

EXPLORER PROGRAMMING CONCEPTS

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Insert latest changed pages and discard superseded pages.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™ Programming Concepts (2549830-0001 *A)

Original Issue June 1987
Change 1 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

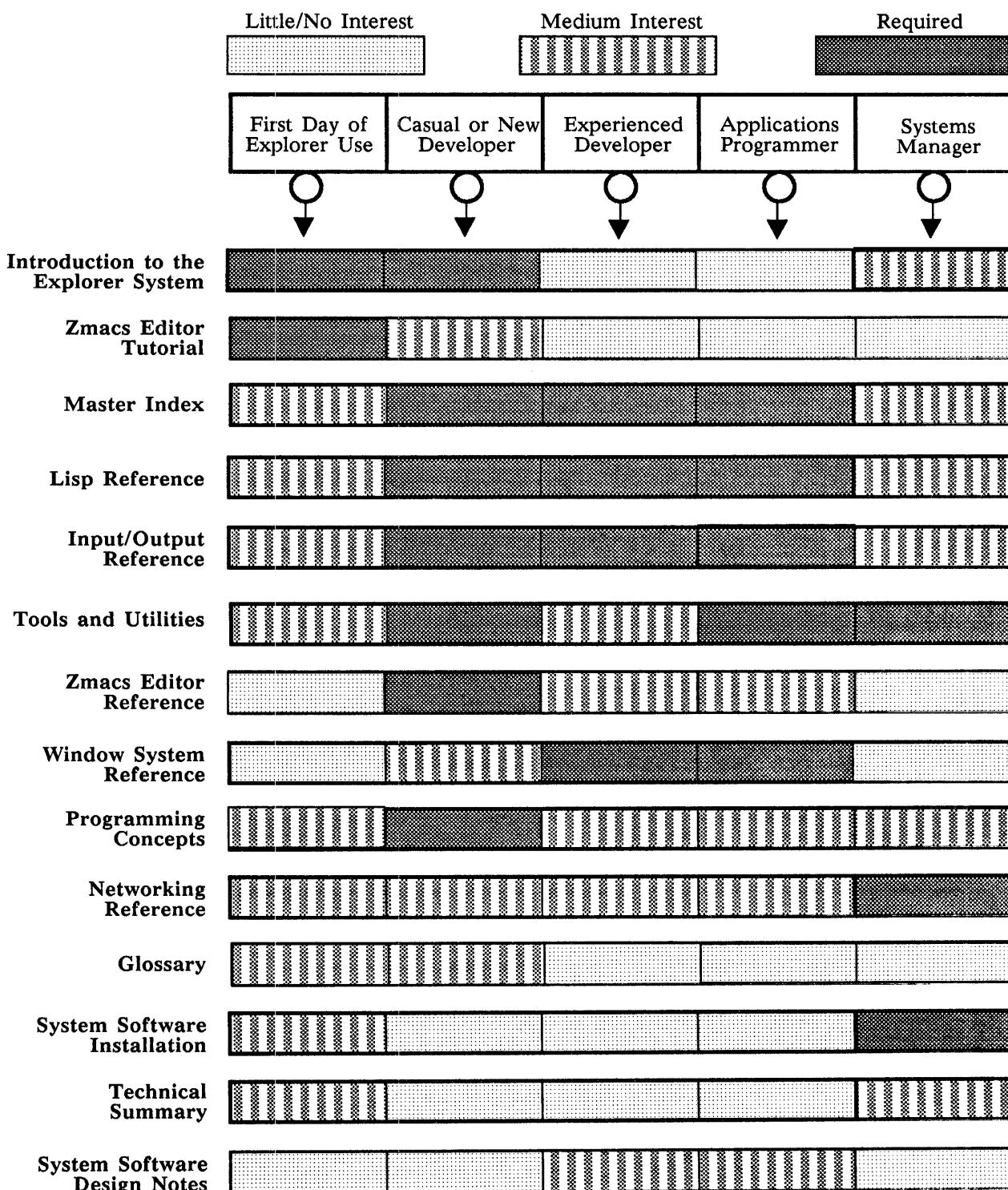
Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

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.
Cover	1	xx - xxi	0	8-1 - 8-10	0
Inside Cover	0	1-1 - 1-11	0	9-1 - 9-17	0
Title Page	0	2-1 - 2-20	0	10-1 - 10-18	1
Effective Pages	1	3-1 - 3-18	0	Index-1 - Index-6	1
Manuals Frontispieces(6pp)	1	4-1 - 4-12	0	Doc Questionnaire	0
xi	1	5-1 - 5-15	0	Business Reply	0
xii - xvii	0	6-1 - 6-28	0	Inside Cover	0
xviii - xix	1	7-1 - 7-16	0	Cover	1

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

Explorer, Explorer II, and NuBus are trademarks of Texas Instruments Incorporated.
TRINITRON and Sony are registered trademarks of Sony Corporation.

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 Xl 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
	About This Manual
1	Conventional Use of the Standard Streams
2	Flavors
3	Condition Signaling and Handling
4	defsystem and make-system
5	Loading and Patching
6	Pathnames
7	Processes and Scheduling
8	Hints to Macro Writers
9	Preparing a Program Product for Delivery
10	Color Concepts

CONTENTS

Section	Paragraph	Title	Page
About This Manual			
		Introduction	xix
		Caveat and Assumptions	xix
		Contents of This Manual	xix
		Notational Conventions	xx
		Keystroke Sequences	xx
		Mouse Clicks	xx
		Lisp Language Notation	xxi
1	Conventional Use of the Standard Streams		
	1.1	Introduction	1-1
	1.2	Why So Many?	1-1
	1.3	*standard-input* and *standard-output*	1-2
	1.4	*query-io*	1-3
	1.5	*error-output*	1-3
	1.6	*trace-output*	1-5
	1.7	*debug-io*	1-5
	1.8	Why Is *terminal-io* Different?	1-5
	1.8.1	The Background Helper Function	1-6
	1.8.2	Failure To Use Synonym Streams	1-7
	1.8.3	Passing *terminal-io* as an Argument	1-7
	1.8.4	How To Guarantee Your Defaults	1-8
	1.9	Other Interesting Streams	1-8
	1.9.1	sys:cold-load-stream	1-8
	1.9.1.1	Advantages	1-8
	1.9.1.2	Disadvantage	1-9
	1.9.1.3	Stream Operations	1-9
	1.9.2	sys:*null-stream*	1-10
	1.10	Odds and Ends	1-10
	1.10.1	Streams Versus Variables	1-10
	1.10.2	strempp Predicate	1-10
	1.10.3	Dribble Files	1-10
2	Flavors		
	2.1	Introduction	2-1
	2.2	Action-Oriented Programming	2-1
	2.2.1	Limitations	2-1
	2.2.2	An Action-Oriented Solution	2-2
	2.3	Object-Oriented Programming	2-2
	2.3.1	An Alternative	2-3
	2.3.2	Choice of Alternative	2-3
	2.3.3	An Advantage	2-4

Section	Paragraph	Title	Page
	2.4	Message Passing	2-4
	2.5	A Generalization	2-5
	2.6	Flavor Programming	2-5
	2.7	Large-Scale Software	2-6
	2.8	Trivial Flavors	2-6
	2.8.1	Instance Variables Without Methods	2-7
	2.8.2	Methods Without Instance Variables	2-8
	2.8.3	Illustration Versus Reality	2-8
	2.9	A Simple Flavor	2-8
	2.10	A Recap	2-11
	2.11	Flavor Instances	2-12
	2.12	Mixing Flavors	2-13
	2.12.1	Defining <code>super-foo-flvr</code>	2-13
	2.12.2	Implementing <code>super-foo-flvr</code>	2-15
	2.13	<code>compile-flavor-methods</code>	2-16
	2.14	Data Hiding and Mixing Methods	2-17
	2.14.1	Data Hiding	2-17
	2.14.2	Flavor Trees	2-18
	2.14.3	Other Features	2-20
	2.15	What's in a Name?	2-20

