item: Keywords

Allows you to add keywords to the current message or to delete or modify existing keywords. Note that this command differs depending on whether you invoke it with a keystroke or from the Message Command menu. The K key displays any existing keywords for editing in the minibuffer. You can either delete or modify these existing keywords or add new keywords. (The ESCAPE key or the space bar provides keyword completion, based on previously-assigned keywords; CTRL-? displays a list of mouse-selectable completions.) The keyword list can contain any number of keywords as long as they are separated by commas.

Selecting the Keywords command from the Message menu displays a menu of existing keywords. You can then select or deselect keywords from the keywords menu by toggling between a highlighted (selected) or unhighlighted (deselected) state, then selecting the Do It item. Selecting the New Keyword item allows you to use the minibuffer to add or delete keywords in the same way as with the K key.

Delete Keyword From All Messages Command

Keystroke: META-X Delete Keyword From All Messages

Mail menu item: Delete Keywords

Deletes specified keywords from all messages. When you invoke the command by pressing K, you are prompted in the minibuffer for the keywords to be deleted; invoking the command via the Mail Command menu displays a menu of the keywords from which you can select those to be deleted. A deleted keyword is no longer available for minibuffer completion or menu selection.

Miscellaneous Mail Commands **31.3.3.7** The miscellaneous mail commands allow you to perform the following operations: sort messages, mark messages for command application (such as forwarding all marked messages to another user), perform marked operations, revert your mail buffer, change message attributes, toggle the reminder attribute, toggle the unseen attribute, and reformat message headers.

Sort Messages **Command**

Keystroke: CTRL-X S or META-X Sort Messages

Mail menu item: Sort Messages

Sorts messages according to the option you chose from the menu, as follows:

- Date (forward) — Sorts by Date field, earliest first.
- Date (backward) — Sorts by Date field, latest first.
- From (forward) — Sorts by From field, alphabetically.
- From (backward) — Sorts by From field, reverse alphabetically.
- To (forward) — Sorts by To field, alphabetically.
- To (backward) — Sorts by To field, reverse alphabetically.
- Size (forward) — Sorts by size in characters, smallest first.
- Size (backward) — Sorts by size in characters, largest first.
- Subject (forward) — Sorts by Subject field, alphabetically.
- Subject (backward) — Sorts by Subject field, reverse alphabetically.
- Keywords (forward) — Sorts by message keywords, alphabetically.
- Keywords (backward) — Sorts by the message keywords, reverse alphabetically.

To restore the original ordering, select Restore Original Order.

Mail Mark for Apply **Command**

Keystroke: A or META-X Mail Mark for Apply

Marks a message for command application, such as forwarding all marked messages to another user or copying all marked messages to another mail file. With a numeric argument *n*, marks *n* messages. (You can use the Undelete (U) command to remove the @ attribute.) To specify the function to apply, use the Execute (X) command. (ESCAPE or the space bar provides completion, and CTRL-? provides a list of possible completions.)

Execute

Command

Keystroke: X or META-X Execute

Performs the requested deletions, printing, or command applications. This command displays the messages you marked and asks you to confirm the operations.

If any messages are marked for command application and you answer Y to the Apply command? query, you are prompted in the minibuffer for the command to apply. The following commands are available:

- **Forward** — Creates a mail template buffer to forward all marked messages to another user.
- **Reply** — Creates a mail template buffer to reply to all recipients of all marked messages. Within the template buffer, you can use the Yank Message (META-CTRL-Y) command to yank the text of all marked messages into the template buffer.
- **Filter** — Creates a separate mail filter buffer that contains only the marked messages.
- **Add Keywords** — Adds specified keywords to the marked messages. You are prompted in the minibuffer for the keywords you want to add.
- **Delete Keywords** — Deletes specified keywords from the marked messages. You are prompted in the minibuffer for the keywords you want to delete.
- **Print** — Prints all marked messages on a printer. You are prompted in the minibuffer for the name of the printer to use.
- **Copy** — Copies all marked messages to another mail file. You are prompted in the minibuffer for the pathname to which you want to copy.
- **Kill History Save** — Copies the text of all marked messages into the kill history.

While you are entering the command, completion is available on the ESCAPE key and a list of possibilities is available on CTRL-?.

If any messages are marked for printing and you answer Y to the Print? query, you are prompted in the minibuffer for the name of the printer to use. While you are entering the printer name, completion is available on the ESCAPE key and a list of all possible printers is available on CTRL-?.

If any messages are marked for deletion and you answer Y to the Expunge? query, all messages marked for deletion are removed from the mail file buffer. The expunged messages can only be retrieved by using the Revert Mail Buffer (CTRL-X CTRL-R) command before the mail file is saved.

Revert Mail Buffer Command

Keystroke: CTRL-X CTRL-R or META-X Revert Mail Buffer

Discards changes made to the buffer by reading the original mail file (that is, most recently saved) back into the buffer. Any associated inboxes are also reread. You are prompted in the minibuffer for the name of the buffer to revert.

Change Message Attributes Command

Keystroke: META-X Change Message Attributes

Allows you to assign attributes to the current message. You are prompted in the minibuffer for the attributes you wish to assign to the current message. Enter the attribute names on a single line, with commas separating them in case of multiple attributes. ESCAPE or the space bar provides attribute completion, and CTRL-? provides a list of possible completions. You can enter any of the following values:

ANSWERED, APPLY, DELETED, FILED, PRINT, RECENT, REMIND, UNSEEN

Toggle Reminder Message Command

Keystroke: ! or META-X Toggle Reminder Message

Toggles between the on or off state of a reminder attribute.

Toggle Unseen Message Command

Keystroke: * or META-X Toggle Unseen Message

Toggles between the on or off state of an unseen attribute.

Reformat Message Headers Command

Keystroke: H or META-X Reformat Message Headers

Reformats message headers to clean up noise headers. Usually, this operation is done before you see a message for the first time.

This command allows you to remove headers that are of no interest to you. Most messages contain a number of headers that provide trace information about where the message originated and the network path it traveled during delivery. When the headers are reformatted, these trace headers are hidden from the display and the remaining headers are listed in a consistent order.

With a numeric argument of 1, this command displays the original headers, including the trace information.

Command Summary 31.3.3.8 Table 31-1 lists all the mail reader keystroke commands.

Table 31-1

Explorer Mail Reader Keystroke Commands

Keystroke	Command Name
<i>Mail File and Inbox Commands:</i>	
I	Read Mail
G	Get New Mail
CTRL-X CTRL-S	Save Mail File
W	Write Mail File
C	Copy Message to Mail File
CTRL-C	Copy Message to Text File
META-X ¹	Change Inboxes
META-X	Change File Options
META-X	Set Mail File Format
<i>Mail Buffer and Window Commands:</i>	
O	Other Mail Buffer
Q or END	Exit Mail Reader
ABORT	Abort Mail Reader
B or CTRL-X CTRL-M ²	List Mail Buffers
CTRL-W	Toggle Mail Window Configuration
META-X	Filtered Mode
S ³	Expand Filter
=	Filter Messages
META-X ³	Delete Filters
<i>Selecting and Viewing Message Commands:</i>	
N	Next Undeleted Message
P	Previous Undeleted Message
CTRL-N	Next Message
CTRL-P	Previous Message
META-X	Next Unseen
META-X	Previous Unseen
SUPER-N	Next Message With Keyword
SUPER-P	Previous Message With Keyword
<	First Message
>	Last Message
J	Jump to Message
META-S	Mail Incremental Search
META-R	Mail Reverse Incremental Search
space bar	Next Message Screen
RUBOUT	Previous Message Screen
V	View Message

NOTES:

¹ To invoke any command whose keystroke is listed as META-X, you press META-X and then type the name of the command as listed in the Command Name column. Note that completion is available.

² The CTRL-X CTRL-M, CTRL-X M, and all mailing list commands can be invoked from anywhere within Zmacs.

³ The Expand Filter command (S) and the Delete Filters command (META-X) only work in filter summary mode.

Table 31-1

Explorer Mail Reader Keystroke Commands (Continued)

Keystroke	Command Name
-----------	--------------

Sending Mail Commands:

M or CTRL-X M ²	Mail
R	Reply
META-X	Reply to Sender
META-X	Reply to All
F	Forward
META-X	Resend
META-X	Mail Template
META-CTRL-Y	Yank Message

Deleting and Expunging Message Commands:

D	Delete Message
CTRL-D	Delete Message Backward
U	Undelete Message
X	Expunge Mail Buffer

Editing Message Commands:

E	Edit Message
---	--------------

Printing Message Commands:

CTRL-SHIFT-P	Print Message
--------------	---------------

Message Keyword Commands:

K	Change Message Keywords
META-X	Delete Keyword From All Messages

Miscellaneous Mail Commands:

CTRL-X S	Sort Messages
A	Mail Mark for Apply
X	Execute
CTRL-X CTRL-R	Revert Mail Buffer
META-X	Change Message Attributes
!	Toggle Reminder Message
*	Toggle Unseen Message
H	Reformat Message Headers

Mailing List Commands² (described later):

META-X	List Mailing List
META-X	List Mailing List Membership
META-X	Add Mailing List Member
META-X	Delete Mailing List Member
META-X	Remove Mailing List

NOTE:

² The CTRL-X CTRL-M, CTRL-X M, and all mailing list commands can be invoked from anywhere within Zmacs.

Mailer

31.4 The mailer system provides facilities for transmitting and receiving mail messages over the network. These facilities include network mail protocols, a queuing system for transmission retries, handling of mail forwarding and mailing lists, and functions to parse and interpret mail addresses and message headers.

The mailer is highly configurable for adaptation to a variety of mail environments. A number of namespace objects and attributes are provided that determine the protocol used to communicate with specific hosts and the routing used to deliver messages to a particular mail address.

Mail Addresses

31.4.1 Mail addresses are the common currency of exchange on a mail network. On a simple network administered by a single organization, mail addresses are typically of the form *user-name@host-name*. Thus, to reach someone by mail, you need to know only their *user-name* and the host from which they read their mail.

A mail address can include a *personal name* that precedes the actual address as follows:

Personal name <username@hostname>

The personal name must be first, can be any length, and should contain only alphanumeric characters. The mail address that is used for delivery is enclosed within the < and > characters.

On large networks, a naming scheme exists to further distinguish between hosts by requiring a domain. A domain specifies the organizational hierarchy in which a host is located. For example, Fred@BOLT.ACME.COM and John@ZAP.EE.UofZAP.EDU are mail addresses containing a domain.

User names and host names in mail addresses can also contain an Explorer namespace qualification. For example, the address JOE.NS1 means to look for a user object JOE in the NS1 namespace. If no such user object is found, the mailer assumes that JOE.NS1 is the actual user name.

For more information on mail address formats, refer to the following specifications:

- *RFC822 — Standard for the Format of ARPA Internet Text Messages*
- *RFC882 — Domain Names — Concepts and Facilities*

These specifications are available from the following address:

ARPANET Network Information Center
SRI International
Room EJ221
333 Ravenswood Avenue
Menlo Park, CA 94025

Mailing Lists

31.4.1.1 A *mailing list* is a mail address that expands into multiple recipients during mail delivery. By using a single mail address, you can reach many people at once without having to enter individual addresses. Members of a mailing list usually share an interest in a particular subject and wish to correspond easily via mail messages.

The Explorer mail system supports two ways of specifying a mailing list: one is based on disk files and one is based on the Explorer namespace system.

Disk-based mailing lists are files stored in a directory named MAILING-LISTS with a file type of MLIST. The name component of each file is the name by which you refer to the mailing list. Each line of the file contains a mail address for each member of the mailing list. A line starting with ! or # in column 0 is considered to be a comment.

For example, suppose you want to start a mailing list to reach all those who are interested in eating out for lunch. On the host GRUB, you can create the file GRUB:MAILING-LISTS;LETS-DO-LUNCH.MLIST and insert the mail addresses of all interested parties, as follows:

```
! List of those interested in eating out for lunch
Wimpy@Burger
Popeye@Sea
Olive@Oil
```

Then, from any machine, you can send a mail message to LETS-DO-LUNCH@GRUB, and your message is delivered to Wimpy, Popeye, and Olive on their respective hosts.

Namespace-based mailing lists are stored in a namespace as **:mailing-list** class objects that contain attributes to specify the list of recipients:

:mailing-list — A class of namespace object. All objects under this namespace class are mailing lists. The name of each object is the name of the mailing list and can be used within mail addresses.

:address-list — An attribute of a namespace **:mailing-list** class object. This attribute specifies the mail addresses (a list of strings) that should receive a copy of any messages sent to a particular mailing list.

NOTE: When you specify a string for the value of a namespace object or attribute, the strings are compared without regard for alphabetic case (although the case is preserved in the namespace as you entered it.)

:remark — An attribute of a namespace **:mailing-list** class object. This attribute specifies a brief description (a string) of appropriate topics for a particular mailing list. This attribute exists only for the purpose of documentation.

For information on using the namespace editor to edit a namespace, refer to Section 32, Namespace Utilities.

The following mailing list commands can be invoked from anywhere within Zmacs. (That is, you do not need to be within a mail buffer to invoke them.)

NOTE: These commands only work on mailing lists in a namespace.

When you are prompted in the minibuffer for a mailing list *name* in any of the following commands, you can enter any number of mailing lists by separating them with commas, and the command operates on all of them. When you are entering a mailing list, completion is available on ESCAPE and a mouse-sensitive list of possible completions is available on CTRL-?. When you are prompted for a mailing list *member*, you can enter any number of mail addresses by separating them with commas.

List Mailing List Command

Keystroke: META-X List Mailing List

Prints the members of the specified mailing list(s).

List Mailing List Membership Command

Keystroke: META-X List Mailing List Membership

Lists all mailing lists that contain the specified member name (which can be any string). The member name can contain wildcard characters: an asterisk (*) matches any number of characters, and a question mark (?) matches any single character.

Add Mailing List Member Command

Keystroke: META-X Add Mailing List Member

Adds the specified mail address(es) to the specified mailing list(s). Any mailing lists that do not already exist are created automatically.

Delete Mailing List Member Command

Keystroke: META-X Delete Mailing List Member

Deletes the specified mail address(es) from the specified mailing list(s).

Remove Mailing List Command

Keystroke: META-X Remove Mailing List

Deletes all members of the specified mailing list(s), and deletes the mailing list from the namespace. You are asked for confirmation before the removal takes place.

User Mail Forwarding

31.4.1.2 You can use the following namespace attribute for user mail forwarding:

:mail-address — An attribute of a namespace **:user** class object. This attribute specifies a string that is the preferred mail address for a particular user. For instance, if there is a **:user** object with the name “fred” and the object has a **:mail-address** attribute with a value of “freddy@ready”, then mail addressed simply as “fred” is delivered to “freddy@ready”.

Also, if the namespace name is “acme”, then the mail address “fred.acme” is also delivered to “freddy@ready”.

Note that the mailer always attempts to look up the `:user` object. Even if a message is sent to another Explorer system with the address “fred@fast”, the mailer first goes to the host “fast”; however, when the mailer looks up the `:user` object “fred” on the host “fast”, the mailer sees the `:mail-address` attribute and forwards the message to “freddy@ready”.

This attribute can be overridden on a particular machine by setting the global variable `mail:*user-mail-address*` to a value such as “freddy@ready” or “Fred Freed <Freddy@ready>”. The default value of this attribute is `nil`.

Address Routing

31.4.2 Several configuration options are available to determine how a message is delivered to a particular address. Some addresses are reachable only through certain network paths, or it may be preferred that all messages are handled by a particular host for forwarding.

Options Affecting Address Routing

31.4.2.1 The options that affect address routing are discussed in the following paragraphs.

Note that the `:user`, `:host`, and `:site` classes of a network namespace take advantage of overriding attribute values. When duplication occurs in any one of these classes, the value of the attribute in the `:user` class takes precedence, followed by the attribute in the `:host` class, and finally the attribute in the `:site` class. The *Explorer Networking Reference* describes the network namespace in detail.

:local-mail-domains — An attribute of `:site` namespace objects. This attribute specifies a list of mail domains (strings) of which this site is a member. For example, if the local site is known as “ACME.COM” on a network, in order for the mailer to recognize that the string “BOLT.ACME.COM” refers to a local machine, it must know that ACME.COM is the local domain. In this case, the value of `:local-mail-domains` should be `(“ACME” “ACME.COM”)`.

:default-mail-host — An attribute of `:user`, `:host`, or `:site` namespace objects. This attribute specifies a string used for the host in addresses that do not specify a host. If this option is not set, it defaults to the local machine. Thus, a message sent to JOEBOB from a host named TEX is delivered directly to the file on the local file system `TEX: JOEBOB; MAIL.TEXT`. If this option is set to “DRIVEIN”, a message sent to JOEBOB is delivered to `JOEBOB@DRIVEIN`.

Note that this attribute applies only to addresses entered by the user. It does not affect an address once it is put on the network.

This attribute can be overridden on a particular machine by setting the global variable `mail:*default-mail-host*` to a value such as “DRIVEIN”. This variable has no default value; that is, it is unbound.

:primary-mail-servers — An attribute of **:user**, **:host**, or **:site** namespace objects. This attribute specifies a list of hosts (strings) that act as servers to store and forward mail. Often the primary mail servers are *smart* mail hosts; that is, they have access to a wider range of accessible networks, mailing lists, and/or user forwarding.

This attribute can be overridden on a particular machine by setting the global variable **mail:*primary-mail-servers*** to a value such as (“RELAY” “BIG-BROTHER”). This variable has no default value; that is, it is unbound.

:use-primary-mail-servers — An attribute of **:user**, **:host**, or **:site** namespace objects. This attribute specifies under what conditions to forward mail to a primary server.

With the value **:always**, the mailer never attempts delivery to any other host but a primary server. In this case, nonlocal addresses are not verified beyond the simplest syntax check; it is the server’s responsibility to provide verification and final delivery.

With the value **:unknown-addresses**, the mailer attempts delivery to all known users and host, but any unknown address are sent to a primary server for handling.

With the value **:after-first-attempt**, the mailer fully verifies and attempts delivery to all addresses, but the mail for hosts that do not respond is forwarded to a primary server for delivery retries.

With the value **:never**, the mailer never uses a primary server. All verification, delivery, and retries are performed by the local host.

This attribute can be overridden on a particular machine by setting the global variable **mail:*use-primary-mail-servers*** to a value such as **:after-first-attempt**. This variable has no default value; that is, it is unbound.

:top-level-mail-domain-servers — An attribute of **:site** namespace objects. This attribute specifies an association list of hosts that handle mail domains and which domains they can handle. For example, if host MOM handles the domains EDU and COM and if host POP handles UUCP and BITNET, then this attribute should have the following value:

```
(("MOM "EDU" "COM") ("POP" "UUCP" "BITNET"))
```

Thus, mail sent to MINNIE@DISNEY.COM is directed to MOM, and mail sent to MICKEY@HOME.UUCP is directed to POP.

This list is used as a last resort after first checking whether an address contains a known host or one of the **:local-mail-domains**.

:mail-gateway-host — An attribute of **:host** namespace objects. This attribute specifies a host name (a string) to use as a substitute for sending mail to a particular host. For example, if the host “TALK” is on the network but for some reason cannot accept mail from Explorer machines and if the host “LISTEN” can deliver mail to “TALK” as well as accept mail from Explorer machines, then the **:mail-gateway-host** attribute of “TALK” should be set to “LISTEN”.

:uucp-gateway-host — An attribute of **:site** namespace objects. This attribute specifies a host name (a string) that can accept and relay mail on the UUCP (UNIX-to-UNIX copy) network. A UUCP address is one that contains no @ character but one or more ! characters. Explorer machines cannot communicate on the UUCP network; however, if a UUCP address is encountered and a **:uucp-gateway-host** is defined, the mail is forwarded to it.

:reject-mail — An attribute of **:host** namespace objects. This attribute specifies when a host should reject mail. With a value of **:always**, this host never accepts mail from other machines. With a value of **:non-local-addresses**, this host rejects any messages that are not for this host (that is, this host does not forward mail to others). With a value of **:never**, this host accepts any mail for a valid address.

*Address-Routing
Algorithm*

31.4.2.2 The following outline illustrates the algorithm used to determine how a particular address is routed and how the configuration options can affect routing:

1. If the address has no host specified and this is an interactive delivery (for example, invoked by pressing END in a mail template buffer), the mailer supplies the host by using either the **:default-mail-host**, if there is one, or the local host.
2. If **:use-primary-mail-servers** has the value of **:always**, the mailer delivers the mail to a server and exits. Otherwise, the mailer proceeds to the next step.
3. If the address is for the local host, the mailer checks for user forwarding or a mailing list in the following order:
 - a. Disk mailing lists
 - b. Namespace mailing list
 - c. **:mail-address** attribute on a **:user** namespace object

If user forwarding or a mailing list is found, the mailer restarts this algorithm on the new address(es). Otherwise, the mailer delivers the mail to the MAIL.TEXT file in the proper directory on the local machine and exits. (For example, mail for the address Barney@Bedrock is written to the inbox BEDROCK:BARNEY; MAIL.TEXT.) If the directory does not exist, the mailer rejects the address.

4. If the address is not for the local host, the mailer looks up the host in the namespace. If the host is known, the mailer delivers the mail to the host or its **:mail-gateway-host** attribute, if it has one, and exits.
5. If the host is unknown and does not contain a . (period), the mailer rejects the address.
6. If the domain is one of the **:local-mail-domains**, the mailer removes the domain and retries step 4.
7. The mailer checks if one of the **:top-level-mail-domain-servers** will accept the domain, and if so, the mailer forwards the mail to the specified server. Otherwise, the mailer rejects the address.

Mail Daemon 31.4.3 The mail daemon, which is part of the mailer, runs as a background process to deliver outgoing mail, store and forward incoming mail, and probe inboxes for new mail.

Mail waiting to be delivered is kept in the *mail queue*. The mail queue is stored on disk in the MAILER directory and read into memory by the mail daemon when the system is booted.

The mail daemon attempts to deliver mail waiting in the queue every 20 minutes and probes inboxes for new mail every 10 minutes. If a message cannot be fully delivered after 36 hours, a warning message is returned to the original sender indicating which recipients have not yet received a copy, how long the message has been in the queue, and how much longer delivery attempts will continue. If a message cannot be fully delivered after 4 days, the message is removed from the queue and returned to the original sender as undeliverable.

mail:print-mail-queue &optional (*host-queues t*) *messages* Function

Prints the state of the mail queue and the mail daemon. First, information is printed about the state of the mail daemon. Then, if *messages* is non-nil, information is printed about each message waiting in the queue. Finally, if *host-queues* is non-nil, information is printed about each host that has mail waiting to be delivered to it.

mail:reset-mail-daemon &optional *enable* &key (:read-queue t) (:log-pathname mail:*mailer-log-pathname*) Function

Resets the mail daemon process. If *enable* is t, this function restarts the mail daemon process; otherwise, this function leaves the daemon process disabled. If :read-queue is t, this function rebuilds the memory image of the mail queue from the files stored in the MAILER directory; otherwise, the mail queue currently in memory is used again. If :log-pathname is specified, this function logs mailer events to the specified pathname. (The default is "LM: MAILER; LOG.TEXT".)

mail:*log-enabled* Variable

Specifies the level of mailer logging that you want. The default value t logs only important events, such as mail arrival and delivery. A value of nil disables logging altogether. A value of :all produces a large amount of debugging information in the log file. To enable logging of debug information for only certain parts of the mailer, the value can also be a list with one or more of the following elements:

:mail-queue	:delivery	:chaos-mail-client
:chaos-mail-server	:smtp-client	:smtp-server

Customizing the Mail System

31.5 To supplement the system-defined filters and template buffers described previously, you can customize the mail utility by defining your own filters and template buffers with the following macros. You can also change the value of the following variables to access a mail file different from your usual mail file, to activate a mode different from the default mode, to access template buffers for messages, replies, and forwarded messages different from those provided by the mail utility, and so on. The definitions for the system-defined default template buffers are in the file SYS:MAIL-READER; SEND-

MAIL.LISP. The definitions for the system-defined filters are at the end of the file SYS:MAIL-READER; FILTER.LISP.

However, note that when you are defining mail filters and template buffers with the macros described in the following paragraphs, they last only for your current session on the Explorer system unless you save the definitions in your environment. One way to do this is to put your filter and template buffer definitions into a LOGIN-INIT file. If you have your own Explorer system, you can perform a disk-save on your environment.

The following flavor and its associated methods are sometimes used by the macros that define mail filters and message template buffers.

mail:message Flavor

This flavor, which includes the **mail:message-node** flavor as a mixin, represents the message object. The special variable **mail:*msg*** is always bound to the current instance of **mail:message**. The **mail:message** flavor has several methods, described immediately following.

:headers-end-bp Method of **mail:message**

The value returned by this method is a buffer pointer to the beginning of the text of the message, after the headers.

:attributes Method of **mail:message**

The value returned by this method is a list of message attributes, such as **:unseen**, **:deleted**, **:answered**, **:recent**, and **:remind**.

:keywords Method of **mail:message**

The value returned by this method is a list of keywords for the message. All keywords are in the **KEYWORD** package. For example, if one of your keywords is named **mail**, it appears in this list as **:mail**.

The **mail:message** flavor also has three important instance variables inherited from the **mail:message-node** flavor.

:first-bp Method of **mail:message**

The value returned by this method points to the beginning of the message headers.

:last-bp Method of **mail:message**

The value returned by this method points to the end of the message text.

:name Method of **mail:message**

The value returned by this method is a string representing the contents of the **:Subject** field of the message.

Defining Mail Filters 31.5.1 The following macro can be used to define your own mail filters.

mail:define-mail-filter *function-name filter-name-string* Macro
doc-string &rest body

This macro defines a mail filter with the name specified by *filter-name-string*.

Arguments: *function-name* — The *function-name* argument names the symbolic function specification that you want to define as a function (the same as the *name* argument to `defun`).

filter-name-string — The *filter-name-string* argument specifies the name for the new mail filter. The value supplied for this argument must be a string.

doc-string — The *doc-string* argument specifies the documentation string describing the filter.

body — The *body* argument should return `t` or `nil`. A variable called `mail:msg` is automatically bound to the message object by the macro and is given to the user as a local variable. You can send the `:keywords` and `:attributes` methods to this object. The `:keywords` method returns a list of keywords for the message object. The `:attributes` method returns a list that can contain `:unseen`, `:recent`, `:deleted`, `:remind`, and `:answered`.

Example: The following are two uses of `mail:define-mail-filter` to create new filters. The first filter will list all messages that both have the `:remind` attribute and are associated with the `:bug` keyword:

```
(mail:define-mail-filter bug-remind-filter "Reminder-Bugs"
 "Messages in this filter will contain messages with the keyword
 of Bug that have been marked with the Remind attribute."
 (and
  (member :remind (send mail:msg :attributes))
  (member :bug (send mail:msg :keywords))))
```

The second filter lists all messages that have the word *bug* in their subject fields:

```
(mail:define-mail-filter bug-filter "bugs"
 "Messages in this filter will contain the word bug in the
 subject field."
 (string-search "bug" (send mail:msg :name)))
```

Defining Mail Template Buffers 31.5.2 The `mail:define-mail-template` function allows you to define custom mail template buffers to use in addition to the system-supplied ones. Several mail variables are described that control which mail template buffer is used for each mail-sending command. Also described are some useful helper functions. The following method can be used in the body of `mail:define-mail-template` for formatting purposes.

`:set-point` (*bp*) Method of `zwei:interval-stream`

This method moves the buffer pointer for the buffer associated with the current instance of `zwei:interval-stream` to the position specified by *bp*. Thus, for example, you can set *bp* to be the point immediately after the colon in the `To` field so that when the defined template buffer is retrieved, the keyboard cursor will be where the user should begin typing—rather than in the upper left corner of the screen. For such a use of this method, see the example for `mail:define-mail-template` immediately following.

The following macro can be used to create your own mail template buffers.

mail:define-mail-template *function-name* *template-name-string* Macro
template-type *doc-string* &rest *body*

This macro allows you to define custom mail template buffers to use other than the system-supplied ones. Thus, you can create personalized template buffers for reports and the like.

Arguments:

function-name — The *function-name* argument names the symbolic function specification you want to define as a function (the same as the *name* argument to `defun`).

template-name-string — The *template-name-string* argument specifies the string describing the template buffer; this string is then included in the Mail Template command.

template-type — The *template-type* argument specifies the type of template buffer you are defining. Valid values for this argument are `:reply` for a reply template buffer, `:forward` for a forwarding template buffer, `:bug-from-error-handler` for a bug template buffer provided by the error handler, and `:mail` for any other kind of template buffer. These values are sometimes used by the mail utility to set a message's attribute. For example, if you define a template buffer with a *template-type* of `:reply`, the `:answered` attribute is set for any message that is replied to with this template buffer.

doc-string — The *doc-string* argument contains the documentation string describing this template buffer.

body — The *body* argument is a set of arbitrary forms that should be used to initialize the buffer pointed to by the `mail:*interval*` variable. The `mail:*interval*` variable is bound to the buffer in which the message is being composed. The `mail:*msg*` variable is bound to the current instance of the `mail:message` flavor.

Example: The following code creates a forwarding template buffer with `to` and `subject` headers.

```
(mail:define-mail-template my-mail-forwarding-template
  "My Forward Template" :forward
  "Template used to forward mail"
  (let (bp)
    (with-open-stream (stream (zwei:interval-stream
                              mail:*interval*))
      (format stream "TO      :")
      (setf bp (send stream :read-bp)) ;; Save buffer pointer
                                           ;; to point after the TO:
      (format stream "-%SUBJECT :")
      (format stream "-2% -----2%")
      ;; Copy message being replied to into buffer
      (with-open-stream (in-stream (zwei:interval-stream
                                    mail:*msg*))
        (stream-copy-until-eof in-stream stream 8))
      (format stream "-2% -----2%")
      (send stream :set-point bp)      ;; Position the cursor at the
                                           ;; buffer pointer set above
      )))
```

The following mail variables control which mail template buffer is used for each mail-sending command. The values of these variables can be set by using the profile utility.

- mail:*mail-template*** Variable
 This variable contains the mail template buffer used by the Mail (M) command to send a new message. The default value of this variable is **mail:default-mail-template**.
- mail:*reply-template*** Variable
 This variable contains the mail template buffer used when a numeric argument is not passed to the Reply (R) command. The default value of this variable is **mail:default-reply-to-sender-template**.
- mail:*reply-template-1*** Variable
 This variable contains the mail template buffer used when a numeric argument of 1 is passed to the Reply (R) command. The default value of this variable is **mail:default-reply-to-all-template**.
- mail:*reply-to-sender-template*** Variable
 This variable contains the mail template buffer used by the META-X Reply to Sender command. The default value of this variable is **mail:default-reply-to-sender-template**.
- mail:*reply-to-all-template*** Variable
 This variable contains the mail template buffer used by the META-X Reply to All command. The default value of this variable is **mail:default-reply-to-all-template**.
- mail:*forward-template*** Variable
 This variable contains the mail template buffer used by the Forward (F) command. The default value of this variable is **mail:default-forward-template**.
- mail:*resend-template*** Variable
 This variable contains the mail template buffer used by the META-X Resend Message command. The default value of this variable is **mail:default-resend-template**.

The following functions are useful when defining a mail template buffer:

- mail:insert-default-header-fields *bp*** Function
 This function inserts defaults for From:, Reply-To:, FCC:, and BCC: fields at the buffer pointer *bp*.
- mail:insert-header-field *bp header-type &optional contents (trailing-newline-p t)*** Function
 This function inserts a header of *header-type* (a keyword) at the buffer pointer *bp*, followed by the *contents* (a string) and finally by a newline unless *trailing-newline-p* is nil. Example of *header-type* are :from, :subject, :to, and :cc, and so on; any message header keyword is valid.

mail:insert-address-list *bp header-type address-list* Function
 &optional (*trailing-cr-p t*)

This function inserts a header of *header-type* (a keyword) at the buffer pointer *bp*, followed by the *address-list* (a list of address objects) separated by commas and followed by a newline unless *trailing-newline-p* is *nil*.

Mail Functions 31.5.3 The following functions provide a variety of features for reading, sending, and receiving mail.

mail:preload-mail-file &optional *mail-file* Function

This function loads your mail file or several mail files in the background so that you can later quickly read the desired files. The *mail-file* argument, which defaults to the value of the **mail:*user-default-mail-file*** variable, can either be a single filename or a list of filenames.

mail:submit-mail *msg-stream &key :to :subject :other-headers* Function
 (:background (not mail:*try-mail-now-p*))

This function sends a mail message. The function reads the message from *msg-stream*, collects recipients from the headers of *msg-stream*, and attempts to send. Required fields, such as From:, Date:, and Message-ID:, that are not supplied are generated automatically.

msg-stream — Either a stream or a string.

:background — Specifies that the message is not sent immediately (the default). The message is queued for background delivery.

:to — A list of recipients (strings or address objects). In this case, *msg-stream* is assumed to contain only text and is not scanned for headers. Proper headers are generated automatically.

:subject — A string to use for the Subject: field when **:to** is supplied.

:other-headers — A list of strings that contain other headers to insert when **:to** is supplied.

Examples: The following example sends a mail message to Me@Work, including the current time in the text:

```
(mail:submit-mail (format nil "To: Me@Work-%Subject: Go Home-%%Get
                          outta here... it's -A."
                          (time:print-current-time nil)))
```

The following example sends a message to Timekeeper@Watch, containing only the current time:

```
(mail:submit-mail (time:print-current-time nil)
  :to `("Timekeeper@Watch") :subject "Current Time"
  :other-headers `("From: Hickory@Dickory"
                  "Reply-To: Dock@Clock"))
```

mail:add-mail-inbox-probe *pathname* Function

This function adds a probe for new mail arriving at the file specified by *pathname*. When new mail arrives at this *pathname*, you receive a notification message. Because the name of an inbox file is usually MAIL.TEXT, you should specify *host:directory;MAIL.TEXT* for *pathname*, where *host* and *directory* are the same as for your mail file. The *pathname* argument can be a pathname object or a pathname string. Probes for new mail are added automatically for inboxes associated with the mail file(s) you are reading (including new inboxes specified with the Change Inboxes command).

mail:remove-mail-inbox-probe *pathname* Function

This function removes an inbox probe (added by the preceding function) from the specified *pathname*. The *pathname* argument can be a pathname object or a pathname string. Inbox probes are removed automatically when you log out.

Mail Variables 31.5.4 The following variables modify various characteristics of the mail reader. The values of these variables can be set by using the profile utility.

Mail File and Inbox Variables 31.5.4.1 The mail file and inbox variables are as follows:

mail:*user-default-mail-file* Variable

This variable specifies the default name of your personal mail file. This variable can be set by the user, but should only be accessed by calling the **mail:default-mail-file** function. The default value of this variable is a pathname of the form *login-host:login-directory;BABYL.TEXT#>*.

Example: (setq mail:*user-default-mail-file* "thor:jones;babyl.text")

mail:*preload-mail-file-p* Variable

When this variable is set to **t**, your mail file (specified by the **mail:*user-default-mail-file*** variable) is automatically loaded in the background when you log in. The default value is **nil**.

mail:*always-check-inboxes* Variable

When this variable is **t**, the mail reader probes the inboxes and reads new mail from them each time you enter the mail reader. This operation can be time-consuming for inboxes on a remote host; therefore, it might be desirable to set this variable to **nil**. Its default value is **t**.

mail:*save-mail-file-in-background* Variable

This variable determines when a modified mail file is to be saved in a background process. The permissible values are as follows:

- **:always** — Always save in background without asking
- **:never** — Never save in background
- **:ask** (the default value) — Query about saving a mail file, but only if it has been modified since the last query

mail:*mail-file-versions-kept* Variable

This variable specifies the number of versions of the mail file to keep after writing a new version. When this variable is set to `nil` (or equal to or less than 1), old versions are never cleaned up. The default value is 6.

mail:*unix-inbox-pathname* Variable

The pathname(s) of inbox(es) to check when the mail file specified by the variable **mail:*user-default-mail-file*** is a UNIX mail file. UNIX mail files do not support built-in inboxes as do BABYL mail files. The default value is `nil`.

mail:*inhibit-mail-file-format-warnings* Variable

When this variable is `t`, you are not warned about the limitations of non-BABYL mail file formats after the mail file is read.

mail:*delete-message-after-copy* Variable

When this variable is `t`, this variable indicates that a message is to be deleted from the source buffer after copying it to another file or buffer. The default value is `nil`.

mail:*default-other-mail-file* Variable

This variable specifies the default pathname of the file to be used by the Copy Message command. The default value is `nil`.

mail:*probe-for-new-mail-p* Variable

When this variable is `t`, you are notified when new mail arrives in one of your inboxes.

mail:*try-mail-now-p* Variable

When this variable is `t`, the mailer attempts to send mail in foreground before queuing.

mail:*kill-mail-buffers-at-logout-p* Variable

When this variable is `t`, mail buffers are killed when you log out.

Mail Window and Buffer Variables 31.5.4.2 The mail window and buffer variables are as follows:

mail:*user-mail-reading-mode* Variable

This variable specifies the default display mode for reading mail. The variable can be set to any of the following values:

- **:summary** — This value displays the summary buffer in a single window on entering the mail reader. This is the default viewing mode.
- **:message** — This value displays the message buffer in a single window on entering the mail reader.
- **:both** — This value displays the summary buffer and the message buffer in two separate windows.

mail:*mail-summary-mode* Variable

This variable specifies the default display mode for the summary buffer. The variable can be set to either of the following values:

- **:basic** — This value, which is the initial value of the variable, displays the summary buffer in its regular mode.
- **:filtered** — This value displays the summary buffer in filtered mode.

mail:*sticky-mail-window-configuration-p* Variable

When this variable is **t**, the window configuration (such as two-window mode) that is used when switching to a mail buffer is the same as the last time the buffer was selected, or if some other mail buffer is currently selected, its window configuration is used. Otherwise, the default configuration is always used. The default value is **t**.

mail:*sticky-mail-buffer-selection-p* Variable

When this variable is **t**, selecting a mail file buffer invokes whatever message sequence (such as a filter buffer) was selected before. Otherwise, the mail file buffer is always selected. The default value is **t**.

mail:*mail-summary-window-fraction* Variable

This variable determines the fraction of the Zmacs frame that is devoted to the summary buffer when in the two-window viewing mode. The value must be in the range of 0.0 through 1.0. The default value is 0.35.

mail:*mail-summary-template* Variable

This variable is a property list of alternating keyword-value pairs that determine the contents of the summary lines in the summary buffer and the order of items within each line. Negative values can be used to indicate right justification. The keywords and their values are as follows:

- **:length** — The permissible values are **:lines** and **:chars**. When this keyword is set to **:lines**, the message size is shown by the number of lines; when this keyword is set to **:chars** (the default value), the message size is shown by the number of characters.
- **:from** — The number of spaces to allow for the From: field. The default value is 30.
- **:subject** — The number of spaces to allow for the Subject: field. The default value is 30.
- **:date** — These values determine the contents of the Date: field; the permissible values are as follows:
 - **:date-and-time** — Day, month, year, and time of message origin
 - **:date** — Day, month, and year of message origin
 - **:brief-date** (the default value) — Day and month of message origin
- **:keywords** — The number of spaces to allow for the Keywords field. The default value is 30.

mail:*mail-summary-attribute-char-alist*

Variable

This variable contains *attribute character column* sets that specify the message attributes, the characters for the attribute, and the columns (0 through 5) in which to print the attributes. The default value of this variable is a list with the following items:

- (:apply #\@ 0) — Indicates that the apply attribute (@) is to be printed in column 0.
- (:print #\P 1) — Indicates that the print attribute (P) is to be printed in column 1.
- (:unseen #* 2) — Indicates that the unseen attribute (*) is to be printed in column 2.
- (:deleted #\D 3) — Indicates that the delete attribute (D) is to be printed in column 3.
- (:answered #\A 4) — Indicates that the answered attribute (A) is to be printed in column 4.
- (:remind #\! 5) — Indicates that the remind attribute (!) is to be printed in column 5.

mail:*box-summary-lines*

Variable

When this variable is *t*, the mouse blinker forms a box around the lines that describe messages in the summary buffer. When this variable is *nil*, the lines are not boxed. The default value is *t*.

*Mail Template
Buffer Variables*

31.5.4.3 The mail template buffer variables are as follows:

mail:*two-window-reply*

Variable

When this variable is *t*, this variable causes the Reply command to display the message being replied to in one window and to display the message being edited in another. The default value is *nil*.

mail:*dont-reply-to*

Variable

A list of mail addresses (strings) to exclude from replies. The default value is *nil*.

mail:*in-reply-to-template*

Variable

A list that determines the pattern of the In-reply-to: field of reply template buffers. An element of this list can be a string or any one of the keywords *:date*, *:from*, *:to*, *:message-id*, or *:phrase*. The keyword *:phrase* (if present) is the *foo* part of *foo <bar@baz>*; otherwise, it is the same as the keyword *:from*. The default value of the variable **mail:*in-reply-to-template*** is ("Msg of" :date " from " :from). If the variable is set to *nil*, no In-reply-to: field is generated.

The default template buffer generates In-reply-to: fields such as the following:

```
In-reply-to:  Msg of 13-Nov-86 8:48:13-CST from John Smith <JS@Blue>
```

The list (:phrase \"'s message of \" :date) generates the following:

```
In-reply-to:  John Smith's message of 13-Nov-86 8:48:13-CST
```

- mail:*default-bcc-string*** Variable
 This variable is a string containing mail addresses to use in a blind carbon copy (BCC:) field of all mail template buffers. When a message is sent, a copy goes to these addresses, but the field does not physically appear in the message header. The default value is `nil`.
- mail:*default-fcc-string*** Variable
 A string containing a pathname to use in a file carbon copy (FCC:) field of all mail template buffers. When the message is sent, it is appended to the file specified by this pathname. The default value is `nil`.
- mail:*default-reply-to-string*** Variable
 A string containing a mail address to be used in a Reply-to: field of all mail template buffers. This field specifies where replies are to be sent, and it overrides the From: field. The default value is `nil`.
- mail:*user-mail-address*** Variable
 This variable overrides the `:mail-address` attribute of the namespace `:user` class object. (Refer to paragraph 31.4.1.2, User Mail Forwarding.)
- mail:*reply-to-all-header-types*** Variable
 This variable contains the header types from which addresses are collected for the Reply to All command. (Examples of header types are To: and :From.)
- mail:*mail-mode-hook*** Variable
 This variable specifies the function that is called after you set up a new mail template buffer to provide custom initializations.