3**Condition Signaling and Handling**

3.1	Introduction	3-1
3.2	Some General Terminology	3-1
3.2.1	Condition	3-1
3.2.1.1	Condition Events	3-1
3.2.1.2	Condition Data Structure Definition	3-2
3.2.1.3	Condition Instance	3-2
3.2.2	<code>make-condition</code>	3-2
3.2.3	Signal	3-3
3.2.4	Handler	3-3
3.3	A Simple Handler Example	3-3
3.3.1	A <code>condition-bind</code> Version	3-3
3.3.2	A <code>condition-case</code> Version	3-5
3.3.3	A <code>condition-call</code> Version	3-6
3.3.4	Handler Functions Versus Handler Forms	3-7
3.3.5	Default <code>condition-bind</code> Handlers	3-8
3.3.6	Conditional Condition Handlers	3-8
3.4	The Condition Hierarchy	3-9
3.4.1	Component Flavors	3-9
3.4.2	An Example	3-10
3.5	Ad Hoc Condition Names	3-11
3.5.1	Unique Messages	3-11
3.5.2	Unique Errors	3-12
3.5.3	Common Examples	3-12
3.5.4	A Confession	3-13
3.5.5	Ad Hoc Hierarchies	3-13
3.6	Provisional Handlers	3-14
3.7	Error Recovery	3-14
3.7.1	Throwing and Restarting	3-15
3.7.2	Proceeding	3-15
3.7.3	A Sketchy Example	3-15

Section	Paragraph	Title	Page
	3.7.4	Resume Handlers	3-16
	3.7.5	Ignoring Errors	3-16
	3.8	Signal Processing Summary	3-17
<hr/>	<hr/>	<hr/>	<hr/>
4		defsystem and make-system	
	4.1	Introduction	4-1
	4.2	Conditions and Dependencies	4-2
	4.2.1	Common Conditions	4-2
	4.2.2	Common Dependencies	4-2
	4.2.3	Summary of General Conditions and Dependencies	4-3
	4.3	Partitioning a Program Into Files	4-3
	4.3.1	Suppressing Needless Recompiles	4-4
	4.4	It Worked Last Night—Why Won’t It Compile This Morning?	4-4
	4.4.1	Variation on a Theme	4-5
	4.4.2	Embarrassment Insurance	4-6
	4.5	Summary of Compile Conditions and Dependencies	4-6
	4.6	defsystem and make-system Tips	4-10
	4.6.1	defsystem ’s :compile-load Transform	4-10
	4.6.2	defsystem ’s :compile-load-init Transform	4-11
	4.6.3	make-system ’s :compile Option	4-11
	4.6.4	make-system ’s :reload Option	4-12
<hr/>	<hr/>	<hr/>	<hr/>
5		Loading and Patching	
	5.1	Introduction	5-1
	5.2	Redefinition Versus Patching	5-1
	5.3	How the Loader Works	5-2
	5.4	How Redefinition Works	5-3
	5.4.1	More Than Just Patching	5-3
	5.4.2	When Is It Really a Patch?	5-3
	5.4.3	The Down Side of Patching	5-5
	5.5	How Creating a Patch Works	5-5
	5.5.1	Patch File Details	5-5
	5.5.2	Using Zmacs To Create a Patch	5-6
	5.5.2.1	Correct the Source	5-6
	5.5.2.2	Mark the Region	5-6
	5.5.2.3	Execute Add Patch	5-6
	5.5.2.4	Execute Finish Patch	5-7
	5.5.2.5	How To Avoid Common Mistakes	5-7
	5.6	How To Patch the Environment	5-8
	5.6.1	Patching a Data Structure Definition	5-8
	5.6.2	Patching a Flavor Method	5-9
	5.6.3	Things You Can’t Patch	5-10
	5.7	How Installing a Patch Works	5-10
	5.7.1	A Little Nomenclature	5-11
	5.7.2	Patch Directory	5-11
	5.7.3	load-patches	5-12

Section	Paragraph	Title	Page
	5.8	Miscellaneous	5-13
	5.8.1	Explorer System Patch Directories	5-13
	5.8.2	How Do You Find defsystems	5-13
	5.8.3	The Strange Case of defvar	5-14
6	Pathnames		
	6.1	Introduction	6-1
	6.2	Internal Versus External Names	6-1
	6.3	Pathname Objects	6-2
	6.3.1	Device Component	6-2
	6.3.2	Directory Component	6-3
	6.3.3	Filename Component	6-4
	6.3.4	File Type Component	6-4
	6.3.5	Version Component	6-5
	6.3.6	A Re-Illustration	6-5
	6.4	Namestrings	6-5
	6.5	Parsing Functions	6-7
	6.5.1	Interned Pathnames	6-7
	6.5.2	Why Intern?	6-8
	6.5.3	Interning Our Way Around the Problem	6-9
	6.6	Defaulting Versus Merging	6-9
	6.6.1	Defaulting	6-9
	6.6.1.1	A Good Example	6-9
	6.6.1.2	A Bad Example	6-10
	6.6.1.3	An Aside	6-11
	6.6.2	Merging	6-11
	6.6.3	Defaulting Defaults	6-11
	6.6.3.1	The Common Lisp Approach	6-12
	6.6.3.2	The Explorer's Approach	6-12
	6.6.4	Problems With the UNIX OS	6-12
	6.6.4.1	Implied UNIX File Types	6-12
	6.6.4.2	Unimplied UNIX File Types	6-13
	6.6.4.3	A Compromise	6-13
	6.7	Canonical File Types	6-14
	6.7.1	The Role of File Types	6-14
	6.7.2	Portable File Types	6-15
	6.7.3	Two-Way Mappings	6-15
	6.8	Interchange Format	6-16
	6.9	Generic Pathnames	6-18
	6.10	Logical Pathnames — Part I	6-19
	6.10.1	A Program Development Scenario	6-19
	6.10.1.1	Choice of Host	6-19
	6.10.1.2	Choice of Device	6-20
	6.10.1.3	Choice of Directory	6-20
	6.10.1.4	Choice of Name	6-20
	6.10.1.5	Choice of Type	6-20
	6.11	Logical Pathnames — Part II	6-21
	6.11.1	A Better Program Development Scenario	6-22
	6.11.1.1	Choice of Logical Host	6-22
	6.11.1.2	Choice of Device	6-23
	6.11.1.3	Choice of Directory	6-23
	6.11.1.4	Choice of Name	6-23

Section	Paragraph	Title	Page
	6.11.1.5	Choice of Type	6-23
	6.11.1.6	Choice of Version	6-23
	6.11.1.7	Miscellaneous	6-24
	6.11.2	Tying It All Together	6-24
	6.12	SYS-Host	6-24
	6.12.1	The Site Directory	6-24
	6.12.2	System Source Files	6-25
	6.12.3	Why SYS-Host Causes Problems	6-25
	6.13	Absolute Versus Relative Directories	6-26
	6.13.1	Simple Relative Directories	6-27
	6.13.2	General Relative Directories	6-27
	6.13.3	Pathname Object Notation	6-28
7		Processes and Scheduling	
	7.1	Introduction	7-1
	7.2	Some Terminology	7-1
	7.2.1	Stack Group	7-1
	7.2.2	Coroutines	7-1
	7.2.3	Processes	7-1
	7.2.4	Scheduler	7-1
	7.3	Why Do I Need Processes?	7-2
	7.4	Why Does the System Need Processes?	7-2
	7.5	Explorer Coroutines	7-3
	7.5.1	Call/Return Versus Resume	7-3
	7.5.2	Argument Passing	7-3
	7.5.3	Keeping Track of Coroutines	7-4
	7.5.4	An Aside	7-4
	7.5.5	Coroutine Programming	7-4
	7.6	Stack Groups	7-5
	7.6.1	Stack Group Contents	7-5
	7.6.2	Global Versus Private Variables	7-5
	7.6.3	Initial Process Bindings	7-6
	7.6.4	Why Does Anyone Care?	7-7
	7.6.5	Bypassing Bindings	7-8
	7.7	Processes	7-8
	7.7.1	Process Initial Function	7-9
	7.7.2	Wait Function and Arguments	7-10
	7.7.3	process-wait Versus Wait Function	7-10
	7.7.4	More About Initial and Wait Functions	7-11
	7.7.5	Simple Processes	7-12
	7.7.6	Errors Inside the Scheduler	7-12
	7.8	Process Activity States	7-13
	7.8.1	Simple Is Not Good Enough	7-13
	7.8.1.1	An Example of the Problem	7-13
	7.8.1.2	The Example Revisited	7-14
	7.9	All Things Considered	7-15
	7.9.1	Bashing a Process	7-15
	7.10	Scheduler	7-16