Reformat Header Variables 31.5.4.4 The reformat header variables are as follows:

- mail:*reformat-headers-automatically*** Variable
 When this variable is `t`, message headers are always reformatted automatically before they are displayed. The default value is `t`.
- mail:*reformat-headers-include-list*** Variable
 This variable contains a list of headers to be kept when reformatting message headers. The order of values in the list determines the order of the header fields in the reformatted headers. If the variable **mail:*reformat-headers-exclude-list*** also has a value, it takes precedence in specifying which headers to keep, but this list still determines the order of the fields. The default values are `:date`, `:from`, `:subject`, `:reply-to`, `:to`, and `:cc`, in that order.
- mail:*reformat-headers-exclude-list*** Variable
 This variable contains a list of headers to be eliminated when reformatting message headers. In those cases where items on this list are also included in the **mail:*reformat-headers-include-list***, this list takes priority. However, **mail:*reformat-headers-include-list*** still determines the order of the headers. In general, the headers included in this list are those considered to be of little or no importance to the message viewer.

mail:*reformat-headers-case* Variable

This variable specifies the case to be used for reformatted header labels. The permissible values are **:upcase** (uppercase), **:downcase** (lowercase), or **:capitalize** (initial letter capitalized). The default value is **:capitalize**. If this variable is set to any other value, the header case is untouched.

mail:*reformat-headers-body-goal-column* Variable

This variable specifies the column in which the bodies of the header fields begin. The default value is 6.

Miscellaneous Variables 31.5.4.5 The miscellaneous variables are as follows:

mail:*yank-message-headers-include-list* Variable

This variable specifies which headers to include when copying a message being replied to. The default values are **:date**, **:from**, and **:subject**.

mail:*yank-message-prefix* Variable

This variable specifies the prefix string for each line of a yanked message. The default value is " " (that is, three spaces).

mail:*unsent-message-query-p* Variable

When this variable is **t**, this variable causes the mail reader to ask if you wish to continue editing a previously started but unsent message. This situation occurs when you initiate a send message operation. The default value is **t**.

mail:*upcase-message-keywords-p* Variable

When this variable is **t**, message keywords are converted to all uppercase. When this variable is **nil**, message keywords remain in the case in which they were created. The default value is **t**.

mail:*choose-from-all-mail-keywords-p* Variable

When this variable is **t**, the menus for keyword selection allow you to choose from all keywords known to the system. Otherwise, the choices are limited to the keywords present in the current mail buffer.

NAMESPACE UTILITIES

Overview

32.1 The namespace utilities allow you to create a database for your network configuration in which you can specify items such as mailing lists, hosts (either logical or physical), printers, sites, and users on the network. You can also use the namespace utilities to create other types of databases, such as databases for your personal use. These utilities allow you to easily change items in the database.

This section includes discussions of the following:

- **Namespace Concepts** — Discusses namespace concepts including network operations, definitions of terms, and general operations.
- **Namespace editor** — Tells how to use the namespace editor (also referred to as NSE), which displays the database in an easy-to-read format. All the namespace editor commands are described in detail.
- **Customizing the namespace editor** — Tells how to customize the namespace editor to suit different needs.
- **User functions** — Describes the user functions that perform operations similar to the namespace editor operations except that these functions are used from within your program.

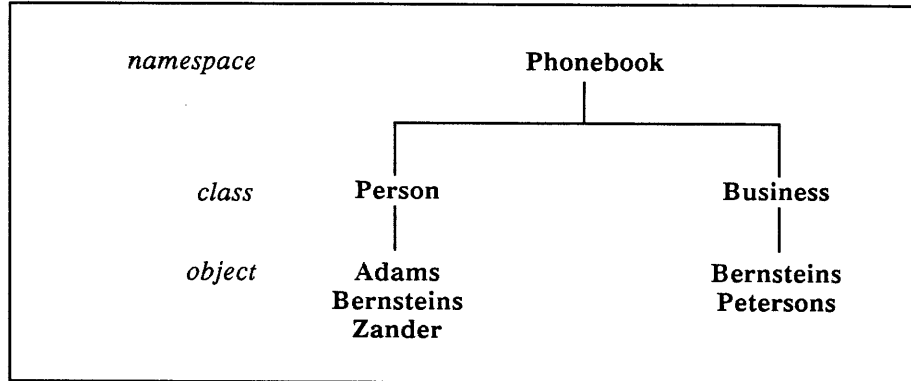
For detailed information on network namespaces, refer to the *Explorer Networking Reference*.

Namespace Concepts

32.2 The following paragraphs describe namespace concepts that apply to both the namespace editor and the user functions. Among the concepts discussed are the following:

- The entries (*objects*) in a namespace
- A special kind of object called an *alias*
- Retrieving objects from a namespace
- Types of namespaces
- Pathnames associated with namespaces
- Default attributes of namespaces
- General namespace operations

Objects 32.2.1 A namespace is a collection of objects. Each object within a namespace is identified by a name and a class. A class is somewhat analogous to a type. No fixed choices exist for a class (you can create your own). An object name must be unique only within a class. The combination (*object-name, object-class*) must be unique only within a particular namespace. A namespace can then be thought of hierarchically as a collection of classes, each containing objects. The following diagram shows this hierarchy:



Note that the object (Bernsteins, Person) is a different object than (Bernsteins, Business).

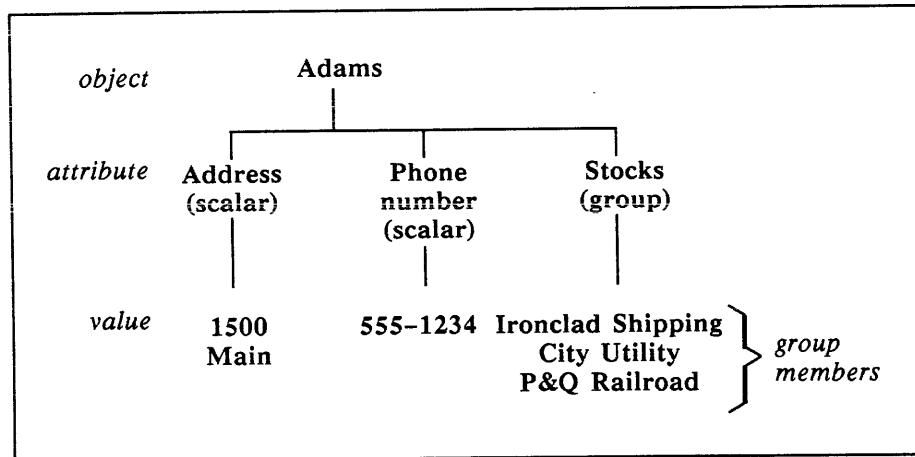
A namespace object can have a set of attributes, with each attribute having an associated value. Attributes are of two types: scalar and group. These types do not refer to the values that may be associated with the attributes but to the way in which those values are changed.

The value of a *scalar* attribute is completely replaced when the attribute is updated. A *group* attribute has a list of members as its value; the attribute value can be incrementally changed by adding and deleting members from the group without accessing the entire attribute value. A typical example of this is two people concurrently adding themselves to a mailing list without interfering with each other.

Suppose you have an object attribute such as `:mailing-list` ("Rhonda" "Bill") and you want to add "Mark" to the list.

- If the attribute is a scalar attribute, you must get the entire value, put "Mark" on the list, and put the value back.
- If the attribute is a group attribute, you do not need to edit the value; you just ask for "Mark" to be added to the list. If two people are concurrently changing the value of the group attribute, they do not overwrite each other.

The hierarchical structure of a namespace can then be extended by viewing objects as follows:



Namespace object content is very flexible. There are no fixed sets of classes or attributes. An object name, object class, attribute name, scalar attribute value, or group attribute value can be any Lisp object. However, since namespaces can span hosts, types should be defined that are reconstructable on all hosts. (A specific flavor instance, for example, probably makes sense only on the host where it was created.)

An object name can be a string, keyword, list, and so on. However, if it is a string, wildcarded lookup operations can be performed.

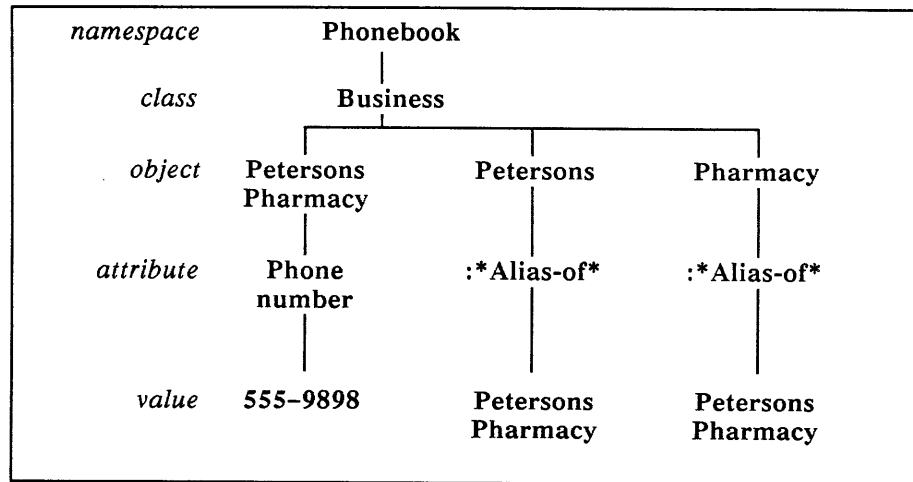
Furthermore, objects can have hidden property lists that are used primarily for internal information about the object, although there are user functions that allow you to manipulate them if you want.

Aliases 32.2.2 You can also create a namespace object that is an *alias* of another object. The alias object has an attribute that points to another object of the same class. The alias can be used as simply a nickname of the other object, or it can be used in a more complicated way by attaching attributes to the alias object itself. The attributes of both objects are then combined during some types of object lookup. An alias object can point to an object in another namespace. There can be multiple aliases for the same object and any number of levels of aliases.

Here are some important facts about aliases:

- Aliases must be of the same class as the object to which they refer.
- Alias objects can have additional attributes attached, which in some cases shadow (hide) lower-level attributes of the same name.
- Multiple levels of aliases are allowed.
- Both the editor and the user functions provide ways of accessing an object by referring to an alias.

The following shows an example of a simple alias:



Retrieving Objects From a Namespace

32.2.3 Objects can be retrieved from a namespace by name and class or by specifying criteria for a subset of objects. These criteria can include a wildcarded name, a class, and/or a list of attributes and values. A user-specified test function can also be used for more complicated restrictions.

Multiple namespaces can exist, and these can be hierarchically structured. (A namespace can exist whose name is contained in another namespace.) The name of an object in a namespace can be *qualified* or *unqualified*. Object names can be explicitly qualified to indicate the containing namespace, or defaults can be assumed.

An unqualified (or *relative*) object name does not contain a namespace name that specifies which namespace to search for the object. In this situation, the namespace utilities rely upon a *search list*. The search list contains a list of available namespaces; the *search rules* search this list in the order the namespaces are listed. For example, if you look up NEIL (an unqualified name), the namespace utilities search for the object NEIL in the namespaces on the search list in the order they are listed. The lookup operation stops either when NEIL is found, or when the namespace search list is exhausted.

A qualified object name contains the namespace name in which to search for the object. Only object names that are strings can be qualified. In a qualified object name, the vertical bar character (|) separates the namespace name from the object name.

For example, suppose you have an Explorer system called NEIL in Austin and an Explorer system called NEIL in Dallas, and both are on the same network. How would you access one or the other? You can put them in different namespaces that are named AUSTIN and DALLAS. To access the Explorer system in Austin, you use AUSTIN|NEIL (a qualified name). The namespace utilities try to find the object NEIL in the namespace AUSTIN. In this situation, the namespace search list is not used.

To conclude, a namespace provides a way to partition names so that two names that are the same do not conflict.

NOTE: You should not use objects of a class with the same name that any network namespace uses because these namespaces are also on the search list. Refer to the *Explorer Networking Reference* for information on network namespaces.

Types of Namespaces 32.2.4 You can create one of the following types of namespaces:

- Public — Multiuser namespace; used for network namespaces.
- Personal — Designed for a single user; maintained in a file.
- Symbolics — Allows you to talk to a Symbolics server for a network namespace.
- Basic — Mainly used as a data structure with the same access interface as any other type of namespace; resides in memory and has no server or associated files.

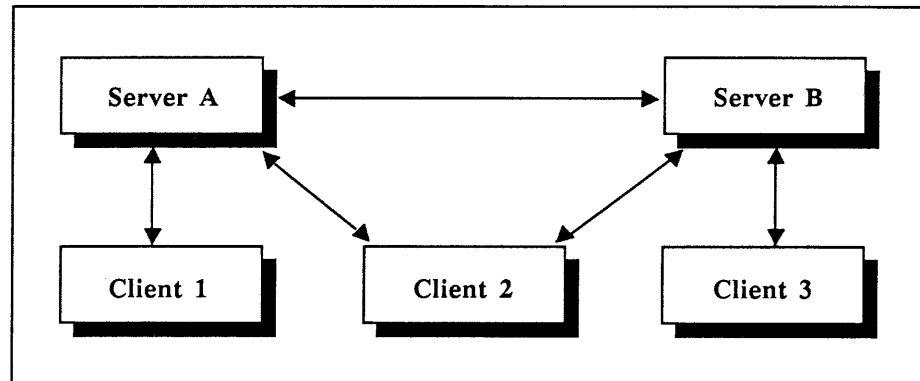
Public Namespace 32.2.4.1 The public namespace is designed for multiusers; it is used for (but not limited to) namespaces that contain network configuration information. When creating a public namespace, you specify one or more host names to be used as servers for the namespace. These servers work in conjunction with each other across the network to handle concurrent access. An individual server is called a *co-server* (even if it is the only one).

A co-server does the following:

- Accepts changes that you make to the namespace
- Changes its copy of the namespace
- Propagates the changes to other co-servers

A *client* (that is, a user) contacts a co-server to perform operations on the namespace. The client has a *cache*, which contains temporary copies of some of the namespace objects. On the other hand, the co-server maintains a permanent (recreatable across boot operations) copy of the namespace, which is shared between multiple clients.

The following diagram shows a sample relationship between servers and clients:



- A server and a client can be on the same machine (Client 1 and Server A, for example).
- A client can be on a machine with no server, but it can talk to servers (Client 2 can talk to both Server A and Server B).

The cache entries of the clients can time out, be explicitly flushed, or be marked for refresh by user functions or by using the namespace editor. A cache entry times out after a certain length of time unless you have made local-only changes.

A client can make local or global changes. *Local* changes are temporary updates to the cache. You might make a local change because you want an environment different from the information on the server. You can also make a global change (or update). A *global* update means to tell a server about the change, which in turn tells the other servers and makes the change available to other clients.

Personal Namespace **32.2.4.2** A personal namespace is designed for use by one person. Rather than having servers, this namespace is loaded at the user's machine and maintained using an xld file. It does not have a cache (thus, global and local updates have the same effect). This type of namespace is most useful for small, infrequently modified namespaces. For any change to be saved, the entire namespace must be written to a file. For this reason, it is not suitable for use as a network namespace because the network can make updates that are intended to be local, but in fact are permanently saved by writing out the entire namespace.

Symbolics Namespace **32.2.4.3** A Symbolics namespace client can talk to a Symbolics server for a network namespace. Only client services are provided; co-serving with a Symbolics server is not allowed (that is, an Explorer system and a Symbolics system cannot both be servers for the same namespace). You can perform updates on a Symbolics database, as well as read operations.

The differences between Explorer and Symbolics namespace models are handled as follows:

- A Symbolics namespace usually has a fixed set of classes (although this can be extended *at the Symbolics*). The default classes are shown in the namespace editor when you are editing a Symbolics namespace unless the associated list of known classes is extended either as an attribute of a Symbolics namespace object (see paragraph 32.2.6.3, Default Attributes for Symbolics Namespace) or when describing a new namespace in response to namespace editor prompts (see paragraph 32.3.2, Configuring the Namespace).
- Symbolics allows replicated attributes (that is, more than one attribute of the same name). These are turned into group attributes when transferred from the Symbolics server to the Explorer client, and the process is reversed when transferring the other way. Some specified Symbolics attributes and/or associated values are changed when cached into the form that the Explorer network interface uses. When an update is made, the process is reversed before sending to the Symbolics server.
- Symbolics does not allow the manipulation of individual object attributes, so these operations are simulated.
- Some options for finding objects by specifying their properties are not supported at the Symbolics server. The class must be specified, neither name wildcarding nor wild attribute names are supported, and no user-specified test function is allowed. (These operations are discussed in detail in the Find Objects With Specific Properties command description.) The Explorer system transfers the minimum superset of objects from the Symbolics server and does additional filtering locally. The parameter `name:*expensive-foreign-lookup-by-properties*` (the default is `nil`) can be bound to control whether a large set of objects is transferred in order to do local filtering.

Basic Namespace 32.2.4.4 A basic namespace resides in memory and has no server or associated files. It is mainly used as a data structure with the same access interface as any other type of namespace.

NOTE: You can edit a basic namespace, but your changes will be discarded when you boot because this type of namespace is memory-resident only.

Namespace Pathnames 32.2.5 A basic namespace does not have any associated pathnames because it is memory-resident only, and there is no way to save it.

NOTE: Note that you cannot rename or copy personal and public files to change the type of namespace from one to the other. You can use the namespace editor to copy selected objects or classes to another namespace, or you can use the user function `name:copy-namespace` to copy the entire namespace.

Personal Namespace Pathnames 32.2.5.1 The default directory where personal namespaces are stored is LM: NAME-SERVICE;. The default pathname for a personal namespace is as follows:

LM: NAME-SERVICE; *namespace-name*-PERSONAL.XLD

An example is LM: NAME-SERVICE; CATS-PERSONAL.XLD. You can change the pathname when you create the namespace.

Public Namespace Pathnames 32.2.5.2 A server uses the following files on the local machine for a public namespace:

- LM: NAME-SERVICE; *namespace-name*-AUDIT.TEXT
- LM: NAME-SERVICE; *namespace-name*-PUBLIC.XLD
- LM: NAME-SERVICE; *namespace-name*-LOG.LISP
- LM: NAME-SERVICE; SERVER-BOOT-LIST.LISP

These files are used transparently except for the LM: NAME-SERVICE; *namespace-name*-AUDIT.TEXT file, which you may want to view. When auditing is enabled, a more readable and complete history of namespace changes is saved in this file.

When a version of the namespace is written, it is stored in the file LM: NAME-SERVICE; *namespace-name*-PUBLIC.XLD.

Changes between xld writes for a public namespace are saved in the file LM: NAME-SERVICE; *namespace-name*-LOG.LISP.

The LM: NAME-SERVICE; SERVER-BOOT-LIST.LISP file contains the list of public namespaces for which this machine is a server. This file is maintained automatically by the namespace system, although you can edit the file.

NOTE: Any of these pathnames may in the future contain additional characters for version identification if incompatible namespace changes mean old files are no longer valid. For example, a new filename might be LM: NAME-SERVICE; *namespace-name*-PUBLIC-A.XLD.

Default Attributes

32.2.6 Each type of namespace, except basic and clients of foreign namespaces, contains an object whose name is the same as that of the namespace and whose class is `:namespace`. This object describes characteristics of the namespace. The object has a set of default attributes, which differ depending on the type of namespace. The following paragraphs discuss the default attributes for the personal, public, and Symbolics namespaces.

The `:namespace` objects can also be used within one namespace to describe another namespace. For example, in a network namespace, you might want to describe how to access some other namespaces so that you can put them on your search list and/or reference them.

*Default
Attributes for
Personal Namespace*

32.2.6.1 The following are the default attributes for a personal namespace. You can edit these attributes.

:type — Identifies the type of namespace. For a personal namespace, the value of this attribute is `:personal`.

:usage — Governs the way the editor edits the namespace. The default is `nil` (that is, give the namespace no special handling).

You can specify the usage mode corresponding to a set of expert editors with this attribute. (For a namespace used for a network configuration, **:usage** is set to `:network`.) Refer to paragraph 32.4, NSE Customization, for information on expert editors.

:namespace-file-pathname — Specifies the default file where the namespace is stored. The syntax for the default file is as follows:

LM: NAME-SERVICE; *namespace-name*-PERSONAL.XLD

For example, if your namespace is CATS, the default file is LM: NAME-SERVICE; CATS-PERSONAL.XLD.

Note that you can specify a remote file instead of a local file if necessary.

:auto-save-enabled — When this attribute is set to non-`nil` (the default), the entire namespace is written to an xld file after every *n* changes (where *n* is equal to **:changes-before-save**). While this attribute is set to `nil`, the namespace is never written to an xld file.

CAUTION: With a personal namespace, writing to an xld file is the only way that changes are permanently recorded, so you may not want to change this attribute.

:changes-before-save — If **:auto-save-enabled** is non-`nil`, the value of this attribute specifies the number of updates to the namespace that are made before a new copy of the namespace is written to an xld file.

:read-only — When this attribute is set to non-`nil`, disallows updates until this attribute is set to `nil`. The default is `nil`.

*Default Attributes
for Public Namespace*

32.2.6.2 The following are the default attributes for a public namespace. You can edit these attributes.

:type — Identifies the type of namespace. For a public namespace, the value of this attribute is **:public**.

:usage — Governs the way the editor edits the namespace. The default is **nil** for a public namespace, unless the namespace is for a network configuration (in which case, it is set to **:network**).

You can also specify the usage mode corresponding to a set of expert editors with this attribute. Refer to paragraph 32.4, NSE Customization, for information on expert editors.

(:servers :group) — Specifies the list of hosts that are to be co-servers for this namespace. If this is a new namespace, the default server is the local host. If this is not a new namespace, the existing namespace already contains a **:namespace** object that specifies its servers.

:caching-control — If set to **t** or **:always** (the default), this attribute always caches any object obtained from the server. If set to **:selective**, this attribute caches these objects unless the object has the **:*non-cacheable*** attribute set to **t**. Otherwise, objects from this namespace are never cached.

:cache-entry-timeout — Specifies the amount of time until a locally cached copy of an object times out. (The copy is deleted, and a new copy is obtained from a namespace server.) You can specify this value as a number of seconds or as a list in the following form:

```
(n :second or :seconds
  :minute or :minutes
  :hour    or :hours
  :day     or :days
  :week    or :weeks)
```

An example is (30 :minutes). The default is (24 :hours).

:auditing-enabled — If set to **non-nil** (the default), records all namespace changes and significant events in a human-readable audit file. The name of this file is the following:

LM: NAME-SERVICE; *namespace-name*-AUDIT.TEXT.

:auto-save-enabled — When this attribute is set to **non-nil** (the default), the entire namespace is written to an xld file after every *n* changes (where *n* is equal to **:changes-before-save**). While this attribute is set to **nil**, the namespace is never written to an xld file.

With a public namespace, every change between xld saves is written to the LOG file. Therefore, every update is permanently recorded regardless of whether a new xld file is ever written.

:changes-before-save — If **:auto-save-enabled** is **non-nil**, the value of this attribute specifies the number of updates to the namespace that are made before a new copy of the namespace is written to an xld file.

:read-only — When this attribute is set to **non-nil**, disallows updates until this attribute is set to **nil**. The default is **nil**.

*Default
Attributes for
Symbolics Namespace*

32.2.6.3 The following are the default attributes for a Symbolics namespace. You can edit these attributes.

:type — Identifies the type of namespace. For a Symbolics namespace, the value of this attribute is **:symbolics**.

:usage — Governs the way the editor edits the namespace; set to **:network** for a Symbolics namespace.

You can also specify the usage mode corresponding to a set of expert editors with this attribute. Refer to paragraph 32.4, NSE Customization, for information on expert editors.

(:primary-servers :group) — Specifies the list of primary Symbolics servers used first for any type of request.

(:secondary-servers :group) — Specifies the list of secondary Symbolics servers used for reads only (not updates) if no primary servers respond.

:caching-control — If non-nil, this attribute always caches any object obtained from the server. If set to nil, this attribute never caches.

:cache-entry-timeout — Specifies the amount of time until a locally cached copy of an object times out. (The copy is deleted, and a new copy is obtained from a namespace server.) You can specify this value as a number of seconds or as a list in the following form:

```
(n :second or :seconds
   :minute or :minutes
   :hour   or :hours
   :day    or :days
   :week   or :weeks)
```

An example is (30 :minutes). The default is (24 :hours).

:known-classes — Specifies a standard set of Symbolics classes. The default value of this attribute is as follows:

```
(:namespace :site :network :host :printer :user).
```

**Namespace
Operations**

32.2.7 The namespace operations described here are available in both the namespace editor and the user functions, unless stated otherwise.

All namespace comparisons (that is, object names, attribute names, and attribute values) are performed by using these rules:

- Strings are compared without regard for alphabetic case (although the case is preserved in the namespace as you entered it).
- **equal** is used for items other than strings that are not lists.
- Lists match if each member matches by using these rules.

*General
Namespace Operations*

32.2.7.1 Several operations are provided to deal with namespaces on a general level. These operations allow you to do the following:

- Create new namespaces and access old ones.
- Add the name of a namespace to the search list so that the namespace can be searched when you enter an unqualified name (that is, you do not indicate the namespace in the name of the object).
- List the available namespaces, classes in a namespace, or objects in a namespace.
- Copy all the objects in one namespace to another namespace by using one of the user functions.
- Copy an object, a class of objects, or an attribute to the same namespace or another namespace by using the editor.

*Changing a
Namespace*

32.2.7.2 You can easily make the following changes to a namespace that is already created:

- Add objects to the namespace, or delete them.
- Make new classes for the namespace.
- Add attributes, either scalar or group, to an object.
- Find and change the value of an attribute, or delete the attribute.
- Add members to a group attribute, or delete members from the attribute.
- Add objects that are aliases for other objects, or delete them.

Note that any change you make to an object (for example, adding an attribute or a group member) results in the object being automatically created if it does not already exist.

*Finding and
Accessing Objects*

32.2.7.3 You can use one of the following techniques to find and access namespace objects:

- Find an object by specifying its name and class.
- Find an object and its aliases by specifying the object's name and class.
- Find objects by their properties, which can be any combination of the following. These properties allow you to filter the objects that are found.
 - Name pattern — An unqualified name or a list of unqualified names. A name that is a string can contain wildcard characters.
 - Attribute list — A list of attribute names and values.
 - Test predicate — Created by the user for more complicated restrictions.

Namespace Editor (NSE)

32.3 The namespace editor (NSE) allows you to view and change namespace objects in a special kind of Zmacs buffer. The display in the buffer works much like the read-only buffer in Dired or Mail. You position the keyboard cursor on the line that contains the item on which you want to perform an operation, and then you type a keystroke command or click middle for a menu of commands.

The initial display contains lines for the classes. Then, as in Dired, you can expand a class line (by typing S on the line) to see the names of the objects contained in the class. You can also expand an object line to see its attributes. Figure 32-1 shows a sample expanded display.

The first line displays the name of the namespace, `Fairy-Tales`. The first column on the left lists symbols such as `*`, `G`, and `+` that indicate various characteristics of a class, object, or attribute. The next column lists the class names, `:CHARACTER` and `:NAMESPACE`. The object names are `"Cinderella"` and `"Ella"`. The attributes are `:shoe-size`, `:skills`, and so on, with their values listed on the right.

If a class contains a large number of objects, additional features of the namespace editor allow you to work with a more convenient subset of objects rather than completely expanding a class.

Figure 32-1

Namespace Editor Display

```

                                NAMESPACE EDITOR FOR "Fairy-Tales"

*   :CHARACTER
    "Cinderella"
      :shoe-size      3
G   :skills          (:CLEANING :COOKING)
    "Motto"          "Someday my pransome hince will come"

+   "Ella"
      :*ALIAS-OF*    "Cinderella"

:NAMESPACE

```

This numbered paragraph describes the following topics:

- Accessing the namespace editor — Tells how to access the namespace editor.
- Configuring the namespace — Describes how to answer the prompts on the Configure Namespace menu, such as the type of namespace you are creating.
- Basic NSE operations — Provides an overview of several basic namespace editor operations, such as cursor movement, updating changes, undoing changes, exiting the namespace editor, and so on.
- Symbol codes — Describes the symbols with which a class, object, or attribute may be marked, such as a deletion mark.

- General commands menu — Discusses the general namespace editor commands that you can use at any time during an edit session.
 - Class commands — Describes the namespace editor commands that perform operations on classes.
 - Object commands — Discusses the namespace editor commands that perform operations on objects.
 - Attribute commands — Describes the namespace editor commands that perform operations on attributes.
 - Group attribute commands — Discusses the namespace editor commands that perform operations on group attributes.
-

**Accessing the
Namespace Editor**

32.3.1 Choose one of the following procedures to access the namespace editor:

- From the System menu, select the item `Namespace Editor` in the Programs column.
- If you are in Zmacs, type `META-X Edit Namespace`.

A menu appears that lists the available namespaces. To create a new namespace or to configure an existing but unknown namespace (that is, unknown to the system), select `<other-namespace>`.

A menu appears that allows you to configure a namespace.

**Configuring the
Namespace**

32.3.2 Follow these steps to answer the prompts on the Configure Namespace menu:

1. Enter the name of the namespace for `Namespace Name`.
2. Select one of the following values for `Namespace Type`.
 - `PUBLIC` — Main type of namespace, which is used for network namespaces. A co-server accepts changes, changes its copy of the namespace, and propagates the new copy to other servers.
 - `PERSONAL` — Designed for your personal use. This namespace does not use servers. Changes are written to an xld file rather than being propagated to servers.
 - `SYMBOLICS` — Allows you to talk to a Symbolics server for a network namespace.
 - `BASIC` — Resides in memory and has no server. This type of namespace is used mainly as a data structure with the same access interface as any other type of namespace.

Note that you can edit a basic namespace, but your changes will be discarded when you boot because this type of namespace is memory-resident only.

For more information on these namespaces, refer to paragraph 32.2.4, Types of Namespace.

3. Select one of the following values for `Local search list placement`:
 - `BEGINNING` — Puts the namespace at the beginning of the search list.
 - `END` — Puts the namespace at the end of the search list.
 - `NONE` — Does not put the namespace on the search list.
4. If you are creating a new namespace, select `Yes` for `New namespace?`.

When you load a personal or public namespace that was specified as new, the following message may appear:

```
WARNING: Namespace X is being initialized without using existing
database files.
```

This message informs you that there are old files by this name that are not being loaded and that new versions are being created. You may have intended to do this, or you may have specified `Yes` for `New namespace?` accidentally. The old versions of the files still exist and can be retrieved if desired.

5. Select `Do It` or press the `END` key when you complete your selections.

6. Depending on what type of namespace you are creating and whether it is a new namespace, other prompts may appear:
 - If you are creating a personal namespace, another prompt appears asking you for a pathname for storing the namespace. Accept the default pathname, or specify another pathname where you want your personal namespace stored.
 - If you are configuring a public namespace that is not new, another prompt appears asking you for the remote servers. Specify a list of host names already serving a namespace.
 - If you are creating a new Symbolics namespace or configuring an old one, three prompts appear:
 - Remote Servers — Specify a list of host names already serving a namespace.
 - Namespace Classes — Accept the default list of Symbolics classes or add to the list any Symbolics classes the Explorer system does not know about.
 - Cache? — Accept the default value of Yes to cache namespace objects obtained from the Symbolics system.

Unless you are creating a basic namespace, the screen now displays one class, `:namespace`. This class contains an object with the same name as the namespace. The object contains several default attributes. A basic namespace does not contain this default object.

Basic NSE Operations

32.3.3 Here are some commands and techniques that you can use at any time during your namespace editor session:

END key

Press the END key at any time to exit the namespace editor. If necessary, you are prompted to save your changes.

ABORT key

Leaves your namespace editor session, switching to another Zmacs buffer. You must return to the buffer to save any unsaved changes.

Click middle or press the M key

The namespace editor contains two context-sensitive commands that you can use at any time:

- Clicking the middle mouse button displays the General Commands menu if the mouse cursor box is *not* on a class, object, or attribute. If the mouse cursor box is on a class, object, or attribute, clicking the middle mouse button displays a menu of commands for the item at the current mouse position.
- Pressing M invokes the General Commands menu if the keyboard cursor is not on a class, object, or attribute. If the cursor is on a class, object, or attribute, pressing M invokes a menu of commands for the item at the current cursor position.

S command

As in Dired, pressing S beside a class line expands (shows) the objects associated with the class. Likewise, pressing S beside an object line shows the attributes associated with the object. Pressing S on an expanded class or object deexpands (closes) it.

HELP M

Pressing HELP M displays a listing of the namespace editor commands.

Update commands

The changes that you make within the namespace editor are not updated until you use one of the update commands. You can make both local and global updates. If you are working on a public namespace, it is a good idea to make local updates, test the results, and then update globally. Commands are available to update an attribute, an object, a class of objects, or the entire namespace. These commands are discussed later in the section.

Revert commands

The namespace editor contains two different levels of revert commands that allow you to undo changes. You can revert to the state shown in the local cache, or you can revert to the state shown in the name server. These commands are discussed later in the section.

Suggestions menus

The Suggestions menus also allow you to execute the namespace editor commands.

Cursor movement

As in Dired or Mail, you can use the following commands (among others) to move the keyboard cursor:

- Move the mouse cursor to the line where you want the keyboard cursor and click left.
- To move down a line, press the ↓ key or the space bar.
- To move up a line, press the ↑ key or RUBOUT.

The namespace editor provides these additional cursor movement commands:

- To move to the next object, press N.
- To move to the previous object, press P.
- To move to the next class, press META-N.
- To move to the previous class, press META-P.

Switching between the namespace editor and Zmacs buffers

As in a regular Zmacs buffer, you can switch between your current namespace buffer and other buffers. Each namespace that you edit with the namespace editor is placed in its own buffer, named according to the name of the namespace.

- You can use the Zmacs List Buffers command (CTRL-X CTRL-B) to move to other buffers.
- You can use the Zmacs Select Previous Buffer command (META-CTRL-L) to go to the previous buffer. Pressing META-CTRL-L again returns you to the first buffer.

Numeric arguments

Many namespace editor commands accept numeric arguments. When you type a number (*n*) or CTRL-*n* before invoking a command, the command is executed *n* times. For example, 3 N moves the cursor down three objects.

Symbol Codes 32.3.4 A class, object, or attribute may be marked with one of these symbols:

- ++ If your namespace has a local cache, then the class, object, or attribute marked with two plus signs is new and needs to be *locally* saved as well as globally saved.
- + The class, object, or attribute marked with one plus sign is new and needs only to be globally saved.

NOTE: A personal or basic namespace has no local cache; therefore, local and global saves are the same.

- ** If your namespace has a cache, then the class, object, or attribute marked with two asterisks needs to be *locally* saved as well as globally saved.
- * The class, object, or attribute marked with one asterisk needs only to be globally saved.
- d The class, object, or attribute marked with a d is marked for deletion. When this item is globally updated, the line is removed from the screen.
- G An attribute marked with a G is a group attribute. This means that the special Add Group Member, Edit Group Member, Delete Group Member, and Add Key for Group Attribute commands are available for this attribute. You can make an attribute become a group attribute with the G command. When you try to edit a group attribute, you are given a menu of group members, and you can edit the group member that you select.

**General
Commands Menu**

32.3.5 The General Commands Menu allows you to perform several operations:

Add Object Command

Keystroke: A or META-X Add Object

Adds an object to the current class. You are prompted in the minibuffer for the name of the class and the name of the object. If you press RETURN when prompted for the class name, a menu of existing class names appears.

Show Documentation Command

Keystroke: CTRL-SHIFT-D or CTRL-SHIFT-A

Shows documentation for the current class, object, or attribute. Pressing CTRL-SHIFT-D shows long documentation; pressing CTRL-SHIFT-A shows short documentation.

Expand All Objects in NSE Command

Keystroke: META-CTRL-S or META-X Expand All Objects in NSE

Expands all object lines in the namespace editor.

Unexpand All Objects in NSE Command

Keystroke: META-X Unexpand All Objects in NSE

Unexpands all objects in the namespace editor.

Revert NSE to Local State Command

Keystroke: CTRL-X CTRL-R or META-X Revert Buffer

Reverts all objects in the namespace editor to their local states and unexpands all class and object lines. Changes that you have not globally saved will be lost.

Update Namespace Locally Command

Keystroke: CTRL-X CTRL-S or META-X Update Namespace Locally

Writes changes to the local cache for the entire namespace.

Update Namespace Globally Command

Keystroke: CTRL-X CTRL-W or META-X Update Namespace Globally

Writes changes to the local cache and to the name server for the entire namespace.

Distribute Namespace Command

Keystroke: META-X Distribute Namespace

Distributes the current namespace to all servers or to the local machine only. You are asked to save the namespace before distributing.

Verify Namespace Command

Keystroke: META-X Verify Namespace

Verifies objects and associated attributes within a namespace. The verification routine(s) that is called by this command is determined by the values of the `:namespace-verification-routine` keyword of the expert editors. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to paragraph 32.4, NSE Customization, for more details.

Write NSE Buffer to File Command

Keystroke: META-X Write NSE Buffer to File

Writes the buffer to a file. This command is necessary because the NSE CTRL-X CTRL-W command is tied to updating and does not work like the Zmacs CTRL-X CTRL-W command, which writes a buffer to a file.

Class Commands **32.3.6** If you press M when the keyboard cursor is beside a class name (or click middle when the mouse cursor box is on a class name), a menu of commands available for the class appears.

Expand/Unexpand Class Command

Keystroke: S or META-X Expand or Unexpand

Toggles the expansion of a class of objects.

Expand All Objects in Class Command

Keystroke: META-S or META-X Expand All Objects

Expands all object lines of the current class.

Unexpand All Objects in Class Command

Keystroke: META-X Unexpand All Objects in Class

Unexpands all object lines of the current class.

Find Object Command

Keystroke: F or META-X Find Object

Finds the object that you specify (searches only within the current class). You are prompted for the name of the object. If the object does not currently appear in the buffer, the appropriate class is expanded to include the desired object. Otherwise, the cursor is simply moved to the desired object.

Find Object and Aliases Command

Keystroke: CTRL-F or META-X Find Object and Aliases

Finds the object that you specify and its aliases. You are prompted for the class and the name of the object. You are also prompted on whether to search for aliases on multiple levels. If *no*, only immediate aliases of the object are found. If *yes*, aliases of those aliases and so on are found.

Finally, you are prompted to merge with the displayed objects. If *yes*, the object and its aliases are mixed with the objects already displayed within the class. If *no*, the appropriate class is expanded to contain only the object and its aliases.

Note that unsaved changes are not considered in the search.

Find Non-Alias Objects Command

Keystroke: META-F or META-X Find Non-Alias Objects

Expands the current class of objects, and does *not* include aliases in the expansion. Note that unsaved changes are not considered in the search.

Find Objects With Specific Properties

Command

Keystroke: META-CTRL-F
or META-X Find Objects With Specific Properties

Expands the current class with the objects that match all the qualifications that you specify. Note that unsaved changes are not considered in the search.

You are prompted for a name pattern, an attribute list, and/or a test predicate. You are also prompted for a decision on whether to return only the first object found and for a display format.

- A *name pattern* can be an unqualified object name or a list of unqualified names. A name that is a string can contain the wildcard characters * and ?. The * character looks for a name with any number of characters in this position. The ? character looks for a name with any single character in this position. If you specify a list of names for the name pattern, the search operation returns all of the objects that match those names and any other criteria (that is, the attribute list and/or test predicate); these names can include wild characters if they are strings.

The following shows an example of specifying a name pattern:

```
"M*.*"
```

MYCROFT.X is an example of this type of name pattern, while HYCROFT.X and MYCROFT.XX are not.

The following shows an example of entering the name pattern as a list of names. Notice that you can use a wild character.

```
("Tom" "Bill" "Lis*")
```

- An *attribute list* is a list of attribute names and values used for restricting the objects. Either attribute names or values can be the wild keyword :*. If :* is specified as a value, the objects that are found have properties of the specified name with any value. If :* is specified as a name, the objects that are found have any attribute with the specified value.

The following shows an example of specifying an attribute list:

```
(:sex :male  
:age :*  
:* "Dinosaurs"  
:members "Tom")
```

This attribute list restricts the objects found to those containing all of the following criteria: a :sex attribute with a value of :male, an :age attribute with any value (since the wild keyword :* is used), any attribute with the value "Dinosaurs", and a :members attribute with the value "Tom". If the :members attribute is a group attribute, the member "Tom" is looked for in the attribute value.

You cannot specify something such as :age < 30 in the attribute list because only comparison by equality is done. You can specify a test function, though (described next). When :age 30 is a member of the attribute list, your test function can use 30 as a ceiling. To conclude, you can perform the comparison as you wish in your own test function.

- The test predicate identifies a function taking arguments *object*, *name-pattern*, *class*, and *attribute-list*. If the test predicate is a symbol, it is assumed that the symbol is defined at the remote host and that it is applied as a filter at the remote host (to limit the set of objects returned). Otherwise, the nonsymbol (for example, a function object or closure) is applied locally after a superset of the objects has been transmitted. The latter is obviously less efficient. The *attribute-list* can actually be anything else you want to pass to your test function. It should return non-nil to include an object in the subset. The following shows an example of a test function:

```
(defun user:age-check-function (object name-pattern class
                               attribute-list)
  (let ((age (name:get-attribute-value object :age)))
    (and age (> age (cdr attribute-list)))))
```

- *First only* means that only the first object meeting the test is returned.
- You have a choice of four display formats:

NORMAL — Within the class, displays only the objects that meet the qualifications that you specify.

MERGE-WITH-DISPLAYED-OBJECTS — Mixes the objects found with the objects already displayed within the class.

MAKE-PERSONAL-FILTER — Defines a personal filter for the objects that are found by this command. In many ways, a filter works like a subclass. You can expand and unexpand a filter as you can a class, and you can perform the same commands within a filter as you can within a class. However, when you add an object, you cannot specify a filter name as the name of the class of the object.

A personal filter goes away when you reboot. You can store it by using the `nse:define-personal-filter` macro in your LOGIN-INIT file. For more information on filters, refer to paragraph 32.4.2, Filters.

MAKE-PUBLIC-FILTER — Defines a public filter for the objects that are found. A public filter is stored in the `:editor-customization` class of the namespace. If you update the namespace globally, the filter is permanently available to anyone who edits the namespace.

Find Changed Objects

Command

Keystroke: META-X Find Changed Objects

Expands the current class with changed or new objects only.

Expand Horizontally

Command

Keystroke: CTRL-H or META-X Expand Horizontally

Expands all object lines in a class horizontally so that you can view certain attributes in column form. A menu allows you to specify how the display appears.

Example:

```
attribute-1 : :addresses
length-1   : 50
attribute-2 : :location
length-2   : 25
```

If you enter this format into the choose-variable-values (CVV) menu, you obtain a columnar display (Figure 32-2) showing one line for each object with the following items:

- The object name (25 characters wide). Note that this width is set internally in the code and cannot be changed. However, you can specify the width for the object name if you create a personal horizontal format with the `nse:define-personal-horizontal-format` macro.
- The value of the `:addresses` attribute for that object (50 characters wide).
- The value of the `:location` attribute for that object (25 characters wide).

For more information on creating and saving horizontal formats, refer to paragraph 32.4.3, Horizontal Formats.

Toggle Horizontal Display

Command

Keystroke: H or META-X Toggle Horizontal Display

Toggles between the horizontal and standard display formats for the current class. The default horizontal display is used. However, if no default horizontal display is found, a CVV menu allows you to specify a format.

If no default horizontal format is found, you are given a CVV menu in which to enter the information.

To determine the default horizontal display, the following search rules are used:

1. Looks for the latest horizontal format used for the current class during this session.
2. Looks for a personal horizontal format specified by the `nse:define-personal-horizontal-format` macro (searches the list `nse:*personal-horizontal-format-list*`).
3. Looks for a public horizontal format in the `:editor-customization` class in the namespace.

Refer to paragraph 32.4.3, Horizontal Formats, for information on creating a public horizontal format.

Figure 32-2 Horizontally Expanded Class

```

                                NAMESPACE EDITOR FOR "JAY"
                                This is a namespace of type :NETWORK.

█ :HOST
  :OBJECT-NAME      :ADDRESSES      :LOCATION
  *CONRAD*          ((:CHAOS 1483))  *John Smith's office*
  *SYS*             NIL              NIL

:MAILING-LIST
:NAMESPACE
:PRINTER
:SITE
:USER

ZHAOS Namespace Editor for "JAY" NETWORK (After distribution, a cache will be available for local-only updates.)

L: Move point, L2: Move to point, M: Namespace-Editor Commands Menu, M2: Save/Kill/Yank, R: General Menu, R2: System Menu

```

Copy Objects in Class

Command

Keystroke: CTRL-C or META-X Copy Objects in Class

Copies the current class of objects to the specified namespace buffer. If the buffer is not there, it is created but not selected. If the class is expanded, only the currently displayed objects are copied.

Write (Update) Class Locally

Command

Keystroke: W or META-X Update Class Locally

Writes changes to the local cache for the current class.

NOTE: If you have no local cache, Write (Update) Class Locally and Write (Update) Class Globally work the same.

Write (Update) Class Globally Command

Keystroke: CTRL-W or META-X Update Class Globally

Writes changes to the local cache and to the name server for the current class.

Verify Class Incrementally Command

Keystroke: META-X Verify Class Incrementally

Verifies objects within the current class. If verification fails, you are warned that the object content is invalid. The verification routine(s) that is called by this command is determined by the value of the **:incremental-verification-routine** keyword of the expert editor. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to paragraph 32.4, NSE Customization, for more details.

Revert Class to Local State Command

Keystroke: R or META-X Revert Class to Local State

Reverts the current class to the state shown in the local cache.

NOTE: If you have no local cache, Revert Class to Local State and Revert Class to Global State work the same.

Revert Class to Global State Command

Keystroke: META-R or META-X Revert Class to Global State

Reverts the current class to the state shown in the name server.

The following describes a class command that is not on the menu:

Copy Class to This Namespace Command

Keystroke: C or META-X Copy Class to This Namespace

Copies the current class of objects to a different class within this namespace.

Object Commands 32.3.7 If you press M when the keyboard cursor is beside an object name (or click middle when the mouse cursor box is on an object name), a menu of commands available for the object appears. These commands are described in the following paragraphs.

If you enter one of these commands by its keystroke or META-X, the keyboard cursor must be on the same line as the object name.

Expand/Unexpand Object Command

Keystroke: S or META-X Expand or Unexpand

Toggles the expansion of the attributes of an object.

Add Attribute Command

Keystroke: CTRL-A or META-X Add Attribute

Adds an attribute to the current object.

Add Group Attribute Command

Keystroke: META-X Add Group Attribute

Adds a group attribute to the current object.

Delete Object Command

Keystroke: D or META-X Delete Object

Marks the current object for deletion. The object is deleted when you perform a global update. If you have a local cache and you update locally, the deletion marks remain.

Undelete Object Command

Keystroke: U or META-X Undelete Object

Undeletes the current object.

Add Alias for Object Command

Keystroke: META-CTRL-A or META-X Add Alias for Object

Adds an alias for the current object.

Delete Aliases for Object Command

Keystroke: META-CTRL-D or META-X Delete Aliases for Object

Marks one or more aliases of the current object for deletion. You specify which aliases from a menu that appears. The aliases are deleted when you perform a global update. If you have a local cache and you update locally, the deletion marks remain.

Write (Update) Object Locally Command

Keystroke: W or META-X Update Object Locally

Writes changes to the local cache for the current object.

NOTE: If you have no local cache, Write (Update) Object Locally and Write (Update) Object Globally work the same.

Write (Update) Object Globally Command

Keystroke: CTRL-W or META-X Update Object Globally

Writes changes to the local cache and to the name server for the object.

Verify Object Incrementally Command

Keystroke: META-X Verify Object Incrementally

Verifies the current object. If verification fails, you are warned that the object content is not correct. The verification routine(s) that is called by this command is determined by the value of the `:incremental-verification-routine` keyword of the expert editor. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to paragraph 32.4, NSE Customization, for more details.

Copy Object Command

Keystroke: CTRL-C or META-X Copy Object

Copies the current object to the specified namespace buffer. If the buffer is not there, it is created, but not selected.

Revert Object to Local State Command

Keystroke: R or META-X Revert Object to Local State

Reverts the current object to the state shown in the local cache.

Revert Object to Global State Command

Keystroke: META-R or META-X Revert Object to Global State

Reverts the current object to the state shown in the name server.

The following describes an object command that is not on the menu:

Copy Object to This Namespace Command

Keystroke: C or META-X Copy Object to This Namespace

Copies the current object to the current namespace. You are prompted for the name of the new object and for its class.

Attribute Commands

32.3.8 Attributes may be marked with one of the symbols described earlier in paragraph 32.3.4, Symbol Codes.

If the nature of the attribute implies incremental changes, you probably want to make the attribute a group attribute. Then several people can modify the attribute without interfering with each other or having to know what the current value is.

If you press M when the keyboard cursor is beside an attribute name (or click middle when the mouse cursor box is on an attribute name), a menu of commands available for the attribute appears. These commands are described in the following paragraphs. (An attribute command that is not on the menu is also described at the end of these commands.)

NOTE: If you invoke this menu on a *group* attribute name, the menu of commands that appears contains some commands specific to group attributes in addition to containing most of the attribute commands described here. Refer to paragraph 32.3.9, Group Attribute Commands.

If you enter one of these attribute commands by its keystroke, the keyboard cursor must be on the same line as the attribute name.

Edit Attribute Command

Keystroke: E or META-X Edit Attribute

Edits the current attribute. The currently selected line must describe an attribute.

NOTE: If you execute this command on a group attribute (marked with a g), a menu of group members appears. You can choose which group member to edit.

Delete Attribute Command

Keystroke: D or META-X Delete Attribute

Deletes the current attribute. The currently selected line must describe an attribute. The attribute is deleted when you perform a global update. If you have a local cache and you update locally, the deletion marks remain.

Undelete Attribute Command

Keystroke: U or META-X Undelete Attribute

Undeletes the current attribute. The currently selected line must describe an attribute.

Copy Attribute Command

Keystroke: CTRL-C or META-X Copy Attribute

Copies the current attribute to the specified object in the specified namespace buffer. If the buffer is not there, one is created but not selected.

Toggle Group Status Command

Keystroke: G or META-X Toggle Group Status

If the current attribute is a group attribute, this command makes it a scalar (normal) attribute. If the current attribute is a normal attribute, this command makes it a group attribute.

Refer to paragraph 32.3.9, Group Attribute Commands, for more information.

View Attribute Command

Keystroke: V or META-X View Attribute

Views the current attribute value.

Write (Update) Attribute Locally Command

Keystroke: W or META-X Update Attribute Locally

Writes changes to the local cache for the current attribute.

NOTE: If you have no local cache, Write (Update) Attribute Locally and Write (Update) Attribute Globally work the same.

Write (Update) Attribute Globally Command

Keystroke: CTRL-W or META-X Update Attribute Globally

Writes changes to the local cache and to the name server for the current attribute.

Verify Attribute Incrementally Command

Keystroke: META-X Verify Attribute Incrementally

Verifies the value of the current attribute. If verification fails, you are warned that the attribute value is invalid. The verification routine(s) that is called by this command is determined by the value of the **:incremental-verification-routine** keyword of the expert editor. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to paragraph 32.4, NSE Customization, for more details.

Revert Attribute to Local State Command

Keystroke: R or META-X Revert Attribute to Local State

Reverts the value of the current attribute to the state shown by the local cache. If there is no local cache, all unsaved changes are discarded (the command reverts to the global state).

Also on this menu are commands for the current object.

Add Attribute to Current Object Command

Keystroke: CTRL-A or META-X Add Attribute

Adds an attribute to the current object.

Add Group Attribute to Current Object Command

Keystroke: META-X Add Group Attribute

Adds a group attribute to the current object. (Refer to paragraph 32.3.9, Group Attribute Commands, for more information.)

Unexpand Current Object Command

Keystroke: S or META-X Expand or Unexpand

Toggles the expansion of the attributes of an object.

The following describes an attribute command that is not on the menu:

Copy Attribute to This Namespace Command

Keystroke: C or META-X Copy Attribute to This Namespace

Copies the current attribute to the specified object in the current namespace buffer.

**Group Attribute
Commands**

32.3.9 If you press M when the keyboard cursor is beside a group attribute name (or click middle when the mouse cursor box is on a group attribute name), a menu of commands appears that contains several commands for group attributes. Other commands on this menu are discussed in previous paragraphs.

If you enter one of these commands by its keystroke, the keyboard cursor must be on the same line as the attribute name.

Add Group Member Command

Keystroke: META-A or META-X Add Group Member

Adds a group member to the current group attribute.

Delete Group Member Command

Keystroke: META-D or META-X Delete Group Member

Deletes one or more group members from the current group attribute. You specify which group members from a menu.

Edit Group Member Command

Keystroke: E or META-X Edit Group Member

Edits a group member of the current attribute. You choose the member to edit from a menu.

Add Key for Group Attribute Command

Keystroke: META-X Add Key for Group Attribute

Adds a key function to a group attribute. The key function is used for comparison of group members, and it affects the behavior of the Add Group Member command.

You are prompted in the minibuffer for the key function, which is a symbol for the name of a function. The default is `car`, which checks to see if a group member by that name already exists. If the name does exist, the new group member replaces the old group member. For example, suppose you have a group attribute with the following value and a key function of `car`:

```
((:eagles 2) (:falcons 8))
```

If you add the group member `(:eagles 4)`, this new group member replaces the old group member `(:eagles 2)`, as follows:

```
((:eagles 4) (:falcons 8))
```

Also, you can specify the value `:none` for the key function to remove a key function that you previously specified for the attribute.

Command Summary 32.3.10 Table 32-1 lists the namespace editor keystroke commands.

Table 32-1

Namespace Editor Keystroke Commands	
Keystroke	Command Name
<i>General Commands:</i>	
A	Add Object
CTRL-SHIFT-D or CTRL-SHIFT-A	Show Documentation
META-CTRL-S	Expand All Objects in NSE
META-X*	Unexpand All Objects in NSE
CTRL-X CTRL-R	Revert NSE to Local State
CTRL-X CTRL-S	Update Namespace Locally
CTRL-X CTRL-W	Update Namespace Globally
META-X	Distribute Namespace
META-X	Verify Namespace
META-X	Write NSE Buffer to File
<i>Class Commands:</i>	
S	Expand/Unexpand Class
META-S	Expand All Objects in Class
META-X	Unexpand All Objects in Class
F	Find Object
CTRL-F	Find Object and Aliases
META-F	Find Non-Alias Objects
META-CTRL-F	Find Objects With Specific Properties
META-X	Find Changed Objects
CTRL-H	Expand Horizontally
H	Toggle Horizontal Display
CTRL-C	Copy Objects in Class
W	Write (Update) Class Locally
CTRL-W	Write (Update) Class Globally
META-X	Verify Class Incrementally
R	Revert Class to Local State
META-R	Revert Class to Global State
C	Copy Class to This Namespace
META-X	Create Public Horizontal Format

NOTE:

* To invoke any command whose keystroke is listed as META-X, you press META-X and then type the name of the command as listed in the Command Name column. Note that completion is available.

Table 32-1

Namespace Editor Keystroke Commands (Continued)	
Keystroke	Command Name
<i>Object Commands:</i>	
S	Expand/Unexpand Object
CTRL-A	Add Attribute
META-X	Add Group Attribute
D	Delete Object
U	Undelete Object
META-CTRL-A	Add Alias for Object
META-CTRL-D	Delete Aliases for Object
W	Write (Update) Object Locally
CTRL-W	Write (Update) Object Globally
META-X	Verify Object Incrementally
CTRL-C	Copy Object
R	Revert Object to Local State
META-R	Revert Object to Global State
C	Copy Object to This Namespace
<i>Attribute Commands:</i>	
E	Edit Attribute
D	Delete Attribute
U	Undelete Attribute
CTRL-C	Copy Attribute
G	Toggle Group Status
V	View Attribute
W	Write (Update) Attribute Locally
CTRL-W	Write (Update) Attribute Globally
META-X	Verify Attribute Incrementally
R	Revert Attribute to Local State
CTRL-A	Add Attribute to Current Object
META-X	Add Group Attribute to Current Object
S	Unexpand Current Object
C	Copy Attribute to This Namespace
<i>Group Attribute Commands:</i>	
META-X	Add Group Attribute to Current Object
G	Toggle Group Status
META-A	Add Group Member
META-D	Delete Group Member
E	Edit Group Member
META-X	Add Key for Group Attribute

**NSE
Customization**

32.4 The following facilities are provided to allow you to customize the namespace editor:

- Customization variables — Allow you to change the way the namespace editor works, such as which fonts are displayed.
- Filters — Represent a subset of objects within a certain class.
- Horizontal formats — Used for the namespace editor display.
- Expert editors — Provide additional help in editing the objects found in a particular type of namespace.

NOTE: You only need to define expert editors if you are building a utility on top of the namespace editor.

Customization Variables **32.4.1** Several miscellaneous variables are provided to allow you to change how the namespace editor works:

nse:*prompt-in-mini-buffer-p*

Variable

When this variable is set to non-nil, you are prompted in the minibuffer for most of the input that the namespace editor needs. Setting this variable to non-nil allows the use of Zmacs keyboard macros because keyboard macros can use only input obtained from the minibuffer. (Refer to the *Explorer Zmacs Reference* for information on keyboard macros.)

When this variable is set to nil (the default), you are prompted for input in a variety of ways, depending on the command you are executing.

When this variable is set to :maybe, you are prompted for most input in the minibuffer unless an expert editor is found. This applies only to the Edit Attribute, Add Attribute, Edit Group Member, and Add Group Member commands.

nse:*nse-fonts*

Variable

A list of fonts used for displaying information in the namespace editor. This list should consist of one to four elements. Each element specifies a font to be used as follows:

- Font 0 — For class line information and headings at the top of the buffer
- Font 1 — For horizontal display headings
- Font 2 — For attribute line information (fixed-width font preferred)
- Font 3 — For object line information

If you specify only one font in the list, the font is used for all display items. The following shows the default font list that the namespace editor uses:

```
^(:higher-medfnb :cptfontb :cptfont :wider-medfnt)
```

nse:*personal-filter-list* Variable

A list of personal filters to be used during the current session. This variable is set by using the `nse:define-personal-filter` macro. To clear all personal filters, set this variable to nil.

nse:*personal-horizontal-format-list* Variable

A list of personal horizontal formats to be used during the current session. This variable is set by using the `nse:define-personal-horizontal-format` macro. To clear all system filters, set this variable to nil.

nse:*truncate-attribute-lines-nicely* Variable

If this variable is set to non-nil (the default), the attribute name and value are always displayed on one line of the namespace editor screen. (Both are truncated if necessary.) Use the View Attribute (V) command to display the entire name and value. If the value of this variable is nil, the attribute lines are allowed to wraparound.

If you want to obtain a handy representation of the namespace, you can do the following:

1. Execute the form `(setf *nse:*truncate-attribute-lines-nicely* nil)`.
2. Execute the command META-X Expand All Objects in NSE.
3. Execute the command META-X Print Buffer or META-X Write NSE Buffer to File.

Filters 32.4.2 A *filter* represents a subset of objects within a certain class. A filter line looks almost like a class line. In many ways, a filter behaves like a subclass (for the purpose of display) although it is not a class. You can expand and unexpand a filter as you can a class. You can perform any command within a filter that you can perform within a class. However, when you add an object, you cannot specify a filter name as the name of the class of the object.

The `nse:define-personal-filter` macro allows you to create personal filters. To use this personal filter across edit sessions, include the definition in your LOGIN-INIT file. The Find Objects With Specific Properties command (META-CTRL-F) allows you to create both personal and public filters but does not allow you to save them across edit sessions.

The following shows an example of a screen with unexpanded classes and filters:

```
:HOST
:HOSTS-BEGINNING-WITH-THE-LETTER-A      {PERSONAL FILTER OF:HOST}
:SITE
:PRINTER
:GRAPHICS-PRINTERS                      {PUBLIC FILTER OF :PRINTER}
```

nse:define-personal-filter *namespace-name filter-name class* Macro
 &key :name-pattern :attribute-list :test :first-only

Defines a personal filter for the *class* within *namespace-name* and calls the filter *filter-name*.

namespace-name — The name of the namespace (a string) for which you are creating this filter. If the value is :any, this filter exists for any namespace.

filter-name — The name of the filter.

class — The class of objects for which this filter exists.

Keywords: Keyword arguments are passed to the **name:list-objects-from-properties** function to filter the objects. Refer to that function or to the Find Objects With Specific Properties command (META-CTRL-F) for details on the keywords.

Horizontal Formats

32.4.3 The CTRL-H and H commands (described in paragraph 32.3.6, Class Commands) allow you to create or use a horizontal format. The META-X Create Public Horizontal Format command and the **nse:define-personal-horizontal-format** macro also allow you to create horizontal formats and provide a way to save them.

Create Public Horizontal Format Command

Keystroke: META-X Create Public Horizontal Format

Defines a public horizontal format. This command is to be executed while your cursor is beside a horizontally expanded class. During execution of this command, an object is created that represents the current horizontal format, and it is placed in the **:editor-customization** class. If you update this object globally, this horizontal format becomes permanently (and publicly) available for use by the H command. By editing the attributes of this object, you affect the way information is displayed when using horizontal formats.

nse:define-personal-horizontal-format *namespace-name class format-list* Macro
 &optional (*display-width-for-object-name 25*)

Defines a horizontal format for your personal use. To use this horizontal format across edit sessions, include the definition in your LOGIN-INIT file.

namespace-name — The *namespace-name* argument is the name of the namespace (a string) that will use your horizontal format. If you specify :any as the value of *namespace-name*, this horizontal format is available for all namespaces.

class — The *class* argument is the name of the class (or filter) to which this horizontal format applies.

format-list — The *format-list* argument is an alternating list of attribute names and their corresponding display widths. For example:

```
^ (:addresses 50 :location 30)
```

display-width-for-object-name — The *display-width-for-object-name* argument specifies the width for the object name that automatically appears at the beginning of each line in the horizontal display. You can specify a display width other than the default of 25.

Expert Editors

32.4.4 If you are writing a utility built on top of the namespace editor, you probably want to customize certain aspects of the editor. To do this, you use expert editors. The most common uses of expert editors are the following:

- To provide documentation to the user for certain objects and attributes within a namespace
- To provide a menu of values to choose from when the user edits the value of an attribute

You can use expert editors for many other operations, all of which are discussed in the following paragraphs.

NOTE: You only need to define expert editors if you are building a utility on top of the namespace editor.

The following list briefly describes the topics discussed about expert editors:

- Paragraph 32.4.4.1 discusses the `nse:define-nse-expert-editor` macro, which allows you to define an expert editor.
- Paragraph 32.4.4.2 discusses in detail the *spec-list* of the `nse:define-nse-expert-editor` macro.
- Paragraph 32.4.4.3 describes several variables that you can use when creating an expert editor.
- Paragraph 32.4.4.4 discusses the verification macros that you must use when defining verification routines to be called by expert editors.
- Paragraph 32.4.4.5 describes how to view expert editors that are currently defined.

nse:define-nse-expert-editor Macro 32.4.4.1 The following macro defines an expert editor.

```
nse:define-nse-expert-editor spec-list usage Macro
&key :incremental-verification-routine :edit-routine :side-effect
:documentation-string :add-group-member-routine :before-update
:after-update :before-edit :default-attribute-list-for-add-object
:attribute-menu-list-for-add-attribute :namespace-verification-routine
:force-object-names-to-string-p
```

NOTE: The `nse:define-nse-expert-editor` macro does not have a permanent effect unless you perform a disk-save.

spec-list — The *spec-list* argument is a three-element list specifying the elements of the namespace that call this expert editor when the elements are being edited, verified, or queried for documentation. It is a list of the form (*class object attribute*). The value `:any` matches any item. For example:

```
^ (:host :any :addresses)
```

This expert editor helps edit, verify, and/or document network addresses of hosts. Since *object* is `:any`, all *host* objects will use this expert editor for network addresses.

If you need more information on the *spec-list* argument, refer to paragraph 32.4.4.2, Spec-List Formats.

usage — The kind of namespace for which you are writing the expert editor. The *usage* argument is `:network` for a network namespace.

NOTE: You can specify an arbitrary value for *usage*. However, to make the expert editors work, you must set the `:usage` attribute in the namespace object to the same value. (You can have several expert editors that are used for a particular namespace `:usage`. You might have a different expert editor for each attribute in your namespace, for instance.) Refer to paragraph 32.2.6, Default Attributes, for information on setting the `:usage` attribute.

You can set *usage* to `:any`, which means that the expert editor works for any type of namespace. Suppose that two expert editors exist with the same *spec-list*: one specifies `:any` for *usage* and the other specifies a specific *usage*. The expert editor with the specific *usage* overrides the one with the `:any usage` (but only if the *usage* matches the namespace `:usage`).

`:incremental-verification-routine` — Specifies an incremental verification routine to be performed when you press META-CTRL-V (verification command) beside an attribute, object, or class of objects. This routine is also used for the menu items Verify Class Incrementally, Verify Object Incrementally, and Verify Attribute Incrementally. When you write your verification routines, you should use the verification macros described in paragraph 32.4.4.4, Verification Routine Macros.

- :edit-routine** — Specifies an edit routine to be used when you press E beside an attribute. The routine should return the new value of the attribute. (The namespace editor takes care of updating the attribute.) This routine is used when editing a group member as well as when editing a regular attribute. Interesting variables to be used in this routine are `nse:*old-group-member-value*` and `nse:*old-value*`.
- :side-effect** — Specifies a routine to be called after an attribute or object has been changed. For example, after a host object has been deleted, you can call a special routine to write that object to a file for future reference.
- Consider another example: after you edit the attribute "FILE PRINTER", you may want to see if the specified printer is defined. Since this routine is called after any change (deletion, addition, editing, and so on), there is a special variable `nse:*operation*` whose value reflects the operation that the user just performed. (Paragraph 32.4.4.3, Expert Editor Variables, describes the `nse:*operation*` variable.)
- Within your edit routine, you may want to check the value of `nse:*operation*` to be sure that your `:side-effect` is executed only when you intend it to be.
- :documentation-string** — Specifies a string to be displayed when the user presses CTRL-SHIFT-D (Long Documentation command) or CTRL-SHIFT-A (Short Documentation command). You can specify documentation strings for a class line, object line, or an attribute line.
- :add-group-member-routine** — Specifies a routine to be executed in order to add a group member (META-A). The routine should return the new group member value to be added. (The namespace editor takes care of adding the group member.)
- :before-update** — Specifies a routine to be executed before an update is made.
- :after-update** — Specifies a routine to be executed after an update is made.
- :before-edit** — Specifies a routine to be executed before an attribute is edited.
- :default-attribute-list-for-add-object** — Specifies an attribute list (`((att1 val1) (att2 val2))`) to appear by default on a newly added object.
- :attribute-menu-list-for-add-attribute** — Specifies a menu list of attributes to add to the object.
- :namespace-verification-routine** — Specifies a routine to be used for large-scale verification of the namespace, class, object, or attribute. When you write your verification routines, you should use one of the verification macros described in paragraph 32.4.4.4, Verification Routine Macros.

:force-object-names-to-string-p — If this keyword is `t` (the default is `nil`), the namespace editor assumes that all objects are strings during commands such as Add Object, Find Object, and so on. This keyword is useful on the expert editor for the namespace.

When this keyword is `t` for a general expert editor on the namespace, you can override it in a specific expert editor by specifying the keyword `:dont`. Then objects are not interpreted as strings when that specific expert editor is being used.

Examples: The following example defines an expert editor to provide the user with documentation, an edit routine, and an incremental verification routine on the `:timezone` attribute. The list ``(:any :any :timezone)` represents (*class-name object-name attribute-name*). Because `:any` is the value of *class-name* and *object-name*, this expert editor is called for the `:timezone` attribute regardless of the class or the object with which it is associated. The *usage* argument is `:network`, which means that this expert editor is used only when the `:usage` attribute of the namespace object is set to `:network`. (Refer to paragraph 32.2.6, Default Attributes, for information on setting the `:usage` attribute.)

When the user presses META-CTRL-V (verification command) beside any `:timezone` attribute, the incremental verification routine `nse:verify-timezone` is called. When the user presses E (Edit command) beside any `:timezone` attribute, the edit routine `nse:timezone-mlist` is called. When the user presses CTRL-SHIFT-D or CTRL-SHIFT-A beside any `:timezone` attribute, the string specified by `:documentation-string` is displayed.

```
(nse:define-nse-expert-editor `(:any :any :timezone) :network
  :edit-routine 'nse:timezone-mlist
  :documentation-string "Time zone where this particular host console
    is located. Typically, this differs from the default only if
    the site is widely dispersed."
  :incremental-verification-routine 'verify-timezone)
```

The next example defines an expert editor for the class `:host`. The list ``(:host :any nil)` specifies that this expert editor is called for the class `:host` and any (`:any`) object on the class `:host`. Because the *attribute-name* is `nil`, this expert editor is not called for attributes on the objects of the class `:host`.

This expert editor provides the user with documentation and an incremental verification routine for the class `:host`. Furthermore, the `:default-attribute-list-for-add-object` keyword specifies an attribute list to be added by default on any newly created object for the class `:host`. The `:attribute-menu-list-for-add-attribute` keyword specifies a menu list of attributes that can be added to the object. The user decides which item to choose from this menu.

In the `defparameter` that defines the menu list, compare the syntax for specifying the group attributes `:addresses` and `:aliases` to the syntax for specifying a scalar attribute such as `:associated-machine`. Also, notice the `:other` item. By selecting `:other` from the menu, the user is prompted to enter an attribute name. (The prompting is built into the namespace editor software.)

```
(define-nse-expert-editor `(:host :any nil) :network
  :documentation-string
    "A network host object represents a computer on your network."
  :incremental-verification-routine 'verify-host
  :default-attribute-list-for-add-object
    'make-attribute-list-from-host-info
  :attribute-menu-list-for-add-attribute '*host-attribute-mlist*)
```

```

;; Menu list for :attribute-menu-list-for-add-attribute
(defparameter *host-attribute-mlist*
  (list `(:addresses :value (:addresses :group))
        `(:aliases :value (:aliases :group))
        :associated-machine
        :bitmap-printer
        :sys-host
        :sys-host-translations-file
        :system-type
        :other))

;; Function called for :default-attribute-list-for-add-object
(defun make-attribute-list-from-host-info (&optional host-name)
  (list :short-name (or host-name *object-name* "New Explorer")
        :string-for-printing (or host-name *object-name*
                                  "New Explorer")
        `(:aliases :group) ()
        `(:addresses :group) (list (list :chaos nil) (list :ip nil))
        `(:services :group)
          (or *default-services-list* *net-default-services-list*)
        :machine-type "Explorer"
        :system-type :lisp))
))

```

The next example defines an expert editor for the namespace itself (at the top level). The `:namespace-verification-routine` keyword specifies a top-level routine to call to verify a network namespace. If you do not specify this routine, there is a default routine provided by the system that looks for `:namespace-verification-routine` editors on objects and attributes. This default routine executes all existing routines for existing objects and attributes in the namespace. Because this expert editor is for the entire namespace, the list `(nil nil nil)` specifies that these routines are not used for a class, object, or attribute, but for the namespace itself.

The `:before-edit` keyword specifies a routine to execute before the user edits an attribute. In this example, the `network-mode-add-default-classes` routine sets up classes for `:host`, `:printer`, and so on. The `:side-effect` keyword specifies a routine to call after the namespace is displayed on the screen for editing. In this case, the `network-mode-add-default-objects` routine adds a `SYS` host for a new namespace. The `:force-object-names-to-string-p` keyword makes the namespace editor assume that all objects are strings during commands such as `Add Object`, `Find Object`, and so on.

```

;; These are editors for the namespace itself — top level.
(define-nse-expert-editor '(nil nil nil) :network

  ;; Always set up classes for :host, :printer, etc.
  :before-edit 'network-mode-add-default-classes

  ;; Add a SYS host for a new namespace.
  :side-effect 'network-mode-add-default-objects

  ;; Object names should always be strings.
  :force-object-names-to-string-p t
  :namespace-verification-routine 'com-network-verify-namespace)

```

The following example defines an expert editor for the `:default-mail-host` attribute. The list ``(:any :any :default-mail-host)` specifies that the editor is called for the `:default-mail-host` attribute, regardless of the class or the object with which it is associated. The class should not matter here because the `:default-mail-host` attribute can exist in the `:host`, `:site`, or `:user` classes.

```
(define-nse-expert-editor `(:any :any :default-mail-host) :network
  :incremental-verification-routine `verify-string
  :namespace-verification-routine `verify-string
  ;; Called by the namespace verification routine to verify this particular attribute.
  :documentation-string
    "The host to use as the default in mail addresses that have an
    unspecified host. This applies only to new messages entered
    into the mail system and does not override the options
    relating to mail servers and gateways.")
```

Spec-List Formats 32.4.4.2 This paragraph describes the *spec-list* formats of the `nse:define-nse-expert-editor` macro for the following types of expert editors:

- Attribute expert editors
- Object expert editors
- Class expert editors
- Namespace expert editors

Invalid *spec-list* formats are discussed after these formats.

Spec-List Format for Attributes An expert editor that always operates on an attribute uses the following *spec-list*:

(non-nil non-nil non-nil) usage

This attribute expert editor operates either on a particular attribute (if you supply the name of an attribute) or on any attribute (if you specify `:any` in the attribute slot). The *class*, *host*, and *usage* requirements must also be met so that the expert editor can operate on a particular attribute.

The following is the *spec-list* of a specific expert editor for an attribute. The routines specified for this expert editor override any routines specified by a general attribute expert editor. (That is, a specific expert editor overrides a general expert editor.)

(non-nil non-nil attribute-name)

The following *spec-list* is the *spec-list* of a general expert editor for an attribute. Any routine specified by this expert editor is used unless that same routine is specified by a matching specific attribute expert editor.

(non-nil non-nil :any)

The following routines are appropriate for an attribute expert editor:

```
:documentation-string      :before-update
:edit-routine              :after-update
:before-edit               :namespace-verification-routine
:side-effect               :incremental-verification-routine
:add-group-member-routine
```

The following routines are not appropriate for an attribute expert editor:

```
:default-attribute-list-for-add-object
:attribute-menu-list-for-add-attribute
:force-object-names-to-string-p
```

Spec-List Format for Objects An expert editor that always operates on an object uses the following *spec-list*:

```
(non-nil non-nil nil) usage
```

This object expert editor operates either on a particular object (if you supply the name of an object) or operates on any object (if you specify `:any` in the object slot). The *class* and *usage* requirements must also be met so that the expert editor can operate on a particular object.

The following is the *spec-list* of a specific expert editor for an object. The routines specified for this expert editor override any routines specified by a general attribute expert editor. (That is, a specific expert editor overrides a general expert editor.)

```
(non-nil object-name nil)
```

The following is the *spec-list* of a general expert editor for an object. Any routine specified by this expert editor is used unless that same routine is specified by a matching specific object expert editor.

```
(non-nil :any nil)
```

The following routines are appropriate for an object expert editor:

```
:documentation-string
:before-update
:after-update
:namespace-verification-routine
:incremental-verification-routine
:default-attribute-list-for-add-object
:attribute-menu-list-for-add-attribute
```

The following routines are not appropriate for an object expert editor:

```
:edit-routine (Note that objects cannot be edited; only attributes.)
:before-edit
:side-effect
:add-group-member-routine
:force-object-names-to-string-p
```

Spec-List Format for Classes An expert editor that always operates on a class uses the following *spec-list*:

(non-nil nil nil) usage

This class expert editor operates either on a particular class (if you supply the name of a class) or on any class (if you specify *:any* in the class slot). The *usage* requirement must also be met so that the expert editor can operate on a particular class.

The following is the *spec-list* of a specific expert editor for a class. The routines specified for this expert editor override any routines specified by a general attribute expert editor. (That is, a specific expert editor overrides a general expert editor.)

(class-name nil nil)

The following is the *spec-list* of a general expert editor for a class. Any routine specified by this expert editor is used unless that same routine is specified by a matching specific class expert editor.

(:any nil nil)

The following routines are appropriate for a class expert editor:

:documentation-string
:before-update
:after-update
:namespace-verification-routine
:incremental-verification-routine
:default-attribute-list-for-add-object
:attribute-menu-list-for-add-attribute

The following routines are not appropriate for a class expert editor:

:edit-routine (Note that classes cannot be edited; only attributes.)
:before-edit
:side-effect
:add-group-member-routine
:force-object-names-to-string-p

Spec-List Format for Namespace Expert Editors An expert editor that always operates on a namespace uses the following *spec-list*:

(nil nil nil) usage

This expert editor operates either on a particular type of namespace (if you supply a particular *usage*) or on any namespace (if you specify *:any* for *usage*).

The following is the *spec-list* of a specific expert editor for a network namespace. The routines specified for this expert editor override any routines specified by a general attribute expert editor. (That is, a specific expert editor overrides a general expert editor.)

```
(nil nil nil) :network
```

The following is the *spec-list* of a general expert editor for any type of namespace. Any routine specified by this expert editor is used unless that same routine is specified by a matching specific namespace expert editor.

```
(nil nil nil) :any
```

The following routines are not appropriate for a namespace expert editor:

```
:before-edit
:side-effect
:namespace-verification-routine
:force-object-names-to-string-p
```

The following routines are not appropriate for a namespace expert editor:

```
:edit-routine
:add-group-member-routine
:before-update
:after-update
:default-attribute-list-for-add-object
:attribute-menu-list-for-add-attribute
:incremental-verification-routine
:documentation-string
```

NOTE: The *usage* argument is illustrated for namespace expert editors, but it works the same way for all expert editors. The value `:any` is general and is overridden by a specific *usage*. If you specify an expert editor for a specific *usage*, it always takes precedence over one specified for a general (`:any`) *usage*.

Invalid Spec-List Formats The following *spec-list* formats are invalid:

```
(nil non-nil non-nil)
```

```
(nil nil non-nil)
```

These invalid formats do not cause errors, but the routines specified by expert editors with invalid *spec-lists* are never used.

Expert Editor Variables 32.4.4.3 You can use the following variables when making an expert editor.

nse:*object* Variable

This variable is bound to the object currently (or most recently) edited or verified in the namespace editor. It is of the form (*object-name class attribute-list*). This variable is also bound when an attribute is being added to an object.

nse:*attribute* Variable

This variable is bound to the attribute name currently being edited, verified, or queried for documentation.

nse:*stream* Variable

This variable is bound to the stream that is used for printing verification errors and warnings during namespace verification.

nse:*value* Variable

This variable is bound to the value of the attribute currently being edited, verified, or queried for documentation. At the time a **:side-effect** of editing is performed, **nse:*value*** is set to the newly chosen value of an attribute.

nse:*old-value* Variable

At the time a **:side-effect** of editing is performed, **nse:*old-value*** is set to the old value of an attribute (the value of the attribute before it was edited).

nse:*buffer* Variable

This variable is bound to the instance of the namespace editor buffer currently being edited.

nse:*group-member-value* Variable

This variable is bound to the value of the group member currently being edited. At the time a **:side-effect** of editing is performed, this variable is bound to the newly selected value of the group member.

nse:*old-group-member-value* Variable

At the time a **:side-effect** of editing is performed, this variable is bound to the old value of the group member (the value of the group member before it was edited).

nse:*group-members-to-delete* Variable

If the user has performed the Delete Group Members command, the variable is bound to the list of group members that were chosen for deletion.

nse:*operation* Variable

This variable is bound to the name of the current command being executed. This variable can have the following values:

**:add-object, :delete-object, :undelete-object,
:edit-attribute, :add-attribute, :delete-attribute, :undelete-attribute,
:edit-group-member, :add-group-member, :delete-group-member,
:update-class, :update-object, :update-attribute**

nse:*local* Variable

This variable is bound at update time to let the programmer know if the update is local or global. The value of this variable is useful only if **nse:*operation*** is **:update-class, :update-object, or :update-attribute**.

nse:*verification-level* Variable

This variable specifies the default level of verification, which applies to incremental verification as well as namespace verification. The following lists the possible values for this variable:

- **:errors-only** — Warnings are not printed for incremental verification or namespace verification.
- **nil** or **:none** — Automatic incremental verification is not performed before updates.
- **:full** — Full verification is performed; warnings and error messages are printed for all verifications.

The default value of this variable is **:full**.

Verification Routine 32.4.4.4 The following macros are useful in a verification routine called by
Macros expert editors.

nse:verify-err Macro

Prints a verification warning to **nse:*stream*** and also to ***standard-output*** (if different from **nse:*stream***). The value of **nse:*stream*** and the arguments you supply to the macro are passed to the **format** function. (Refer to the *Explorer Input/Output Reference*.)

nse:verify-wrn Macro

Prints a verification error to **nse:*stream*** and also to ***standard-output*** (if different from **nse:*stream***) unless the value of the **nse:*verification-level*** variable is **:errors-only**. (This variable is described in paragraph 32.4.4.3, *Expert Editor Variables*.) The value of **nse:*stream*** and the arguments you supply to the macro are passed to the **format** function. (Refer to the *Explorer Input/Output Reference*.)

Viewing Expert Editors 32.4.4.5 You can use the following command to view expert editors that are currently defined.

Edit Internal Namespace

Command

Keystroke: META-X Edit Internal Namespace

Displays expert editors that are currently defined. This command prompts for a symbol whose value is a namespace object (flavor instance). The namespace editor for the corresponding namespace is then selected.

All of the expert editors for the namespaces currently defined are stored in an internal namespace. You can display this namespace of expert editors by entering the symbol `nse:nse-namespace` in response to the prompt. Class names of the internal `nse:nse-namespace` correspond exactly to the *usage* arguments given with the `nse:define-nse-expert-editor` macro. Object names correspond to the *spec-list* arguments exactly. Attribute names correspond to the keyword arguments, and the attribute values are the corresponding values of those keyword arguments. You can edit this namespace, but the changes are discarded when you boot.

The following shows an example screen:

```

                                NAMESPACE EDITOR FOR "NSE NAMESPACE"

:ANY
:NAMESPACE

(:HOST :ANY :ADDRESSES)

:INCREMENTAL-VERIFICATION-ROUTINE  VERIFY-ADDRESSES
:EDIT-ROUTINE                       NIL
:ADD-GROUP-MEMBER-ROUTINE           NIL
:DOCUMENTATION-STRING                "A list of network addresses for this host."
:SIDE-EFFECT                         SIDE-EFFECT-FOR-ADDRESSES
:BEFORE-UPDATE                       NIL
:AFTER-UPDATE                         NIL
:BEFORE-EDIT                          NIL
:DEFAULT-ATTRIBUTE-LIST-FOR-AD...    NIL
:ATTRIBUTE-MENU-LIST-FOR-ADD-A...    NIL
:NAMESPACE-VERIFICATION-ROUTINE     NIL
:FORCE-OBJECT-NAMES-TO-STRING-P     NIL

```

User Functions

32.5 The user functions provide a way to perform the same operations as the namespace editor except that these functions are used from within your program. They also provide some other operations. The following topics are discussed:

- **Namespace functions** — Allow you to list the namespace search rules, list the known namespaces, copy all the objects from one namespace to another, display information about the available namespaces, create or load a personal or basic namespace, or delete a namespace.
- **Modification functions** — Allow you to add or delete objects, attributes, members of group attributes, and aliases.
- **Retrieval functions** — Allow you to perform several retrieval operations on objects, such as looking up the objects in one or more namespaces, looking up the value of an attribute, or looking up a set of objects from their properties.
- **Object manipulation functions** — Allow you to perform the following operations on namespace object structures that you have already obtained: get the value of an attribute from an object, get an association list of attribute names and values from an object, or generate a list of objects including their name, class, and attribute list.
- **Miscellaneous functions** — Allow you to attach to an object a hidden property with a value, get the value of a hidden property, find out if a namespace has a local cache, mark one object (or all objects) in a namespace to be refreshed on the next lookup, and clear the local cache of a namespace. You can also find out if a namespace is a client of a non-Explorer namespace, force a server to be booted from local namespace files even if it is not on the server boot list, or initialize the name service as is done during a cold boot.

The following notes explain some arguments that are used in many of these functions:

- Several functions take an optional namespace argument. This can be a namespace name (string or symbol), an instance of a namespace, or `nil` (which is the default).
- A name argument (such as the name of an object) can be qualified or relative. A qualified name indicates the containing namespace. If the name is relative (unqualified), either the namespace argument or the search rules are used. A namespace argument takes precedence over a qualified name.
- Group attribute members work differently in these areas: merges, updates, and lookup by properties. Unlike scalar attributes, which override succeeding attributes of the same name, group attributes are combined on these operations according to the rules of set union.
- Aliases are implemented in many functions. These functions take a chase and a merge argument. A *chase* argument means to follow an alias chain and return the terminal (or real) object. A *merge* argument means that attributes along an alias chain are merged to form a synthesized view.