Section	Paragraph	Title	Page
8	Hints to Macro Writers		
	8.1	Introduction	8-1
	8.2	Name Conflicts	8-1
	8.3	Block-Name Conflicts	8-3
	8.4	Macros Expanding Into Many Forms	8-3
	8.5	Macros That Surround Code	8-4
	8.6	Multiple and Out-of-Order Evaluation	8-6
	8.7	Nesting Macros	8-7
	8.8	Functions Used During Expansion	8-9
9	Preparing a Program Product for Delivery		
	9.1	Introduction	9-1
	9.2	Checklist	9-1
	9.3	Choose a Package	9-2
	9.3.1	Standard System Package	9-2
	9.3.2	What's An Extension?	9-2
	9.3.3	Exporting Symbols	9-3
	9.3.4	Placement of the Package Declaration	9-3
	9.3.5	A Naming Convention	9-4
	9.3.6	Problems with Package Names	9-4
	9.3.7	A Catch 22	9-5
	9.4	File Attribute List	9-5
	9.4.1	The Common Lisp Standard	9-6
	9.5	Logical Pathnames	9-6
	9.5.1	Pathname Translations	9-7
	9.5.2	Customer-Specific Translations	9-7
	9.5.3	Patchable Products	9-8
	9.5.4	Multiple Logical Directories	9-9
	9.6	Creating and Manipulating Internal Pathnames	9-9
	9.7	<code>defsystem</code> and <code>make-system</code>	9-10
	9.7.1	The System File	9-11
	9.8	The DEFSYSTEM File	9-11
	9.9	Inter- and Intra- Product Independence	9-12
	9.9.1	Your Top-Level Form	9-13
	9.9.2	Running in Your Own Process	9-14
	9.9.3	Running in Your Own Window	9-14
	9.10	Initializing Your Product	9-15
	9.11	A Helping Hand For <code>copy-file</code>	9-17

Section	Paragraph	Title	Page
10		Color Concepts	
	10.1	Introduction	10-1
	10.2	Basics of Color Perception	10-1
	10.2.1	Nature of Light	10-1
	10.2.2	Nature of the Eye	10-1
	10.2.3	The Analytic Model	10-2
	10.3	Basics of Computer-Generated Color	10-3
	10.3.1	Color CRT Operation	10-3
	10.3.2	Color Models	10-3
	10.4	Guidelines for Effective Use of Color	10-4
	10.4.1	General Principles	10-4
	10.4.2	Designing the Color User Interface	10-5
	10.5	Basics of Programming the Color Explorer	10-7
	10.5.1	Considerations for Programming Applications	10-7
	10.5.2	Considerations for Programming the Explorer System	10-13
	10.6	References	10-18

Index

Table	Title	Page
Tables	10-1 Glossary of Color Terms	10-12

ABOUT THIS MANUAL

Introduction

This manual didn't start out as a manual. It started as a series of informal papers written by a single programmer (Merrill Cornish) in his own special style to help train new programmers on the Explorer project. (So he wouldn't have to keep answering the same questions for each new person who came on board.) The use of these papers, known as *Gentle Introductions*, became so widespread that our Education Center started using them in formal training classes. So...

Here are the collected Gentle Introductions, now to be known as eight sections of the *Programming Concepts* manual. The papers have been lightly edited, updated for the current release, and put into one book with an index for your edification and enjoyment.

Included with Merrill's Gentle Introductions are Hints to Macro Writers (from an earlier release of the *Explorer Lisp Reference* manual) and Color Concepts.

Caveat and Assumptions

Some people claim that the Gentle Introductions are rather brutal. The sections do assume previous knowledge—sometimes a great deal of previous knowledge. Specifically, the sections in this manual are geared to be read by a programmer who is familiar with Lisp and who has experience working on the Explorer system. Some sections assume that the reader has experience with the joys and troubles of dealing with very large software applications.

A novice programmer can still benefit greatly from the explanations in this manual. It's well worth reading.

Contents of This Manual

This manual consists of 10 sections, as follows:

- Section 1 Conventional Use of the Standard Streams
- Section 2 Flavors
- Section 3 Condition Signaling and Handling
- Section 4 `defsystem` and `make-system`
- Section 5 Loading and Patching
- Section 6 Pathnames
- Section 7 Processes and Scheduling
- Section 8 Hints to Macro Writers
- Section 9 How to Prepare a Program Product for Delivery on an Explorer System
- Section 10 Color Concepts

Each of the sections can be treated as a standalone paper. So you can read these in any order, skipping around as you choose. Typically, you should read each section sequentially—most examples in a section are carried through the discussion.

Oh, yes—the remainder of this section is the standard boilerplate information about notational conventions. So, if you're familiar with our system and typographical conventions, feel free to skip ahead to the meat of the manual.

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 <i>chorded</i> , 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 <i>find file</i> 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 <i>clicking</i> . 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 once.
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 <code>monowidth</code> 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.

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.

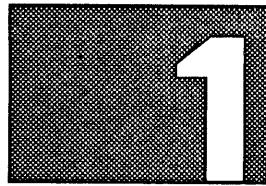
Within manual text, each example of actual Lisp code is shown in the monowidth font. For instance:

```
(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 Section 1, Introduction, of the *Explorer Lisp Reference* manual.



CONVENTIONAL USE OF THE STANDARD STREAMS

Introduction

1.1 Common Lisp provides you with seven default input/output streams. These streams are represented by the following global special variables:

```
*terminal-io*
*standard-input* and *standard-output*
*error-output*
*query-io*
*debug-io*
*trace-output*
```

After a boot, ***terminal-io*** is bound to the Lisp Listener, and all the other standard streams are bound to a synonym stream of the ***terminal-io*** stream (see **make-synonym-stream**).

Notice the distinction. The other streams are *not* bound to the Listener themselves. Instead, they are bound to whatever ***terminal-io*** is currently bound to. Rebinding ***terminal-io*** (it's bad form to setf a global special variable) has the effect of changing all of its synonym streams simultaneously. In contrast, if all streams had been directly bound to the Listener, then changing one would have no effect on the others.

These are two questions to be answered here:

- If all of these streams are going to the same place, then why have seven different ways of doing it?
 - Why is ***terminal-io*** different? Why does it seem to be the parent of the other standard streams?
-

Why So Many?

1.2 If you ignore ***terminal-io*** itself for a moment, the other six names represent a general classification of the kinds of I/O a program might need to do during its development and production lifetime:

- Basic data input and output
- Error reporting
- End user prompting
- Program debugging
- Execution tracing

While it is often useful to use the monitor and keyboard for all I/O, you eventually find yourself wishing you could split things up. For example,

- While tracing a program from a monitor, you wish you could either suppress the program's normal output or at least redirect it to a disk file.

- You want to run a program in pseudo-batch mode by reading ordinary keyboard input from one disk file and saving output to another disk file. But you still want any unexpected errors to be displayed on the monitor right away.
- For testing purposes, you are running your program in two windows on a split screen. You want one window to handle all normal keyboard input and user prompts just as the end user would see. But you want to use the other window to handle the debugging and tracing things you need during development.

Each of these standard streams is available to handle a different, well-defined class of I/O. You will eventually need to switch input between keyboard and disk or to switch output among monitor, disk, and a bit bucket that throws all output away.

Given that you have these six standard streams, the next question is: "Which one is appropriate to use when?" In general, Lisp itself doesn't care what you do as long as you use an output stream for output and an input stream for input. However, bug reports we receive show that if your program violates the user's expectations too much, he complains. The following paragraphs give guidelines for when to use each stream. Then we get to an explanation of what's so special about ***terminal-io***.

standard-input
and
standard-output

1.3 ***standard-input*** and ***standard-output*** are the workhorse streams that all basic I/O functions default to. They are analogous to C's **STDIN** and **STDOUT** streams, to Pascal's **INPUT** and **OUTPUT** files, and to Fortran's Unit 5 and Unit 6. In general, you should use these unless you have a specific reason to use one of the others.

These streams are frequently bound to disk files to make a program run in a batch-like mode. Therefore, you should

- *Never* use these to prompt the user for input.
- *Never* use ***standard-output*** for an error or warning message you want the user to take immediate action on.

standard-output is the appropriate stream for progress messages, comments, summaries, and report information. For example, **print-herald**, **print-disk-label**, and **describe** use it.

Also, certain functions, such as **load**, offer a **:verbose** keyword option that, if **nil**, suppresses output. If you write a function that outputs general commentary, you should output it to this stream, and it would be considered friendly if you provided a **:verbose** keyword to control it.

When deciding whether to use ***standard-output***, ask yourself the question:

Can I afford to ignore the information in this message during execution if I am able to look at it on disk later?

As a rule of thumb:

- If you find that the user needs that information in real time, then you should use another stream such as ***query-io***.

- If other software needs that information, then perhaps you really need to signal a condition instead.

There is a point of confusion about default streams: the standard I/O functions such as `read`, `read-line`, `print`, `princ`, `prin1`, and so on all default to ***standard-input*** for input and to ***standard-output*** for output. Furthermore, as a notational convenience (and a matter of necessity as we will see later), these functions also accept a stream argument of `t` as meaning ***terminal-io***. The popular `format` function has no default output stream, but it accepts `t` as a shorthand for ***standard-output***—*not *terminal-io**. Beware.

Some programmers have empirically discovered that providing the symbol `t` as the stream argument to a function seems to work. It works, of course, because of the shorthand notation mentioned above. But those programmers may ultimately pay for being too lazy to type in a real stream name. Depending upon what output functions are used internally, their output may do the following:

- Go to ***standard-output*** (if used by `format`)
- Go to ***terminal-io*** (if used by `print` and friends)
- Go into the debugger (if a message was sent to `t`)
- Do any or all combinations of the above

In other words, the symbol `t` is not a stream surrogate. It is just an argument-defaulting convention that some, but not all, I/O functions happen to use.

query-io

1.4 *query-io* is the appropriate stream for prompting the user in real time with some information and then reading the response interactively. ***query-io*** should *never* be bound to a disk file. It should always be bound to an interactive display.

This is the stream used by the `y-or-n-p`, `yes-or-no-p`, `fquery`, and `prompt-and-read` functions. As a rule of thumb, use ***query-io*** when your program is doing something similar to one of these functions. That is, use ***query-io*** when you want your I/O to go to the same place `y-or-n-p` would go.

error-output

1.5 *error-output* is the stream used by the `warn`, `error`, `ferror`, `cerror`, and `signal` functions and by the error-checking case statements such as `ecase` and `etypecase`. You should use this stream for messages you want to appear in the company of `ferror` and friends. On the other hand, if you do output to ***error-output***, then you are probably doing something wrong. You should probably either be outputting to ***standard-output*** or be signaling an error.

Let's say your program comes upon a strange situation that should be noted. Here are some alternative actions:

- Output a comment to ***standard-output***
- Output an error message to ***error-output***
- Formally signal an error

If you output to ***standard-output***, you must consider that ***standard-output*** may be bound to a disk file. The user at the monitor will not see the message when it happens and may never see it if he or she doesn't look at the disk file later.

Furthermore, no matter what ***standard-output*** is bound to, higher level software will not be aware that you detected anything amiss, much less have any idea of what you are trying to report. Remember, software understands signal names—not ASCII text.

If you output to ***error-output***, then you have a better chance that the message will show up on the user's monitor in real time—but you still have no guarantee. Also, higher level software can't understand ASCII text on ***error-output*** any better than on ***standard-output***. Therefore, don't use either ***standard-output*** or ***error-output*** if you want anyone to see your message right away.

If you signal a condition (whether or not that condition is an error), then you can still provide the same message text as before, but this time you know that it will be seen. Furthermore, higher level software can detect your signal and respond to it, if appropriate. Therefore, anything you would normally output to ***error-output*** could probably have been better handled by an error or warning reporting function.

A common shortcoming of error handling code is that the programmer thinks in terms of telling the user at the monitor about what has happened. But any function the user can enter from the keyboard, another piece of software can call internally. If there is something important enough to tell the user about immediately, then it is probably important enough to tell any calling software about too.

Therefore, simple messages to ***error-output*** are suspect. In the same vein, the Explorer's **ferror** function with a first argument of **nil** and Common Lisp's **error** function are also suspect. Although they formally signal an error so that higher level software knows something has happened, the software still doesn't know what kind of error has occurred. So, if you are going to signal an error, you should at minimum use **ferror** with a symbol as its first argument that uniquely identifies the error.

As an aside: the Explorer system's implementation of the **error** function causes confusion because it offers outward compatibility between the incompatible functionality of Zetalisp's old **error** function and Common Lisp's **error** function. That is, the functionality of the Explorer's **error** function's functionality actually changes depending upon the *types* of the arguments it is called with.

- If called with **ferror**-like arguments (a signal name, a format string, and format args), then it is identical to **ferror**.
- If called with only a format string and format args, then it is identical to the Common Lisp **error** function.

A good rule of thumb is to use **error** as the Common Lisp standard function and use **ferror** if you want the extra functionality of an explicit signal name.

trace-output

1.6 ***trace-output*** is the stream used by the trace facility. You should not have much need for it unless you want your program to provide extra annotation during a trace. This stream exists mainly so you can separately control where trace output goes rather than having it be something you output to yourself.

debug-io

1.7 ***debug-io*** is another stream that exists mainly so you can separately control where it goes. The debugger uses this stream so that most I/O to it is done while your program is suspended. Therefore, there is little practical value in reading or writing to it yourself from your program.

The ***debug-io*** stream is unique among the standard streams in one respect. If your program were to enter the debugger and the debugger were to find that the value of ***debug-io*** is nil, then the debugger uses the ***terminal-io*** stream of your program (its normal default) instead. From your point of view, this convention means that if, in the heat of debugging, you have set ***debug-io*** to some other stream and you want to return it to its default, then you only have to set it to nil rather than finding a synonym stream for ***terminal-io***.

This defaulting is a convenience feature provided to you by the debugger before it begins to do its work assuming that ***debug-io*** is used only by the debugger. The rest of the system knows nothing of this convention. If you were to set ***debug-io*** to nil and then use it as a stream argument to the standard I/O functions, then they would see a stream of nil that they assume is a shorthand notation for something else (usually ***standard-output***).

**Why Is
terminal-io
Different?**

1.8 Now it is time to answer the second of the two questions posed earlier. Although, as we have seen above, the standard streams can be bound to disk files, they are frequently bound to windows. Now, what happens if your program tries to write to its window using one of these streams while your window is buried?

- If your window was created with the :save-bits option, then your output goes to a memory image of your window to be displayed when your window is eventually unburied.
- Otherwise, you have essentially told the Screen Manager to write some text without giving it anywhere to write it. Not good. You end up with the dreaded Window Lock.

Window Lock is the Screen Manager's rather graceless way of pointing out that you haven't thought your program logic through completely. It freezes the system and displays Window Lock in the status line until you manually find and expose the offending window so the typeout can complete.

In an attempt to avoid the most common causes of Window Lock, the system has implemented certain conventions involving ***terminal-io***, the other standard streams, and the window exposure state.

1. If you have a window, bind ***terminal-io*** to it.
 2. If you want one of your streams (standard or otherwise) to do I/O to that window, then bind that stream to a synonym stream of ***terminal-io***.
-

3. Never pass the variable ***terminal-io*** as a stream argument to any function. Instead, pass a synonym stream of ***terminal-io***.
4. When the window becomes exposed, then the system will see to it that ***terminal-io*** is bound to that window (thereby connecting all those synonym streams to the window also).
5. If the window becomes deexposed, then the system will bind ***terminal-io*** to a special-purpose function that knows how to handle window lock conditions without actually causing Window Lock.

It is the operation of this special-purpose function that leads to those admonitions about not passing ***terminal-io*** as a stream argument. If you will just accept the conditions listed above, then you can skip the next few paragraphs. However, if you want to find out just how much trouble you can cause by ignoring system conventions, then read on.

The Background Helper Function

1.8.1 The easiest way to explain what the mysterious background helper function does is to work out an example. Let's start by assuming that your window is buried so that your copy of ***terminal-io*** is bound to that function. That your window is buried does not necessarily mean that your process is stopped. You could still be running in background. Now, what happens if your background process writes to, say, ***standard-output***?

If you have followed convention, ***standard-output*** will be bound to a synonym stream of ***terminal-io***, which means that the helper function the system bound to ***terminal-io*** will actually receive your output request. Oddly enough, the function starts execution by totally ignoring its arguments (that is, your request to ***standard-output***).

Instead, the function selects a Background Stream Typeout window from a resource of such windows it maintains. It then setsfs your ***terminal-io*** to that window. Finally, it blindly takes the arguments you originally sent to ***standard-output*** and resends them to the new ***terminal-io***. At some point along the way, the user receives a notification worded something like:

Process X wants to typeout.

This is the user's cue to look for the new Background Stream window. From the System Menu, if you click on Select in the Windows column, the windows that include Background Stream in their name were created by the mechanism described above.

NOTE: If you *kill* the Background Stream window, then you will kill the process that is outputting to that window too.

Any later attempts by your program to write to any synonym stream of ***terminal-io*** now go directly to that background window. When your window eventually is exposed, the system once more binds ***terminal-io*** (taking all of its synonym streams along with it) to the window, and the temporary background window is returned to the resource.

So far, we've talked only in terms of a process that had a window. Of course, some processes don't have windows even though they might still contain writes to ***standard-output*** and friends. You can think of these windowless processes as processes with windows that are never exposed. That is, attempts by windowless processes to output to ***terminal-io*** or to any synonym stream of it will get the background window described above.

Failure To Use Synonym Streams

1.8.2 Now, what happens if you ignore the warnings and bind ***standard-output*** directly to the value of ***terminal-io*** rather than to a synonym stream of it? Remember that the system switches the binding of ***terminal-io*** as the window switches between exposure and deexposure. Synonym streams of ***terminal-io*** will be switched with it. However, a stream that simply has the same value as ***terminal-io*** is not switched.

Sooner or later, one of these nonsynonym streams will end up trying to output to a deexposed window (which causes Window Lock). Or else it will output to a background window when the real window is exposed and waiting for output.

For everything to work smoothly, all streams must be switched together as their window alternates between exposure and deexposure. The system has no way of knowing which standard streams you might be using or what other streams you may have created yourself. Therefore, it switches only ***terminal-io*** and relies on you to make use of its value.

Passing ***terminal-io*** as an Argument

1.8.3 If you should ignore the warnings not to pass ***terminal-io*** as a stream argument to a function, then the system continues its retribution. If you pass ***terminal-io*** as a stream argument to some function, then that function executes with a local *copy* of the value of ***terminal-io*** that existed on entry to the function, not with the current value ***terminal-io*** itself, which is being switched.

For example, let's suppose the following function appeared in your code:

```
(defun print-foo (foo stream)
  "Print all elements of FOO to STREAM."
  (dotimes (i (length foo))
    (print (elt foo i) stream)))
```

Now if you called this function as, say,

```
(print-foo foo-object *standard-output*)
```

then all would be well because ***standard-output*** is a synonym stream of ***terminal-io***. But if you called it as, say,

```
(print-foo foo-object *terminal-io*)
```

while your window was deexposed, then amazing things happen.

- On the first call to **print-foo**, each element of **foo** gets printed in its own background window.
- On all succeeding calls to **print-foo**, all elements are printed out in one background window as expected.

The reason for this odd behavior is left as an exercise for the reader. The moral of all this is:

Don't mess with ***terminal-io***.

Now you've seen the real reason that the standard output functions such as **print**, **princ**, **prin1** have the special convention that a stream argument of **t** tells the function to output directly to ***terminal-io***. It's safe to output *directly* to ***terminal-io*** itself since any change to ***terminal-io*** by the background helper function is seen immediately by the **print** function.

How To Guarantee Your Defaults

1.8.4 We've seen that the system depends heavily upon each process having a private ***terminal-io*** and having all other standard streams bound to a synonym stream of it. Unfortunately, the system doesn't guarantee these defaults when it starts a new process.

Therefore, if your program includes its own process, then you should include the following set of bindings around the body of the process's Initial Function:

```
(let* ((*terminal-io*           *terminal-io*)
       (*standard-output*      (make-synonym-stream '*terminal-io*))
       (*standard-input*       (make-synonym-stream '*terminal-io*))
       (*error-output*        (make-synonym-stream '*terminal-io*))
       (*trace-output*        (make-synonym-stream '*terminal-io*))
       (*query-io*            (make-synonym-stream '*terminal-io*))
       (*debug-io*            (make-synonym-stream '*terminal-io*)))
  ...body of Initial Function...)
```

When your process is started, all of its standard streams including ***terminal-io*** are bound to the same thing as their counterparts in the parent process. Binding ***terminal-io*** to itself assures that if ***terminal-io*** should be setfed somewhere in your program (despite all warnings to the contrary), then only your version of ***terminal-io*** will be changed. Otherwise, your parent process' ***terminal-io*** would have been setfed too. The binding of the other standard streams to a synonym stream of ***terminal-io*** guarantees that your process will have the proper defaults despite anything your parent process has done.

Other Interesting Streams

1.9 There are two other streams provided by default on the Explorer system although they are not part of the Common Lisp standard. These streams are represented by the global special variable **sys:cold-load-stream** and the system constant **sys:*null-stream***. These symbols must *never* be either set or bound—treat them as constants.

sys:cold-load-stream

1.9.1 This is the most primitive stream available for I/O to an Explorer monitor. It is commonly known as *THE cold load stream* because there is one and only one. The principal feature of the cold load stream is that it completely bypasses the window system. Such a primitive tool has both advantages and disadvantages.

Advantages

1.9.1.1 The main advantage of **sys:cold-load-stream** is that it is guaranteed to work. When you output to it, you never get into a Window Lock state or any other locked state unless you explicitly request it (see **:tyi** below). Any process—even a background process—can output to the cold load stream.

The reason programmers dread being *thrown into the cold load stream* is that by implication, the window system has become so wedged that the cold load stream is the only way left for the system to get a message to you. Good luck.

Disadvantages

1.9.1.2 The main disadvantage is that **sys:cold-load-stream** ignores the window system, and the window system is the centerpiece of much of the Explorer software.

Output to this stream makes a mess of the monitor. Writing to the cold load stream just writes to successive characters on the monitor, independently of where the window system is writing. It overwrites any windows that it happens to cross, including the thin line around the edge of the monitor and the mouse documentation window or status line (which most programmers don't even know can be done). You can clear the main screen with the CLEAR-SCREEN key, but not the mouse documentation window or status line if they were overwritten.

If you enable **more** processing for this stream, then *all* processes must wait until your user responds to a ****more**** prompt from the cold load stream in your process.

These disadvantages mean that the cold load stream should be used only when needed for debugging. If your user ever sees any of your output via the cold load stream, then you have done something wrong.

Stream Operations

1.9.1.3 The simplest way to output to the cold load stream is to use **format** with **sys:cold-load-stream** as its stream argument. In addition, the cold load stream supports the :line-out, :string-out, :clear-eol, :set-cursor-pos, and :tyi messages. A typical debugging sequence might be:

```
(send sys:cold-load-stream :set-cursor-pos 0 0)
(send sys:cold-load-stream :clear-eol)
(send sys:cold-load-stream :string-out (format nil "..." args...))
```

The last two forms together effectively accomplish the following:

```
(format sys:cold-load-stream "..." "args...")
```

However, the **format** function does not offer you any way of specifying where the output starts, as the :set-cursor-pos message in the first form does. Also, the raw :string-out message just ORs its text with whatever was already on the screen. Therefore, you need to preface the :string-out with a :clear-eol to make sure it's writing on a "clean" line.

The :tyi method of **sys:cold-load-stream** is also very primitive. It stops execution of all other processes until a character is typed (which it does not echo, so the monitor is unchanged). Therefore, if you want your program to pause while you examine the monitor, then insert

```
(send sys:cold-load-stream :tyi)
```

into your program. Your program, and everything else, hangs waiting for you to press any key.

sys:*null-stream*

1.9.2 **sys:*null-stream*** is designed to do nothing, but to do it intelligently. It is analogous to the dummy files that exist on conventional operating systems. Its main (non-) action is to throw away any output sent to it and to return an immediate end of file for any attempt at input from it. For example, the following