Here are the four object lookup modes for aliases:

Chase	Merge	Action
nil	nil	Returns the top-level object only
nil	non-nil	Returns a synthesized object with the name of the top-level object (merged attributes are attached)
non-nil	nil	Returns the bottom-level object only
non-nil	non-nil	Returns a synthesized object with the name of the bottom-level object (merged attributes are attached)

- In merges, the higher-level attributes always have priority, regardless of which object is being returned.
- Group attribute values are combined according to the rules of set union during the merge process, if both are groups. Thus, a lower-level group attribute can be overridden by making the higher-level attribute a scalar attribute (perhaps with a list value if that is what clients expect).
- An alias-of pointer can be a qualified or unqualified name. If it is unqualified, the first attempt is in the same namespace as the alias.

Suppose you have the following objects:

```
"A" :EXAMPLE
     :*Alias-of*           "B"
     :Unique-scalar-a     "A"
     :Dup-scalar          "A"
G    :Dup-group           (A C)
```

```
"B" :EXAMPLE
     :Unique-scalar-b     "B"
     :Dup-scalar          "B"
G    :Dup-group           (B C)
```

The `name:list-object` function (described in more detail later) illustrates how the `:chase-aliases` and `:merge-aliases` arguments work on aliases.

Example 1: `(name:list-object "A" :example :chase-aliases nil :merge-aliases nil)`

The following is returned:

```
("A" :EXAMPLE (:*alias-of* "B"
                    :Unique-scalar-a "A"
                    :Dup-scalar "A"
                    (:Dup-group :Group) (A C)))
```

Example 2: (name:list-object "A" :example :chase-aliases nil :merge-aliases t)

The following is returned:

```
("A" :EXAMPLE (:Unique-scalar-b "B"
:Dup-scalar "A"
(:Dup-group :Group) (B C A)
:*alias-of* "B"
:Unique-scalar-a "A"))
```

Example 3: (name:list-object "A" :example :chase-aliases t :merge-aliases nil)

The following is returned:

```
("B" :EXAMPLE (:Unique-scalar-b "B"
:Dup-scalar "B"
(:Dup-group :Group) (B C)))
```

Example 4: (name:list-object "A" :example :chase-aliases t :merge-aliases t)

The following is returned:

```
("B" :EXAMPLE (:Unique-scalar-b "B"
:Dup-scalar "A"
(:Dup-group :Group) (B C A)
:*alias-of* "B"
:Unique-scalar-a "A"))
```

Namespace Functions 32.5.1 The following are the available namespace functions.

- | | |
|--|----------|
| name:list-namespace-search-rules | Function |
| Returns a list of the names of the namespaces in the current search list. | |
| name:list-known-namespaces | Function |
| Returns a list of the names of the namespaces for which there are available instances, preserving the search-list order. | |
| name:show-namespace-configuration &optional (<i>stream</i> *standard-output*) | Function |
| Displays information about the namespaces available on this machine. The following information is displayed: | |
| <ul style="list-style-type: none"> ■ Name — The name of the namespace. ■ Type — The type of namespace, such as :personal. ■ Usage — The usage mode corresponding to a set of expert editors. (Refer to paragraph 32.4, NSE Customization, for information on expert editors.) ■ Search # — Where in the search list the namespace appears (1 for first, 2 for second, and so on). ■ Server? — Whether the namespace has a server at this machine. | |

name:copy-namespace *source-namespace destination-namespace* Function
 &optional *local-p*

Copies all the objects in *source-namespace* to *destination-namespace*. Both namespaces must already exist. If *local-p* is non-nil and *source-namespace* has a cache, only cached items are copied. You can use *local-p* to make a copy of the local cache.

Example: (name:copy-namespace "ROBIN" "BIRDS")

name:load-personal-namespace *namespace-name* Function
 &key (:search-list-location :beginning) :pathname :new
 :changes-before-save (:auto-save-enabled t) :read-only :usage

Creates or loads a personal namespace called *namespace-name*.

:search-list-location — Specifies where to put the namespace in the search list: **:beginning**, **:end**, or **nil** (that is, do not put it on the search list).

:pathname — Specifies the default file where the namespace is stored. The syntax for the default file is as follows:

LM: NAME-SERVICE; *namespace-name*-PERSONAL.XLD

For example, if your namespace is CATS, the default file is LM: NAME-SERVICE; CATS-PERSONAL.XLD.

Note that you can specify a remote file instead of a local file if necessary.

:new — Either non-nil or nil (nil is the default). You should use **:new t** if no files need to be loaded because this is a new namespace.

When you load a personal namespace that was specified as new, the following message may appear:

WARNING: Namespace X is being initialized without using existing database files.

This message informs you that there are old files by this name that are not being loaded and that new versions are being created. You may have intended to do this, or you may have specified **:new** accidentally. The old versions of the files still exist and can be retrieved if desired.

:auto-save-enabled — When this keyword is non-nil (the default), the entire namespace is written to an xld file after every *n* changes (where *n* is equal to **:changes-before-save**). While this keyword is nil, the namespace is never written to an xld file.

CAUTION: With a personal namespace, writing to an xld file is the only way that changes are permanently recorded.

:changes-before-save — If **:auto-save-enabled** is non-nil, the value of this keyword specifies the number of updates to the namespace that are made before a new copy of the namespace is written to an xld file.

:read-only — If this keyword is non-nil, writes are disallowed. (The default is nil.)

:usage — Namespace content (used by namespace editor). You can specify an expert editor here (for example, **:network**).

Example: (name:load-personal-namespace "PHONE-LIST")

name:configure-namespace &optional *name* Function

Creates a namespace instance based on user descriptions. This function uses the Configure Namespace menu (described in paragraph 32.3.2, Configuring the Namespace) to obtain the user input. The optional *name* argument is the name of the namespace. If you do not specify *name*, you can specify it on the menu. The returned values for this function are namespace instance, usage, and search list location.

name:add-namespace *namespace-name type* &optional *search-list-location* Function
&rest *init-args*

Creates a new namespace called *namespace-name*. The *type* argument specifies the type of namespace (for example, **:personal**).

CAUTION: The **name:add-namespace** function is mainly intended for internal use, but it might be useful for personal and basic namespaces. In general, do not use this function for more complicated types of namespaces.

search-list-location — Specifies where to put the namespace in the search list: **:beginning**, **:end**, or **nil** (that is, do not put it on the search list).

init-args — These initialization arguments (also called *init-args*) differ for the type of namespace you are creating, as follows:

The *init-arg* for the **:basic** type of namespace is as follows:

:read-only — If **:non-nil**, writes are disallowed (**nil** is the default).

The *init-args* for the **:personal** type of namespace are as follows:

:new — Either **non-nil** or **nil** (**nil** is the default). You should use **:new t** if no files need to be loaded because this is a new namespace.

When you load a personal namespace that was specified as **new**, the following message may appear:

```
WARNING: Namespace X is being initialized without using
existing database files.
```

This message informs you that there are old files by this name that are not being loaded and that new versions are being created. You may have intended to do this, or you may have specified **yes** for **New namespace?** in error. The old versions of the files still exist and can be retrieved if desired.

:namespace-file-pathname — Specifies the default file where the namespace is stored. The syntax for the default file is as follows:

LM: NAME-SERVICE; *namespace-name*-PERSONAL.XLD

For example, if your namespace is CATS, the default file is LM: NAME-SERVICE; CATS-PERSONAL.XLD.

Note that you can specify a remote file instead of a local file if necessary.

:auto-save-enabled — When this *init-arg* is non-*nil* (the default), the entire namespace is written to an xld file after every *n* changes (where *n* is equal to **:changes-before-save**). While this keyword is *nil*, the namespace is never written to an xld file.

CAUTION: With a personal namespace, writing to an xld file is the only way that changes are permanently recorded.

:changes-before-save — If **:auto-save-enabled** is non-*nil*, the value of this *init-arg* specifies the number of updates to the namespace that are made before a new copy of the namespace is written to an xld file.

:usage — Governs the way the editor edits the namespace. You can specify the usage mode corresponding to a set of expert editors here. Refer to paragraph 32.4, NSE Customization, for information on expert editors.

:read-only — If non-*nil*, writes are disallowed (*nil* is the default).

The *init-args* for the **:public:** type of namespace are as follows:

NOTE: The **:public** *init-args* are for the client instance of a public namespace, whereas the **:explorer-server** *init-args* are for the server instance of a public namespace. The **:public** and the **:explorer-server** types correspond to two different namespace flavors.

:setup-from-object — Specifies an object of class **:namespace** used to initialize the namespace.

:bootstrap-servers — Specifies a list of initial servers.

:source-domain — Specifies the namespace from which the namespace object specified in **:setup-from-object** is obtained.

The *init-args* for the **:explorer-server** type of namespace are as follows:

:new — Either non-*nil* or *nil* (*nil* is the default). You should use **:new t** if no files need to be loaded because this is a new namespace.

When you load a public namespace that was specified as *new*, the following message may appear:

WARNING: Namespace X is being initialized without using existing database files.

This message informs you that there are old files by this name that are not being loaded and that new versions are being created. You may have intended to do this, or you may have specified `:new` accidentally. The old versions of the files still exist and can be retrieved if desired.

NOTE: The rest of the *init-args* for `:explorer-server` are mutually exclusive options that govern the behavior of the namespace instance being created. The default is `nil` for all of them except that `:re-boot` is used if nothing else is specified.

:configure — Either non-`nil` or `nil`. If set to non-`nil`, changes are not propagated until `:distribute` is specified (that is, until you send the `:distribute` method to the server namespace instance or invoke the `name:distribute-namespace` function described in paragraph 32.5.5, Miscellaneous Functions).

You should use the `:configure` option when creating a new namespace from scratch. Changes are not propagated to other co-servers (actually, there are not any yet) until you finish the initial editing. Changes are logged so that you can restart without losing work in the event of a crash.

Your changes are made directly to the new server instance and *not* through a client that is caching information.

:bootstrap — Specifies a server name. This *init-arg* allows this namespace server to be initialized with all the information known to another server.

:standalone — Either non-`nil` or `nil`. If set to non-`nil`, this *init-arg* does not propagate or resynchronize with co-servers.

:re-boot — Either non-`nil` or `nil`. If set to non-`nil`, this *init-arg* resynchronizes with co-servers.

:dormant — Either non-`nil` or `nil`. If set to non-`nil`, this *init-arg* does not do anything.

:convert — Either non-`nil` or `nil`. If set to non-`nil`, this *init-arg* does not even log, save, or audit until `:distribute` is specified (that is, until you send the `:distribute` method to the server namespace instance or invoke the `name:distribute-namespace` function described in paragraph 32.5.5, Miscellaneous Functions).

You should use the `:convert` option when you are automatically (that is, from a program) obtaining information from one source (for example, a `SITEINFO` file) and putting it in the new namespace. Changes are not propagated to the co-servers until `:distribute` is specified. Also, you do not need to log the changes because you can always restart the program in the event of a crash.

Your changes are made directly to the new server instance and *not* through a client that is caching information. (The `:convert` option is used by the Convert Siteinfo utility.)

The *init-args* for the `:symbolics` type of namespace are as follows:

`:setup-from-object` — Specifies an Explorer namespace object used to initialize the namespace.

`:bootstrap-servers` — Specifies a list of initial servers, which are host names.

`:source-domain` — Specifies the namespace from which the namespace object specified in `:setup-from-object` is obtained.

`:class-list` — Specifies a list of Symbolics classes. The default is the value of the `name:*known-symbolics-classes*` variable.

This function returns an instance of the namespace.

Example:

```
(name:add-namespace "Tom" :personal :beginning :new t)
;; Use :new if is not already created.
```

name:delete-namespace *name &optional search-list-only* Function

Deletes a namespace from the search list and from the list of currently known namespaces (unless *search-list-only* is non-nil).

Example:

```
(name:delete-namespace "Tom" t)
```

Modification Functions **32.5.2** The following are the available functions for modifying a namespace.

name:add-object *object-name object-class &key :alias-of :attributes (:cacheable t) :local :namespace :pre-delete* Function

Adds (or updates) an object of a given name and class to have the specified attributes.

NOTE: Adding an object that replaces an existing object of the same name overwrites old attributes.

object-name — Can be qualified or relative (unqualified).

object-class — The object class.

`:alias-of` — The name of an object for which this object is an alias.

`:attributes` — A list that contains pairs of either of the following for an attribute whose value is a list that can be incrementally changed:

attribute-name, value

(attribute-name :group [key]), value

The *attribute-name*, in the latter case, should not be a list whose second member is `:group`. Optionally, a group attribute can have an associated *key* function that is used to access the member part and test for equality during merging. A common function to use is `car`.

`:cacheable` — This object can be cached after client access (the default is `t`).

`:local` — Makes this update only to the cache (if this namespace has a local cache).

- :namespace** — The namespace to which this object is to be added. If you do not specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.
- :pre-delete** — Deletes any previously existing version first. Otherwise, the new attributes are merged with any existing ones.

This function returns the new database object if the object is successfully added.

Examples:

```
(add-object "Fairy-Tales|Cinderella" :character
:attributes '(:shoe-size 3 (:skills :group)
              (:cleaning :cooking)
              "Motto" "Someday my pransome hince will come"))
```

This example adds the object *cinderella* to the `:character` class in the `Fairy-Tales` namespace. (The namespace must already exist.) It also adds the attributes `:shoe-size`, `:skills` (a group attribute), and `"Motto"`.

The following example illustrates the use of a key function with a group attribute:

```
(add-object "Fairy-Tales|Cinderella" :character
:attributes '((sizes :group 'car) (:shoe 3) (:dress 5)))
```

Later, if a group member is added to the `sizes` attribute that matches the `car` of one of the members, the new group attribute and its value replace the old group member and its value. For example, the group member `(:shoe 2)` would replace the old group member `(:shoe 3)`. (Refer to the `name:add-group-member` function for information on adding group members.)

name:delete-object *object-name object-class &key :local :namespace* Function
:delete-aliases **:chase-aliases**

Deletes the object identified by *object-name* and *object-class*.

NOTE: This function does not map across the search rules; either the namespace that you specify with **:namespace** or the default namespace is used. The default namespace is the first namespace on the search list.

- :local** — If you specify **:local** as non-`nil`, only the locally cached copy is deleted.
- :namespace** — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.
- :delete-aliases** — If you specify **:delete-aliases** as non-`nil`, this function also deletes any aliases of this object within the same namespace.
- If *object-name* is an alias, the alias objects that point to this alias are deleted.
- :chase-aliases** — If you specify **:chase-aliases** as non-`nil`, this function first goes to the terminal object if *object-name* is an alias and you are deleting aliases. Otherwise, the function goes to the object identified by *object-name* and *object-class*.

Example:

```
(name:delete-object "Fairy-Tales|Cinderella" :character)
```

name:add-attribute *object-name object-class attribute-name attribute-value* Function
 &key :namespace :group :key :local :chase-aliases

Adds an attribute, *attribute-name*, having *attribute-value* to the object identified by *object-name* and *object-class*.

NOTE: Adding an attribute with the same name as one that already exists overwrites the old attribute and its value.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:group — If you specify **:group**, a group attribute is created.

:key — A group attribute can have an associated key function that is used to access the member part and test for equality during merging. You can specify the key function as (*attribute-name* :group *key-function*) or by using **:key** *key-function*. Also, you can specify the value **:none** for *function* to remove a key function that you previously specified for this attribute.

:local — If you specify **:local** as non-**nil**, only the locally cached copy is modified.

:chase-aliases — If you specify **:chase-aliases** as non-**nil**, the attribute is attached to the terminal object to which a chain of aliases refers. Otherwise, the attribute is attached to the object identified by *object-name* and *object-class*.

Examples: (name:add-attribute "Bluebird" :bird
 :favorite-food "Sunflower Seeds"
 :namespace "Birds")

The following shows an example of using a key function:

```
(name:add-attribute "Bluebird" :bird `(:meals :group car)
                                     `(("Sunflower Seeds" 400)
                                       ("Cups of Water" 3))
                                     :namespace "Birds")
```

If a group member is added to the **:meals** attribute that matches the **car** of one of the members, the new group attribute and its value replace the old group member and its value. (Refer to the **name:add-group-member** function for information on adding group members without disturbing the rest of the members.)

name:delete-attribute *object-name object-class attribute-name* Function
 &key :namespace :local :chase-aliases

Deletes the attribute, *attribute-name*, from the object identified by *object-name* and *object-class*.

:namespace — Unless you specify **:namespace**, the name qualification of the *object-name* or the first namespace on the search list is used.

:local — If you specify **:local** as non-**nil**, only the locally cached copy is modified.

:chase-aliases — If you specify **:chase-aliases** as non-**nil**, the object modified is the terminal object to which a chain of aliases is resolved.

Otherwise, the object modified is the object identified by *object-name* and *object-class*.

Example: `(name:delete-attribute "Bluebird" :bird :favorite-food
:namespace "Birds")`

name:add-group-member *object-name object-class group-name member* Function
&key :key :namespace :local :chase-aliases

Adds *member* to the attribute *group-name* of the object identified by *object-name* and *object-class*.

:key — A group attribute can have an associated key function that is used to access the member part and test for equality during merging. You can specify the key function by using *:key function*. Also, you can specify the value *:none* for *function* to remove a key function that you previously specified for this attribute.

:namespace — Unless you specify *:namespace*, the name qualification of *object-name* or the first namespace on the search list is used.

:local — If you specify *:local* as non-nil, only the locally cached copy is modified.

:chase-aliases — If you specify *:chase-aliases* as non-nil, the object modified is the terminal object to which a chain of aliases is resolved. Otherwise, the object modified is the object identified by *object-name* and *object-class*.

Examples: `(name:add-group-member "Home|Cats" :pets :census "Calico")`

The following shows an example of using a key function:

```
(name:add-group-member "Birds|Bluebird" :bird :meals
  ^("Sunflower Seeds" 200) :key ^car)
```

If a group member is added to the *:meals* attribute that matches the *car* of one of the members, the new group attribute and its value replace the old group member and its value. For example, the group member ("Sunflower seeds" 200) replaces an old group member ("Sunflower Seeds" 400).

name:delete-group-member *object-name object-class group-name member* Function
&key :key :namespace :local :chase-aliases

Deletes *member* from the attribute *group-name* of the object identified by *object-name* and *object-class*.

:key — A group attribute can have an associated key function that is used to access the member part and test for equality during member deletion. You can specify the key function as (*attribute-name :group key-function*) or by using *:key key-function*. Also, you can specify the value *:none* for *function* to remove a key function that you previously specified for this attribute.

:namespace — Unless you specify *:namespace*, the name qualification of *object-name* or the first namespace on the search list is used.

:local — If you specify *:local* as non-nil, only the locally cached copy is modified.

:chase-aliases — If you specify *:chase-aliases* as non-nil, the object modified is the terminal object to which a chain of aliases is resolved.

Otherwise, the object modified is the object identified by *object-name* and *object-class*.

Example: (name:delete-group-member "Home|Cats" :pets :census "Calico")

name:add-alias *alias-name object-name object-class* Function
 &key :if-exists :namespace :local :chase-aliases

Adds an object named *alias-name* that is an alias of the object identified by *object-name* and *object-class*. The new object will be of the same class.

:if-exists — If **:if-exists** is **:error**, an error is signaled if an object named *alias-name* already exists. Otherwise, only the **:alias-of** attribute of a previously existing object is updated.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:local — If you specify **:local** as non-nil, only the locally cached copy is modified.

:chase-aliases — If you specify **:chase-aliases** as non-nil, the function first goes to the terminal object if *object-name* is an alias. Otherwise, the function goes to the object identified by *object-name* and *object-class*.

Example: (add-alias "Cinderella" "Fairy-Tales|Ella" :character)

This example adds the alias *cinderella* to the object *Ella* in the **:character** class in the *Fairy-Tales* namespace.

NOTE: Since the alias is an object, you use the **name:delete-object** function to delete an alias.

Retrieval Functions **32.5.3** Objects are structures that contain a name, class, timestamp (time of last modification), deleted flag, hidden property list, and attribute list. The retrieval functions perform both lookup and list operations on objects. A *lookup* function returns an object structure, for example:

```
#<NAME::OBJECT 50577160>
```

A *list* function returns a list representation of the structure, which is useful because you do not need to know anything about the internal representation of namespace objects. For example:

```
("Toms-friends" :mailing-list (:members ("Snowball" "Tiger")))
```

name:lookup-object *object-name object-class* &key :namespace Function
 :chase-aliases :merge-aliases :read-only :local :if-does-not-exist

Returns the namespace object identified by *object-name* and *object-class*.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:chase-aliases — If you specify **:chase-aliases** as non-nil, alias chains are followed, returning the terminal object.

:merge-aliases — If you specify **:merge-aliases** as non-nil, attributes along an alias chain are merged to form a synthesized view.

:read-only — If you specify **:read-only** as non-nil, the real database object (not a copy) is returned. This saves memory if you only want to look at the object.

CAUTION: If you modify the real database object, you are modifying the database in an illegal way.

:local — If you specify **:local** as non-nil, only the locally cached copy is accessed.

:if-does-not-exist — If **:if-does-not-exist** is **:error**, an error is signaled if the namespace object is not found. Otherwise, **nil** is returned when the object does not exist.

This function also returns as a second value, the namespace instance where this object was found.

Example: (name:lookup-object "Ronnie" :person :chase-aliases nil
:merge-aliases nil)

name:list-object *object-name object-class &key :namespace* Function
:chase-aliases :merge-aliases :brief :local :if-does-not-exist

Returns a list representation of the namespace object identified by *object-name* and *object-class*.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:chase-aliases — If you specify **:chase-aliases** as non-nil, alias chains are followed, returning the terminal object.

:merge-aliases — If you specify **:merge-aliases** as non-nil, attributes along an alias chain are merged to form a synthesized view.

:brief — If you specify **:brief** as non-nil, the function returns a list format that includes only the name and the class of the object, for example, ("beagle" :canine).

:local — If you specify **:local** as non-nil, only the locally cached copy is accessed.

:if-does-not-exist — If **:if-does-not-exist** is **:error**, an error is signaled if the namespace object is not found. Otherwise, **nil** is returned when the object does not exist.

This function also returns as a second value the instance where this object was found.

Example: (name:list-object "Ronnie" :person :chase-aliases nil
:merge-aliases nil :brief t)

name:lookup-attribute-value *object-name object-class attribute-name* Function
 &key :namespace :chase-aliases (:merge-aliases t) :local

Returns the value of *attribute-name* for the object identified by *object-name* and *object-class*.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:chase-aliases — If you specify **:chase-aliases** as non-nil, alias chains are followed, returning the terminal object.

:merge-aliases — If you specify **:merge-aliases** as non-nil, attributes along an alias chain are merged to form a synthesized view.

:local — If you specify **:local** as non-nil, only the locally cached copy is accessed.

Example: (name:lookup-attribute-value "Reagan" :person :full-name)

name:lookup-objects-from-properties &key :namespace Function
 :name-pattern :class :attribute-list :first-only :local :test :read-only

Returns a subset of namespace objects meeting the specified criteria. These are returned as a list unless you specify **:first-only** as non-nil, in which case only the first object meeting the test is returned. This function is particularly useful if, for example, you have 300 hosts in a configuration.

:namespace — Unless you specify **:namespace**, the first namespace on the search list is used.

:name-pattern — A *name pattern* can be an unqualified object name or a list of unqualified names. A name that is a string can contain the wildcard characters * and ?. The * character looks for a name with any number of characters in this position. The ? character looks for a name with any single character in this position. If you specify a list of names for the name pattern, the search operation returns all of the objects that match those names and any other criteria (that is, the attribute list and/or test predicate); these names can include wild characters if they are strings.

:class — Limits the class of the objects returned.

:attribute-list — A list of attribute names and values used for restricting the objects. Either attribute names or values can be the wild keyword **:***. If **:*** is specified as a value, the objects that are found have properties of the specified name with any value. If **:*** is specified as a name, the objects that are found have any attribute with the specified value.

:local — If you specify **:local** as non-nil, only the locally cached copy is accessed.

:test — Identifies a function that takes the arguments *object*, *name-pattern*, *class*, and *attribute-list*. If **:test** is a symbol, it is assumed that the symbol is defined at the remote host and that it is applied as a filter at the remote host to limit the set of objects returned. Otherwise, the nonsymbol (for example, a function object or closure) is applied locally after a superset of the objects has been transmitted. The latter is obviously less efficient. The *attribute-list* can actually be anything else you want to pass to your test function. It should return non-nil to include an object in the subset.

:read-only — If you specify **:read-only** as non-nil, the real database object (not a copy) is returned. This saves memory if you only want to look at the object.

CAUTION: If you modify the real database object, you are modifying the database in an illegal way.

Examples: The following example shows one way of using `:name-pattern`. The `*` means any number of wild characters are allowed in this position. The `?` means one wild character is allowed in this position.

```
(name:lookup-objects-from-properties
 :name-pattern "M*.?")
```

`MYCROFT.X` is an example of this type of name pattern, while `HYCROFT.X` and `MYCROFT.XX` are not. The following example shows another way of using `:name-pattern`. This time it is a list of names. Notice that you can use wild characters.

```
(name:lookup-objects-from-properties
 :name-pattern ("Tom" "Bill" "Lis*")
 :class :person
 :attribute-list `(:sex :male
                  :age :*
                  :* "Dinosaurs"
                  :members "Tom")
 :test 'my-function)
```

This attribute list restricts the objects found to those containing all of the following criteria: a `:sex` attribute with a value of `:male`, an `:age` attribute with any value (since the wild keyword `:*` is used), any attribute with the value `"Dinosaurs"`, and a `:members` attribute with the value `"Tom"`. If the `:members` attribute is a group attribute, the member `"Tom"` is looked for in the attribute value.

You cannot specify something such as `:age < 30` in the attribute list because only comparison by equality is done. You can specify a test function, though. When `:age 30` is a member of the attribute list, you can use `30` as a ceiling in your test function. To conclude, you can perform the comparison as you wish in your own test function.

The following example returns all objects of class `:person` with `:age` equal to `30`:

```
(name:lookup-objects-from-properties :class :person
 :attribute-list `(:age 30))
```

The next example returns all objects of class `:person` with `:age` greater than `30`:

```
(name:lookup-objects-from-properties :class :person :attribute-list
 `(:age 30) :test 'user:age-check-function)
```

```
(defun user:age-check-function (object name-pattern class
 attribute-list)
 (let ((age (name:get-attribute-value object :age)))
 (and age (> age (cdr attribute-list)))))
```

name:list-objects-from-properties &key :namespace :name-pattern Function
 :class :attribute-list :first-only :local :brief :test

Returns a formatted subset of namespace objects meeting the specified criteria. These are returned as a list of lists unless you specify **:first-only** as non-nil, in which case only the first object meeting the test is returned.

The keyword arguments are the same as the keywords for the **name:lookup-objects-from-properties** function except for **:brief**. Also, this function does not have a **:read-only** keyword.

:brief — If you specify **:brief** as non-nil, only the names and classes of objects are included in the formatted display, for example, ("beagle" :canine).

name:universal-lookup-objects-from-properties Function
 &key (:namespace-list (name:list-known-namespaces)) :name-pattern
 :class :attribute-list :first-only :local :test :read-only

Returns a subset of objects from **:namespace-list** meeting the specified criteria. These are returned as a list of sublists of the form (*namespace-name*, *object-list*).

The other keyword arguments are the same as the keywords for the **name:lookup-objects-from-properties** function.

name:universal-list-objects-from-properties Function
 &key (:namespace-list (list-known-namespaces))
 :name-pattern :class :attribute-list :first-only :local :brief :test

Returns a subset of formatted objects from **:namespace-list** meeting the specified criteria. These are returned as a list of sublists of the form (*namespace*, *object-list*).

The other keyword arguments are the same as the keywords for the **name:lookup-objects-from-properties** function except for **:brief**. Also, this function does not have a **:read-only** keyword.

:brief — If you specify **:brief** as non-nil, only the names and classes of objects are included in the formatted display, for example, ("beagle" :canine).

name:lookup-object-and-aliases *object-name object-class* Function
 &key :namespace :local (:chase t) (:multi-level t) :read-only

Returns a list of namespace objects, the first of which is the object identified by *object-name* and *object-class* and the remainder of which are objects that are direct aliases of the first that exist *within the same namespace*. If the original object does not exist, nil is returned.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:local — If you specify **:local** as non-nil, only the locally cached copy is accessed.

:chase — If you specify **:chase** as non-nil, the function first goes to the terminal object if *object-name* is an alias.

:multi-level — If you specify **:multi-level** as non-nil, the function checks for multiple levels of aliases (which makes the function slower).

:read-only — If you specify **:read-only** as non-nil, the real database object (not a copy) is returned. This saves memory if you only want to look at the object.

CAUTION: If you modify the real database object, you are modifying the database in an illegal way.

Example: (name:lookup-object-and-aliases "Reagan" :person)

name:list-object-and-aliases *object-name object-class* Function
&key :namespace :local :brief (:chase t) (:multi-level t)

Returns a list of formatted namespace objects, the first of which is the object identified by *object-name* and *object-class* and the remainder of which are objects that are direct aliases of the first that exist *within the same namespace*.

:namespace — Unless you specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:local — If you specify **:local** as non-nil, only the locally cached copy is accessed.

:brief — If you specify **:brief** as non-nil, only the names and classes of objects are included in the formatted display, for example, ("beagle" :canine).

:chase — If you specify **:chase** as non-nil, the function first goes to the terminal object if *object-name* is an alias.

:multi-level — If you specify **:multi-level** as non-nil, the function checks for multiple levels of aliases (which makes the function slower).

Example: (name:list-object-and-aliases "Reagan" :person)

name:namespace-summary **&key :namespace :list :brief :local** Function
:read-only

Lists all the objects in **:namespace**. If you do not specify **:namespace**, the first namespace on the search list is used.

:list — If you specify **:list** as non-nil, the namespace summary is shown in display format.

:brief — If you specify **:brief** as non-nil, only the names and classes of objects are included in the formatted display, for example, ("beagle" :canine).

:local — If you specify **:local** as non-nil, the function includes only items in the local cache.

:read-only — If you specify **:read-only** as non-nil, the real database object (not a copy) is returned. This saves memory if you only want to look at the object.

CAUTION: If you modify the real database object, you are modifying the database in an illegal way.

Example: (name:namespace-summary :namespace "Birds" :list t)

name:namespace-classes &optional *namespace* Function
 Returns a list of all classes in *namespace*. If you do not specify *namespace*, the first namespace on the search list is used.

Example: (name:namespace-classes "Birds")

Object Manipulation Functions **32.5.4** The object manipulation functions allow you to perform operations on namespace object structures that you have already obtained.

name:get-attribute-value *object attribute-name* Function
 Returns the value of an attribute named *attribute-name* on *object*. Returns nil if the attribute is deleted or does not exist.

Example: (name:get-attribute-value
 (name:lookup-object "giraffe" :animals) :herd)

name:get-attribute-list *object* Function
 Returns an association list of attribute names and values from a namespace *object*. If the attribute is a group, its indicator is (*name* :group) and its value is a list of members.

Example: (name:get-attribute-list (name:lookup-object "giraffe" :animals))

name:format-objects *object-or-object-list* &optional *brief* Function
 Generates a list of sublists of the form (*name class attribute-list*) for a namespace *object* or for a list of namespace objects that you specify in *object-list*.

brief — If you specify *brief* as non-nil, only the names and classes of objects are included in the formatted display, for example, ("beagle" :canine).

Example: (name:format-objects (list
 (name:lookup-object "giraffe" :animals)
 (name:lookup-object "zebra" :animals)))

Miscellaneous Functions 32.5.5 The following paragraphs describe the miscellaneous functions.

The `name:put-hidden-property` and `name:get-hidden-property` functions are used for clients wishing to manipulate the property list (not attributes) of an object. This update is always done locally only. The property list is preserved across cache refreshes. Note that this manipulation of the property list is done on the real database object, not a copy as returned by lookup.

name:put-hidden-property *object-name object-class property value* Function
&key :namespace (:chase t)

Attaches a hidden *property* having *value* to the object identified by *object-name* and *object-class*.

:namespace — You can directly specify the namespace with **:namespace**. If you do not specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:chase — If you specify **:chase** as non-nil, the object modified is the object to which a chain of aliases is resolved.

Example: `(name:put-hidden-property "Home|Cats" :pets :crimes
'("Furniture slashing"))`

name:get-hidden-property *object-name object-class property* Function
&key :namespace (:chase t)

Returns the value of a hidden *property* if the *property* is attached to the object identified by *object-name* and *object-class*.

:namespace — You can directly specify the namespace with **:namespace**. If you do not specify **:namespace**, the name qualification of *object-name* or the first namespace on the search list is used.

:chase — If you specify **:chase** as non-nil, alias chains are followed, returning the value of the hidden property from the terminal object.

Example: `(name:get-hidden-property "Home|Cats" :pets :crimes)`

name:namespace-has-cache *namespace* Function

Returns **t** if *namespace* has a local cache.

Example: `(name:namespace-has-cache "Birds")`

name:refresh-cache *namespace &optional class* Function

Marks all objects in *namespace* (or only those of *class*, if specified) as needing refresh on the next lookup. This preserves any hidden properties attached to the object.

Example: `(name:refresh-cache "Birds")`

CAUTION: The `name:refresh-cache` function, when applied to a network namespace, can delete local updates made by the network during boot and may make your system unable to use the network.

name:refresh-cached-object *object-name object-class* &optional *namespace* Function

Marks a particular object, identified by *object-name* and *object-class*, as needing refresh on the next lookup. If you do not specify *namespace*, the name qualification of *object-name* or the first namespace on the search list is used.

name:flush-namespace *namespace* &optional *class* Function

Completely clears the local cache of *namespace* (if one exists). If your namespace does not have a cache, this function clears the namespace. If you specify *class*, only objects of that class are cleared.

Example: (name:flush-cache "Birds")

CAUTION: The name:flush-namespace function, when applied to a network namespace, can delete local updates made by the network during boot and may make your system unable to use the network.

name:foreign-namespace *namespace* Function

Returns the type of namespace if *namespace* is a client of a non-Explorer namespace.

Example: (name:foreign-namespace "Birds")

name:distribute-namespace *namespace-name* Function

&key :local-only (:save-first t) :server-list
(:search-list-loc :beginning) (:notify t)

Terminates the Configure or Convert mode for a new public namespace. These modes are described in the *:explorer-server init-args* of the name:add-namespace function in paragraph 32.5.1, Namespace Functions.

CAUTION: The name:distribute-namespace is mainly intended for internal use.

You can optionally write the namespace to a binary file. You *must* do this if you are in Convert mode because no changes have been written to the log file.

Propagating to servers of subsequent changes is enabled.

A local client instance is created, and all further namespace accesses at this machine now go through the client instance rather than directly to the server instance.

Optionally, you can start other servers, which obtain their information from this host. This works only if those servers are already on the network.

:local-only — When non-nil, this keyword distributes only to this host. The default is nil.

- :save-first** — When non-nil (the default), this keyword writes the namespace to an xld file first.
- :server-list** — Specifies a list of servers (host names) to receive distribution. The default is all servers.
- :search-list-loc** — Specifies where to put the namespace in the search list: **:beginning**, **:end**, or **nil** (that is, do not put it on the search list).
- :notify** — When non-nil (the default), this keyword notifies you if some servers did not answer.

name:force-local-server-boot *namespace-name* Function
 &key (**:search-list-loc** **:beginning**)
:standalone **:permanent** **:new**

Forces a server to be booted from local namespace files even if it is not on the server boot list. This applies only to public namespaces.

- :search-list-loc** — Specifies where to put the namespace in the search list: **:beginning**, **:end**, or **nil** (that is, do not put it on the search list).
- :standalone** — If you specify **:standalone** as non-nil, no communication with other servers will be done.
- :permanent** — If you specify **:permanent** as non-nil, this server is added to the server boot list.
- :new** — Either non-nil or nil (nil is the default). You should use **:new t** if no files need to be loaded because this is a new namespace.

name:initialize-name-service &optional *namespace* (*display t*) Function

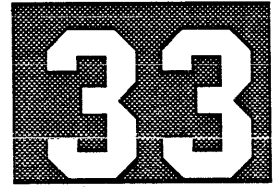
Performs all name service (and network) initializations just like those done during a cold boot (destroying any local namespace updates). If you specify *namespace*, this function broadcasts for a server of that *namespace* (but does not override a qualified pack host name for this machine). If *display* is non-nil (the default), the resulting namespace configuration is shown on ***standard-output***.

Error Messages

32.6 The following errors are signaled when a namespace access fails. They all share the common base condition **'name:rpc-error**.

- 'NAME:NO-LOCAL-HANDLER** — No local server is available for the namespace, and no other servers are known.
- 'NAME:LOCAL-CONNECTION-REJECTED** — The local server for the namespace is not currently accepting requests, and no other servers are known or could be contacted.
- 'NAME:REMOTE-CONNECTION-REJECTED** — The last server tried is not accepting requests, and no other servers are known or could be contacted.
- 'NAME:REMOTE-CONNECTION-UNSUCCESSFUL** — No servers could be contacted.

MISCELLANEOUS NETWORK FUNCTIONS



Introduction

33.1 The miscellaneous network functions allow you to perform the following operations:

- Print information on the status of hosts
- Reset the network
- Eval serve (that is, set up a read-eval-print loop on a remote host)
- Finger hosts (that is, display information about users logged in at various machines in your network)
- Send and print notifications

For more details on the operations discussed in this section, refer to the *Explorer Networking Reference*.

Host Status

33.2 To check on the status of hosts, you use the `net:host-status` function:

`net:host-status &rest hosts`

Function

If *hosts* is nil, this function polls all hosts known by the locally-cached version of the network namespace. Otherwise, the function polls the specified *hosts*. A status message is returned for each medium supported by each host. (For related information, see the description of the `net:*poll-each-status-p*` variable later in this numbered paragraph.)

The following example shows a `net:host-status` display:

ADDR	HOST	STATUS
1464	"Brigham-Young"	is responding on medium CHAOS.
1107327720	"Brigham-Young"	is responding on medium IP.

The following list explains the meaning of the headers:

ADDR	The address of the listed host
HOST	The name of the host
STATUS	The status (responding or not responding) of the host and the medium in which the particular protocol has been implemented

net:*poll-each-status-p*

Variable

When this variable is **nil**, the **net:host-status** function returns only the status of the most desirable protocol of each host. With less information to check, the **net:host-status** function operates more quickly. If **net:*poll-each-status-p*** is non-**nil** (**t** is the default), the **net:host-status** function operates as previously described, returning the status for each protocol on the target host.

Resetting the Network

33.3 When proper network transmission and reception breaks down, the first corrective action you should take is to reset the network. To do so, use the following function:

net:reset &optional *enable-p*

Function

Resets the network for all protocols currently loaded. If the value of *enable-p* is **nil** (the default), the function turns off the network for this host. If *enable-p* is not **nil**, the network is enabled and turned on.

Eval Serving

33.4 The Eval server allows you to set up a read-eval-print loop on a remote host. Before you can use the Eval server, it must be enabled on the remote host. Use the following functions to turn on the Eval server at the host site.

chaos:eval-server-on *mode*

Function

This function enables and disables the Eval server. *mode* can take on the following values: **t** for on, **nil** for off, **:notify** for on, with the condition that the user is notified when a connection is made, or **:not-logged-in** for on if no one is logged in. The default is **nil**.

After you enable the server at the remote host, start the Eval session by issuing the **remote-eval** function at your host.

chaos:remote-eval *host*

Function

This function initiates an Eval server session to *host*, where the read-eval-print loop is handled. Results are returned to your terminal until you terminate the session.

Each time a complete symbolic expression arrives, the Eval server reads it, evaluates it, and sends back the result. Terminate the Eval session by pressing **ABORT**.

Fingering Hosts

33.5 The `finger` function and the TERM F key sequence display information about users logged in at various machines in your network. The `chaos:find-hosts-or-lispms-logged-in-as-user` function returns a list of hosts on which a user is logged in.

The `finger` function and the TERM F key sequence display the following information about a user logged in at a machine in your network:

- Login name
- Full name
- Process now running
- Location

`finger &optional spec (stream standard-output) hack-brackets-p` Function

Prints brief information about a user as specified by *spec*. The *spec* argument can be *user@host* or *@host*. The *stream* argument specifies where to print the information. If *hack-brackets-p* is *t*, the first line shows what host you are fingering.

If you enter the `finger` function with no arguments, your own machine is fingered.

You can also obtain finger information by pressing TERM 0 F. You are prompted for *user@host* or *@host*.

Pressing TERM F displays information about all users logged in at the various machines in your network.

Both the `finger` function and the TERM F key sequences are interfaces to the Finger protocol. The Finger protocol is a Lisp Machine version of the ARPANET Name protocol that uses a simple transaction instead of a stream connection.

During network configuration, finger digits can be assigned that specify which machines to finger. For example, pressing TERM 3 F might display information about the users logged on the machines KING-ARTHUR, GUINEVERE, and LANCELOT. Refer to the *Explorer Networking Reference* for more information on this.

Example: (finger "@young")

The following is displayed:

```
LISA -                ZMACS                TI-AUSTIN
```

`chaos:find-hosts-or-lispms-logged-in-as-user user` Function
&optional *hosts no-lispms-p*

Returns a list of hosts on which *user* is logged in. The *hosts* argument is the list of hosts to check (in addition to all Lisp machines). If *no-lispms-p* is *t*, the function does not return any Lisp machines.

Sending and Printing Notifications

33.6 The `chaos:shout`, `chaos:notify-all-lms`, and `chaos:notify` functions send notifications to Lisp machines. The `print-notifications` function reprints any notifications that have been received.

In addition to the functions that send notifications, a notification can come from utilities such as Mail and Converse. Also, a notification is not restricted to the network. It can be an asynchronous message from the Explorer system itself.

- chaos:shout** Function
Sends a message to all Lisp machines. The message is read from the terminal. The message should be brief; otherwise, you should use mail.
- chaos:notify-all-lms** &optional (*message* (`notify-get-message`)) Function
Sends a *message* to all Lisp machines. The message is printed as a notification. If you omit *message*, the message is read from the terminal. The message should be brief; otherwise, you should use mail.
- chaos:notify** *host* &optional (*message* (`notify-get-message`)) Function
Sends a *message* to *host*. The message is printed as a notification. If you omit *message*, the message is read from the terminal. The message should be brief; otherwise, you should use mail.
- print-notifications** Function
Reprints any notifications that have been received. The difference between a notification and a send is that a send comes from other users, while a notification is usually an asynchronous message from the Explorer system itself. However, the default way for the system to inform you about a send is to make a notification. So `print-notifications` *normally* includes all sends as well as notifications.

COLOR MAP EDITOR

Introduction

34.1 The Color Map editor allows you to define and edit software *color maps*. With the present implementation of color hardware, you can choose from over 16 million different colors, 256 of which can be displayed at any one time. These 256 colors are associated with a particular color map. You can create and save as many color maps as desired. Only one of these color maps can be loaded in the hardware at any time, thus determining the 256 colors that can be displayed.

Each window can have its own color map. When multiple windows are displayed on a screen, the color map in effect is that of the selected window, or if no window is selected, the window pointed to by the global variable `w:default-screen`. When you invoke the Color Map editor, the initial color map is that of the window under the mouse.

NOTE: The first 32 slots in a color map are reserved for system software usage, such as the color of the background, foreground, borders, labels, blinkers, and so on. If you attempt to edit one of these reserved colors, you are notified by a display of either the name corresponding to the color or by the phrase `system color` if a name is not assigned to the color.

If you only want to change the color of an object, you do not need to edit the color by using the Color Map editor. For example, suppose that a window has a default color for text of yellow on a black background, and you want green text on a pink background. You should use the Edit Attributes item from the System menu. (Refer to the Customizing Your Environment section in the *Introduction to the Explorer System*.) You can change the foreground and background colors of the window, the label color, the label background color, and the border color.

Do not use the Color Map editor to edit the color yellow to be green and the color black to be pink. If you did, anything drawn in yellow would be green, and anything drawn in black would be pink. Editing the color changes the value of the logical color, while editing the attribute changes the attribute to a different logical color.

This section discusses the following topics:

- Loading the Color Map editor software
- Invoking the Color Map editor
- Editing and defining colors
- Color editor commands, which affect the general operation of the Color Map editor

- Color map commands, which deal specifically with the color map
- Command summary, which provides a list of the Color Map editor commands

For more information on using color, refer to the following documents:

- *Explorer Programming Concepts*, which provides a detailed overview on color concepts
- *Explorer Window System Reference*
- *Fundamentals of Interactive Computer Graphics* by J. D. Foley/A. Van Dam
- *Computer Graphics* by Donald Hearn/M. Pauline Baker
- *Explorer Color System Interface Board General Description*

Loading the Color Map Editor Software

34.2 To load the Color Map editor software, enter the following form:

```
(make-system 'color-map-editor :noconfirm)
```

This form lists the files that are being loaded. Once the files are loaded, you can invoke the Color Map editor.

Note that you can avoid having to create the Color Map editor each session by saving a band that includes the Color Map editor. Refer to the *Explorer Input/Output Reference* for information about saving a band.

Invoking the Color Map Editor

34.3 To invoke the Color Map editor, choose one of the following methods:

- Press TERM D.
- Select the Color Map Editor item from the System menu.
- Enter the `color:cme` function in a Lisp Listener.

The initial (default) color to edit is the color value of the pixel at which the mouse is pointing. The color map of the window under the mouse is the map being edited.

`color:cme` &optional *color-to-edit color-map* Function

Invokes the Color Map editor. Optionally, you can specify the color that you initially want to edit and the color map.

color-to-edit — The value or name of a color in the color map. The value can range from 0 to 255; an example of a color name is `w:green`. The system-defined color names are listed in the *Explorer Window System Reference*. The default color to edit is the color value of the pixel at which the mouse is pointing.

color-map — The name of the color map to display. The default is the color map of the window under the mouse. When you save a color map to a file by using the Save Color Map command, the name of the file becomes a global symbol that refers to the color map.

Examples: (color:cme 239 rainbow-map)
(color:cme 239 (send w:selected-window :color-map))

To exit the Color Map editor, click left outside of the Color Map editor window or press the END key.

Editing and Defining Colors

34.4 To select a color for editing, choose one of the following methods:

- Display the entire color map by selecting the Display Color Map command, and use the mouse to select the color to edit. Refer to Figure 34-1.
- Invoke the Screen Object command, and select some object (that is, text, window border/label) from the screen.
- Click left on the color number, and enter a color name or number in the prompt window.

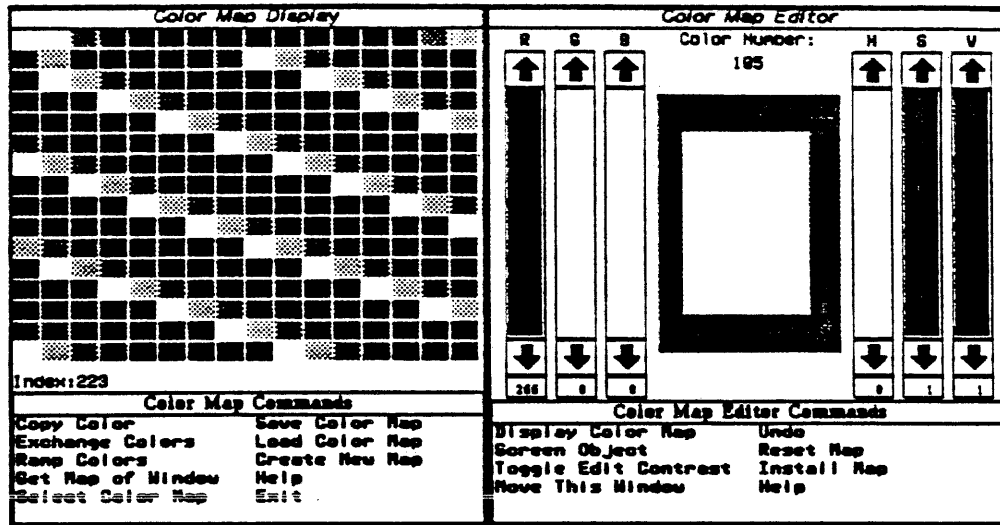
The *color box* displays the color currently being edited. A label appears above the color box designating the color's index into the color map. If the index is between 0 and 31 (inclusive), either the color name (if there is one) or the message `system color` also appears.

The color box is surrounded by another box, the *contrast box*, which helps you compare or contrast one color with another. You can edit the color of the contrast box by selecting the Toggle Edit Contrast command. This command toggles the Color Map editor mode between editing the color of the color box and editing the color of the contrast box.

The Color Map editor allows you to edit or define a particular color by specifying parameters in either the RGB or the HSV model. In the RGB model, the three parameters are red, green, and blue; in the HSV model, they are hue, saturation, and value. You can use both of these models when defining a color. The resultant color is a combination of all the parameters.

The RGB model for representing color is based on the three color photo-receptors in the human eye, which detect either red, green, or blue. The RGB model is *not* based on the three primary colors red, blue, and yellow, which many people might expect. The HSV model describes the color in terms of color (hue), purity (saturation), and brightness (value).

Figure 34-1 Color Map Editor Display



The RGB model is often used for adding a color to the color you are editing, such as adding more green or more red by moving the R or G sliders up. The HSV model is often used for modifying a color, such as making purple brighter by moving the V slider up. When the sliders for saturation and value are at the top, a color such as red is considered pure. This concept is somewhat analogous to the pure colors an artist combines to make colors. Moving the saturation slider down (that is, decreasing the saturation) is comparable to adding white to a color. Moving the value slider down is comparable to adding black to a color.

For more information regarding the color RGB and HSV models, refer to the *Fundamentals of Interactive Computer Graphics* by J. D. Foley/A. Van Dam.

Each slide bar for the RGB and HSV models consists of the following four panes:

- Up arrow pane
- Vertical, rectangular slider pane
- Down arrow pane
- Numeric readout pane

A filled rectangle in the slider pane extends from the bottom up to the current value. The color box always displays the color defined by the levels of the slide bars. The following paragraphs describe each pane in detail.

Slider Pane When you hold the left button down and move the mouse cursor up or down in a slider region, the slider moves along with the mouse cursor. A single left mouse click in the slider pane adjusts the value to the level of the mouse cursor.

Up/Down Arrow Panes When the mouse cursor is on or near the up/down arrows, a single click adjusts the value by one, and the slider is repositioned accordingly. Holding the left button continuously changes the color and adjusts the slider accordingly.

Numeric Readout Pane This pane displays the numeric value associated with the level of the slider. The range of values is dependent upon the associated color model. Any mouse click on the numeric readout pane displays a window that accepts input from the keyboard; the slider is repositioned to reflect the new value.

Color Editor Commands

34.5 The Color Editor Commands menu provides the following commands that affect the general operation of the Color Map editor:

Display Color Map Command

Keystroke: CTRL-D

Displays the entire color map of the current window in a window called the Color Map Display. The current window is the window under the mouse. Each color is displayed in a small, mouse-selectable rectangle. To edit a color, select it with the mouse by clicking on it. When you move the mouse cursor over the colors in the displayed color map, the index number associated with each particular color is displayed.

Screen Object Command

Keystroke: CTRL-S

Allows you to click left on some color on the screen and edit that color. Note that any other objects displayed on the screen that use this color are also affected.

Toggle Edit Contrast Command

Keystroke: CTRL-T

Toggles the Color Map editor mode between editing the color of the color box and editing the color of the contrast box.

Help Command

Keystroke: HELP

Displays documentation on how to use the Color Map editor. Each command is described.

Undo Command

Keystroke: UNDO

Aborts all changes made to the color currently being edited since the Color Map editor was most recently selected.

Move This Window Command

Keystroke: CTRL-M

Allows you to reposition the Color Map editor window by moving it with the mouse.

Reset Map Command

Keystroke: CTRL-R

Undoes all the changes that you made to the current map of the Color Map editor.

Install Map Command

Keystroke: CTRL-I

Allows you to click left on a window to make its color map the current color map of the Color Map editor.

Color Map Commands

34.6 The Color Map Display, which is activated by the Display Color Map command, provides the following commands:

Load Color Map Command

Keystroke: CTRL-L

Loads a color map previously saved to a file. This command displays a window that contains a menu of saved color maps. Initially, the command attempts to find the color maps residing in the COLOR-MAPS directory on your local machine. You can change the default setting of the pathname by selecting the <Enter Pathname> option. This option allows you to enter a pathname for the desired color map file. The specified color map is downloaded to the hardware, and the screen colors change accordingly.

Save Color Map Command

Keystroke: CTRL-S

Saves the current color map to a file. You are prompted for the pathname in a pop-up window. The name component of the filename designated for the color map is made global so that the color map can be referred to by this symbol name even after you exit the Color Map editor. The variable `w:*color-maps*` lists the symbol names of defined color maps.

Ramp Colors Command

Keystroke: CTRL-R

Executes a linear ramp of the RGB values between selected indices. You are prompted to select the starting and the ending indices.

Copy Color Command

Keystroke: CTRL-C

Copies a single color from one location in the color map to another. To copy a color, you are prompted to click left on the source color and then click left on the destination color.

Exchange Colors Command

Keystroke: CTRL-E

Exchanges the locations of two colors. You are prompted to click left on two colors to exchange them.

Select Color Map Command

Keystroke: CTRL-M

Displays a menu from which you can select one of the color maps in the environment. The system default color map is listed in the menu. The selected color map is invoked, and the screen colors change accordingly.

Create New Map Command

Keystroke: CTRL-N

Creates a new color map that is a copy of the original system default color map. This command is useful when you have edited the current color map in such a manner that it is no longer useful.

Get Map of Window Command

Keystroke: CTRL-W

Displays the color map of a window that you select with the mouse. You can use this command to copy the color map of one window to another.

Command Summary 34.7 Table 34-1 lists the Color Map Editor commands.

Table 34-1

Color Map Editor Commands

Command Name	Keystroke
--------------	-----------

Color Editor Commands:

Display Color Map	CTRL-D
Screen Object	CTRL-S
Toggle Edit Contrast	CTRL-T
Help	HELP
Undo	UNDO
Move This Window	CTRL-M
Reset Map	CTRL-R
Install Map	CTRL-I

Color Map Commands:

Load Color Map	CTRL-L
Save Color Map	CTRL-S
Ramp Colors	CTRL-R
Copy Color	CTRL-C
Exchange Colors	CTRL-E
Select Color Map	CTRL-M
Create New Map	CTRL-N
Get Map of Window	CTRL-W

Introduction

35.1 The Visual Interactive Documentation (Visidoc) Online Manual Viewer allows you to view documentation from online reference manuals. Visidoc retrieves online reference material and places it in a buffer where the text and graphics information can be viewed. You can also perform operations such as evaluating examples and copying text. Among the manuals from which you can view documentation are the following:

- *Explorer Window System Reference*
- *Explorer Lisp Reference*
- *Explorer Tools and Utilities*
- *Explorer Input/Output Reference*
- *Explorer Zmacs Reference*
- *Explorer Networking Reference*

This section discusses the following topics:

- **Installing the Visidoc Client Software** — Tells how to install the Visidoc client software (if it is not already installed). This software is necessary before you can invoke Visidoc.
- **Invoking Visidoc** — Tells how to invoke Visidoc.
- **Exiting Visidoc** — Tells how to exit Visidoc.
- **Features of Visidoc** — Describes the major features of Visidoc. Because Visidoc is easily used online, this section does not describe in great detail how to use it.
- **Making a Host a Visidoc Server** — Tells the system administrator how to set up a Visidoc server. (These instructions are duplicated in the *Explorer System Software Installation*.)
- **Maintenance of the Visidoc server namespace** — Tells how to modify the Visidoc server's namespace information. Normally, you do not need to do this.

**Installing the
Visidoc Client
Software**

35.2 Before you invoke Visidoc, execute the `print-herald` function to see if Visidoc is loaded. If not, enter the following form to install the client software for Visidoc:

```
(make-system 'visidoc :noconfirm)
```

If you need more information on `make-system`, refer to the *Explorer Lisp Reference*.

NOTE: If an error message appears when you try to invoke Visidoc that states there is no Visidoc server, the system administrator needs to set up the Visidoc server as described in paragraph 35.6, Making a Host a Visidoc Server.

Invoking Visidoc

35.3 To access Visidoc, you can use one of the following methods:

- Select the item Visidoc from the System menu. A menu appears from which you can select the manual you want to view. Upon selection, the Visidoc screen and the Table of Contents for the manual appears.
- Press META-CTRL-SHIFT-D or type META-X Visidoc in Zmacs. A prompt appears in the minibuffer asking you what to document. Specify a word or phrase of interest in the minibuffer, or select the word or phrase with the mouse.

Exiting Visidoc

35.4 To exit Visidoc, you can use one of the following methods:

- Press the END key.
- Select the Exit command from the Visidoc commands window.

Exiting Visidoc returns you to the previous buffer.

Features of Visidoc 35.5 The features of Visidoc are briefly discussed in the paragraphs that follow.

Memory Characteristics 35.5.1 The documentation for the Visidoc manuals is stored in files on disk. Indexing information is stored in server machines, not in each machine. This reduces the size of the load band by megabytes for client machines.

Display Characteristics 35.5.2 The following list describes the display characteristics of Visidoc:

- The text in the main Visidoc viewing window appears the same as in the manuals.
- Graphics drawings are included within text. However, screen dumps, such as the figures in the Peek section of the *Explorer Tools and Utilities*, are not included. Most screen dumps in the manuals attempt to duplicate real-time conditions. You can reproduce one of the Peek screen dumps by selecting Peek online.
- A window of popular Visidoc commands is available. When Suggestions menus are on, they provide a Visidoc commands menu in place of this window.
- A history list of the items you have requested is available.
- The page number of the item you select is displayed in the minibuffer each time you request the item.

Using Visidoc 35.5.3 The following list describes many of the user interface features of Visidoc:

- Pressing HELP M while in Visidoc displays documentation on using Visidoc, which includes a list of keystrokes for several commands.
- A Visidoc commands menu can be invoked by clicking right in the main Visidoc viewing screen. This menu provides several commands that are not in the Visidoc commands window.
- A list of the manuals that you can view is available.
- The Table of Contents for each manual can be viewed.
- Cross-references can be selected with the mouse. That is, if you see a word in boldface, you can select it with the mouse and the documentation for the item appears. If there are several different topics relating to the documentation, a menu listing the various topics appears.

NOTE: Sometimes no documentation exists for an item that is mouse-sensitive.

- Complex searches (using apropos capabilities) for topics can be performed.
- The history window lists items that you previously selected for apropos searches. You can select these items from the history window.
- Visidoc builds the manual you are viewing in *Hypertext* style. Hypertext refers to the "read in any order" capability. That is, you do not have to start a section at the beginning. For example, if the `w:menu` function is described in the middle of a section, you can go immediately to it. Also, the manual keeps itself in the correct order.
- Navigation through a manual is available by selecting the commands that find the next and previous topics.
- The surrounding topic can be read for context information. For example, suppose you are viewing the login function in the *Explorer Tools and Utilities*. When you request to view the surrounding topic, the introduction that precedes the login function in Section 3, Login Initialization File, appears.
- Visidoc is built in as part of the Zmacs editor, which means you can use the standard Zmacs capabilities. Some of these capabilities are as follows:
 - The Visidoc buffer for the manual you are viewing can be used like any other Zmacs buffers. For example, the List Buffers command (`CTRL-X CTRL-B`) displays the Visidoc buffer with the rest of your buffers and allows you to select it.
 - Examples can be evaluated by marking the code and evaluating the marked region.
 - Text or code can be marked or yanked.
 - The Visidoc buffer can be scrolled like any other Zmacs buffer.
 - Search capabilities, such as Incremental Search (`CTRL-S`) and Reverse Incremental Search (`CTRL-R`), are available.

Refer to the *Explorer Zmacs Editor Reference* if you need information on Zmacs.

- Each manual that you view is in its own Zmacs buffer.
- The page number for an item can be displayed in the minibuffer by selecting the page number command. The page number is initially displayed when you request the item; however, other operations in the buffer may cause it to be erased from the minibuffer.
- Previously requested items are viewable by invoking the Visidoc utility again, by selecting the Visidoc buffer through standard Zmacs procedures, or by selecting the item from the history menu.
- The history list of the items you requested can be saved to a file. You can also restore them from the file.
- Once you have seen the text for a topic, you can get it out of the way by positioning your cursor on the topic and contracting it into a single line that represents the presence of undisplayed text.

**Making a Host a
Visidoc Server**

35.6 The system administrator needs to make at least one host a Visidoc server. Perform the following steps:

1. Make sure your file system has 9.5 megabytes of free space for the index information and for the basic set of manual data files.
2. Create a top-level directory called NAME-SERVICE if the host does not already have one. Use the META-X Create Directory command in Zmacs to do this. (Refer to the *Explorer Zmacs Editor Reference* if you need information on creating directories).
3. Prepare the Visidoc Manuals Files tape (TI part number 2549303-0001) for installation. (This tape is included with Explorer Release 3.2.) Perform the following steps:
 - a. Ensure that the tape is write-protected.
 - b. Put the tape into your tape drive and select the Backup System window by pressing SYSTEM B.
 - c. Click on the Prepare Tape command, and then select 1/4 inch cartridge Tape.
 - d. After the Prepare Tape command completes, click on the Load Tape command. The tape is now positioned at the start of the VISIDOC-SERVER.XLD file.
 - e. Turn off MORE processing by pressing TERM 0 M.
4. Restore the file VISIDOC-SERVER.XLD to the NAME-SERVICE directory by using the Restore Directory command. Answer the prompts as follows:

NOTE: Do not change the name of the VISIDOC-SERVER.XLD file. This file contains the namespace index information that the server needs to locate names, files, and other information when a client asks for Visidoc information.

Prompt	Value
Source Pathname	LM: NAME-SERVICE; VISIDOC-SERVER.XLD
Destination pathname	LM: NAME-SERVICE; VISIDOC-SERVER.XLD
Use host name from tape?	No

This step takes about 2 to 5 minutes to restore.

5. Next, restore the MANUALS directory from the tape. These are the data files that the Visidoc viewer will access to display. You can restore these manual data files to some directory other than the one specified. Select the Restore Directory command and answer the prompts as follows:

Prompt	Value
Source Pathname	LM:MANUALS;*. *
Destination pathname	LM:MANUALS;*. *
Use host name from tape?	No
Query?	No
Create directories automatically?	Yes

This step takes about 30 minutes to restore.

6. Rewind the tape.
7. Use the Verify File command to verify the VISIDOC-SERVER.XLD file, which you restored from tape. Answer the prompts as follows:

Prompt	Value
Source Pathname	LM: NAME-SERVICE; VISIDOC-SERVER.XLD
Destination pathname	LM: NAME-SERVICE; VISIDOC-SERVER.XLD
Use host name from tape?	No

This step takes about 2 to 5 minutes to verify.

8. Use the Verify Directory command to verify the MANUALS directory, which you restored from tape. Answer the prompts as follows:

Prompt	Value
Source Pathname	LM:MANUALS;*. *
Destination pathname	LM:MANUALS;*. *
Use host name from tape?	No

This step takes about 20 to 25 minutes.

9. Rewind the tape.
10. Make a system on the Visidoc server as follows:

```
(make-system 'visidoc-server :noconfirm)
```

If you need more informaton on make-system, refer to the *Explorer Lisp Reference*.

This step takes about 30 seconds.

11. Enter the following form to initialize the Visidoc server:

```
(dox:initialize-visidoc-server)
```

The VISIDOC-SERVER namespace file is loaded at this time. This is a large file that takes about 8 minutes to load.

12. After the namespace file is loaded, you are prompted for the directories that contain the manual data files. You can use the defaults, provided that you have restored the MANUALS directory and subdirectories from tape to the same directory names on the host. Otherwise, provide the names of any new directories.

If you change the names of the directories, the Visidoc server namespace, along with the new directory names, are saved to disk. This save occurs in the background and takes about 8 minutes. The save also requires that you have 1,250K bytes more space on your file system. You can then delete and expunge the old version of the file. If you did not change the names of the directories, the namespace is not saved.

NOTE: It is strongly recommended that the Visidoc server host also be the file server host for the manual data files. This decreases network activity and namespace access delays required to contact multiple hosts. However, you may specify the logical pathname of any known host at the time you are prompted for the directory names that contain the manual data files. (Be sure that the manual data files from the directories on tape reside on the specified host and directories.)

13. After the initialization process is complete and the new set of Visidoc server index information has been saved, save the environment to disk by entering the form (disk-save "LODn"). If you do not perform a disk-save, you will need to perform the make-system of the Visidoc server and execute the dox:boot-visidoc-server function after every cold boot.

Maintenance of the Visidoc Server Namespace

35.7 In most cases, you do not need to do anything to the Visidoc server's namespace of index information. However, should you need to make changes subsequent to installation, the following paragraphs describe the structure of the top-level index information that you might need to change. To make these changes, edit the `visidoc-server` namespace using the normal namespace editing techniques described in Section 32, Namespace Utilities.

Objects in the :namespace Class

35.7.1 The keyword objects (such as `:io`) under the `:namespace` class in the `visidoc-server` namespace contain the `:files-info` attribute, which you can modify if you want to change directory and host information. The `:files-info` attribute is an association list that identifies the host, directory, and file extension of the pathname components, as follows:

- `:host` component — A string that identifies the host where the data files for the particular manual are kept.
- `:directory` component — A list of strings in which the first string is the top-level directory, the second string is the first sublevel directory, and so on. (This list is accepted as an argument to the `:directory` keyword in the `make-pathname` function.)
- `:extension` component — A string that identifies the file as a Visidoc file. The extension can actually be any file type, but it must be the same file type for all data files of any one manual.

The `"VISIDOC-SERVER"` object in the `:namespace` class holds information specific to this namespace instance. The attribute of interest in this object is the `:release-version`. This attribute identifies the version of manual data files with which this set of index information belongs. The attribute's value should be a keyword with a structure similar to the following:

```
:REL-<major version>-<minor version>
```

In the following example, this namespace contains index information for manuals that document Explorer Release 3.2:

```
:RELEASE-VERSION :REL-3-2
```

Saving Your Changes

35.7.2 To save any changes you make to your Visidoc server's namespace, you need to save the changes to your namespace as you normally do, and you also need to save the namespace back to disk. Note that this is very important. To save the namespace to disk, execute the following function:

```
(dox:save-visidoc-server)
```

Maintenance of the Network Namespace :site Option

35.7.3 At some time, you may not want a Visidoc server host to be a server for a particular version of online documentation, or you may not want the host to be an available server at all. The following information helps you identify what information to modify in order to remove the host from the appropriate list(s).

The `:visidoc-servers` attribute on the network namespace object in the `:site` class of your network namespace contains an association list of release versions and hosts that are Visidoc servers for that version of manual documentation. The attribute's value should be a keyword with a structure similar to the following:

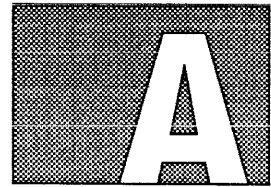
```
:REL-<major version>-<minor version>
```

The rest of the association list is a list of host names. For example, suppose the attribute line for your network object in the `:site` class is the following:

```
:VISIDOC-SERVERS ((:REL-3-2 ("foo" "bar")) (:REL-4-0 ("baz" "foo")))
```

The hosts `foo` and `bar` are Visidoc servers for manual data files for Explorer Release 3.2, and the hosts `baz` and `foo` are hosts for Explorer Release 4.0. Notice that the host `foo` is a server for multiple versions of manual data. You should use the appropriate namespace techniques to modify and globally save the new attribute value.

EXPLORER FONTS



This appendix shows the fonts that are automatically loaded with the Explorer system. The fonts are listed in alphabetical order, as follows:

43vxms	higher-medfnb	metsb	tr12bi
5x5	higher-tr8	metsbi	tr12i
bigfnt	hl10	metsi	tr18
bottom	hl10b	mouse	tr18b
cmb8	hl12	search	tr8
cmdunh	hl12b	ti-logo	tr8b
cmold	hl12bi	tiny	tr8i
cmr10	hl12i	top	tvfont
cmr18	hl6	tr10	vt-graphics
cmr5	hl7	tr10b	vt-graphics-bottom
courier	icons	tr10bi	vt-graphics-top
cptfont	medfnt	tr10i	vt-graphics-wider-font
cptfontb	medfntb	tr12	wider-font
cptfontbi	mets	tr12b	wider-medfnt
cptfonti			

Each listing consists of a line of `cptfont` and then a line of the actual font. This allows you to see quickly what character a particular glyph is associated with. The numbers along the left are the decimal character codes of the first character in the line.

In some cases, if a font has several characters that are not defined, those lines are not shown. All fonts have non-displayable characters between character codes 128 and 160, inclusive. These characters cannot be redefined.

Most fonts include the complete ISO character set. The exceptions are as follows:

- Fonts that are not used to display text (specifically, `icons`, `mouse`, `search`, and `ti-logo`).
- Fonts that are copies of raster fonts available in the graphics editor (specifically, `bottom`, `cmb8`, `cmdunh`, `cmold`, `cmr10`, `cmr18`, `cmr5`, and `top`).
- Fonts that are used on the VT100 emulator, which does not support ISO characters (specifically `vt-graphics`, `vt-graphics-bottom`, `vt-graphics-top`, and `vt-graphics-wider-font`).
- Very complex fonts, such as `43vxms`.

40	43VXMS							
	{	}	*	+	,	-	.	/
				x	-	x		
48	0	1	2	3	4	5	6	7
	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
	8	9				=		?
64	A	B	C	D	E	F	G	
	A	B	C	D	E	F	G	
72	H	I	J	K	L	M	N	O
	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
	P	Q	R	S	T	U	V	W
88	X	Y	Z	[\]	^	
	X	Y	Z					
96	a	b	c	d	e	f	g	
	a	b	c	d	e	f	g	
104	h	i	j	k	l	m	n	o
	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	/
	x	y	z		 			

0	5X5																														
	!	@	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
32	!	@	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	/
128	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	/
160	!	@	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
192	A	A	A	A	A	A	R	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
224	A	A	A	A	A	A	R	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E

BIGFNT																																
0	·	↓	α	β	^	~	ε	π	λ	δ	δ	↑	±	⊕	⊗	∂	c	∞	n	u	υ	ε	⊗	z	+	+	κ	+	≤	≥	#	√
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	e	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ
128																																
160	ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°	±	²	³	´	µ	¶	·	,	ı	º	»	¼	½	¾	¿		
192	h	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

BOTTOM																											
0	·	↓	α	β	^	~	ε	π	λ	U	δ	↑	±	⊕	⊗	∂											
16	c	∞	n	u	υ	ε	⊗	z	+	+	κ	+	≤	≥	#	√											
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/												
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?											
64	e	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O											
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_											
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o											
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ											

CMBB																																
0	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	Ω	ı	j	'	ˆ	^																
16	v	υ	~	ε	π	λ	δ	δ	↑	±	⊕	⊗	∂	β	æ	œ	Æ	Œ														
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/																	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?																
64	Ø	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O																
80	P	Q	R	S	T	U	V	W	X	Y	Z	["]	-	-																
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o																
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ	p	q	r	s	t	u	v	w	x	y	z	ff	fi	fl	ff	ff

0	CMUNH	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	Ω	ı	ı	ı	ı	ı
16	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘
32	!	”	’	ft	%	&	’	()	*	+	,	-	.	/			
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	Ø	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	X	Y	Z	[“]	ˆ	˘	
96	ı	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	x	y	z	ff	fi	fl	ffi	ffl	

0	CMOLD	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	Ω	ı	ı	ı	ı	ı
16	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘	˘
32	!	”	’	ft	%	&	’	()	*	+	,	-	.	/			
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	Ø	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	X	Y	Z	[“]	ˆ	˘	
96	ı	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	x	y	z	ff	fi	fl	ffi	ffl	

0	CHR10	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν		
		Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	Ω	ı	ı		
16	˘	˙	˚	˛	˜	˝	ˆ	˜	˝	ˆ	˜	˝	ˆ	˜		
		˘	˙	˚	˛	˜	˝	ˆ	˜	˝	ˆ	˜	˝	ˆ		
32	!	”	/	ft	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	Ø	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[“]	-	—
96	ı	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	ı	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	/
	p	q	r	s	t	u	v	w	x	y	z	ff	fi	fl	ffi	ffl

0	CHR18	↓	α	β	ˆ	˘	˙	˚	
		Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ
8	λ	δ	δ	†	‡	˙	˚	˘	
		Φ	Ψ	Ω	ı	ı	ı	ı	
16	˘	˙	˚	˛	˜	→	„	˚	
		˘	˙	˚	˛	˜	→	„	˚
24	˘	˙	˚	˛	˜	→	„	˚	
		˘	˙	˚	˛	˜	→	„	˚
32	!	”	’	ft	%	&	’	’	
		!	”	’	ft	%	&	’	
40	()	*	+	,	-	.	/	
	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	
	0	1	2	3	4	5	6	7	
56	8	9	:	;	<	=	>	?	
	8	9	:	;	<	=	>	?	
64	Ø	A	B	C	D	E	F	G	
	Ø	A	B	C	D	E	F	G	
72	H	I	J	K	L	M	N	O	
	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	
	P	Q	R	S	T	U	V	W	
88	X	Y	Z	[“]	-	-	
	X	Y	Z	[“]	-	-	
96	‘	a	b	c	d	e	f	g	
	‘	a	b	c	d	e	f	g	
104	h	i	j	k	l	m	n	o	
	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	
	p	q	r	s	t	u	v	w	
120	x	y	z	ff	fi	fl	ffi		
	x	y	z	ff	fi	fl	ffi		

0	CHR5	· ↓ α β ^ ~ ε π λ ρ δ † ± ● ○ ð
16		Γ Δ Θ Α Β Π Σ Τ Φ Ψ Ω ¡ ¨ ð
32		· ↓ α β ^ ~ ε π λ ρ δ † ± ● ○ ð
48		0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64		Ø Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
80		P Q R S T U V W X Y Z [\] ^ _
96		' a b c d e f g h i j k l m n o
112		p q r s t u v w x y z { } ~ f

0	COURIER	· ↓ α β ^ ~ ε π λ ρ δ † ± ● ○ ð
16		· ↓ α β ^ ~ ε π λ ρ δ † ± ● ○ ð
32		! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64		Ø Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
80		Ø Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
96		' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
112		' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128		
160		! ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
176		! ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192		À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
208		À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
224		à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó
240		à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó

0	CPTFONT	· ↓ α β ^ ~ ε π λ ρ δ † ± ● ○ ð
16		· ↓ α β ^ ~ ε π λ ρ δ † ± ● ○ ð
32		! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64		Ø Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
80		Ø Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
96		' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
112		' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128		
160		! ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
176		! ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192		À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
208		À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó
224		à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó
240		à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó

CPTFONTB

0	• ↓ α β ^ ~ * π λ δ δ † ± ⊙ ∞ ∂ c ∩ n u υ ∃ ⊙ z + + × † ‡ ∑ ∏ √ ∞
32	" # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128	
160	€ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

CPTFONTBI

0	• ↓ α β ^ ~ * π λ δ δ † ± ⊙ ∞ ∂ c ∩ n u υ ∃ ⊙ z + + × † ‡ ∑ ∏ √ ∞
32	" # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128	
160	€ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

CPTFONTI

0	• ↓ α β ^ ~ * π λ δ δ † ± ⊙ ∞ ∂ c ∩ n u υ ∃ ⊙ z + + × † ‡ ∑ ∏ √ ∞
32	" # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128	
160	€ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

HIGHER-MEDFNB

0	· ↓ α β ^ ~ ε π λ ρ δ † ± ⊙ ⊖ ⊗ ⊞ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿ ⊿
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	Ⓐ Ⓑ Ⓒ Ⓓ Ⓔ Ⓕ Ⓖ Ⓗ Ⓘ Ⓜ Ⓝ Ⓟ Ⓠ Ⓡ Ⓢ Ⓣ Ⓤ Ⓥ Ⓦ Ⓧ Ⓨ Ⓩ [\] ^ _
96	` a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ ¡ ¢
128	
160	ı ċ ĕ ħ Ÿ š ˆ ⊙ ⊖ ⊗ ⊞ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

HIGHER-TRB

0	· ↓ α β ^ ~ ε π λ ρ δ † ± ⊙ ⊖ ⊗ ⊞ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	Ⓐ Ⓑ Ⓒ Ⓓ Ⓔ Ⓕ Ⓖ Ⓗ Ⓘ Ⓜ Ⓝ Ⓟ Ⓠ Ⓡ Ⓢ Ⓣ Ⓤ Ⓥ Ⓦ Ⓧ Ⓨ Ⓩ [\] ^ _
96	` a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ ¡ ¢
128	
160	ı ċ ĕ ħ Ÿ š ˆ ⊙ ⊖ ⊗ ⊞ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

HL10

0	· ↓ α β ^ ~ ε π λ ρ δ † ± ⊙ ⊖ ⊗ ⊞ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	Ⓐ Ⓑ Ⓒ Ⓓ Ⓔ Ⓕ Ⓖ Ⓗ Ⓘ Ⓜ Ⓝ Ⓟ Ⓠ Ⓡ Ⓢ Ⓣ Ⓤ Ⓥ Ⓦ Ⓧ Ⓨ Ⓩ [\] ^ _
96	` a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ ¡ ¢
128	
160	ı ċ ĕ ħ Ÿ š ˆ ⊙ ⊖ ⊗ ⊞ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

HL10B																																			
0	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	●	⊖	∂	c	∩	n	u	ψ	Ξ	⊙	z	+	+	≠	+	≤	≥	■	▽			
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?				
64	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓙ	⓫	⓬	⓭	⓯	⓰	⓱	⓲	⓳	⓴	⓵	⓶	⓷	⓸	⓹	⓺	⓻	⓼	⓽	⓾	⓿	[\]	^	_		
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ			
128																																			
160	ı	ċ	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	°	±	²	³	´	µ	¶	·	,	ı	²	»	¼	½	¾	¿					
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß			
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ			

HL12																																			
0	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	●	⊖	∂	c	∩	n	u	ψ	Ξ	⊙	z	+	+	≠	+	≤	≥	■	▽			
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?				
64	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓙ	⓫	⓬	⓭	⓯	⓰	⓱	⓲	⓳	⓴	⓵	⓶	⓷	⓸	⓹	⓺	⓻	⓼	⓽	⓾	⓿	[\]	^	_		
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ			
128																																			
160	ı	ċ	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	°	±	²	³	´	µ	¶	·	,	ı	²	»	¼	½	¾	¿					
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß			
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ			

HL12B																																			
0	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	●	⊖	∂	c	∩	n	u	ψ	Ξ	⊙	z	+	+	≠	+	≤	≥	■	▽			
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?				
64	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓙ	⓫	⓬	⓭	⓯	⓰	⓱	⓲	⓳	⓴	⓵	⓶	⓷	⓸	⓹	⓺	⓻	⓼	⓽	⓾	⓿	[\]	^	_		
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ			
128																																			
160	ı	ċ	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	°	±	²	³	´	µ	¶	·	,	ı	²	»	¼	½	¾	¿					
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß			
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ			

HL12B1

0	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	⊙	⊙	⊙	⊙
16	c	▷	n	υ	ϒ	Ξ	⊙	z	+	+	≠	+	≤	≥	#	~	
32		"	#	\$	%	&	'	()	*	+	,	-	.	/		
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	Ⓢ	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-	
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	/	
128																	
144																	
160	ı	ç	€	¤	¥	ı	š	-	⊙	⊙	<	-	-	⊙	-		
176	°	±	²	³	´	µ	¶	·	,	ı	⊙	⊙	⊙	⊙	⊙	⊙	
192	h	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	

HL12I

0	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	⊙	⊙	⊙	⊙
16	c	▷	n	υ	ϒ	Ξ	⊙	z	+	+	≠	+	≤	≥	#	~	
32		"	#	\$	%	&	'	()	*	+	,	-	.	/		
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	Ⓢ	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-	
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	/	
128																	
144																	
160	ı	ç	€	¤	¥	ı	š	-	⊙	⊙	<	-	-	⊙	-		
176	°	±	²	³	´	µ	¶	·	,	ı	⊙	⊙	⊙	⊙	⊙	⊙	
192	h	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	

HL6

0	• ↓ α β ^ ~ ε π λ ϑ δ † ± • ω ϑ c ρ n u v y z • z + + x + s z = v
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? ! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _ ⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f ' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128	
160	ı ċ ĕ ğ Ÿ ! § - © ¢ ‹ › - - © - ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ ı ċ ĕ ğ Ÿ ! § - © ¢ ‹ › - - © - ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

HL7

0	• ↓ α β ^ ~ ε π λ ϑ δ † ± • ω ϑ c ρ n u v y z • z + + x + s z = v
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? ! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _ ⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f ' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128	
160	ı ċ ĕ ğ Ÿ ! § - © ¢ ‹ › - - © - ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ ı ċ ĕ ğ Ÿ ! § - © ¢ ‹ › - - © - ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

ICONS

0	• ↓ α β ^ ~ ε π λ ϑ δ † ± • ω ϑ c ρ n u v y z • z + + x + s z = v
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f Ⓜ

MEDFNB

0	• ↓ α β ^ ~ ε π λ ϑ δ † ± • ω ϑ c ρ n u v y z • z + + x + s z = v • ↓ α β ^ ~ ε π λ ϑ δ † ± • ω ϑ c ρ n u v y z • z + + x + s z = v
32	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? ! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _ ⊙ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _
96	' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f ' a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~ f
128	
160	ı ċ ĕ ğ Ÿ ! § - © ¢ ‹ › - - © - ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ ı ċ ĕ ğ Ÿ ! § - © ¢ ‹ › - - © - ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

MEDFNT

0	·	↓	α	β	^	-	ε	π	λ	ø	δ	†	±	•	•	•	•	•	•																	
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?					
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-	~	ƒ		
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ƒ	ƒ			
128																																				
150	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß				
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ				

METS

0	·	↓	α	β	^	-	ε	π	λ	ø	δ	†	±	•	•	•	•	•																			
16	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı		
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/																						
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?																					
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O																					
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-	~	ƒ																			
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o																					
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ƒ	ƒ																				
128																																					
144																																					
160	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	
176	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿																					
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß					
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß																					
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ					
240	đ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ																					

0	METS	•	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	•	•	δ
16	¢	ƒ	∞	∪	∩	∩	∩	∩	∩	∩	∩	∩	∩	∩	∩	∩	∩
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/		
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ	
128																	
144																	
160		ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
176	•	±	±	±	±	±	±	±	±	±	±	±	±	±	±	±	±
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	
240	đ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	


0	METSBI	α	β	^	~	ε	π	λ	ξ	ς	†	±	•	◦	∂	
16	◌̂	◌̃	◌̄	◌̅	◌̆	◌̇	◌̈	◌̉	◌̊	◌̋	◌̌	◌̍	◌̎	◌̏	◌̐	
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	‘	’	‚	‛	„	…	‰	‡	•	◦	◌̂	◌̃	◌̄	◌̅	◌̆	◌̇
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ƒ
128																
144																
160	ı	ċ	ĕ	ı̇	ı̈	ı̉	ı̊	ı̋	ı̌	ı̍	ı̎	ı̏	ı̐	ı̑	ı̒	ı̓
176	•	±	²	³	⁴	⁵	⁶	⁷	⁸	⁹	∞	∫	∑	∏	∑	∏
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	đ	ñ	ò	ó	ô	õ	ö	+	ø	ù	ú	û	ü	ý	þ	ÿ

0	METS1															
	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	•	∞	∂
16	ç	ç	ñ	ü	ü	Ë	⊙	z	+	+	≠	+	≤	≥	≡	√
32	!	!	#	\$	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ
128																
144																
160	ı	ç	£	¤	¥	ı	š	-	•	•	◀	-	-	•	-	
175	±	²	³	´	µ	¶	·	,	ı	q	»	¼	½	¾	¿	
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	đ	ñ	ò	ó	ô	õ	ö	+	ø	ù	ú	û	ü	ý	þ	ÿ

0	MOUSE															
	↑	→	↓	←	↕	↔	↖	×	↑	→	↓	←	↕	•	∞	∂
16	☐	☐	☐	☒	☒	☒	☒	+	+	+	+	+	+	+	+	+
32	•	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
48	☐	×	↶	↷	↶	↷	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ

SEARCH							
0	"	↓	α	β	^	~	π
8	λ	δ	δ	†	±	•	ð
16	c	ç	n	u	ÿ	•	¿
24	+	+	≠	+	¿	≠	∨
32		!	"	#	\$	%	'
40	()	*	+	,	-	/
48	0	1	2	3	4	5	6
56	8	9	:	;	<	=	>
64	0	A	B	C	D	E	F
72	H	I	J	K	L	M	N
80	P	Q	R	S	T	U	V
88	X	Y	Z	[\]	^
96	`	a	b	c	d	e	f
104	h	i	j	k	l	m	n
112	p	q	r	s	t	u	v
120	x	y	z	{		}	~