```
(let ((*standard-output* sys:*null-stream*))  
...body...)
```

would cause all output from within the body of the let to ***standard-output*** or to any of its synonym streams to be thrown into the bit bucket.

On the Explorer system **sys:*null-stream*** handles many (but not all) stream messages with legal but trivial responses. However, this stream does not handle some of the higher level stream messages that do not have acceptable defaults answers. It signals an **:unclaimed-message** error if you ask it to do something it doesn't understand.

Odds and Ends

1.10 The following topics are interesting odds and ends that don't fit elsewhere in this section.

Streams Versus Variables

1.10.1 Lisp programmers loosely talk about things like ***terminal-io*** being a stream when actually it is a global special variable whose value is a stream object. As with most of the sloppy Lisp jargon, the distinction usually isn't important for one reason or another. But if the subject should come up, remember: you saw it here.

streamp Predicate

1.10.2 On the Explorer system, the Common Lisp **streamp** predicate is actually a heuristic that in extreme cases can be tricked into falsely identifying something as a stream. There is no stream data type on the Explorer system, and there is no stream structure type, so no direct, unambiguous check is possible. Any object capable of accepting the stream messages can serve as a stream. Therefore, on the Explorer system, there is a wide variety of things that might be streams.

The checks **streamp** makes are tight enough that you should never have a problem. However, if some programmer were to, say, write a function which, given **:which-operations** as an argument, returns a list containing either **:tyi** or **:tyo**; then **streamp** is going to be fooled.

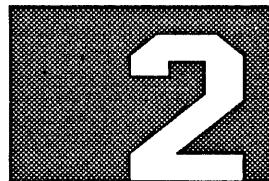
Notice that since **streamp** sends a **:which-operations** message to the object as a last resort, the object may react strangely if it isn't a stream. In particular, it is not safe to use **streamp** on nonstream functions that have side effects. While **streamp** is trying to deduce whether the object is a stream, it may inadvertently trigger the side effect.

Dribble Files

1.10.3 As a matter of record (literally), you can use the Common Lisp **dribble** function in a Lisp Listener to cause input from ***standard-input*** and output to ***standard-output*** to be saved to a disk file in addition to being output to the Listener as usual. To start a dribble file (also known as a Wall Paper file), simply call the **dribble** function with a file namestring as an argument.

You cannot look at the contents of a dribble file until you have closed it. You close it by simply calling **dribble** again with *no* arguments.

This simple dribble file will not record error messages output to ***error-output*** or queries to ***query-io***. If you wish to record I/O to *all* standard streams, then use the **dribble-all** function, an Explorer extension. As before, you close this dribble file with a call to **dribble** without arguments.



FLAVORS

Introduction

2.1 The terms *object-oriented programming*, *message passing*, and *flavors* are often used interchangeably—but there is a difference.

- *Object-oriented programming* is just a way of looking at a problem. It can be done in several distinctly different ways by simply adopting certain programming techniques and data structures. In particular, you do not need a special software system.
- *Message passing* is a style of programming. It is basically an alternate syntax that conveys the appearance that you are manipulating objects rather than actions.
- *Flavors* is a set of software which has the principal purpose of constructing new functions out of existing functions at your direction. Its technique of constructing new from old is called *inheritance* in the literature. The code it constructs uses message passing.

The object-oriented programming principle is only a small part of everything that goes to make up flavors. However, many people will refer to the whole flavor system as object-oriented programming. When this naming of the whole by the part is used in poetry, it's called synecdoche; when it's used in conversation, it's called confusing.

If we want to sort out this confusion about what should be called object-oriented programming, then we need to first understand what it is replacing: action-oriented programming.

Action-Oriented Programming

2.2 Our normal mode of programming could be described as action-oriented programming since we call a function to perform a specific action and provide it with specific arguments. In many situations, action-oriented programming is the only programming technique that makes sense.

For example, let's assume that the only output device available to our Lisp machines is a printer. In such an impoverished situation, we could get by with a single action-oriented character output function named, say, `char-out` that would be called with the character to output as its argument.

Because there is only one very specific action to be performed in our example (outputting a character to a printer), there is nothing to be gained in this case from object-oriented programming or message passing. The program notation would be different, but nothing else.

Limitations

2.2.1 The limitations of action-oriented programming show up when we have to deal with several closely related actions. Those basic action-oriented techniques still work, but we start to get the vague feeling that there ought to be a better way.

For example, let's now assume that we have several output devices on our machine: a printer, a CRT, and a network port. Our action-oriented programming technique would need three action-specific functions: **char-out-printer**, **char-out-crt**, and **char-out-net**. Each time a programmer wants to output a character, it is up to him to select the correct function.

From this example, we can begin to see the problem. Although the programmer tends to think in generic terms "output this character," he must also decide how to output it so that he can select the correct function name. The problem becomes worse if the program is, say, a utility that may be outputting to different devices at different times.

An Action-Oriented Solution

2.2.2 If the programmer needs some sort of general-purpose character output routine, he must code some sort of **case** statement that chooses the right function at run time based on the current output device. For example, assume that *out-device* is the argument to the utility identifying which output device to use. To output the letter **A**, our programmer would have to write

```
(case out-device
  (printer (char-out-printer #\A))
  (crt    (char-out-crt    #\A))
  (net    (char-out-net    #\A)))
```

and would have to include a similar **case** statement everywhere he had to output a character.

Usually, when programmers are faced with the problem of a messy piece of code that is repeated in many places, they're going to try to make a subroutine out of it. In the process, they will probably be reinventing object-oriented programming whether they know it or not.

Object-Oriented Programming

2.3 What our programmers really want is a generic **char-out** function that takes, say, a device identifier symbol and a character to output as arguments. If they specifically want to output the letter **A** to the printer, then they would call

```
(char-out 'printer #\A)
```

On the other hand, if the device identifier had been passed in as the argument *out-device*, then they would call

```
(char-out out-device #\A)
```

Internally, the function **char-out** would probably be implemented much as our original open-coded **case** statements:

```
(defun char-out (device-id char)
  (case device-id
    (printer (char-out-printer char))
    (crt    (char-out-crt    char))
    (net    (char-out-net    char))))
```

This little **char-out** function seems like an obvious and straightforward way to solve the problem. It does implement the basic notion of object-oriented programming: a generic function performs one of several possible related actions based on the type or value of one of its arguments (in this case, *device-id*).

However, the previous example does not clearly show why this style of programming should be dignified with the formal title of *object-oriented* programming. After all, a lot of functions modify their execution based on the value of their arguments—that's what arguments are for.

An Alternative

2.3.1 There is an alternative implementation of our **char-out** function available in Lisp that illustrates stronger object orientation. In this alternative example, we are going to record the device-specific function on the property list of each device identifier symbol:

```
(setf (get 'printer 'char-out) #'char-out-printer)
(setf (get 'crt      'char-out) #'char-out-crt)
(setf (get 'net      'char-out) #'char-out-net)
```

That is, the **char-out** property of the symbol holds a function object that knows how to output a character to the device represented by that symbol. If the above notation looks unfamiliar, just remember that **setf** is Common Lisp's generic assignment function so that

```
(setf (get symbol property-name) property-value)
```

is Common Lisp's way of doing the traditional

```
(putprop symbol property-value property-name).
```

To see how we would use these properties, assume that the argument *output-device* mentioned above has the symbol **CRT** as its value. Then

```
(get output-device 'char-out)  =>
(get 'crt          'char-out)  => #'char-out-crt
```

meaning that the symbolic name of each device (PRINTER, CRT, or NET) identifies to the programmer which device to use while the **char-out** property on the symbol identifies the specific function needed to output a character to that device. Now we can rewrite **char-out** to use these properties.

```
(defun char-out (device-id char)
  (funcall (get device-id 'char-out) char))
```

Choice of Alternative

2.3.2 From one viewpoint, these two implementations of **char-out** are just a matter of personal programming style. From another viewpoint, the property-list version seems more object-oriented. In a sense, the object—the device identifier—*knows* how to output its own characters since it carries its own unique character output function on its property list.

You will find a practical difference between the two **char-out** implementations when it comes time, for example, to upgrade the software to handle character output to a tape unit using a new **char-out-tape** function. If you had used the **CASE** statement implementation, then you would have to *modify* the existing **char-out** function to add a new clause to the **CASE** statement as shown below:

```
(defun char-out (device-id char)
  (case device-id
    (printer (char-out-printer char))
    (crt     (char-out-crt   char))
    (net     (char-out-net   char)))
  new ==> (tape    (char-out-tape  char))))
```

If, however, you had used the property-list implementation, then you would have had to do nothing more than put a **char-out** property onto the **tape** symbol:

```
(setf (get 'tape 'char-out) #'char-out-tape)
```

The version of the **char-out** function that uses the **funcall** would not have to be changed.

An Advantage

2.3.3 This example is simple enough that you might not see why adding new code is better than modifying existing code. The first reason is purely pragmatic: if it works, don't fix it. The modification needed in our example above was simple, but other upgrades might require a significant rewrite. The more you change, the more chance there is for fumble fingering an error into working code.

The second reason applies to systems that do not offer dynamic linking as Lisp machines do. Modifying and then recompiling a function of the Lisp machine allows everyone who uses that function to immediately use the new version. On systems that require, say, a link edit step, you would have to relink the entire program.

The third reason is that you may simply not have the source program available to modify.

Message Passing

2.4 With either of the alternatives above, a call to output a character would have looked the same:

```
(char-out device-id char)
```

The differences would have been inside the **char-out** function. However, our property-list version could be cast in a different call format that emphasizes the fact that it is the object that knows how to do a given operation:

```
(send device-id 'char-out char)
```

This statement would be read as “send *device-id* a **char-out** message” and would be understood to mean “tell the *device-id* object to output *char*”. This message passing notation is really just syntactic sugar. Remember that