TI-LOGO													
64	0	A	B	C	D	E	F	G	H	I	J	K	L
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\
96	`	a	b	c	d	e	f	g	h	i	j	k	l
112	p	q	r	s	t	u	v	w	x	y	z	{	



```
TINY  
0  . ↓ α β ^ ~ ε π λ ρ δ † ± • ◊ ∂ c ∃ n u υ Ω ◊ 2 + + ≠ + ‰ ‹ ∑ ■ √  
32  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
64  @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _  
96  ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ¡  
128  
160  ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿  
192                              ¡ ¢ ¤ ¥ ¦ § ¨ ¨ ª « ¬ ­ ¯  
224  ¨ ª « ¬ ­ ¯ ° ± ² ³ ´ µ ¶ · ¸ ¨ ª « ¬ ­ ¯ ° ± ² ³ ´ µ ¶ · ¸ ¨ ª « ¬ ­ ¯ ° ± ² ³ ´ µ ¶ · ¸
```

```
TOP  
0  . ↓ α β ^ ~ ε π λ ρ δ † ± • ◊ ∂  
16   ∃ n u υ Ω ◊ 2 + + ≠ + ‰ ‹ ∑ ■ √  
32  ! " # $ % & ' ( ) * + , - . /  
48  0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
64  @ A B C D E F G H I J K L M N O  
80  P Q R S T U V W X Y Z [ \ ] ^ _  
96  ` a b c d e f g h i j k l m n o  
112 p q r s t u v w x y z { | } ~ ¡
```

```
TRI0  
0  . ↓ α β ^ ~ ε π λ ρ δ † ± • ◊ ∂ c ∃ n u υ Ω ◊ 2 + + ≠ + ‰ ‹ ∑ ■ √  
32  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
64  @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _  
96  ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ¡  
128  
160  ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿  
192                              ¡ ¢ ¤ ¥ ¦ § ¨ ¨ ª « ¬ ­ ¯  
224  ¨ ª « ¬ ­ ¯ ° ± ² ³ ´ µ ¶ · ¸ ¨ ª « ¬ ­ ¯ ° ± ² ³ ´ µ ¶ · ¸ ¨ ª « ¬ ­ ¯ ° ± ² ³ ´ µ ¶ · ¸
```


0	TR10B	·	↓	α	β	^	~	ε	π	λ	ø	δ	†	±	•	∞	∂
16		c	>	n	u	υ	Ξ	⊙	z	+	+	≠	+	≤	≥	■	▽
32			°	#	\$	%	&	'	()	*	+	,	-	.	/	
48		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
80		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96		'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112		p	q	r	s	t	u	v	w	x	y	z	{		}	~	/
128																	
144																	
160			€	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	–	
176		°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿
192		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208		Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224		à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240		ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

0	TR10B1	·	↓	α	β	^	~	ε	π	λ	ø	δ	†	±	•	∞	∂
16		c	>	n	u	υ	Ξ	⊙	z	+	+	≠	+	≤	≥	■	▽
32			°	#	\$	%	&	'	()	*	+	,	-	.	/	
48		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
80		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96		'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112		p	q	r	s	t	u	v	w	x	y	z	{		}	~	/
128																	
144																	
160			€	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	–	
176		°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿
192		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208		Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224		à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240		ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

TR101

0	·	↓	α	β	ˆ	˜	ε	π	λ	δ	δ	†	±	•	∞	∂	c	∞	n	u	u	∃	•	z	+	+	≠	+	≤	≥	■	√
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	⊙	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓗ	Ⓘ	Ⓢ	Ⓚ	Ⓛ	Ⓜ	Ⓝ	Ⓟ	Ⓠ	Ⓡ	Ⓢ	Ⓣ	Ⓤ	Ⓥ	Ⓦ	Ⓧ	Ⓨ	Ⓩ	[\]	ˆ	-	
96	`	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ
128																																
160		¢	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	–	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿	
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

TR12

0	·	↓	α	β	ˆ	˜	ε	π	λ	δ	δ	†	±	•	∞	∂																
16	·	↓	α	β	ˆ	˜	ε	π	λ	ϕ	δ	†	±	•	∞	∂																
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/																	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?																
64	⊙	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓗ	Ⓘ	Ⓢ	Ⓚ	Ⓛ	Ⓜ	Ⓝ	Ⓟ	Ⓠ	Ⓡ	Ⓢ	Ⓣ	Ⓤ	Ⓥ	Ⓦ	Ⓧ	Ⓨ	Ⓩ	[\]	ˆ	-	
80	Ⓟ	Ⓠ	Ⓡ	Ⓢ	Ⓣ	Ⓤ	Ⓥ	Ⓦ	Ⓧ	Ⓨ	Ⓩ	[\]	ˆ	-																
96	`	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ																
128	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ																
144																																
160		¢	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	–	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿	
176	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿																
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
208	Ⓟ	Ⓠ	Ⓡ	Ⓢ	Ⓣ	Ⓤ	Ⓥ	Ⓦ	Ⓧ	Ⓨ	Ⓩ	[\]	ˆ	-																
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
240	đ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ																

TR12B																
0	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	•	•	δ
	·	↓	α	β	^	~	ε	π	λ	δ	δ	†	±	•	•	δ
16	c	>	n	u	υ	Ξ	•	z	+	+	≠	+	≤	≥	■	▼
	c	>	n	u	υ	Ξ	•	z	+	+	≠	+	≤	≥	■	▼
32			"	#	\$	%	&	'	()	*	+	,	-	.	/
			"	#	\$	%	&	'	()	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	•	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ƒ
	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ƒ
128																
144																
160	ı	ç	£	¤	¥	¦	§	-	•	•	<	-	-	•	-	
	ı	ç	£	¤	¥	¦	§	-	•	•	<	-	-	•	-	
176	•	±	²	³	´	µ	¶	·	,	ı	ǂ	>	¼	½	¾	ı
	•	±	²	³	´	µ	¶	·	,	ı	ǂ	>	¼	½	¾	ı
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	•	Ñ	ò	ó	ô	õ	ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
	•	Ñ	ò	ó	ô	õ	ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	•	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
	•	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

0	TR1281															
	·	↓	α	β	^	~	ε	π	λ	ø	δ	†	±	•	∞	∂
	·	↓	α	β	^	~	ε	π	λ	ø	δ	†	±	•	∞	∂
16	c	o	n	u	v	∃	•	z	+	+	≠	+	≤	≥	≡	∨
	c	o	n	u	v	∃	•	z	+	+	≠	+	≤	≥	≡	∨
32		"	#	\$	%	&	'	()	*	+	,	-	.	/	
		"	#	\$	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	!
	p	q	r	s	t	u	v	w	x	y	z	{		}	~	!
128																
144																
160		€	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	–	
		€	£	¤	¥	¦	§	¨	©	ª	«	¬	–	•	–	
176	°	±	²	³	´	µ	¶	·	,	‘	²	»	¼	½	¾	¿
	°	±	²	³	´	µ	¶	·	,	‘	²	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

0	TR121
0	· ↓ α β ^ ~ e π λ δ δ † ± ● ∞ ∂
16	· ↓ α β ^ ~ e π λ δ δ † ± ● ∞ ∂
16	c > n u v w x y z + - * / < > = > ?
32	! " # \$ % & ' () * + , - . /
48	0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64	Ⓜ R B C D E F G H I J K L M N O
80	P Q R S T U V W X Y Z [\] ^ _
96	‘ a b c d e f g h i j k l m n o
112	p q r s t u v w x y z { } ~ /
128	
144	
160	ı ċ ĕ ħ ŷ ı ſ - © ¢ < ~ - © -
175	• ± ² ³ ´ μ ¶ · ¸ ¹ º » ¼ ½ ¾
192	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï
208	Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
224	à á â ã ä å æ ç è é ê ë ì í î ï
240	đ ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

0	TR18	·	↓	α	β	^	¬	ε	π	λ	δ	δ	†	±	•	∞	∂
16		◡	◢	∩	∪	∩	∪	⊗	⊘	↔	↔	≠	+	≤	≥	≡	√
32		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
48		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
96		'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112		p	q	r	s	t	u	v	w	x	y	z	{		}	~	ƒ
128																	
144																	
160		ı	ç	£	¤	¥	ı	š	"	©	ª	«	¬	–	•	-	
176		°	±	²	³	´	µ	¶	·	,	ı	º	»	¼	½	¾	¿
192		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208		Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224		à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240		đ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

0	TR188	·	↓	α	β	^	¬	ε	π	λ	ø	δ	†	±	⊙	∞	ð
16		·	↓	α	β	^	¬	ε	π	λ	ø	δ	†	±	⊙	∞	ð
32		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
48		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
96		'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112		p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ
128																	
144																	
160		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°	±
176		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°	±
192		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208		Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224		à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240		đ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

0	TR8	·	↓	α	β	^	¬	ε	π	λ	ø	δ	†	±	⊙	∞	ð	c	ç	n	u	ü	ÿ	⊙	z	+	+	×	+	≤	≥	■	√		
32		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?			
64		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-		
96		'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ		
128																																			
160		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯	°	±	z	ç	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
192		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß		
224		à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	đ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ		

VT-Graphics-Bottom

0	·	↓	α	β	^	~	ε	π	λ	ø	δ	†	±	•	°	∂
16	↳	↳	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	⊙	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	∫

VT-Graphics-Top

0	·	↓	α	β	^	~	ε	π	λ	ø	δ	†	±	•	°	∂
16	↳	↳	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	⊙	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	∫

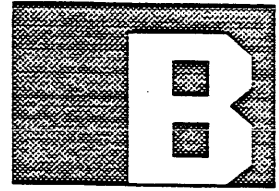
VT-GRAPHICS-WIDER-FONT																											
0	•	↓	α	β	^	∪	ε	π	λ	α	δ	†	±	⊙	⊗	⊘	⊙	⊙									
16	c	u	∩	∪	∩	∪	⊙	⊗	⊘	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙									
32		"	#	\$	2	&	'	()	*	+	,	-	.	/												
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?											
64	⊙	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O											
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_											
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o											
112	p	q	r	s	t	u	v	w	x	y	z	()	~	∫											

WIDER-FONT																											
0	•	↓	α	β	^	∪	ε	π	λ	α	δ	†	±	⊙	⊗	⊘	⊙	⊙									
16	c	u	∩	∪	∩	∪	⊙	⊗	⊘	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙									
32		"	#	\$	2	&	'	()	*	+	,	-	.	/												
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?											
64	⊙	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O											
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_											
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o											
112	p	q	r	s	t	u	v	w	x	y	z	()	~	∫											
128																											
144																											
160		⊕	⊗	⊘	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪	∩	∪									
176	°	±	z	∩	'	∫	∫	.	,	∩	∩	∩	∩	∩	∩	∩	∩	∩									
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï											
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß											
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï											
240	đ	ř	š	š	š	š	š	+	š	š	š	š	š	š	š	š	š	š									

WIDER-MEDFNT

0	•	↓	α	β	^	~	ε	π	λ	ø	δ	↑	±	⊙	⊖	∂	∃	∞	∴	+	→	≠	≠	≤	≥	≡	∨											
32	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?							
64	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓗ	Ⓘ	⓵	⓶	⓷	⓸	⓹	⓺	⓻	⓼	⓽	⓾	⓿	⓰	⓱	⓲	⓳	⓴	⓵	⓶	⓷	⓸	⓹	⓺	[\]	^	_		
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	ſ						
128																																						
160	ı	ç	ş	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß						
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ						

COMMAND TABLES



This appendix provides a convenient reference to all of the keystroke command tables described previously in the manual. These tables are copies of the tables from their respective sections. The following tables are listed:

- Completion Commands — Table B-1
- UCL Keystroke Commands — Table B-2
- Font Editor Keystroke Assignments — Table B-3
- Graphics Editor Keystroke Assignments — Table B-4
- Summary of Debugger Commands — Table B-5
- Window-Based Debugger Commands — Table B-6
- Summary of Inspector Keyboard Commands — Table B-7
- General Flavor Inspector Commands — Table B-8
- Peek Modes and Commands — Table B-9
- Stepper Commands — Table B-10
- Telnet Commands — Table B-11
- VT100 Commands — Table B-12
- Converse Commands — Table B-13
- Explorer Mail Reader Keystroke Commands — Table B-14
- Namespace Editor Keystroke Commands — Table B-15
- Color Map Editor Commands — Table B-16

Table B-1

Completion Commands	
Command Key	Completion Description
ESCAPE	<p><i>Recognition completion</i> treats the typed input as the initial substring and completes the word when possible. If several words start with the substring, the word is completed as far as possible.</p> <p>For instance, in an application that has commands named <code>print-results</code> and <code>print-query</code>, the typed character <code>p</code> followed by pressing the ESCAPE key causes the word to complete as <code>print-.</code> Whereas, if <code>print-query</code> is not in the application, the same typed input using the ESCAPE key completes as the full command name <code>print-results</code>.</p>
SUPER-ESCAPE	<p><i>Apropos completion</i> treats the typed input as a substring included anywhere in the completing word and attempts to complete it.</p> <p>For instance, <code>(output-to-string</code> followed by pressing SUPER-ESCAPE completes as follows, which is the only Lisp function that contains this string:</p> <p><code>(with-output-to-string</code></p>
HYPER-ESCAPE	<p><i>Spelling-corrected completion</i> attempts to complete a word by treating the input string as slightly misspelled with missing, extra, or misplaced characters.</p>
Space bar	<p><i>Auto-completion</i> is recognition completion on typed command names. This allows you to type part of an application command name and then press the space bar to complete the name.</p>
CTRL-/	<p>Displays a mouse-sensitive menu of possible recognition completions.</p>
SUPER-/	<p>Displays a mouse-sensitive menu of possible apropos completions. Refer to Figure 6-8.</p>
HYPER-/	<p>Displays a mouse-sensitive menu of possible spelling-corrected completions.</p>

Table B-2

UCL Keystroke Commands	
Command Name	Keystroke
Help	HELP
Application Help	Select from Help menu
Tutorial	Select from Help menu
Explorer Overview	Select from Help menu
System Menu	Mouse right
System Application	SYSTEM HELP
TERM Key Help	TERM HELP
Command Type-In Help	Select from Help menu
Command Display	HYPER-CTRL-HELP
Command History	HYPER-CTRL-P
Command Name Search	HYPER-CTRL-N
Keystroke Search	HYPER-CTRL-K
Command Editor	HYPER-CTRL-STATUS
Build Keystroke Macro	HYPER-CTRL-M
Build Command Macro	HYPER-CTRL-C
Save Commands	HYPER-CTRL-S
Load Commands	HYPER-CTRL-L
Top Level Configurer	HYPER-CTRL-T
Redo	HYPER-CTRL-R

Table B-3 Font Editor Keystroke Assignments

Menu Name	Keystroke	Menu Name	Keystroke
<i>Font-io</i> ¹			
Select	SUPER-A	Rotate Left	SUPER-←
Display	SUPER-D	Rotate Right	SUPER-→
Copy	SUPER-C	Rotate 180	SUPER-↓ or SUPER-↑
Load	SUPER-L	Italicize	SUPER-I
Write	SUPER-W	Stretch	SUPER-S
Create	SUPER-M	Thicken	SUPER-T
Remove	SUPER-K	Unthicken	SUPER-U
Directory	SUPER-F	Reverse	SUPER-V
<i>Character-io</i>			
Get Char	CTRL-G	Save Char	CTRL-P
Get Char Num	META-CTRL-G	Save X	META-CTRL-P
Get Char Gray	META-G		
<i>Editing</i> ²			
Reflect	META-R	Stretch Char	META-S
Left Rotate	META-←	Char Thicken	META-T
Right Rotate	META-→	Char Unthicken	META-U
180 Rotate	META-↓, META-↑	Char Reverse	META-V
Char Italicize	META-I		
<i>Line</i> ³			
Spline ³	META-CTRL-L	Erase Both ⁴	META-CTRL-E
Box ³	META-CTRL-S	Erase Black ⁴	CTRL-E
	META-CTRL-B	Erase Gray ⁴	META-E
<i>Merge Gray</i> ⁴			
Merge Menu	META-X	Move Both ⁴	META-CTRL-M
Swap Planes ⁴	META-CTRL-X	Move Gray ⁴	META-M
	CTRL-X		
<i>Screen</i>			
Home	CTRL-H	Left	META-CTRL-←
Redisplay	CLEAR SCREEN	Up	META-CTRL-↑
Set Scale	CTRL-@	Right	META-CTRL-→
Change Mode	CTRL-M	Down	META-CTRL-↓
Change Var	HYPER-V	New Mouse	HYPER-N
Clear Registers	SUPER-E		

NOTES:

- ¹ All the font-io commands use SUPER.
- ² All the character editing commands use META.
- ³ The drawing operations (line, spline, and box) use META-CTRL and the first letter of the command.
- ⁴ Most of the commands that affect the planes are on separate modifier keys:
 Black plane only — CTRL key
 Gray plane only — META key
 Both planes — META-CTRL keys

Table B-4 Graphics Editor Keystroke Assignments

Command Name	Assigned Keystroke	Menu or Submenu
Arc Mode	META-A	Draw
Circle Mode	META-C	Draw
Clear Background Picture	META-CTRL-CLEAR INPUT	Picture
Clear Picture	CLEAR INPUT	Picture
Copy Objects Mode	META-F	Function
Crosshair On or Off	CTRL-H	Status
Define Presentation	SUPER-D	Presentation
Define Subpicture	META-G	Subpicture
Delete Objects Mode	META-D	Function
Display Presentation	SUPER-Y	Presentation
Drag-Move Mode	META-CTRL-M	Function
Drag-Copy Mode	META-CTRL-E	Function
Edit Parameters	META-E	Function
Exit	END	Function
Explode Subpicture	META-X	Subpicture
Find Picture	META-CTRL-F	Picture
Grid On or Off	CTRL-G	Status
Insert Picture	META-CTRL-I	Picture
Insert Subpicture	META-I	Subpicture
Interrupt Drawing On or Off	CTRL-I	Status
Kill or Save Pictures	META-CTRL-STATUS	Picture
Kill or Save Presentations	SUPER-STATUS	Presentation
Kill Picture	META-CTRL-K	Picture
Kill Presentation	SUPER-K	Presentation
Line Mode	META-L	Draw
List or Select Presentation	SUPER-L	Presentation
List Pictures	META-CTRL-L	Picture
Load Presentation	SUPER-F	Presentation
Modify Presentation	SUPER-M	Presentation
Mouse Down One	↓	—
Mouse Down Three	HYPER-↓	—
Mouse Down Two	SUPER-↓	—
Mouse Left One	←	—
Mouse Left Three	HYPER-←	—
Mouse Left Two	SUPER-←	—
Mouse Right One	→	—
Mouse Right Three	HYPER-→	—
Mouse Right Two	SUPER-→	—
Mouse Up One	↑	—
Mouse Up Three	HYPER-↑	—
Mouse Up Two	SUPER-↑	—
Move Objects Mode	META-M	Function
Next Picture	META-CTRL-N	Picture
Paint Mode	META-B	Draw
Pan Down	CTRL-↓	Windowing
Pan Down Half	META-↓	Windowing
Pan Down Third	META-CTRL-↓	—
Pan Left	CTRL-←	Windowing
Pan Left Half	META-←	Windowing
Pan Left Third	META-CTRL-←	—
Pan Right	CTRL-→	Windowing

Continued
Continued

Table B-4 Graphics Editor Keystroke Assignments (Continued)

Command Name	Assigned Keystroke	Menu or Submenu
Pan Right Half	META-→	Windowing
Pan Right Third	META-CTRL-→	—
Pan Up	CTRL-↑	Windowing
Pan Up Half	META-↑	Windowing
Pan Up Third	META-CTRL-↑	—
Polyline Mode	META-P	Draw
Previous Picture	META-CTRL-P	Picture
Print Graphics File	CTRL-P	Picture
Read Background Picture	META-CTRL-B	Picture
Rectangle Mode	META-R	Draw
Redraw Picture	CLEAR SCREEN	Picture
Reorder Pictures	META-CTRL-O	Picture
Restore Presentation	SUPER-R	Presentation
Restore Subpicture	HYPER-R	Subpicture
Restore User Status	HYPER-SUPER-R	Function
Revert Picture	META-CTRL-R	Picture
Revert Picture Status	HYPER-SUPER-V	Function
Revert User Status	HYPER-SUPER-T	Function
Ruler Mode	META-N	Draw
Save Picture	META-CTRL-S	Picture
Save Picture Compiled Mode	CTRL-E	Status
Save Presentation	SUPER-S	Presentation
Save Subpicture	HYPER-S	Subpicture
Save User Status	HYPER-SUPER-S	Function
Scale Objects Mode	META-Q	Function
Scale Thickness On or Off	CTRL-Q	Status
Set Fill and Edge Color	CTRL-C	Status
Set Font	CTRL-F	Status
Set Layout	CTRL-L	Status
Set Min Dot Delta	CTRL-B	Status
Set Min Nil Delta	CTRL-N	Status
Set Pick Tolerance	CTRL-R	Status
Set Rewindow Area	META-V	Windowing
Set Tab Width	CTRL-S	Status
Set Thickness	CTRL-T	Status
Set X Grid Size	CTRL-X	Status
Set Y Grid Size	CTRL-Y	Status
Show Default Window	META-CLEAR SCREEN	Windowing
Show Entire Picture	CTRL-CLEAR SCREEN	Windowing
Spline Mode	META-S	Draw
Text Mode	META-W	Draw
Triangle Mode	META-T	Draw
Undefine Subpicture	META-U	Subpicture
Undo	UNDO	Function
View or Modify Status	STATUS	—
Zoom Grid On or Off	CTRL-Z	—
Zoom In	CTRL->	Windowing
Zoom In Three	META-CTRL->	—
Zoom In Two	META->	Windowing
Zoom Out	CTRL-<	Windowing
Zoom Out Three	META-CTRL-<	—
Zoom Out Two	META-<	Windowing

Table B-5 Summary of Debugger Commands

Key Sequence	Explanation of Command
CTRL-A	Prints the argument list of the function in the current frame.
META-CTRL-A	Sets * to the <i>n</i> th argument of the function in the current frame and + to the locative to the argument.
CTRL-B	Prints a brief backtrace of the stack.
META-B	Prints a more detailed backtrace of the stack.
META-CTRL-B	Prints a more detailed backtrace of the stack with no censoring of interpreter functions.
META-C	Attempts to continue, like RESUME, but executes <code>setq</code> on the unbound variable or defines the undefined function. This command uses the <code>:store-new-value proceed</code> type and is available only if this <code>proceed</code> type is also available.
CTRL-D	Attempts to continue, like RESUME, but traps on the next function call.
META-D	Toggles the flag that causes a trap on the next function call after you continue or otherwise exit the debugger.
CTRL-E	Edits the source code for the function in the current frame.
META-CTRL-F	Sets * to the function in the current frame.
META-CTRL-H	Prints a list of the condition handlers for the current frame.
CTRL-L or CLEAR SCREEN	Clears the screen and redisplay the error message and current frame.
META-L	Clears the screen and redisplay the error message and the current frame's function, arguments, locals, and compiled code.
META-CTRL-L	Sets * to the <i>n</i> th local of the function in the current frame and sets + to the locative to the local.
CTRL-M	Sends a bug report containing the error message and a backtrace of <i>n</i> frames (the default is 5).
CTRL-N or LINE FEED	Moves down to the next frame. With an argument, this command moves down <i>n</i> frames.
META-N	Moves down to the next frame with full-screen typeout. With an argument, this command moves down <i>n</i> frames.
META-CTRL-N	Moves down to the next frame even if it is uninteresting or still accumulating arguments. With an argument, this command moves down <i>n</i> frames.
CTRL-P or RETURN	Moves up to the previous frame. With an argument, this command moves up <i>n</i> frames.
META-P	Moves up to the previous frame with full-screen typeout. With an argument, this command moves up <i>n</i> frames.
META-CTRL-P	Moves up to the previous frame even if it is uninteresting or still accumulating arguments. With an argument, this command moves up <i>n</i> frames.
CTRL-R	Returns a value (or values) from the current frame.
META-R	Reinvokes the function in the current frame with possibly altered arguments.
META-CTRL-R	Reinvokes the function in the current frame (throws back to it and starts it over at its beginning).
CTRL-S	Searches for a frame containing a function with the specified string.
META-S	Prints the value of a special variable within the context of the current frame. Instance variables of <code>self</code> can also be specified even if not special.
META-CTRL-S	Prints a list of special variables bound by the current frame and the values they are bound to by the frame. If the frame binds <code>self</code> , all the instance variables of <code>self</code> are listed even if not special.
CTRL-T	Throws a value to a tag.
META-CTRL-U	Moves down the stack to the next interesting frame if the current frame is uninteresting.
CTRL-X	Toggles the flag in the current frame that causes a trap on exit.
META-X	Sets the flag that causes a trap on exit for the current frame and all the frames outside of it.
META-CTRL-X	Clears the flag that causes a trap on exit for the current frame and all the frames outside of it.
META-CTRL-W	Enters the window-based debugger.
META-<	Moves to the top of the stack.
META->	Moves to the bottom of the stack.

Table B-5 Summary of Debugger Commands (Continued)

Key Sequence	Explanation of Command
ABORT	Quits to command level. This is not a command but something you can press to escape from typing a form.
HELP	Prints a help message.
CTRL-0 through CTRL-9, META-0 through META-9, META-CTRL-0 through META-CTRL-9	Numeric arguments to a debugger command are specified by typing a number (in base 10) with CTRL and/or META held down.
RESUME	Attempts to continue, using the first proceed type on the list of available ones for this error.
SUPER-A to SUPER-Z	The debugger assigns these commands to all the available proceed types for this error. The assignments are different each time the debugger is entered, so it prints a list of them when it starts up.

Table B-6 Window-Based Debugger Commands

Name	Keystroke	Description
<i>Examine commands:</i>		
Doc	CTRL-HELP	Shows documentation for each of the window-based debugger panes.
Search	CTRL-S	Prompts you for a string, then searches for this string in a stack frame. Search then selects this frame and displays its arguments, variables, and code. This works like CTRL-S in the regular debugger.
Error	CTRL-L	The Error command reprints the error message for the current error in the Lisp window. This works like CTRL-L in the regular debugger.
Stay	HYPER-S	Toggles the value of <code>eh:*enter-window-debugger*</code> to use the regular or window-based debugger.
Inspect	CTRL-I	Inspects either a mouse-sensitive item that you have selected or the value of a form that you enter from the keyboard. The display appears in the top window.
Edit	CTRL-E	The Edit command invokes the editor on a function after reading the name of the function or allowing you to select a function with the mouse. This works like CTRL-E in the regular debugger.
Report	CTRL-M	Mails a bug report containing the error message and a backtrace of the stack. This works like CTRL-M in the regular debugger.
Arglist	CTRL-A	The Arglist command asks for the name of a function that can be typed on the keyboard or selected with the mouse if it is on the screen. When you pick a flavor instance or a closure, the Arglist command asks for the message name to this flavor and prints its arguments. When you pick a line of a stack frame from the stack window, the Arglist command tries to align the printout of the arguments with the value supplied in this line in this frame. This works like CTRL-A in the regular debugger.
Modify	META-CTRL-A or META-CTRL-L	Prompts you to select an argument or local variable with the mouse and allows you to type (or select with the mouse) a new value to be substituted.

Table B-6 Window-Based Debugger Commands (Continued)

Name	Keystroke	Description
<i>Resume commands:</i>		
Retry	META-CTRL-R	Attempts to restart the current frame, like the META-CTRL-R key sequence in the regular debugger.
Resume	RESUME	Resumes from the error. Clicking left on Resume is like pressing RESUME in the regular debugger.
Return	CTRL-R	Asks for the name of a value (which can be selected with the mouse) and returns it from the current frame, like CTRL-R in the regular debugger.
Abort	META-CTRL-ABORT	Leaves the debugger and aborts the program.
Bk Next	META-D	Toggles the flag that causes a trap on the next function call after you continue or otherwise exit the debugger. This works like META-D in the regular debugger.
Bk Exit	CTRL-X	Toggles the flag in the current frame that causes a trap on exit or throw through the frame. This works like CTRL-X in the regular debugger.
Bk All	SUPER-X	Toggles the flag that causes a trap on exit for the current frame and all outer frames. This works like META-X in the regular debugger.
Step	CTRL-D	Steps from the current point of execution in the current frame, but traps on the next function call. This works like CTRL-D in the regular debugger. Note that this command does not use the Stepper on the function; actually, it steps through the stack frame by frame, reinvoking the debugger on every function call.
End	END	Exits the debugger window, returning to the regular debugger.

Table B-7 Summary of Inspector Keyboard Commands

Name	Keystroke	Description
Bottom	META->	Scrolls the bottom or leftmost inspection pane to the bottom.
Break	BREAK	Runs a break loop in the typeout window of the bottom or leftmost inspection pane.
Config	SUPER-C	Invokes a menu that allows you to choose between four configurations: one with three small inspection panes, one with a single large inspection pane, one with two horizontal inspection panes, and one with two vertical inspection panes. You can set the default configuration in the Profile variable <code>tv:*inspector-configuration*</code> .
Delete	CTRL-CLEAR SCREEN	Erases the contents of the inspection panes and deletes the items in the history pane and history cache.
Doc	CTRL-HELP	Displays a pop-up window briefly describing each inspection pane.
Down	CTRL-V or CTRL-↓	Scrolls the bottom or leftmost inspection pane's display forward by one page.
Edit	HYPER-E	Edits the definition of the object at which you are pointing with the mouse or the definition of the form that you evaluate.
Exit	END	Exits the Inspector, returning the value of = (if you entered the Inspector with the <code>inspect*</code> function). Refer to <code>Set=</code> .
FlavIns	HYPER-F	Invokes the Flavor Inspector on a flavor or method. Refer to Section 16, Flavor Inspector, for details.
Mode	SUPER-M	Enters/exits the Lisp mode. This command causes the Lisp Listener pane to evaluate Lisp forms as in a regular Lisp Listener.
Modify	CTRL-M	Allows you to change the value of an object. You are prompted to select the object by clicking on it with the mouse; then enter its replacement value through the Lisp Listener pane or click on another value. You can also invoke this command by pressing HYPER while clicking on an item.
Print	SUPER-P	Invokes a menu that allows you to modify variables that control the printing format, such as <code>*print-base*</code> .
Refresh	CTRL-R	Redisplays the inspected objects reflecting any value changes to the components of those objects.

Table B-7 Summary of Inspector Keyboard Commands (Continued)

Name	Keystroke	Description
Set=	CTRL-=	Sets the value of the symbol = (equal sign) to an object that you select with the mouse or to the result of a form that you type in the Lisp Listener pane, for later reference and use. You can then do something such as <code>(setq foo (car =))</code> , if = was bound to a list. Also, the value of the = symbol is returned when exiting the Inspector if the Inspector was invoked with <code>inspect*</code> .
Up	META-V or CTRL-↑	Scrolls the bottom or leftmost inspection pane's display back by one page.
Top	META-<	Scrolls the bottom or leftmost inspection pane to the top.

Table B-8 General Flavor Inspector Commands

Name	Keystroke	Description
All Flavors	SUPER-A	Displays all flavor names currently defined in the system in an inspection pane. You might want to browse through flavor structures.
Bottom	META->	Scrolls the main inspection pane to the bottom.
Break	BREAK	Enters the break read-eval-print loop.
Config	SUPER-C	Invokes a menu that allows you to choose between four configurations: one with three small inspection panes, one with a single large inspection pane, one with two horizontal inspection panes, and one with two vertical inspection panes. You can set the default configuration in the Profile variable <code>tv:*flavor-inspector-configuration*</code> .
Delete	CTRL-CLEAR SCREEN	Deletes all inspected objects from the history and inspection panes.
Down	CTRL-V or CTRL-↓	Scrolls the main inspection pane's display forward one page.
Exit	END	Exits the Flavor Inspector.
Help on Syntax	META-HELP	Provides input editor help.
Mode	SUPER-M	Toggles between Lisp mode and inspect mode.
Refresh	CTRL-R	Redisplays the inspected objects, updating any fields that have changed values.
Top	META-<	Scrolls the main inspection pane to the top.
Trace	SUPER-T	Turns on Trace for a specified method. This command is equivalent to <code>(trace method-spec)</code> . Use <code>(untrace)</code> to turn off all tracing.
Up	META-V or CTRL-↑	Scrolls the main inspection pane's display back one page.

NOTE:

When you position the mouse cursor over a flavor or method name in one of the inspection panes, several commands relating to flavors or methods are available on the mouse as described in the mouse documentation window.

Table B-9

Peek Modes and Commands	
Name	Keystroke
<i>Modes:</i>	
Processes	P
Counters	C
Areas	A
File Status	F
Windows	W
Servers	S
Network	N
Function Histogram	M
<i>Commands:</i>	
Host Status	H
Set Timeout	T
Exit	END
Documentation	CTRL-HELP

Table B-10 Stepper Commands

Key Sequence	Explanation of Command
CTRL-N (Next)	Steps to the item to be evaluated. The Stepper continues until the next item to print out, and it accepts another command.
SPACE	Advances to the next item at this level. In other words, this command continues to evaluate at this level but does not step anything at lower levels. This is a good way to skip over parts of the evaluation that do not interest you.
CTRL-A (Args)	Skips over the evaluation of the arguments of this form but pauses in the Stepper before calling the function that is the car of the form.
CTRL-U (Up)	Continues evaluating until the Stepper goes up one level. This resembles the space command, except that it skips over anything on the current level as well as on lower levels.
END (exit)	Exits; finishes evaluating without any more stepping.
CTRL-T (Type)	Retypes the current form in full (without truncation).
CTRL-G (Grind)	Grinds (that is, pretty-prints) the current form.
CTRL-E (Editor)	Switches windows to the editor.
CTRL-B (Breakpoint)	<p>Activates a breakpoint (that is, a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:</p> <ul style="list-style-type: none"> ■ *step-form* — The current form ■ *step-values* — The list of returned values ■ *step-value* — The first returned value <p>If you change the values of these variables, you affect execution.</p>
CTRL-L	Clears the screen and redisplay the last 10 pending forms (forms that are being evaluated).
META-L	Resembles CTRL-L but does not clear the screen.
META-CTRL-L	Resembles CTRL-L but redisplay all pending forms.
HELP	Pops up the Universal Command Loop (UCL) Help menu.

Table B-11

Telnet Commands	
Keystroke	Description
NETWORK A	Send the Abort Output (AO) Telnet command to the remote host.
NETWORK CLEAR INPUT	Send the Erase Line (EL) Telnet command to the remote host.
NETWORK D	Disconnect from the current remote host and ask for the name of another remote host to which you want to connect.
NETWORK END	Expose the previously selected window and leave the Telnet connection open.
NETWORK HELP	Describes the Telnet commands. Also, you can obtain this information by pressing HELP and clicking on Command Display.
NETWORK M	Toggle the <i>*more*</i> processing variable on the Telnet window.
NETWORK O	Toggle between Insert mode (the default mode) and Overwrite mode.
NETWORK P	Send the Interrupt Process (IP) command to the remote host.
NETWORK Q	Expose the previously selected window and disconnect from the remote host.
NETWORK STATUS	Send the Are You There (AYT) Telnet command to the remote host.

Table B-12 VT100 Commands

Name	Keystroke	Description
Answerback	NETWORK B	Sends the Answerback message string in *vt100-answerback-message*
80/132 Columns	NETWORK C	VT100 Setup-A 80/132 Columns
Set Lines	NETWORK L	Sets the number of lines for the VT100 screen and reconfigure screen
Reset	NETWORK R	VT100 Setup-A Reset
VT100 Switch	NETWORK S	Enables/Disables VT100 escape sequence processing
Truncate	NETWORK T	Toggles truncating of VT100 screen pane
Reverse Video	NETWORK V	Toggles between standard video and reverse video for the VT100 screen

Table B-13 Converse Commands

Key Sequence	Explanation of Command
END	Send the current message without exiting from Converse.
CTRL-END	Send the current message and exit from Converse.
ABORT	Eliminate the current Converse window.
CTRL-M	Mail the current message instead of sending it with Converse.
META-{	Move to the previous To: line.
META-}	Move to the next To: line.
META-X Delete Conversation	Delete the current conversation.
META-X Write Buffer	Write all of the conversations into a file.
META-X Write Conversation	Write only the current conversation into a file.
META-X Append Conversation	Append the current conversation to the end of a file.
META-X Regenerate Buffer	Rebuild the buffer structure. This command is useful if you edit in or across the thick dividing lines that separate conversations, which damages the buffer structure. Some error messages may suggest that you execute this command before retrying an operation. Note that this command deletes anything you have inserted in the buffer but have not yet sent.
META-X Gag Converse	Toggle the value of the <code>zwei:*converse-gagged*</code> variable. If set to <code>t</code> , the variable tells Converse to reject incoming messages; if set to <code>nil</code> , it tells Converse to accept incoming messages.

Table B-14

Explorer Mail Reader Keystroke Commands

Keystroke	Command Name
<i>Mail File and Inbox Commands:</i>	
I	Read Mail
G	Get New Mail
CTRL-X CTRL-S	Save Mail File
W	Write Mail File
C	Copy Message to Mail File
CTRL-C	Copy Message to Text File
META-X ¹	Change Inboxes
META-X	Change File Options
META-X	Set Mail File Format
<i>Mail Buffer and Window Commands:</i>	
O	Other Mail Buffer
Q or END	Exit Mail Reader
ABORT	Abort Mail Reader
B or CTRL-X CTRL-M ²	List Mail Buffers
CTRL-W	Toggle Mail Window Configuration
META-X	Filtered Mode
S ³	Expand Filter
=	Filter Messages
META-X ³	Delete Filters
<i>Selecting and Viewing Message Commands:</i>	
N	Next Undeleted Message
P	Previous Undeleted Message
CTRL-N	Next Message
CTRL-P	Previous Message
META-X	Next Unseen
META-X	Previous Unseen
SUPER-N	Next Message With Keyword
SUPER-P	Previous Message With Keyword
<	First Message
>	Last Message
J	Jump to Message
META-S	Mail Incremental Search
META-R	Mail Reverse Incremental Search
space bar	Next Message Screen
RUBOUT	Previous Message Screen
V	View Message

NOTES:

¹ To invoke any command whose keystroke is listed as META-X, you press META-X and then type the name of the command as listed in the Command Name column. Note that completion is available.

² The CTRL-X CTRL-M, CTRL-X M, and all mailing list commands can be invoked from anywhere within Zmacs.

³ The Expand Filter command (S) and the Delete Filters command (META-X) only work in filter summary mode.

Table B-14

Explorer Mail Reader Keystroke Commands (Continued)

Keystroke	Command Name
-----------	--------------

Sending Mail Commands:

M or CTRL-X M ²	Mail
R	Reply
META-X	Reply to Sender
META-X	Reply to All
F	Forward
META-X	Resend
META-X	Mail Template
META-CTRL-Y	Yank Message

Deleting and Expunging Message Commands:

D	Delete Message
CTRL-D	Delete Message Backward
U	Undelete Message
X	Expunge Mail Buffer

Editing Message Commands:

E	Edit Message
---	--------------

Printing Message Commands:

CTRL-SHIFT-P	Print Message
--------------	---------------

Message Keyword Commands:

K	Change Message Keywords
META-X	Delete Keyword From All Messages

Miscellaneous Mail Commands:

CTRL-X S	Sort Messages
A	Mail Mark for Apply
X	Execute
CTRL-X CTRL-R	Revert Mail Buffer
META-X	Change Message Attributes
!	Toggle Reminder Message
•	Toggle Unseen Message
H	Reformat Message Headers

Mailing List Commands²:

META-X	List Mailing List
META-X	List Mailing List Membership
META-X	Add Mailing List Member
META-X	Delete Mailing List Member
META-X	Remove Mailing List

NOTE:

² The CTRL-X CTRL-M, CTRL-X M, and all mailing list commands can be invoked from anywhere within Zmacs.

Table B-15

Namespace Editor Keystroke Commands	
Keystroke	Command Name
<i>General Commands:</i>	
A	Add Object
CTRL-SHIFT-D or CTRL-SHIFT-A	Show Documentation
META-CTRL-S	Expand All Objects in NSE
META-X*	Unexpand All Objects in NSE
CTRL-X CTRL-R	Revert NSE to Local State
CTRL-X CTRL-S	Update Namespace Locally
CTRL-X CTRL-W	Update Namespace Globally
META-X	Distribute Namespace
META-X	Verify Namespace
META-X	Write NSE Buffer to File
<i>Class Commands:</i>	
S	Expand/Unexpand Class
META-S	Expand All Objects in Class
META-X	Unexpand All Objects in Class
F	Find Object
CTRL-F	Find Object and Aliases
META-F	Find Non-Alias Objects
META-CTRL-F	Find Objects With Specific Properties
META-X	Find Changed Objects
CTRL-H	Expand Horizontally
H	Toggle Horizontal Display
CTRL-C	Copy Objects in Class
W	Write (Update) Class Locally
CTRL-W	Write (Update) Class Globally
META-X	Verify Class Incrementally
R	Revert Class to Local State
META-R	Revert Class to Global State
C	Copy Class to This Namespace
META-X	Create Public Horizontal Format

NOTE:

* To invoke any command whose keystroke is listed as META-X, you press META-X and then type the name of the command as listed in the Command Name column. Note that completion is available.

Table B-15

Namespace Editor Keystroke Commands (Continued)

Keystroke	Command Name
<i>Object Commands:</i>	
S	Expand/Unexpand Object
CTRL-A	Add Attribute
META-X	Add Group Attribute
D	Delete Object
U	Undelete Object
META-CTRL-A	Add Alias for Object
META-CTRL-D	Delete Aliases for Object
W	Write (Update) Object Locally
CTRL-W	Write (Update) Object Globally
META-X	Verify Object Incrementally
CTRL-C	Copy Object
R	Revert Object to Local State
META-R	Revert Object to Global State
C	Copy Object to This Namespace
<i>Attribute Commands:</i>	
E	Edit Attribute
D	Delete Attribute
U	Undelete Attribute
CTRL-C	Copy Attribute
G	Toggle Group Status
V	View Attribute
W	Write (Update) Attribute Locally
CTRL-W	Write (Update) Attribute Globally
META-X	Verify Attribute Incrementally
R	Revert Attribute to Local State
CTRL-A	Add Attribute to Current Object
META-X	Add Group Attribute to Current Object
S	Unexpand Current Object
C	Copy Attribute to This Namespace
<i>Group Attribute Commands:</i>	
META-X	Add Group Attribute to Current Object
G	Toggle Group Status
META-A	Add Group Member
META-D	Delete Group Member
E	Edit Group Member
META-X	Add Key for Group Attribute

Table B-16

Color Map Editor Commands	
Command Name	Keystroke
<i>Color Editor Commands:</i>	
Display Color Map	CTRL-D
Screen Object	CTRL-S
Toggle Edit Contrast	CTRL-T
Help	HELP
Undo	UNDO
Move This Window	CTRL-M
Reset Map	CTRL-R
Install Map	CTRL-I
<i>Color Map Commands:</i>	
Load Color Map	CTRL-L
Save Color Map	CTRL-S
Ramp Colors	CTRL-R
Copy Color	CTRL-C
Exchange Colors	CTRL-E
Select Color Map	CTRL-M
Create New Map	CTRL-N
Get Map of Window	CTRL-W

INDEX

Introduction

The indexes for this Explorer software manual are divided into several subindexes. Each subindex contains all the entries for a particular category, such as functions, variables, or concepts. The concepts (or general) index also contains all functions, variables, and so on. If you cannot find an entry in the general index such as a particular function name, use the Functions index.

Note that the format of this general index differs from the format of the general indexes in other Explorer manuals by listing the functions, variables, and so on. The reason for this difference is that there are approximately 33 different utilities discussed in this manual. The format of this index allows you to find all the entries related to a particular utility, such as Mail, in one place.

The various subindexes for this manual and the pages on which they begin are as follows:

Index Name	Page
General	Index-2
Flavors	Index-19
Functions	Index-20
Initialization Options	See Operations
Instance Variables	Index-23
Macros	See Functions
Methods	See Operations
Operations	Index-24
Variables	Index-26

Alphabetization Scheme

The alphabetization scheme used in this index ignores package names and nonalphabetic symbol prefixes for the purposes of sorting. For example, the `name:add-alias` function is sorted under the entries for the letter A rather than under the letter N.

Hyphens are sorted after spaces. Consequently, the `apropos` entry precedes the `apropos-flavor` entry. However, the `apropos-flavor` entry precedes the `aproposb` entry, as follows:

`apropos`, 25-7
`apropos-flavor`, 25-9
`aproposb`, 25-9

General
Symbols

+ variable, 26-2
 ++ variable, 26-2
 +++ variable, 26-2
 - variable, 26-2
 * variable, 26-2
 ** variable, 26-2
 *** variable, 26-2
 / variable, 26-2
 // variable, 26-2
 /// variable, 26-2

A

fs:add-logical-pathname-host function, 3-5
 mail:add-mail-inbox-probe function, 31-52
 advise utility, 21-1–21-5
 sys:advised-functions variable, 21-3
 advising one function within another, 21-4
 :around advice, 21-4
 designing the advice, 21-3
 advise utility functions
 advise, 21-1
 advise-within, 21-5
 unadvise, 21-2
 unadvise-within, 21-5
 ALU values, 10-10–10-12
 color, 10-12
 specifying, 10-24
 status variable for, 10-27
 application building
 Glossary utility, 5-1–5-12
 Suggestions menus, 8-1–8-20, 9-1–9-15
 tree editor, 11-1–11-15
 Universal Command Loop (UCL),
 6-1–6-26, 7-1–7-38
 applyhook, 20-1–20-2
 applyhook variable, 20-1
 apropos functions
 apropos, 25-7
 apropos-flavor, 25-9
 apropos-list, 25-8
 apropos-method, 25-9
 apropos-resource, 25-10
 aproposb, 25-9
 aproposf, 25-8
 sub-apropos, 25-10
 areas
 describe, 25-6
 Peek, 17-7

B

background pictures, 10-30
 backtrace, 13-1

baselines of fonts, 12-3
 black plane, 12-7
 break. *See* Lisp Listener and break
 breakon, 22-1–22-2
 eh:*breakon-functions* variable, 22-2
 breakon functions
 breakon, 22-1
 unbreakon, 22-1
 bug reporting, 4-1
 Bug command (META-X), 4-1
 bug function, 4-1
 build-command-table function, 7-13
 build-menu function, 7-15

C

Call Tree Inspector. *See* metering
 callers, finding. *See* who-calls functions
 clear-mar function, 23-2
 color:cme function, 34-2
 cold-load stream, 14-4
 color
 See also Color Map editor; graphics editor
 Profile variables, 2-2
 Color Map editor, 34-1–34-8
 color:cme function, 34-2
 color box, 34-3
 w:*color-maps* variable, 34-7
 contrast box, 34-3
 editing and defining colors, 34-3
 HSV model, 34-3
 invoking the, 34-2
 loading the, 34-2
 numeric readout pane, 34-5
 RGB model, 34-3
 slide bar, 34-4
 up/down arrow panes, 34-5
 Color Map editor commands
 Copy Color (CTRL-C), 34-7
 Create New Map (CTRL-N), 34-7
 Display Color Map (CTRL-D), 34-5
 Exchange Colors (CTRL-E), 34-7
 Get Map of Window (CTRL-W), 34-8
 Help (HELP), 34-6
 Install Map (CTRL-I), 34-6
 Load Color Map (CTRL-L), 34-6
 Move This Window (CTRL-M), 34-6
 Ramp Colors (CTRL-R), 34-7
 Reset Map (CTRL-R), 34-6
 Save Color Map (CTRL-S), 34-7
 Screen Object (CTRL-S), 34-5
 Select Color Map (CTRL-M), 34-7
 Toggle Edit Contrast (CTRL-T), 34-5
 Undo (UNDO), 34-6

color of graphics objects
 edge color, 10-9
 editing the edge and fill colors, 10-24
 fill color, 10-9
 status variables for the edge and fill colors, 10-27
 values for graphic methods, 10-9
 command context switch for Suggestions, 9-10
 command table summaries, B-1—B-23
 command, UCL
See also Universal Command Loop (UCL)
 history, 6-8
 macros, 6-18
 Converse, 30-1—30-5
 user options, 30-4
 Zmacs commands, 30-2
 Converse functions
 qsend, 30-3
 qsend-off, 30-3
 qsend-on, 30-3
 zwei:reply, 30-4
 Converse variables
 zwei:*converse-append-p*, 30-4
 zwei:*converse-beep-count*, 30-4
 zwei:*converse-end-exits*, 30-5
 zwei:*converse-extra-hosts-to-check*, 30-5
 zwei:*converse-gagged*, 30-5
 zwei:*converse-receive-mode*, 30-4
 zwei:*converse-wait-p*, 30-5
 counters, Peek, 17-6
 crash analysis and reporting, 24-1—24-18
 crash reporting, 24-1
 force crash keychord, 24-15
 hardware crash descriptions and troubleshooting, 24-7—24-14
 mass storage subsystem crashes, 24-9—24-11
 NuBus crashes, 24-8
 NUPI device and controller error crashes, 24-11—24-13
 NUPI special event crashes, 24-14
 power fail crash, 24-9
 processor fault crashes, 24-8
 NVRAM, preparing, 24-2
 shutdown record analysis format, 24-4—24-7
 software crash descriptions, 24-15—24-18
 crash analyzer functions
 report-all-shutdowns, 24-3
 report-last-shutdown, 24-2
 current font, 12-3
 gwin:*current-cache-for-raster-objects* variable, 10-16
 customization
 evaluator, 20-1—20-2
 font editor, 12-1—12-31
 Glossary utility, 5-1—5-12
 login-init file, 3-1—3-6
 Mail, 31-46—31-57

namespace editor (NSE), 32-36—32-50
 New User utility, 1-1—1-2
 Profile utility, 2-1—2-5
 Suggestions, 8-1—8-20, 9-1—9-15
 UCL, 6-1—6-26, 7-1—7-38

D

debugger (error handler), 13-1—13-13
See also breakon
 accessing the, 13-1
 backtrace, 13-1
 commands, 13-5—13-13
 resuming execution, 13-10
 stepping through function calls and returns, 13-10
 transferring to other systems, 13-11
 examining arguments, locals, functions, and values, 13-7
 examining special variables, 13-10
 examining stack frames, 13-6
 special variable bindings in the, 13-4
 Suggestions menus, 8-15
 window-based debugger, 14-1—14-5
 debugger (error handler) functions
 sys:debug-warm-booted-process, 13-11
 eh, 13-2
 eh-arg, 13-8
 eh-fun, 13-9
 eh-loc, 13-9
 eh-val, 13-9
 debugger (error handler) variables
 debug-io, 13-5
 eh:*enter-window-debugger*, 14-1
 eh:*error-backtrace-length*, 13-2
 eh:*inhibit-debugger-proceed-prompt*, 13-2
 debugging
 after a warm boot, 13-11
 cold-load stream, 14-4
 TERM O S, 14-4
 debugging functions, miscellaneous, 25-1—25-20
 defcommand macro, 7-5
 define-glossary function, 5-12
 define-glossary-file-format function, 5-11
 mail:define-mail-filter macro, 31-48
 mail:define-mail-template macro, 31-49
 defstruct, describe, 25-2
 describe functions
 describe, 25-2
 describe-area, 25-6
 describe-defstruct, 25-2
 describe-flavor, 25-3
 describe-package, 25-5
 sys:describe-partition, 25-5
 describe-region, 25-6
 describe-system, 25-4
 documentation, online. *See* Visidoc

dribble file, 25-16—25-18
 dribble-file functions
 dribble, 25-17
 dribble-all, 25-17
 dribble-end, 25-17
 dribble-start, 25-17

E

editors
 font editor, 12-1—12-31
 graphics editor, 10-1—10-42
 namespace editor, 32-13—32-35
 tree editor, 11-1—11-15
 eh function, 13-2
 eh-arg function, 13-8
 eh-fun function, 13-9
 eh-loc function, 13-9
 eh-val function, 13-9
 environment functions
 documentation, 25-18
 identity, 25-20
 lisp-implementation-type, 25-19
 lisp-implementation-version, 25-19
 long-site-name, 25-20
 machine-instance, 25-19
 machine-type, 25-19
 machine-version, 25-19
 short-site-name, 25-20
 software-type, 25-20
 software-version, 25-20
 user-name, 25-19
 environment variables
 sys:associated-machine, 25-20
 features, 25-19
 sys:local-finger-location, 25-20
 sys:local-host, 25-20
 sys:local-host-name, 25-20
 sys:local-pretty-host-name, 25-20
 user-id, 25-20
 error handler (debugger). *See* debugger (error handler)
 Eval server, 33-2
 evalhook, 20-1—20-2
 evalhook variable, 20-1
 evaluator, customizing the, 20-1—20-2
 Explorer Overview, UCL help option, 6-4

F

fed function, 12-12
 file properties, 25-13
 file status, Peek, 17-8
 fs:file-properties function, 25-13
 find-process function, 27-24
 finger function, 33-3
 fingering hosts, 33-3
 fixed-width fonts, 12-2
 flavor
 apropos, 25-9

 describe, 25-3
 Flavor Inspector, 16-1—16-8
 *, **, and *** variables, 16-5
 changing the configuration, 16-5
 command menu pane, 16-8
 flavor commands, 16-5
 tv:*flavor-inspector-configuration* variable, 16-4
 history pane, 16-7
 inspect-flavor function, 16-1
 inspection panes, 16-4
 label, 16-5
 locking the inspection panes, 16-5
 scrolling, 16-5
 Lisp Listener pane, 16-2
 method commands, 16-6
 font, 12-1
 font editor, 12-1—12-31
 adjusting character placement, 12-19
 creating a font from scratch, 12-27—12-28
 creating a modified font, 12-24—12-26
 drawing characters, 12-18
 drawing mode, 12-17
 editing a character, 12-17—12-21
 editing operations on characters, 12-20—12-21
 editing operations on fonts, 12-24—12-26
 erasing the contents of planes, 12-21
 examining a character, 12-22
 executing commands in, 12-12
 exiting the, 12-12
 fed function, 12-12
 grid scale, 12-22
 help features of, 12-9—12-11
 invoking the, 12-12
 keystroke assignments, 12-31
 merging planes, 12-19
 moving the contents of the editing area, 12-19
 sample font for comparison, 12-16
 sample string, 12-22
 saving a character, 12-23
 selecting a character to edit, 12-14—12-16
 selecting a font to edit, 12-12—12-14
 swapping planes, 12-19
 variables. *See* font editor variables
 window, 12-5—12-11
 black plane, 12-7
 character box, 12-6
 editing area, 12-6—12-7
 font display, 12-15
 gray plane, 12-7
 grid, 12-6
 label pane, 12-8
 Lisp Listener pane, 12-8
 registers, 12-7
 writing a font to a file, 12-24

- font editor commands
 - Box (META-CTRL-B), 12-18, 12-21
 - Change Modes (CTRL-M), 12-17
 - Change Variables (HYPER-V), 12-16
 - Copy (SUPER-C), 12-26
 - Create (SUPER-M), 12-24
 - Directory (SUPER-F), 12-13
 - Display (SUPER-D), 12-14
 - Erase Black (CTRL-E), 12-21
 - Erase Both (META-CTRL-E), 12-21
 - Erase Gray (META-E), 12-21
 - Get Char (CTRL-G), 12-14
 - Get Char Num (META-CTRL-G), 12-14
 - Home (CTRL-H), 12-19
 - Italicize (SUPER-I), 12-25
 - Line (META-CTRL-L), 12-18
 - Load (SUPER-L), 12-14
 - Merge Gray (META-X), 12-19
 - Merge Menu (META-CTRL-X), 12-19
 - Move Both (META-CTRL-M), 12-19
 - Move Gray (META-M), 12-19
 - New Mouse (HYPER-N), 12-17
 - Save Char (CTRL-P), 12-23
 - Save X (META-CTRL-P), 12-23
 - Select (SUPER-A), 12-13
 - Set Scale (CTRL-@), 12-22
 - Spline (META-CTRL-S), 12-18
 - Swap Planes (CTRL-X), 12-19
 - Write (SUPER-W), 12-24
- font editor variables
 - fed:*columns*, 12-16
 - fed:*label-base*, 12-16
 - fed:*sample-font*, 12-16
- font map, 12-3
- fonts
 - AST files and, 12-30
 - current font, 12-3
 - descenders, 12-6
 - directory of system fonts, 12-4
 - dumps of all, A-2—A-29
 - families of, 12-3
 - fixed-width, 12-2
 - font map, 12-3
 - list of all, A-1
 - performance considerations and, 12-28—12-29
 - properties of, 12-2
 - selected font, 12-2
 - variable-width, 12-2
 - vertical spacing, 12-3
- force crash keychord, 24-15
- function
 - advising a, 21-1—21-5
 - apropos, 25-8
 - histogram, 27-41—27-45
 - stepping, 19-1—19-3
 - tracing a, 18-1—18-6
- function histogram, 27-41—27-45
- function histogram functions
 - meter:modify-histogram, 27-44
 - meter:report-histogram, 27-44
 - meter:restore-histogram, 27-45
 - meter:save-histogram, 27-45
 - meter:start-histogram, 27-44
 - meter:stop-histogram, 27-44
- function histogram variables
 - meter:*function-histogram-depth*, 27-43
 - meter:*function-histogram-interval*, 27-43
 - meter:*function-histogram-number*, 27-43
- function histogram!!!
 - functions, 27-42
 - overview, 27-2
 - using Peek (M keystroke), 27-41
- G**
- GED system, 10-3
- ged:ged function, 10-4
- gloss:glossary function, 5-2
- Glossary utility
 - accessing the, 5-2
 - defining a glossary from the Lisp Listener, 5-12
 - defining glossary file format, 5-11
 - Glossary Command Menu, 5-3
 - Glossary Expert mode (META-CTRL-T), 5-5
 - Glossary User mode, 5-2—5-4
 - Keyboard Typein Window, 5-4
 - Menu of Glossary Entries, 5-4
 - panes, 5-2—5-4
 - Text of Selected Glossary Entries, 5-4
 - Thumb Index, 5-4
 - using Zmacs to create a glossary file, 5-10
- Glossary utility functions
 - define-glossary, 5-12
 - define-glossary-file-format, 5-11
 - gloss:glossary, 5-2
- Glossary utility!!++, 5-1—5-12
- Glossary Expert mode commands
 - Add or Delete Glossary Entry, 5-7
 - Define Glossary, 5-5
 - Delete Glossary, 5-6
 - Edit Glossary Entry, 5-7
 - Exit Expert Mode, 5-9
 - Merge in Glossary, 5-8
 - (Re)Generate XRefs, 5-9
 - Select Glossary, 5-6
 - Turn On XRef Deletion, 5-9
 - Write Current Glossary, 5-8
- glyph, 12-2
- graphics editor, 10-1—10-42
 - commands
 - See also graphics editor commands
 - keystroke assignments for, 10-40—10-42

- mouse clicks, default values (table), 10-8
- selecting, 10-7
- types of, 10-7
- undoing, 10-29
- creating the, 10-3
- gwin: *current-cache-for-raster-objects* variable, 10-16
- entering the, 10-4
- exiting the, 10-25
- foreground, clearing the, 10-31
- ged:ged function, 10-4
- menus, 10-7
- mouse cursor, moving the, 10-7
- pictures, 10-30—10-33
- presentations, 10-33—10-35
- ged: *save-bits-for-buffers* variable, 10-3
- status variables (table), 10-27
- window, 10-4—10-6
 - configurations, 10-4
 - defining a rewindow area, 10-39—10-40
 - panning the window, 10-38—10-39
 - zooming the window, 10-39
- windowing, 10-38—10-40
- world, 10-2
- graphics editor ALU values
 - color, 10-12
 - w:combine, 10-11
 - w:erase, 10-11
 - w:normal, 10-11
 - w:opposite, 10-11
- graphics editor buffers
 - changing, 10-30—10-31
 - killing, 10-31—10-32
 - reverting, 10-33
 - saving, 10-31—10-32
- graphics editor commands
 - Arc (META-A), 10-13—10-14
 - Circle (META-C), 10-14—10-15
 - Clear Background Picture (META-CTRL-CLEAR INPUT), 10-30
 - Clear Picture (CLEAR INPUT), 10-31
 - Copy (META-F), 10-23
 - Define Presentation (SUPER-D), 10-33—10-34
 - Define Subpicture (META-G), 10-36—10-37
 - Delete (META-D), 10-24
 - Display Presentation (SUPER-Y), 10-34
 - Drag-Copy (META-CTRL-E), 10-24
 - Drag-Move (META-CTRL-M), 10-26
 - Edit Parameters (META-E), 10-24—10-25
 - Exit (END), 10-25
 - Explode Subpicture (META-X), 10-37
 - Find Picture (META-CTRL-F), 10-32
 - Insert Picture (META-CTRL-I), 10-31
 - Insert Subpicture (META-I), 10-37
 - Kill or Save Pictures (META-CTRL-STATUS), 10-32
 - Kill or Save Presentations (SUPER-STATUS), 10-35
 - Kill Picture (META-CTRL-K), 10-31
 - Kill Presentation (SUPER-K), 10-34
 - Line (META-L), 10-15
 - List Pictures (META-CTRL-L), 10-30
 - List Presentations (SUPER-L), 10-35
 - Load Presentation (SUPER-F), 10-34
 - Modify Presentation (SUPER-M), 10-35
 - Move (META-M), 10-25—10-26
 - Next Picture (META-CTRL-N), 10-30
 - Paint (META-B), 10-15—10-17
 - Polyline (META-P), 10-17
 - Previous Picture (META-CTRL-P), 10-31
 - Print Picture (CTRL-P), 10-32
 - Read Background Picture (META-CTRL-B), 10-30
 - Rectangle (META-R), 10-18
 - Redraw Picture (CLEAR SCREEN), 10-33
 - Reorder Pictures (META-CTRL-O), 10-31
 - Restore Presentation (SUPER-R), 10-34
 - Restore Subpicture (HYPER-R), 10-37
 - Restore User Status (HYPER-SUPER-R), 10-29
 - Revert Picture (META-CTRL-R), 10-33
 - Revert User Status (HYPER-SUPER-T), 10-29
 - Ruler (META-N), 10-19
 - Save Picture (META-CTRL-S), 10-31
 - Save Presentation (SUPER-S), 10-34—10-35
 - Save Subpicture (HYPER-S), 10-37
 - Save User Status (HYPER-SUPER-S), 10-28
 - Scale (META-Q), 10-26—10-27
 - Set Rewindow Area (META-V), 10-39—10-40
 - Show Default Window (META-CLEAR SCREEN), 10-40
 - Show Entire Picture (CTRL-CLEAR SCREEN), 10-40
 - Spline (META-S), 10-20
 - Status (STATUS), 10-28
 - Text (META-W), 10-21
 - Triangle (META-T), 10-22
 - Undefine Subpicture (META-U), 10-38
 - Undo (UNDO), 10-29
 - Write Picture (META-CTRL-W), 10-31
- graphics objects, 10-8—10-22
 - aids for drawing objects, 10-3
 - ALU values, 10-10—10-12
 - color, 10-12
 - specifying, 10-24
 - status variable for, 10-27
 - characteristics of, 10-8—10-12

- color
 - edge color, 10-9
 - editing the edge and fill colors, 10-24
 - fill color, 10-9
 - status variables for the edge and fill colors, 10-27
 - values for graphic methods, 10-9
 - copying, 10-23—10-24
 - deleting, 10-24
 - drawing, 10-12
 - editing parameters of, 10-24—10-25
 - filled versus unfilled, 10-8
 - grouping, 10-3
 - moving, 10-25—10-26
 - named colors in the default color map, 10-10
 - positioning, 10-12
 - scaling, 10-26—10-27
 - selecting, 10-12, 10-22—10-23
 - status variables of, 10-27—10-29
 - types of, 10-2
 - arcs, 10-13—10-14
 - background pictures, 10-30
 - circles, 10-14—10-15
 - lines, 10-15
 - paintings, 10-15—10-17
 - polylines, 10-17
 - rectangles, 10-18
 - rulers, 10-19
 - splines, 10-20
 - subpictures, 10-36—10-38
 - text, 10-21—10-22
 - triangles, 10-22
 - graphics window system (GWIN), required for GED, 10-3
 - gray patterns, examples of, 10-9
 - gray plane, 12-7
- ## H
- hardware crash descriptions and troubleshooting, 24-7—24-14
 - histogram, function, 27-41—27-45
 - history
 - Suggestions menu, 8-18
 - UCL command, 6-8
 - host status, Peek, 17-18
 - net:host-status function, 33-1
- ## I
- mail:insert-address-list function, 31-51
 - mail:insert-default-header-fields function, 31-50
 - mail:insert-header-field function, 31-50
 - inspect function, 15-1
 - inspect* function, 15-1
 - inspect-flavor function, 16-1
 - Inspector, 15-1—15-7
 - *, **, and *** variables, 15-5
 - changing the configuration, 15-5
 - command menu pane, 15-7
 - commands, 15-3
 - display according to object type, 15-6
 - Flavor Inspector. *See* Flavor Inspector
 - history pane, 15-6
 - inspection panes, 15-4
 - label, 15-4
 - locking inspection panes, 15-5
 - mouse-sensitive items, 15-4
 - scrolling, 15-5
 - tv:*inspector-configuration* Profile variable, 15-3
 - Lisp Listener pane, 15-3
 - object types in display, 15-6
 - Suggestions menus, 8-14
 - Inspector functions
 - inspect, 15-1
 - inspect*, 15-1
- ## K
- keystroke macros, UCL, 6-17
- ## L
- line height, 12-3
 - Lisp Listener and break, 26-1
 - Suggestions menus, 8-9
 - Lisp Listener and break functions
 - break, 26-1
 - sys:lisp-reinitialize, 26-1
 - sys:lisp-top-level, 26-1
 - sys:lisp-top-level1, 26-1
 - Lisp Listener and break variables
 - +, 26-2
 - ++, 26-2
 - +++, 26-2
 - , 26-2
 - *, 26-2
 - **, 26-2
 - ***, 26-2
 - /, 26-2
 - //, 26-2
 - ///, 26-2
 - sys:*break-bindings*, 26-2
 - sys:load-if function, 3-6
 - login directory, 2-1
 - login function, 3-1
 - login-eval function, 3-3
 - login-fdefine function, 3-3
 - login-forms function, 3-2
 - login-init file, 3-1—3-6
 - creating logical pathnames, 3-5
 - customizations that can be undone, 3-2
 - customizing Zmacs, 3-4
 - using Profile, 3-4
 - using the sys:load-if function, 3-6
 - using the with-timeout macro, 3-5
 - login-init file functions
 - fs:add-logical-pathname-host, 3-5

sys:load-if, 3-6
 login, 3-1
 login-eval, 3-3
 login-fdefine, 3-3
 login-forms, 3-2
 login-setq, 3-2
 logout-list variable, 3-3
 with-timeout, 3-5
 login-setq function, 3-2
 logout-list variable, 3-3

M

macros

tracing, 18-1—18-6
 UCL command, 6-18

Mail, 31-1—31-57

accessing the Mail utility, 31-2
 addresses, 31-28
 details, 31-40
 command summary, 31-38
 commands, mail reader
 See also Mail commands
 basic commands, 31-8
 how to execute, 31-7
 customization, 31-46—31-57
 defining mail filter buffers, 31-48
 defining mail template buffers, 31-48
 functions. *See* Mail functions
 variables. *See* Mail variables
 default mail file, 31-2
 default viewing mode, 31-2
 displays, 31-5
 filter buffer, 31-17
 defining, 31-48
 filtered mode, 31-17
 filtering, 31-16
 forwarding mail, 31-27
 namespace attributes, 31-42
 functions. *See* Mail functions
 help, 31-5
 inbox, 31-11
 keywords, 31-34
 listing mail buffers, 31-18
 mail buffer, 31-17
 mail file, 31-11
 default, 31-2
 mail file and inbox variables, 31-52
 mail file buffer, 31-17
 mail filter buffer, 31-17
 defining, 31-48
 mail reader commands. *See* Mail commands
 mail reader, getting started, 31-1—31-10
 basic commands, 31-8
 entering and exiting, 31-2
 executing commands, 31-7
 help, 31-5
 how to execute, 31-7
 numeric arguments, 31-7

mail reader, getting starting, basic
 commands, 31-8
 mail template buffer, 31-27
 defining, 31-48
 variables, 31-55
 mail window and buffer variables, 31-53
 mailer, 31-40—31-46
 address routing, 31-43
 mail addresses, 31-40
 mail daemon, 31-46
 mailing lists, 31-40
 user mail forwarding, 31-42
 mailing lists, 31-40
 message buffer, 31-2
 message sequence, 31-2
 messages, 31-22
 attributes, 31-23
 command application, 31-35
 deleting and expunging, 31-31
 editing, 31-33
 forwarding, 31-27
 namespace attributes, 31-42
 headers, 31-22
 keywords, 31-34
 performing operations on marked
 messages, 31-36
 printing, 31-33
 reformatting headers, 31-37
 replying to, 31-27
 searching, 31-26
 selecting, 31-24
 sending, 31-27
 sorting, 31-35
 viewing, 31-24
 miscellaneous variables, 31-57
 numeric arguments, 31-7
 reformatting header variables, 31-56
 replying to mail, 31-27
 reverting the mail buffer, 31-37
 selected message, 31-7
 sending mail, 31-27
 summary buffer, 31-2
 template buffer, 31-27
 defining, 31-48
 variables, 31-55
 two-window mode, 31-18
 variables. *See* Mail variables
 viewing mode, 31-2

Mail commands

Add Mailing List Member (META-X),
 31-42
 Change File Options (META-X), 31-16
 Change Inboxes (META-X), 31-15
 Change Message Attributes (META-X),
 31-37
 Change Message Keywords (K), 31-34
 Copy Message to Mail File (C), 31-14

- Copy Message to Text File (CTRL-C), 31-15
- Delete Filters (META-X), 31-21
- Delete Keyword From All Messages (META-X), 31-34
- Delete Mailing List Member (META-X), 31-42
- Delete Message (D), 31-32
- Delete Message Backward (CTRL-D), 31-32
- Edit Message (E), 31-33
- Execute (X), 31-36
- Exit Mail Reader (Q), 31-17
- Expand Filter (S), 31-20
- Expunge Mail Buffer (X), 31-32
- Filter Messages (=), 31-20
- Filtered Mode (META-X), 31-19
- First Message (<<), 31-25
- Forward (F), 31-30
- Get New Mail (G), 31-13
- Jump to Message (J), 31-26
- Last Message (>), 31-25
- List Mail Buffers (B or CTRL-X CTRL-M), 31-18
- List Mailing List (META-X), 31-42
- List Mailing List Membership (META-X), 31-42
- Mail (M or CTRL-X M), 31-29
- Mail Incremental Search (META-S), 31-26
- Mail Mark for Apply (A), 31-35
- Mail Reverse Incremental Search (META-R), 31-26
- Mail Template (META-X), 31-31
- Next Message (CTRL-N), 31-24
- Next Message Screen (space bar), 31-26
- Next Message With Keyword (SUPER-N), 31-25
- Next Undeleted Message (N), 31-24
- Next Unseen (META-X), 31-25
- Other Mail Buffer (O), 31-17
- Previous Message (CTRL-P), 31-25
- Previous Message Screen (RUBOUT), 31-26
- Previous Message With Keyword (SUPER-P), 31-25
- Previous Undeleted Message (P), 31-24
- Previous Unseen (META-X), 31-25
- Print Message (CTRL-SHIFT-P), 31-33
- Read Mail (I), 31-13
- Reformat Message Headers (H), 31-37
- Remove Mailing List (META-X), 31-42
- Reply (R), 31-30
- Reply to All (META-X), 31-30
- Reply to Sender (META-X), 31-30
- Resend (META-X), 31-31
- Revert Mail Buffer (CTRL-X CTRL-R), 31-37
- Save Mail File (CTRL-X CTRL-S), 31-14
- Set Mail File Format (META-X), 31-16
- Sort Messages (CTRL-X S), 31-35
- Toggle Mail Window Configuration (CTRL-W), 31-18
- Toggle Reminder Message (!), 31-37
- Toggle Unseen Message (*), 31-37
- Undelete Message (U), 31-32
- View Message (V), 31-27
- Write Mail File (W), 31-14
- Yank Message (META-CTRL-Y), 31-31
- Mail flavors
 - zwei:interval-stream, 31-48
 - :set-point method, 31-48
 - mail:message, 31-47
 - mail:message methods
 - :attributes, 31-47
 - :first-bp, 31-47
 - :headers-end-bp, 31-47
 - :keywords, 31-47
 - :last-bp, 31-47
 - :name, 31-47
- Mail functions
 - mail:add-mail-inbox-probe, 31-52
 - mail:define-mail-filter, 31-48
 - mail:define-mail-template, 31-49
 - mail:insert-address-list, 31-51
 - mail:insert-default-header-fields, 31-50
 - mail:insert-header-field, 31-50
 - mail:preload-mail-file, 31-51
 - mail:print-mail-queue, 31-46
 - mail:remove-mail-inbox-probe, 31-52
 - mail:reset-mail-daemon, 31-46
 - mail:submit-mail, 31-51
- Mail variables, 31-52–31-57
 - mail:*always-check-inboxes*, 31-52
 - mail:*box-summary-lines*, 31-55
 - mail:*choose-from-all-mail-keywords-p*, 31-57
 - mail:*default-bcc-string*, 31-56
 - mail:*default-fcc-string*, 31-56
 - mail:*default-other-mail-file*, 31-53
 - mail:*default-reply-to-string*, 31-56
 - mail:*delete-message-after-copy*, 31-53
 - mail:*dont-reply-to*, 31-55
 - mail:*forward-template*, 31-50
 - mail:*in-reply-to-template*, 31-55
 - mail:*inhibit-mail-file-format-warnings*, 31-53
 - mail:*kill-mail-buffers-at-logout-p*, 31-53
 - mail:*log-enabled*, 31-46
 - mail:*mail-file-versions-kept*, 31-53
 - mail:*mail-mode-hook*, 31-56
 - mail:*mail-summary-attribute-char-alist*, 31-55
 - mail:*mail-summary-mode*, 31-54
 - mail:*mail-summary-template*, 31-54
 - mail:*mail-summary-window-fraction*, 31-54
 - mail:*mail-template*, 31-50
 - mail:*preload-mail-file-p*, 31-52
 - mail:*probe-for-new-mail-p*, 31-53

- mail: *reformat-headers-automatically*, 31-56
 - mail: *reformat-headers-body-goal-column*, 31-57
 - mail: *reformat-headers-case*, 31-57
 - mail: *reformat-headers-exclude-list*, 31-56
 - mail: *reformat-headers-include-list*, 31-56
 - mail: *reply-template*, 31-50
 - mail: *reply-template-1*, 31-50
 - mail: *reply-to-all-header-type*, 31-56
 - mail: *reply-to-template*, 31-50
 - mail: *reply-to-sender-template*, 31-50
 - mail: *resend-template*, 31-50
 - mail: *save-mail-file-in-background*, 31-52
 - mail: *sticky-mail-buffer-selection-p*, 31-54
 - mail: *sticky-mail-window-configuration-p*, 31-54
 - mail: *try-mail-now-p*, 31-53
 - mail: *two-window-reply*, 31-55
 - mail: *unix-inbox-pathname*, 31-53
 - mail: *unsent-message-query-p*, 31-57
 - mail: *upcase-message-keywords-p*, 31-57
 - mail: *user-default-mail-file*, 31-52
 - mail: *user-mail-address*, 31-56
 - mail: *user-mail-reading-mode*, 31-53
 - mail: *yank-message-headers-include-list*, 31-57
 - mail: *yank-message-prefix*, 31-57
 - make-command macro, 7-12
 - MAR (Memory Address Register), 23-1—23-3
 - MAR condition, sys:mar-break, 23-3
 - MAR functions
 - clear-mar, 23-2
 - mar-mode, 23-2
 - set-mar, 23-2
 - sys:mar-break condition, 23-3
 - mar-mode function, 23-2
 - mass storage subsystem crashes, 24-9—24-11
 - menus
 - creating UCL command, 7-15
 - description of UCL command, 6-10
 - Suggestions. *See* Suggestions
 - messages, sending. *See* Converse; Mail
 - metering, 27-2—27-29
 - analyzing metered data, 27-8—27-23
 - Call Tree Inspector, 27-12
 - Call Tree Inspector examples, 27-14
 - controlling metered data, 27-6
 - customized metering sessions, 27-23
 - edit buffer, 27-12
 - evaluating forms with metering, 27-7
 - finding callers, 27-11
 - LIST-EVENTS analyzer, 27-9
 - meter partition, 27-5
 - output file, 27-12
 - output type, 27-11
 - overview, 27-1
 - resuming garbage collection, 27-23
 - setting up a meter partition, 27-5
 - sort function, 27-10
 - TREE analyzer, 27-9
 - typical session, 27-3
 - metering functions
 - meter:analyze, 27-28
 - meter:disable, 27-25
 - meter:enable, 27-24
 - find-process, 27-24
 - meter, 27-7
 - meter-analyze, 27-8—27-23
 - meter:reset, 27-24
 - meter:resume-gc-process, 27-23
 - meter:run, 27-24
 - meter:test, 27-24
 - metering variables
 - sys:%meter-micro-enables, 27-6
 - meter:metered-objects, 27-25
 - method apropos, 25-9
 - meter:modify-histogram function, 27-44
- ## N
- namespace, 32-1—32-71
 - aliases, 32-3
 - chasing, 32-51
 - merging, 32-51
 - attribute, 32-2
 - group, 32-2
 - group attribute operations with user functions, 32-51
 - NSE commands, 32-29
 - retrieving values, 32-68
 - scalar, 32-2
 - attribute list, 32-22
 - basic namespace, 32-7
 - cache, 32-5
 - class, 32-2
 - NSE commands, 32-20
 - client, 32-5
 - co-server, 32-5
 - command summary for the NSE, 32-34
 - comparisons in a namespace, 32-11
 - concepts, 32-1
 - configuring the namespace
 - by using the NSE, 32-15
 - by using the user functions, 32-53
 - cursor movement in the NSE, 32-17
 - default attributes, 32-9
 - personal namespace, 32-9
 - public namespace, 32-10
 - Symbolics namespace, 32-11
 - error messages, 32-71
 - name
 - *variable:*expensive-foreign-lookup-by-properties, 32-7
 - functions, user. *See* namespace user functions
 - general commands in the NSE, 32-19

- namespace(continued)
 - global changes, 32-6
 - group attribute, 32-2
 - NSE commands, 32-32
 - operations with user functions, 32-51
 - retrieving values, 32-68
 - hierarchy in a namespace, 32-2
 - local changes, 32-6
 - lookup operations, 32-62
 - NSE. *See* namespace editor (NSE)
 - name pattern, 32-22
 - namespace editor (NSE), 32-13
 - accessing the, 32-14
 - attribute commands, 32-29
 - basic NSE operations, 32-16
 - class commands, 32-20
 - command summary, 32-34
 - configuring the namespace, 32-15
 - cursor movement commands, 32-17
 - display, 32-13
 - general commands, 32-19
 - group attribute commands, 32-32
 - numeric arguments, 32-18
 - object commands, 32-27
 - symbol codes, 32-18
 - namespace editor (NSE) customization, 32-36—32-50
 - expert editors, 32-39
 - nse:define-nse-expert-editor macro, 32-40
 - defining expert editors, 32-40
 - spec-list formats for nse:define-nse-expert-editor macro, 32-44
 - variables, 32-48
 - verification routine macros, 32-49
 - viewing, 32-50
 - filters, 32-37
 - horizontal class formats, 32-38
 - variables for customization, 32-36
 - namespace operations, 32-11
 - numeric arguments in the NSE, 32-18
 - object, 32-2
 - manipulation functions, 32-68
 - modification functions, 32-58—32-62
 - NSE commands, 32-27
 - qualified name, 32-4
 - relative name, 32-4
 - retrieval functions, 32-62—32-68
 - retrieving objects from a namespace, 32-4
 - unqualified name, 32-4
 - pathnames, 32-8
 - personal namespace, 32-6
 - default attributes, 32-9
 - pathnames, 32-8
 - property list, 32-69
 - public namespace, 32-5
 - default attributes, 32-10
 - pathnames, 32-8
 - qualified name, 32-4
 - relative name, 32-4
 - retrieving objects from a namespace, 32-4
 - scalar attribute, 32-2
 - NSE commands, 32-29
 - retrieving values, 32-68
 - search list, 32-4
 - search rules, 32-4
 - server, 32-5
 - symbol codes in the NSE, 32-18
 - Symbolics namespace, 32-6
 - default attributes, 32-11
 - test predicate, 32-23
 - types of namespaces, 32-5
 - basic, 32-7
 - personal, 32-6
 - public, 32-5
 - Symbolics, 32-6
 - unqualified object name, 32-4
 - user functions
 - See also* namespace user functions
 - group attribute operations, 32-51
 - miscellaneous functions, 32-69
 - cache refreshing, 32-69
 - property list, 32-69
 - modification functions, 32-58
 - name argument, 32-51
 - namespace argument, 32-51
 - namespace functions, 32-53
 - object manipulation functions, 32-68
 - retrieval functions, 32-62
 - list operations, 32-62
 - lookup operations, 32-62
- namespace editor (NSE) commands
 - Add Alias for Object (META-CTRL-A), 32-27
 - Add Attribute (CTRL-A), 32-27
 - Add Attribute to Current Object (CTRL-A), 32-31
 - Add Group Attribute (META-X), 32-27
 - Add Group Attribute to Current Object (META-X), 32-31
 - Add Group Member (META-A), 32-32
 - Add Key for Group Attribute (META-X), 32-32
 - Add Object (A), 32-19
 - Copy Attribute (CTRL-C), 32-30
 - Copy Attribute to This Namespace (C), 32-32
 - Copy Class to This Namespace (C), 32-26
 - Copy Object (CTRL-C), 32-28
 - Copy Object to This Namespace (C), 32-29
 - Copy Objects in Class (CTRL-C), 32-25
 - Create Public Horizontal Format (META-X), 32-38

- Delete Aliases for Object (META-CTRL-D), 32-28
- Delete Attribute (D), 32-30
- Delete Group Member (META-D), 32-32
- Delete Object (D), 32-27
- Distribute Namespace (META-X), 32-20
- Edit Attribute (E), 32-29
- Edit Group Member (E), 32-32
- Edit Internal Namespace (META-X), 32-50
- Expand All Objects in Class (META-S), 32-20
- Expand All Objects in NSE (META-CTRL-S), 32-19
- Expand Horizontally (CTRL-H), 32-24
- Expand/Unexpand Class (S), 32-20
- Expand/Unexpand Object (S), 32-27
- Find Changed Objects (META-X), 32-23
- Find Non-Alias Objects (META-F), 32-21
- Find Object (F), 32-21
- Find Object and Aliases (CTRL-F), 32-21
- Find Objects With Specific Properties (META-CTRL-F), 32-22
- Revert Attribute to Local State (R), 32-31
- Revert Class to Global State (META-R), 32-26
- Revert Class to Local State (R), 32-26
- Revert NSE to Local State (CTRL-X CTRL-R), 32-19
- Revert Object to Global State (META-R), 32-29
- Revert Object to Local State (R), 32-28
- Show Documentation (CTRL-SHIFT-D), 32-19
- Toggle Group Status (G), 32-30
- Toggle Horizontal Display (H), 32-24
- Undelete Attribute (U), 32-30
- Undelete Object (U), 32-27
- Unexpand All Objects in Class (META-X), 32-21
- Unexpand All Objects in NSE (META-X), 32-19
- Unexpand Current Object (S), 32-31
- Update Namespace Globally (CTRL-X CTRL-W), 32-20
- Update Namespace Locally (CTRL-X CTRL-S), 32-19
- Verify Attribute Incrementally (META-X), 32-31
- Verify Class Incrementally (META-X), 32-26
- Verify Namespace (META-X), 32-20
- Verify Object Incrementally (META-X), 32-28
- View Attribute (V), 32-30
- Write NSE Buffer to File (META-X), 32-20
- Write (Update) Attribute Globally (CTRL-W), 32-31
- Write (Update) Attribute Locally (W), 32-30
- Write (Update) Class Globally (CTRL-W), 32-26
- Write (Update) Class Locally (W), 32-25
- Write (Update) Object Globally (CTRL-W), 32-28
- Write (Update) Object Locally (W), 32-28
- namespace editor (NSE) functions
 - nse:define-nse-expert-editor, 32-40
 - nse:define-personal-filter, 32-38
 - nse:define-personal-horizontal-format, 32-38
 - nse:verify-err, 32-49
 - nse:verify-wrn, 32-49
- namespace editor (NSE) variables
 - nse:*attribute*, 32-48
 - nse:*buffer*, 32-48
 - nse:*group-member-value*, 32-48
 - nse:*group-members-to-delete*, 32-48
 - nse:*local*, 32-49
 - nse:*nse-fonts*, 32-36
 - nse:*object*, 32-48
 - nse:*old-group-member-value*, 32-48
 - nse:*old-value*, 32-48
 - nse:*operation*, 32-49
 - nse:*personal-filter-list*, 32-37
 - nse:*personal-horizontal-format-list*, 32-37
 - nse:*prompt-in-mini-buffer-p*, 32-36
 - nse:*stream*, 32-48
 - nse:*truncate-attribute-lines-nicely*, 32-37
 - nse:*value*, 32-48
 - nse:*verification-level*, 32-49
- namespace user functions
 - name:add-alias, 32-62
 - name:add-attribute, 32-60
 - name:add-group-member, 32-61
 - name:add-namespace, 32-55
 - name:add-object, 32-58
 - name:configure-namespace, 32-55
 - name:copy-namespace, 32-54
 - delete alias. *See* name:add-alias
 - name:delete-attribute, 32-60
 - name:delete-group-member, 32-61
 - name:delete-namespace, 32-58
 - name:delete-object, 32-59
 - name:distribute-namespace, 32-70
 - name:flush-namespace, 32-70
 - name:force-local-server-boot, 32-71
 - name:foreign-namespace, 32-70
 - name:format-objects, 32-68
 - name:get-attribute-list, 32-68
 - name:get-attribute-value, 32-68
 - name:get-hidden-property, 32-69
 - name:initialize-name-service, 32-71
 - name:list-known-namespaces, 32-53
 - name:list-namespace-search-rules, 32-53
 - name:list-object, 32-63