```
(char-out device-id char)
```

in Lisp is always equivalent to

```
(funcall 'char-out device-id char)
```

so that the **send** notation

```
(send device-id 'char-out char)
```

simply substitutes **send** for **funcall** and swaps the first two arguments. This **send** notation emphasizes the message passing aspects of the call without actually changing how things execute very much.

The Explorer's flavor system currently uses this message passing notation using **send**. The Common Lisp committee on object-oriented programming is considering making object-oriented calls look like ordinary function calls.

This *generic* call syntax gives system programmers freedom to choose and change between function and flavor implementations without affecting the users.

A Generalization

2.5 The examples of object-oriented programming above use Lisp symbols as objects and property-list entries as storage locations—but please don’t be lulled into thinking of object-oriented programming only in these terms.

The common thread here is that an *object* is any data structure in any language that can have a little extra information associated with it. That extra information can be in one of two forms:

- An object type identifier (which would allow a `case` statement implementation to select the right function call for a given object)
- An association of an operation name with a function name that can perform that operation (which would allow a property-list-like implementation to find the right function)

In our property-list example, we put only one property, `char-out`, on the symbol that named the device—but we could have put more. For example, the `char-in` property could have held an appropriate character input function, the `device-clear` property could have held a reset function, and so on. Therefore, we need to think of an object as being associated with a *list* of operator-name/operator-function pairs.

Flavor Programming

2.6 In its most trivial form, flavors is a message passing system that defines objects and then associates the necessary extra information with those objects. If this were all flavors did, it would be of little added value over plain message passing.

However, the real value of flavors is not in packaging function names and data objects. The power of flavors comes from its ability to construct new software with added functionality out of existing software. Instead of writing new software from scratch or rewriting existing software, flavors uses the technique of creating new functionality by *adding* to existing software.

As an intellectual exercise, try reading the definition of `defflavor` in the *Explorer Lisp Reference* manual. First, notice the `components` argument, which allows you to specify that your new flavor is to be built on top of one or more other flavors. Next, read through the list of `defflavor` options, which is described later in that same section in the *Lisp Reference* manual. For the moment, don’t worry about understanding *exactly* what they are doing and *exactly* how you should use them. Instead, notice the types of things they allow you to do.

- `:settable-`, `:gettable-`, and `:outside-accessible-instance-variables` offer to automatically generate various simple accessor functions for you.
- `:inittable-instance-variables` and `:default-init-plist` let you fine tune the default values for instance variables when a new instance is created.
- `:included-flavors` allows you more control over who shadows whom when flavors with conflicting instance variables or methods are mixed together.

- Because you can define intermediate flavors that are meant to be used only as components to higher level flavors, `:abstract-flavor` tells the system to warn anyone trying to use this flavor by itself that it will not work.
- The `:required-flavors`, `-methods`, `-instance-variables`, and `-init-keywords` provide further warning to programmers who use a component flavor improperly.
- `:method-combination` gives you more flexibility in defining how methods with the same names should be used together rather than one shadowing all the others.

Now that we have flavors in perspective, the remainder of this section describes flavors from the viewpoint of the software engineering tradeoffs it provides to programmers. From that point, it is up to you to see how they can best work for you. The value of flavors to the designer of large software systems is independent of whether that programmer knows—or even cares—about the advertised wonders of object-oriented programming.

Do not, however, dismiss flavors for more mundane-sized programming projects. The same flavor characteristics that make otherwise unwieldy projects manageable also make ordinary-sized projects unusually straightforward.

Large-Scale Software

2.7 If you are implementing a large-scale software project—and you don't want to make a career out of that one project—then you must build upon what already exists rather than reinventing every wheel. Software reusability has always been known as a good thing, but success is often limited because conventional programming systems do not offer much in the way of data hiding or other modularization features. In our case, *data hiding* is being used in the general sense of

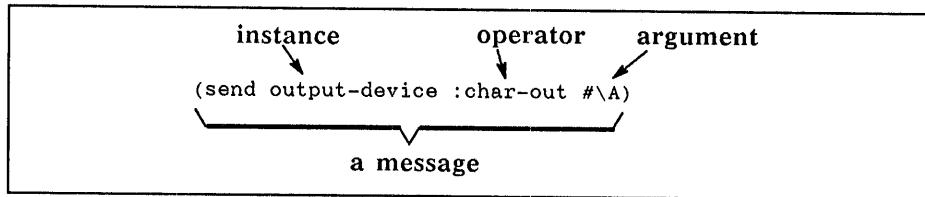
- Using a function (not just data) without having to know about the inner working of that function
 - Being able to change the functionality by adding new code rather than modifying what's already there
-

Trivial Flavors

2.8 In its simplest form (and it has *many* options), a flavor is a set of definitions of variables called *instance variables* and a set of functions called *methods* to go along with those variables. This is a good time to get a little terminology out of the way. Lisp programmers are notorious for using related terms interchangeably, thereby confusing newcomers.

- An *operation* is something like `char-out` mentioned previously.
 - An *instance* is the object on which the operation is done.
 - A *message* is the notation in the program that specifies
 - the operation you wish to execute (for example, `char-out`).
 - the instance you wish to do that operation on (for example, the instance of a CRT that is the system console).
 - any arguments needed by that operation (for example, a character to output).
-

For example,



sends a message to an instance of a device residing in `output-device` telling it to do a `:char-out` operation of the letter `A`.

The piece of software that actually executes the message when it arrives is called a *handler*. Handlers are internally constructed from one or more *methods* you or others have written, and you seldom deal with handlers directly. To summarize the analogy of conventional function terms to flavor terms:

Function Terms	Flavor Terms
Function definition	Method
Function name	Operation
Function call	Message

and there is no direct analogy of a handler in conventional terminology.

Of course, few Lisp programmers bother with these distinctions and just use the terms *method*, *operation*, and *message* interchangeably—but that's programmers for you.

Instance Variables Without Methods

2.8.1 If you were to define a flavor with only instance variable definitions and no methods, then you will have created the equivalent of a Pascal or COBOL record. The flavor definition becomes a sort of template describing the record layout. Just as the `NEW` function in Pascal uses its record template to define a block of memory in that format, the `make-instance` function in Lisp uses the flavor definition template to define an instance of the specified variables.

The functionality of a flavor with only instance variable definitions is the same as a traditional record, although the notation used to access the data is different. And, in truth, accessing flavor instance variables from outside of methods is a little less efficient than accessing simple record components in most other languages. However, access from inside method bodies is *more* efficient than access to Lisp structures or arrays.

If all you wanted was the functionality of an ordinary record in Lisp, then you should use `defstruct` to create a *structure* (Lisp's name for a record). The real point of this illustration is that you are well familiar with the notion of flavor instance variables even though you would have called them by other names.

**Methods Without
Instance Variables**

2.8.2 The other half of this illustration concerns a flavor with only methods and no instance variable definitions. Writing such methods with no associated instance variables and no other flavor options is functionally no different from writing ordinary functions. Once again, the notation for calling methods is different and suffers more overhead than the equivalent function call.

**Illustration
Versus Reality**

2.8.3 The description so far has been less than encouraging. It would appear that flavors do nothing that Lisp structures and functions do not already do except that flavors take longer to do it. Actually, the source of this added overhead is also the source of all the power of flavors. It just so happens that the deliberately trivial example used so far did not make use of flavors' unique capabilities.

A Simple Flavor

2.9 Now, let's start using a simple flavor definition and some methods to go along with it. As an aside: this report uses the suffix `-flvr` on flavor names just to serve as a reminder to you. Actually, you may name them anything you wish as long as all flavor names are unique within a package. In particular, a flavor name may be the same as a function name or a variable name.

Now back to the example: you use the `defflavor` macro to define a flavor as shown below:

```
(defflavor foo-flvr      ; flavor name foo-flvr
          (a           ; instance variables a, b, and c
           b
           c)
          ()          ; no component flavors
...options...)       ; other options
```

This notation defines a template for a flavor named `foo-flvr` with three instance variables, `a`, `b`, and `c`. The empty list identified as component flavors and the options will be ignored for now. This template also establishes another internal data structure called a *method table* that will be used to remember methods defined for this flavor. We will see later how this table is used to create mixtures of flavors.

Next, we can use the `defmethod` macro to define a few methods to go with our flavor as follows:

```
(defmethod (foo-flvr :x) (args...)
  ...body forms...)

(defmethod (foo-flvr :y) (args...)
  ...body forms...)