- namespace user functions (continued)
 - name:list-object-and-aliases, 32-67
 - name:list-objects-from-properties, 32-66
 - name:load-personal-namespace, 32-54
 - name:lookup-attribute-value, 32-64
 - name:lookup-object, 32-62
 - name:lookup-object-and-aliases, 32-66
 - name:lookup-objects-from-properties, 32-64
 - name:namespace-classes, 32-68
 - name:namespace-has-cache, 32-69
 - name:namespace-summary, 32-67
 - name:put-hidden-property, 32-69
 - name:refresh-cache, 32-69
 - name:refresh-cached-object, 32-70
 - name:show-namespace-configuration, 32-53
 - name:universal-list-objects-from-properties, 32-66
 - name:universal-lookup-objects-from-properties, 32-66
- network functions, 33-1—33-4
 - chaos:eval-server-on, 33-2
 - chaos:find-hosts-or-lispms-logged-in-as-user, 33-3
 - finger, 33-3
 - net:host-status, 33-1
 - chaos:notify, 33-4
 - chaos:notify-all-lms, 33-4
 - net:*poll-each-status-p* variable, 33-2
 - print-notifications, 33-4
 - chaos:remote-eval, 33-2
 - net:reset, 33-2
 - chaos:shout, 33-4
- network tools
 - Converse, 30-1—30-5
 - database, namespace, 32-1—32-71
 - Eval server, 33-2
 - fingering hosts, 33-3
 - host status checking, 33-1
 - Mail++%%, 31-1—31-57
 - namespace utilities, 32-1—32-71
 - notifications, sending and printing, 33-4
 - Peek, 17-14
 - remote terminal
 - Eval server, 33-2
 - Telnet, 28-1—28-4
 - VT100 emulator, 29-1
 - resetting the network, 33-2
 - Telnet, 28-1—28-4
 - VT100 emulator, 29-1—29-3
- New User utility, 1-1—1-2
 - new-user function, 1-1
- notifications, sending and printing, 33-4
 - chaos:notify, 33-4
- NuBus crashes, 24-8
- numeric pad menu of the graphics editor, 10-14
- NUPI
 - device and controller error crashes, 24-11—24-13
 - special event crashes, 24-14
- NVRAM, 24-2
- O**
 - online documentation. *See* Visidoc
 - optimizing code, performance tools, 27-1—27-45
- P**
 - package, describe, 25-5
 - partition, describe, 25-5
 - peek function, 17-1
 - Peek, 17-1—17-18
 - Areas (A), 17-7
 - Counters (C), 17-6
 - Documentation (CTRL-HELP), 17-3
 - Exit (END), 17-3
 - File Status (F), 17-8
 - Function Histogram (M), 17-16
 - Host Status (H), 17-18
 - mode and command menu windows, 17-3
 - modes, 17-1
 - Network (N), 17-14
 - peek function, 17-1
 - Processes (P), 17-4
 - Servers (S), 17-12
 - Set Timeout (T), 17-3
 - viewing window, 17-3
 - Windows (W), 17-10
 - performance tools, 27-1—27-45
 - function histogram, 27-41—27-45
 - metering, 27-2—27-29
 - timing macros, 27-30—27-40
 - uci:pop-up-command-menu function, 7-36
 - power fail crash, 24-9
 - mail:preload-mail-file function, 31-51
 - print functions
 - pprint, 25-15
 - pprint-def, 25-16
 - prin1, 25-15
 - princ, 25-15
 - print, 25-15
 - print variables
 - *print-base*, 25-14
 - *print-escape*, 25-14
 - *print-pretty*, 25-14
 - *print-radix*, 25-14
 - mail:Print-mail-queue function, 31-46
 - processes, peek, 17-4
 - processor fault crashes, 24-8
 - Profile utility, 2-1—2-5
 - accessing the, 2-1
 - accessing variables in the, 2-2
 - commands, 2-3

- customizing the, 2-4
- requirements, 2-1
- variables, defining Profile, 2-4
- Profile utility functions
 - profile:define-profile-variable, 2-4
 - profile, 2-1
 - profile:profile-setq, 2-5
- property list functions
 - fs:file-properties, 25-13
 - symbol-plist, 25-13

Q

- qsend function, 30-3
- qsends-off function, 30-3
- qsends-on function, 30-3

R

- read-eval-print loop, 26-1
 - Eval server, 33-2
- read-for-top-level function, 6-19
- region, describe, 25-6
- registers in the font editor, 12-7
- remote terminal
 - Eval server, 33-2
 - Telnet, 28-1—28-4
 - VT100 emulator, 29-1
- mail:remove-mail-inbox-probe function, 31-52
- zwei:reply function, 30-4
- report-all-shutdowns function, 24-3
- meter:report-histogram function, 27-44
- report-last-shutdown function, 24-2
- mail:reset-mail-daemon function, 31-46
- resetting the network, 33-2
- resource apropos, 25-10
- meter:restore-histogram function, 27-45
- roul function, 6-23
- rubber banding to select graphic objects, 10-13

S

- System Menu, UCL help Option, 6-4
- meter:save-histogram function, 27-45
- selected font, 12-2
- sending messages. *See* Converse; Mail
- servers, monitoring with Peek, 17-12
- ucl:set-active-command-tables function, 9-15
- set-mar function, 23-2
- chaos:shout function, 33-4
- software crash descriptions, 24-15—24-18
- meter:start-histogram function, 27-44
- Stepper, 19-1—19-3
 - automatic stepping, 19-2
 - commands, 19-3
 - sys:"step-auto" variable, 19-2
 - symbols, 19-1
- Stepper functions
 - step, 19-1
 - sys:step-auto-off, 19-2

- sys:step-auto-on, 19-2
- meter:stop-histogram function, 27-44
- mail:submit-mail function, 31-51
- Suggestions, 8-1—8-20, 9-1—9-15
 - accessing, 8-2
 - active command table to change menus, using the, 9-15
 - Add a Symbol to a Lisp Expressions Menu, 8-19
 - Add Menu to Buffer, 8-18
 - advising commands to change menus, 9-14
 - Back to Initial Suggestions, 8-18
 - break and Lisp Listener, 8-9
 - command context switch, 9-10
 - debugger, 8-15
 - Display Recent Deletions, 8-9
 - evaluation of a symbol using Listener Suggestions, 8-8
 - Find Commands, 8-18
 - Find Functions for Menu, 8-19
 - incorporating Suggestions in an application, 9-1
 - initializing, 9-9
 - inline macros to change menus, using, 9-12
 - Inspector, 8-14
 - landscape video display, 8-4
 - Lisp expressions, 8-19
 - Lisp Listener and break, 8-9
 - List Apropos Completions, 8-11
 - Menu History, 8-18
 - Menu Tools, 8-17
 - menus, building, 9-8
 - panes, 8-5
 - pop-up keystrokes, 8-19
 - portrait video display, 8-4
 - programming, 9-1—9-15
 - Re-Order Menu Items, 8-19
 - Remove Current Menu, 8-19
 - Select Suggestions Applications, 8-18
 - Suggestions Menu Search, 8-18
 - Suggestions Menus Off, 8-18
 - triggering automatic menu changes, 9-10
 - advising commands, 9-14
 - using inline macros, 9-12
 - using the active command table, 9-15
 - Turn Off Pop-Up Keystrokes, 8-19
 - Turn On Pop-Up Keystrokes, 8-19
 - using Suggestions, 8-1—8-20
 - Zmacs, 8-12
- Suggestions functions
 - sugg:advise-function-to-push-all-menus, 9-14
 - sugg:advise-function-to-push-one-menu, 9-14
 - sugg:declare-suggestions-for, 9-12
 - sugg:initialize-suggestions-for-application, 9-9
 - ucl:set-active-command-tables, 9-15
 - sugg:suggestions-build-menu, 9-8
 - sugg:undeclare-suggestions-for, 9-12
 - sugg:with-suggestions-menus-for, 9-13

Suggestions variables

- sugg:function-wrapped-functions, 9-14
- sugg:suggestions-advised-functions, 9-14
- SYMBOL-HELP, display produced by, 12-11
- System Application, UCL help option, 6-4
- system, describe, 25-4

T

Telnet, 28-1—28-4

- commands, 28-3
- server, 28-4

Telnet functions

- telnet, 28-1
- telnet::telnet-server-on, 28-4

TERM 0 S, 14-4

TERM Key Help, UCL help option, 6-4

thickness of an edge of a graphics object, 10-9

time macro, 27-39

timing macros, 27-30—27-40

- time:define-meter, 27-38
- time:fixnum-microsecond-time, 27-40
- time:microsecond-time, 27-40
- time:microsecond-time-difference, 27-40
- overview, 27-1
- time, 27-39
- timeit, 27-31
- time:*timeit-defaults* variable, 27-34
- time:timeit-report, 27-36

trace utility, 18-1—18-6

trace utility functions

- trace, 18-2
- untrace, 18-6

trace utility variables

- trace-compile-flag, 18-6
- *trace-output*, 18-5

tree editor, 11-1—11-15

- changing how a node is drawn, 11-14
- display, 11-2
- displaying error messages, 11-13
- editing methods. *See* tree editor editing methods
- expanding and contracting nodes, 11-12
- flavor displayer, sample interface, 11-4
- formatting for the scroll window, 11-12
- functions. *See* tree editor functions
- global variables, 11-14
- loading the tree editor software, 11-3
- local variables, 11-14
- panning and zooming, 11-13
- redrawing the tree, 11-12
- running the tree editor, 11-4
- sample interfaces, 11-3
 - flavor displayer, 11-4
 - string displayer, 11-3
- starter kit, 11-3
- string displayer, sample interface, 11-3
- tree editor accessors, 11-5—11-10
 - adding to the list of flavors, 11-5

building a displayable tree, 11-7—11-12

- defining the flavor, 11-7
- defining the function of mouse buttons, 11-6
- defining the methods, 11-7—11-12
 - :children-from-data, 11-7
 - :find-type, 11-8
 - :first-node, 11-7
 - :font-type, 11-8
 - :get-new-tree, 11-10
 - :handle-node, 11-9
 - :highlight-function, 11-8
 - :print-name, 11-8

tree editor editing methods

- :add-brother-node method, 11-11
- :add-node-after, 11-11
- :add-node-before, 11-10
- :delete-subtree, 11-11
- :delete-yourself, 11-11
- :get-user-data, 11-11

tree editor functions

- tree:complain, 11-13
- tree:contract-node-with-redraw, 11-13
- tree:display, 11-4
- tree:expand-contract-with-redraw, 11-13
- tree:expand-node-with-redraw, 11-12
- tree:fill-window, 11-13
- tree:grind-item, 11-12
- tree:move-to-front, 11-13
- tree:pan-window, 11-13
- tree:return-to-default-window, 11-13
- tree:string-item, 11-12
- tree:tree-draw-after-small-changes, 11-12
- tree:tree-redraw, 11-12
- tree:update-node, 11-14
- tree:zoom-window, 11-13

tree editor variables

- tree:*default-adjust-to-sup-size*, 11-15
- tree:*default-application-type*, 11-15
- tree:*default-init-label*, 11-15
- tree:*default-vertical*, 11-15
- tree:*force-recalculate*, 11-14
- tree:*known-application-types*, 11-15
- tree:*max-level*, 11-14
- tree:*minimum-breadth-spacing*, 11-15
- tree:*minimum-depth-spacing*, 11-15
- tree:*root-node*, 11-14
- tree:*scroll-window-height*, 11-15
- tree:*scroll-window-width*, 11-15
- tree:*starting-point-offset*, 11-15
- tree:*tree*, 11-14
- tree:*tree-window*, 11-14
- tree:*truncation-for-scroll-window*, 11-15
- tree:*vertical?, 11-14

U

unbreakon function, 22-1

- Universal Command Loop (UCL), 6-1—6-26, 7-1—7-38

- application development hints, 7-19
- Application Help option, 6-4
- Build Command Macro, 6-18
- Build Keystroke Macro, 6-17
- columns in menus, 7-16
- command context switch, 7-21
- Command Display, 6-7
- Command Editor, 6-16
- Command History, 6-8
- command interpreter
 - basic operation, 7-3
 - flavors, using and customizing, 7-20
 - See also* Universal Command Loop (UCL) flavors
 - global variables, 7-34
- command macros, 6-18
- command menus, 6-10
 - icons, 6-10
 - mouse documentation window, 6-10
- Command Name Search, 6-9
- command names, typed expressions, 6-22
- command summary, 6-26
- command tables
 - active, 7-21
 - all, 7-21
- Command Type-In Help, 6-6
- completion commands, 6-11
- constraint frame, 7-21
- creating commands, 7-5
- development hints for a UCL application, 7-19
- environment customization features, 6-15–6-21
- error catcher, 6-25
- Explorer Overview, 6-4
- flavors, command interpreter, 7-20
 - See also* Universal Command Loop (UCL) flavors
- functions. *See* Universal Command Loop (UCL) functions
- functions, typed expressions, 6-22
- general help options, 6-4
- global variables, command interpreter, 7-34
- Help command, 6-4
- help features, 6-3–6-14
- icons, command menus, 6-10
- implicit message sending (rotl), typed expressions, 6-23
- implicit-paren-functions, typed expressions, 6-22
- input, types of user, 6-2
- instance variables for command interpreter
 - See also* Universal Command Loop (UCL) flavors
 - basic instance variables, 7-21
 - instance variables for printing, 7-24
 - instance variables for reading typed input, 7-23
- keystroke macros, 6-17
- Keystroke Search, 6-10
- Load Commands, 6-19
- making commands, 7-5
- methods for command interpreter. *See* Universal Command Loop (UCL) flavors
- miscellaneous features, 6-21
- miscellaneous functions, 7-36
- mouse documentation window
 - command menus, 6-10
 - help, 6-13
- multicolumn menu, 7-16
- numeric arguments, 6-25
- obtaining arguments, 6-24
- pathname-completion, typed expressions, 6-22
- programmer interface, 7-1–7-38
- Redo command (HYPER-CTRL-R), 6-25
- removing UCL features, 7-31
- Save Commands, 6-19
- setting default UCL options, 6-19
- special expressions, typed expressions, 6-24
- starter kit, 7-25
- symbols, typed expressions, 6-22
- System Application, 6-4
- System Menu
 - help option, 6-4
 - universal command on right mouse button, 6-25
- TERM Key Help, 6-4
- Top Level Configurer, 6-19
- Tutorial option, 6-4
- type-in modes, user configuration of, 6-24
- typed expressions, 6-22
 - algorithm used for typed expressions, 6-23
 - command-names, 6-22
 - functions, 6-22
 - implicit message sending (rotl), 6-23
 - implicit-paren-functions, 6-22
 - kinds of, 6-22
 - pathname-completion, 6-22
 - special expressions, 6-24
 - symbols, 6-22
 - user configuration of type-in modes, 6-24
- universal commands, 6-1
- user interface, 6-1–6-26
- Universal Command Loop (UCL) flavors
 - ucl:basic-command-loop, 7-20
 - ucl:basic-command-loop instance variables
 - ucl:active-command-tables, 7-21
 - ucl:all-command-tables, 7-21
 - ucl:basic-help, 7-21
 - ucl:blip-alist, 7-22
 - ucl:command-entry, 7-22
 - ucl:command-execution-queue, 7-23

Universal Command Loop (UCL) flavors (continued)

- ucl:command-history, 7-23
- ucl:inhibit-results-print?, 7-25
- ucl:input-mechanism, 7-22
- ucl:kbd-input, 7-22
- ucl:max-command-history, 7-23
- ucl:max-output-history, 7-25
- ucl:menu-panes, 7-21
- ucl:numeric-argument, 7-23
- ucl:output-history, 7-25
- ucl:print-function, 7-24
- ucl:print-results?, 7-24
- ucl:prompt, 7-24
- ucl:read-function, 7-24
- ucl:read-type, 7-24
- ucl:tutorial, 7-21
- ucl:typein-handler, 7-23
- ucl:typein-modes, 7-23
- ucl:basic-command-loop methods
 - :command-loop, 7-25
 - :designate-io-streams, 7-25
 - :execute-command, 7-26
 - :fetch-and-execute, 7-26
 - :fetch-input, 7-26
 - :handle-key-input, 7-26
 - :handle-menu-input, 7-26
 - :handle-mouse-input, 7-26
 - :handle-pop-up-typein-and-typeout, 7-26
 - :handle-pop-up-typein-input, 7-26
 - :handle-typein-input, 7-26
 - :handle-unknown-input, 7-27
 - :initialize, 7-25
 - :loop, 7-25
 - :quit, 7-27
- ucl:command, 7-5
- ucl:command-and-lisp-typein-window, 7-28
- ucl:command-loop-mixin, 7-27
 - :process-options initialization option, 7-27
 - :quit method, 7-27
- ucl:command-table, 7-13
- ucl:selective-features-mixin, 7-31
 - :remove-features initialization option, 7-31
- ucl:temporary-command-table, 7-37
- ucl:typein-mode, 7-29
- ucl:typein-mode instance variables
 - ucl:auto-complete-p, 7-30
 - ucl:description, 7-30
 - ucl:documentation, 7-30
- ucl:typein-mode methods
 - :arglist, 7-30
 - :complete, 7-30
 - :complete-p, 7-29
 - :execute, 7-29

:handle-typein-p, 7-29

:help-doc, 7-30

Universal Command Loop (UCL) functions

- build-command-table, 7-13
- build-menu, 7-15
- ucl:build-temporary-command-table, 7-37
- ucl:deallocate-temporary-command-table, 7-38
- defcommand, 7-5
 - :arguments keyword, 7-9
- ucl:display-some-commands, 7-36
- ucl:get-command, 7-36
- ucl:help-menu, 7-38
- ucl:looping-through-command-tables, 7-38
- make-command, 7-12
- ucl:pop-up-command-menu, 7-36
- read-for-top-level, 6-19
- ucl:read-for-ucl, 6-19
- rotl, 6-23
- view-documentation, 7-37
- Universal Command Loop (UCL) variables
 - ucl:*default-max-command-history*, 7-35
 - ucl:*default-max-output-history*, 7-35
 - ucl:*default-print-function*, 7-35
 - ucl:*default-prompt*, 7-34
 - ucl:*default-read-function*, 7-35
 - ucl:*default-typein-modes*, 7-34
- instance variables. *See* Universal Command Loop (UCL) flavors
- ucl:preempting?, 7-35
- ucl:this-application, 7-35
- untrace macro, 18-6

V

- variable apropos, 25-9
- variable-width fonts, 12-2
- vertical spacing (vsp), 12-3
- view-documentation function, 7-37
- Visidoc, 35-1—35-9
 - dox:boot-visidoc-server function, 35-7
 - display characteristics, 35-3
 - exiting Visidoc, 35-2
 - features, 35-3
 - Hypertext, 35-4
 - dox:initialize-visidoc-server function, 35-7
 - installing the Visidoc client software, 35-2
 - invoking Visidoc, 35-2
 - maintenance of the Visidoc server
 - namespace, 35-8
 - making a host a Visidoc server, 35-5
 - memory characteristics, 35-3
 - server, 35-5
 - using Visidoc, 35-3
- Visual Interactive Documentation (Visidoc) Online Manual Viewer. *See* Visidoc
- VT100 emulator, 29-1—29-3

commands, 29-3

W

who-calls functions

 find-all-symbols, 25-12

 what-files-call, 25-12

 where-is, 25-12

 who-calls, 25-11

window-based debugger, 14-1—14-5

 commands, 14-3

 deexposed windows and background
 processes, 14-4

 eh: *enter-window-debugger* variable, 14-1

 panes, 14-1

 using the, 14-1

windows, Peek, 17-10

with-timeout macro, 3-5

world coordinates for graphics, 10-2

Z

Zmacs customizations, login-init file, 3-4

Flavors**B**

ucl: basic-command-loop, 7-20

C

ucl: command-and-lisp-typein-window, 7-28

ucl: command-loop-mixin, 7-27

I

zwei: interval-stream, 31-48

M

mail: message, 31-47

S

ucl: selective-features-mixin, 7-31

T

ucl: temporary-command-table, 7-37

ucl: typein-mode, 7-29

Functions
A

name: add-alias, 32-62
 name: add-attribute, 32-60
 name: add-group-member, 32-61
 mail: add-mail-inbox-probe, 31-52
 name: add-namespace, 32-55
 name: add-object, 32-58
 advise, 21-1
 sugg: advise-function-to-push-all-menus, 9-14
 sugg: advise-function-to-push-one-menu, 9-14
 advise-within, 21-5
 meter: analyze, 27-28
 applyhook, 20-1
 apropos, 25-7
 apropos-flavor, 25-9
 apropos-list, 25-8
 apropos-method, 25-9
 apropos-resource, 25-10
 aproposb, 25-9
 aproposf, 25-8

B

dox: boot-visidoc-server, 35-7
 break, 26-1
 breakon, 22-1
 bug, 4-1
 build-command-table, 7-13
 build-menu, 7-15
 ucl: build-temporary-command-table, 7-37

C

clear-mar, 23-2
 color: cme, 34-2
 tree: complain, 11-13
 name: configure-namespace, 32-55
 tree: contract-node-with-redraw, 11-13
 name: copy-namespace, 32-54

D

ucl: deallocate-temporary-command-table, 7-38
 sys: debug-warm-booted-process, 13-11
 sugg: declare-suggestions-for, 9-12
 defcommand, 7-5
 define-glossary, 5-12
 define-glossary-file-format, 5-11
 mail: define-mail-filter, 31-48
 mail: define-mail-template, 31-49
 time: define-meter, 27-38

nse: define-nse-expert-editor, 32-40
 nse: define-personal-filter, 32-38
 nse: define-personal-horizontal-format, 32-38
 profile: define-profile-variable, 2-4
 delete alias. *See* name:add-alias
 name: delete-attribute, 32-60
 name: delete-group-member, 32-61
 name: delete-namespace, 32-58
 name: delete-object, 32-59
 describe, 25-2
 describe-area, 25-6
 describe-defstruct, 25-2
 describe-flavor, 25-3
 describe-package, 25-5
 sys: describe-partition, 25-5
 describe-region, 25-6
 describe-system, 25-4
 meter: disable, 27-25
 tree: display, 11-4
 ucl: display-some-commands, 7-36
 name: distribute-namespace, 32-70
 documentation, 25-18
 dribble, 25-17
 dribble-all, 25-17
 dribble-end, 25-17
 dribble-start, 25-17

E

eh, 13-2
 eh-arg, 13-8
 eh-fun, 13-9
 eh-loc, 13-9
 eh-val, 13-9
 meter: enable, 27-24
 chaos: eval-server-on, 33-2
 evalhook, 20-1
 tree: expand-contract-with-redraw, 11-13
 tree: expand-node-with-redraw, 11-12

F

fed, 12-12
 fs: file-properties, 25-13
 tree: fill-window, 11-13
 find-all-symbols, 25-12
 chaos: find-hosts-or-lispms-logged-in-as-user, 33-3
 find-process, 27-24
 finger, 33-3
 time: fixnum-microsecond-time, 27-40
 name: flush-namespace, 32-70
 name: force-local-server-boot, 32-71

name: foreign-namespace, 32-70
 name: format-objects, 32-68

G

ged: ged, 10-4
 name: get-attribute-list, 32-68
 name: get-attribute-value, 32-68
 ucl: get-command, 7-36
 name: get-hidden-property, 32-69
 gloss: glossary, 5-2
 tree: grind-item, 11-12

H

ucl: help-menu, 7-38
 net: host-status, 33-1

I

identity, 25-20
 name:in itialize-name-service, 32-71
 sugg: initialize-suggestions-for-application,
 9-9
 dox: initialize-visidoc-server, 35-7
 mail: insert-address-list, 31-51
 mail: insert-default-header-fields, 31-50
 mail: insert-header-field, 31-50
 inspect, 15-1
 inspect*, 15-1
 inspect-flavor, 16-1

L

lisp-implementation-type, 25-19
 lisp-implementation-version, 25-19
 sys: lisp-reinitialize, 26-1
 sys: lisp-top-level, 26-1
 sys: lisp-top-level1, 26-1
 name: list-known-namespaces, 32-53
 name: list-namespace-search-rules, 32-53
 name: list-object, 32-63
 name: list-object-and-aliases, 32-67
 name: list-objects-from-properties, 32-66
 name: load-personal-namespace, 32-54
 login, details on login-init file, 3-1
 login-eval, 3-3
 login-fdefine, 3-3
 login-forms, 3-2
 login-setq, 3-2
 long-site-name, 25-20
 name: lookup-attribute-value, 32-64
 name: lookup-object, 32-62
 name: lookup-object-and-aliases, 32-66
 name: lookup-objects-from-properties,
 32-64
 ucl: looping-through-command-tables,
 7-38

M

machine-instance, 25-19
 machine-type, 25-19
 machine-version, 25-19
 make-command, 7-12
 mar-mode, 23-2
 meter, 27-7
 meter-analyze, 27-8—27-23
 time: microsecond-time, 27-40
 time: microsecond-time-difference, 27-40
 meter: modify-histogram, 27-44
 tree: move-to-front, 11-13

N

name: namespace-classes, 32-68
 name: namespace-has-cache, 32-69
 name: namespace-summary, 32-67
 new-user, 1-1
 chaos: notify, 33-4
 chaos: notify-all-lms, 33-4

P

tree: pan-window, 11-13
 peek, 17-1
 ucl: pop-up-command-menu, 7-36
 pprint, 25-15
 pprint-def, 25-16
 mail: preload-mail-file, 31-51
 prin1, 25-15
 princ, 25-15
 print, 25-15
 mail: print-mail-queue, 31-46
 print-notifications, 33-4
 profile, 2-1
 profile: profile-setq, 2-5
 name: put-hidden-property, 32-69

Q

qsend, 30-3
 qsend-off, 30-3
 qsend-on, 30-3

R

read-for-top-level, 6-19
 ucl: read-for-ucl, 6-19
 name: refresh-cache, 32-69
 name: refresh-cached-object, 32-70
 chaos: remote-eval, 33-2
 mail: remove-mail-inbox-probe, 31-52
 zwei: reply, 30-4
 report-all-shutdowns, 24-3
 meter: report-histogram, 27-44
 report-last-shutdown, 24-2

meter: reset, 27-24
 net: reset, 33-2
 mail: reset-mail-daemon, 31-46
 meter: restore-histogram, 27-45
 meter: resume-gc-process, 27-23
 tree: return-to-default-window, 11-13
 rotl, 6-23
 meter: run, 27-24

S

meter: save-histogram, 27-45
 ucl: set-active-command-tables, 9-15
 set-mar, 23-2
 short-site-name, 25-20
 chaos: shout, 33-4
 name: show-namespace-configuration,
 32-53
 software-type, 25-20
 software-version, 25-20
 meter: start-histogram, 27-44
 step, 19-1
 sys: step-auto-off, 19-2
 sys: step-auto-on, 19-2
 meter: stop-histogram, 27-44
 tree: string-item, 11-12
 sub-apropos, 25-10
 mail: submit-mail, 31-51
 sugg: suggestions-build-menu, 9-8
 symbol-plist, 25-13

T

telnet, 28-1
 telnet: telnet-server-on, 28-4
 meter: test, 27-24
 time, 27-39

timeit, 27-31
 time: timeit-report, 27-36
 trace, 18-2
 tree: tree-draw-after-small-changes, 11-12
 tree: tree-redraw, 11-12

U

unadvise, 21-2
 unadvise-within, 21-5
 unbreakon, 22-1
 sugg: undeclare-suggestions-for, 9-12
 name: universal-list-objects-from-properties,
 32-66
 name: universal-lookup-objects-from-properties,
 32-66
 untrace, 18-6
 tree: update-node, 11-14
 user-name, 25-19

V

nse: verify-err, 32-49
 nse: verify-wrn, 32-49
 view-documentation, 7-37

W

what-files-call, 25-12
 where-is, 25-12
 who-calls, 25-11
 sugg: with-suggestions-menus-for, 9-13
 with-timeout, 3-5

Z

tree :zoom-window, 11-13

Instance Variables

A

- ucl: active-command-tables instance variable of ucl:basic-command-loop, 7-21
- ucl: all-command-tables instance variable of ucl:basic-command-loop, 7-21
- ucl: auto-complete-p instance variable of ucl:typein-mode, 7-30

B

- ucl: basic-help instance variable of ucl:basic-command-loop, 7-21
- ucl: blip-alist instance variable of ucl:basic-command-loop, 7-22

C

- ucl: command-entry instance variable of ucl:basic-command-loop, 7-22
- ucl: command-execution-queue instance variable of ucl:basic-command-loop, 7-23
- ucl: command-history instance variable of ucl:basic-command-loop, 7-23

D

- ucl: description instance variable of ucl:typein-mode, 7-30
- ucl: documentation instance variable of ucl:typein-mode, 7-30

I

- ucl: inhibit-results-print? instance variable of ucl:basic-command-loop, 7-25
- ucl: input-mechanism instance variable of ucl:basic-command-loop, 7-22

K

- ucl: kbd-input instance variable of ucl:basic-command-loop, 7-22

M

- ucl: max-command-history instance variable of ucl:basic-command-loop, 7-23
- ucl: max-output-history instance variable of ucl:basic-command-loop, 7-25
- ucl: menu-panes instance variable of ucl:basic-command-loop, 7-21

N

- ucl: numeric-argument instance variable of ucl:basic-command-loop, 7-23

O

- ucl: output-history instance variable of ucl:basic-command-loop, 7-25

P

- ucl: print-function instance variable of ucl:basic-command-loop, 7-24
- ucl: print-results? instance variable of ucl:basic-command-loop, 7-24
- ucl: prompt instance variable of ucl:basic-command-loop, 7-24

R

- ucl: read-function instance variable of ucl:basic-command-loop, 7-24
- ucl: read-type instance variable of ucl:basic-command-loop, 7-24

T

- ucl: tutorial instance variable of ucl:basic-command-loop, 7-21
- ucl: typein-handler instance variable of ucl:basic-command-loop, 7-23
- ucl: typein-modes instance variable of ucl:basic-command-loop, 7-23

Operations

A

:add-brother-node method of user-defined tree editor flavor, 11-11
:add-node-after method of user-defined tree editor flavor, 11-11
:add-node-before method of user-defined tree editor flavor, 11-10
:arglist method of ucl:typein-mode flavor, 7-30
:attributes method of mail:message, 31-47

C

:children-from-data method of user-defined tree editor flavor, 11-7
:command-loop method of ucl:basic-command-loop, 7-25
:complete method of ucl:typein-mode flavor, 7-30
:complete-p method of ucl:typein-mode flavor, 7-29

D

:delete-subtree method of user-defined tree editor flavor, 11-11
:delete-yourself method of user-defined tree editor flavor, 11-11
:designate-io-streams method of ucl:basic-command-loop, 7-25

E

:execute method of ucl:typein-mode flavor, 7-29
:execute-command method of ucl:basic-command-loop, 7-26

F

:fetch-and-execute method of ucl:basic-command-loop, 7-26
:fetch-input method of ucl:basic-command-loop, 7-26
:find-type method of user-defined tree editor flavor, 11-8
:first-bp method of mail:message, 31-47
:first-node method of user-defined tree editor flavor, 11-7
:font-type method of user-defined tree editor flavor, 11-8

G

:get-new-tree method of user-defined tree editor flavor, 11-10
:get-user-data method of user-defined tree editor flavor, 11-11

H

:handle-key-input method of ucl:basic-command-loop, 7-26
:handle-menu-input method of ucl:basic-command-loop, 7-26
:handle-mouse-input method of ucl:basic-command-loop, 7-26
:handle-node method of user-defined tree editor flavor, 11-9
:handle-pop-up-typein-and-typeout method of ucl:basic-command-loop, 7-26
:handle-pop-up-typein-input method of ucl:basic-command-loop, 7-26
:handle-typein-input method of ucl:basic-command-loop, 7-26
:handle-typein-p method of ucl:typein-mode flavor, 7-29
:handle-unknown-input method of ucl:basic-command-loop, 7-27
:headers-end-bp method of mail:message, 31-47
:help-doc method of ucl:typein-mode flavor, 7-30
:highlight-function method of user-defined tree editor flavor, 11-8

I

:initialize method of ucl:basic-command-loop, 7-25

K

:keywords method of mail:message, 31-47

L

:last-bp method of mail:message, 31-47

:loop method of ucl:basic-command-loop, 7-25

N

:name method of mail:message, 31-47

P

:print-name method of user-defined tree editor flavor, 11-8

:process-options initialization option of ucl:command-loop-mixin, 7-27

Q

:quit method of ucl:basic-command-loop, 7-27

:quit method of ucl:command-loop-mixin, 7-27

R

:remove-features initialization option of ucl:selective-features-mixin, 7-31

S

:set-point method of mail:interval-stream, 31-48

Variables

Symbols

+, 26-2
++, 26-2
+++, 26-2
-, 26-2
*, 26-2
**, 26-2
***, 26-2
/, 26-2
//, 26-2
///, 26-2

A

sys: advised-functions, 21-3
mail: *always-check-inboxes*, 31-52
 applyhook, 20-1
sys: associated-machine, 25-20
nse: *attribute*, 32-48

B

mail: *box-summary-lines*, 31-55
sys: *break-bindings*, 26-2
eh: *breakon-functions*, 22-2
nse: *buffer*, 32-48

C

mail: *choose-from-all-mail-keywords-p*, 31-57
w: *color-maps*, 34-7
fed: *columns*, 12-16
zwei: *converse-append-p*, 30-4
zwei: *converse-beep-count*, 30-4
zwei: *converse-end-exits*, 30-5
zwei: *converse-extra-hosts-to-check*, 30-5
zwei: *converse-gagged*, 30-5
zwei: *converse-receive-mode*, 30-4
zwei: *converse-wait-p*, 30-5
gwin: *current-cache-for-raster-objects*, 10-16

D

debug-io, 13-5
tree: *default-adjust-to-sup-size*, 11-15
tree: *default-application-type*, 11-15
mail: *default-bcc-string*, 31-56
mail: *default-fcc-string*, 31-56
tree: *default-init-label*, 11-15
ucl: *default-max-command-history*, 7-35
ucl: *default-max-output-history*, 7-35
mail: *default-other-mail-file*, 31-53
ucl: *default-print-function*, 7-35
ucl: *default-prompt*, 7-34
ucl: *default-read-function*, 7-35
mail: *default-reply-to-string*, 31-56

uci: *default-typein-modes*, 7-34
 tree: *default-vertical*, 11-15
 mail: *delete-message-after-copy*, 31-53
 mail: *dont-reply-to*, 31-55

E

eh: *enter-window-debugger*, 14-1
 eh: *error-backtrace-length*, 13-2
 evalhook, 20-1
 name: *expensive-foreign-lookup-by-properties*, 32-7

F

features, 25-19
 tv: *flavor-inspector-configuration*, 16-4
 tree: *force-recalculate*, 11-14
 mail: *forward-template*, 31-50
 meter: *function-histogram-depth*, 27-43
 meter: *function-histogram-interval*, 27-43
 meter: *function-histogram-number*, 27-43

G

nse: *group-member-value*, 32-48
 nse: *group-members-to-delete*, 32-48

I

mail: *in-reply-to-template*, 31-55
 eh: *inhibit-debugger-proceed-prompt*, 13-2
 mail: *inhibit-mail-file-format-warnings*, 31-53
 tv: *inspector-configuration*, 15-3

K

mail: *kill-mail-buffers-at-logout-p*, 31-53
 tree: *known-application-types*, 11-15

L

fed: *label-base*, 12-16
 nse: *local*, 32-49
 sys: local-finger-location, 25-20
 sys: local-host, 25-20
 sys: local-host-name, 25-20
 sys: local-pretty-host-name, 25-20
 mail: *log-enabled*, 31-46
 logout-list, 3-3

M

mail: *sticky-mail-window-configuration-p*, 31-54
 mail: *mail-file-versions-kept*, 31-53
 mail: *mail-mode-hook*, 31-56
 mail: *mail-summary-attribute-char-alist*, 31-55
 mail: *mail-summary-mode*, 31-54
 mail: *mail-summary-template*, 31-54
 mail: *mail-summary-window-fraction*, 31-54
 mail: *mail-template*, 31-50
 tree: *max-level*, 11-14
 sys: %meter-micro-enables, 27-6

meter: metered-objects, 27-25
 tree: *minimum-breadth-spacing*, 11-15
 tree: *minimum-depth-spacing*, 11-15

N

nse: *nse-fonts*, 32-36

O

nse: *object*, 32-48
 nse: *old-group-member-value*, 32-48
 nse: *old-value*, 32-48
 nse: *operation*, 32-49

P

nse: *personal-filter-list*, 32-37
 nse: *personal-horizontal-format-list*, 32-37
 net: *poll-each-status-p*, 33-2
 ucl: preempting?, 7-35
 mail: *preload-mail-file-p*, 31-52
 print-base, 25-14
 print-escape, 25-14
 print-pretty, 25-14
 print-radix, 25-14
 mail: *probe-for-new-mail-p*, 31-53
 nse: *prompt-in-mini-buffer-p*, 32-36

R

mail: *reformat-headers-automatically*, 31-56
 mail: *reformat-headers-body-goal-column*, 31-57
 mail: *reformat-headers-case*, 31-57
 mail: *reformat-headers-exclude-list*, 31-56
 mail: *reformat-headers-include-list*, 31-56
 mail: *reply-template*, 31-50
 mail: *reply-template-1*, 31-50
 mail: *reply-to-all-header-type*, 31-56
 mail: *reply-to-all-template*, 31-50
 mail: *reply-to-sender-template*, 31-50
 mail: *resend-template*, 31-50
 tree: *root-node, 11-14

S

fed: *sample-font*, 12-16
 ged: *save-bits-for-buffers*, 10-3
 mail: *save-mail-file-in-background*, 31-52
 tree: *scroll-window-height*, 11-15
 tree: *scroll-window-width*, 11-15
 tree: *starting-point-offset*, 11-15
 sys: *step-auto*, 19-2
 mail: *sticky-mail-buffer-selection-p*, 31-54
 nse: *stream*, 32-48

T

ucl: this-application, 7-35
 time: *timeit-defaults*, 27-34
 trace-compile-flag, 18-6
 trace-output, 18-5

tree: *tree, 11-14
tree: *tree-window, 11-14
nse: *truncate-attribute-lines-nicely*, 32-37
tree: *truncation-for-scroll-window*, 11-15
mail: *try-mail-now-p*, 31-53
mail: *two-window-reply*, 31-55

U

mail: *unix-inbox-pathname*, 31-53
mail: *unsent-message-query-p*, 31-57
mail: *upcase-message-keywords-p*, 31-57
mail: *user-default-mail-file*, 31-52
user-id, 25-20
mail: *user-mail-address*, 31-56
mail: *user-mail-reading-mode*, 31-53

V

nse: *value*, 32-48
nse: *verification-level*, 32-49
tree: *vertical?, 11-14

Y

mail: *yank-message-headers-include-list*, 31-57
mail: *yank-message-prefix*, 31-57

Data Systems Group - Austin Documentation Questionnaire

Explorer Tools and Utilities

Do you use other TI manuals? If so, which one(s)?

How would you rate the quality of our manuals?

	Excellent	Good	Fair	Poor
Accuracy	_____	_____	_____	_____
Organization	_____	_____	_____	_____
Clarity	_____	_____	_____	_____
Completeness	_____	_____	_____	_____
Overall design	_____	_____	_____	_____
Size	_____	_____	_____	_____
Illustrations	_____	_____	_____	_____
Examples	_____	_____	_____	_____
Index	_____	_____	_____	_____
Binding method	_____	_____	_____	_____

Was the quality of documentation a criterion in your selection of hardware or software?

- Yes No

How do you find the technical level of our manuals?

- Written for a more experienced user than yourself
 Written for a user with the same experience
 Written for a less experienced user than yourself

What is your experience using computers?

- Less than 1 year 1-5 years 5-10 years Over 10 years

We appreciate your taking the time to complete this questionnaire. If you have additional comments about the quality of our manuals, please write them in the space below. Please be specific.

Name _____ Title/Occupation _____

Company Name _____

Address _____ City/State/Zip _____

Telephone _____ Date _____

TAPE EDGE TO SEAL

FOLD

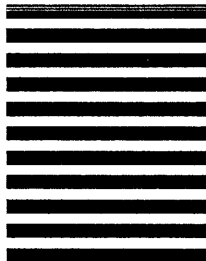


NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP
ATTN: PUBLISHING CENTER
P.O. Box 2909 M/S 2146
Austin, Texas 78769-9990

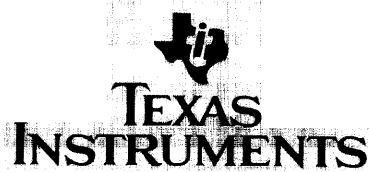


FOLD



Texas Instruments reserves the right to change
its product and service offerings at any time
without notice.

2549831-0001



Printed in U.S.A.