(defmethod (foo-flvr :z) (args...)
  ...body forms...)
```

The most obvious feature of `defmethod` is that it looks almost exactly like a `defun` except that it has a list in the form of

(flavor-name operation-name)

in the place of `defun`'s function spec, which is usually just a symbol. A `defmethod` needs this enhanced notation to identify its parent flavor so it can be recorded in its parent's method table. Otherwise, `defmethods` and `defuns` can have the same lambda list arguments and the same forms in their bodies.

These methods define the operations `:x`, `:y`, and `:z`. Notice that, unlike their function name analogues, operator names are always in the keyword package (that's what the leading colon means) and not in the current package. There are two reasons for this convention: one practical and one philosophical.

The practical reason is that flavors don't need the package designation to keep operations separate. So, it is convenient to put them all in the *same* package (in this case, the keyword package) for easy reference.

The philosophical reason that operation names are not distinguished by packages (as most other symbols are) is that an operation such as `:char-out` should mean the same thing regardless of which piece of code (in whichever package) uses it in a message. A logical operation should be a logical operation regardless of who uses it from where.

However valid these reasons may be now, when object-oriented programming is eventually added to Common Lisp, method names may become package-specific.

There is a notational distinction between methods and ordinary functions. Instance variables are accessed at run time by a different mechanism than other variables. Since the body forms of methods deal with instance variables so frequently, the system allows you to reference an instance variable of the method's parent flavor with the same notation that references an ordinary variable. The special accessing mechanism is invoked for you.

For example, suppose you wanted to set the instance variable `a` to the value of the instance variable `b` plus 2. If you were inside one of `foo-flvr`'s methods, then you could just write

```
(setf a (+ 2 b))
```

The system recognizes that `a` and `b` refer to instance variables, and it does the right thing. In fact, the *only* way you can access the instance variables of a flavor is from within its methods (not quite true, but accept it for now). If you want to access a flavor's instance variables from the outside world (which includes accesses from methods of other flavors), then you will have to write several trivial accessor methods. For example,

```
(defmethod (foo-flvr :get) ()
  a)
```

accesses the value of the instance variable `a` and returns it as the value of the `:get` operation. Similarly,

```
(defmethod (foo-flvr :set) (new-value)
  (setf a new-value))
```

Sets the value of the instance variable `a` to the specified value. To save you the trouble of having to write a whole bunch of trivial accessor methods, you can have the flavor system do it for you by using a couple of the `defflavor` options we brushed over before:

<code>(defflavor foo-flvr</code>	<code>; flavor name</code>
<code>(a b c)</code>	<code>; instance variables</code>
<code>()</code>	<code>; component flavors</code>
<code>:settable-instance-variables</code>	<code>; options</code>
<code>:gettable-instance-variables)</code>	

The default operation names for the get-methods are simply the instance variable names (in the keyword package, of course). For the case of `foo-flvr`, these methods would be `:a`, `:b`, and `:c`. The set-methods use `set-` as a prefix to the instance variable name, for example, `:set-a`, `:set-b`, and `:set-c`.

Therefore, our previous example of adding 2 to `b` and storing the sum into `a` would look like

```
(setf a (+ 2 b))
```

if it appeared in a method of `foo-flvr`, but it would look like

```
(send foo-inst :set-a (+ 2 (send foo-inst :b)))
```

in an ordinary function or in the methods of a different flavor. Actually, you could use the verbose version inside `foo-flvr`'s methods, too, but that would be unnecessarily clumsy and a little slower.

You can, of course, write your own accessor methods and give them any operation names you wish—but there is seldom any reason not to let the system create the default operations for you. The system's convention of using the instance variable's name as the get-method name is perfectly reasonable, but it sometimes causes confusion.

To a new, slightly confused user it appears that sometimes you write the instance variable without the colon and sometimes you use a colon.

Rule of thumb: If you are talking about the instance variable itself, just use its name. If you are talking about the get-method for the variable, then use the colon.

In particular,

- Anywhere you are referring to the instance variables as instance variables, then just use the instance variable name symbol that is in whatever package the `defflavor` was defined (in other words, *not* the keyword package). For example,
 - References from within the body of a method of the parent flavor [for example, `(setf A (+ 2 B))`]
 - References in an instance variable-related `defflavor` option (for example, `:settable-instance-variables`)
- Anywhere you are referring to the get-method of an instance variable, then use the official name of that method (which happens to be the name of the instance variable in the keyword package):
 - Any sends of the get-method [for example, `(send inst :get-a)`]
 - References in a method-related `defflavor` option (for example, `:required-methods`)

A Recap

2.10 Here is a list of what we have learned about flavors so far, plus a few extra pieces of information that we skipped over before:

- You can use a **defflavor** macro to define a flavor with whatever set of instance variables you want. In this macro, you can specify:
 - Default values for any combination of a flavor's instance variables that will be shared by all instances.
 - That any combination of instance variables be made initable when **make-instance** is executed (thereby providing a way to optionally override the default values).
- A flavor's instance variables are directly accessible only from methods of that flavor.
 - Outsiders must call a flavor's accessor methods to read or write any one of that flavor's instance variables.
 - You can request the **defflavor** to write default accessor methods for you to make any combination of instance variables settable or gettable at run time.
- You can use **make-instance** to instantiate as many copies of your flavor as you want.
 - The **make-instance** allows you to optionally specify initial values for instance variables defined as initable in the **defflavor**.
 - All instances of a flavor share the code of the methods you defined for that flavor.
 - Each instance has a unique copy of that flavor's instance variables. (that is, **defflavor** defines what the instance variables will be while **make-instance** allocates memory for them).
 - You must remember the value returned by **make-instance** so that you can send messages to it later.
- You can use **defmethod** to define a method for a flavor you've already defined.
 - The name of a method is a combination of the name of its parent flavor and the operation name keyword that is unique within that flavor.
 - The method name is remembered in the method table of its parent flavor.
 - When a method's code is executed in response to a message sent to a particular instance, then references to instance variables in that code actually access the set of variables unique to that particular instance. (An example of this is shown below.)

So far, the idea of flavors has been interesting, but you have not yet seen that they can do anything you could not already do. The feature that enables the magic is explained next.

Flavor Instances

2.11 Before you can do anything with a flavor—even to call one of its associated methods—you must make an *instance* of it. For example,

```
(setf foo-inst (make-instance 'foo-flvr))
```

creates an instance of our `foo-flvr` flavor and remembers it in a variable named `foo-inst`. You may make as many instances of `foo-flvr` as you want. All instances of one flavor will share the code of that flavor's methods, while each of those instances will have its own private copy of that flavor's instance variables. That's why they're called *instance* variables.

To demonstrate this separation of instance variables, let's assume you have made two instances of `foo-flvr` called `ff1` and `ff2`:

```
(setf ff1 (make-instance 'foo-flvr))
(setf ff2 (make-instance 'foo-flvr))
```

Now, assuming the default accessor methods, let's set instance variable `a` in each instance to a different value:

```
(send ff1 :set-a 11) ; same code for method :set-a called
(send ff2 :set-a 22) ; twice, but on different instances
```

Finally, by reading the same variable in the two different instances, we can see that they are separate:

```
(send ff1 :a) => 11 ; same variable name in different
(send ff2 :a) => 22 ; instances kept separate
```

We previously drew an analogy between doing a `make-instance` on a Lisp flavor name and doing a `NEW` function on a Pascal record name: both allocated a new block of memory for the data. If you have been paying close attention, however, you have noticed that the object returned by `NEW` and the object returned by `make-instance` are radically different. In many Pascal implementations, `NEW` returns little more than the starting address of the newly allocated block of memory.

In contrast, `make-instance` actually returns a custom built function-like object called an *instance* that not only knows the starting address of the newly allocated instance variables but also knows the method hash table that records all method handlers associated with its parent flavor. When you speak of an *instance* of a flavor, you are really talking about this customized function-like object. And, when you speak of *sending a message to an instance*, you are really calling this instance with the operation name keyword as the first argument along with any other arguments that operation needs. By the way, *instance* is a primitive data type on the Explorer system.

When called, this instance object, in turn, decides which method handler function to call based upon the operation name. This function call and lookup step that stands between the caller and the method handler is the source of the overhead mentioned above. Furthermore, if a method of a flavor calls one of its sibling methods of that same flavor, then it, too, must go through this same instance object to be the message decoded. There is no backdoor for getting to a method within the same flavor (unless, of course, you know where to look).

Let's say the code in the body of the `:x` method of `foo-flvr` needs to call the `:y` method of that same flavor. The `:x` code can get at the `:y` code by sending

a `:y` message to *any* instance of `foo-flvr` because all instances share the method code body. But `:x` and `:y` can share instance variables only if `:x` sends the message to its own instance. Therefore, whenever any method of any flavor is executing, the variable `self` is bound to the current instance. Therefore, `:x` could have used

```
(send self :y)
```

to assure that it was calling its `:y` rather than some other instance's `:y`.

Mixing Flavors

Defining super-foo-flvr

2.12 Flavors were designed to be mixed. Rather than continuing with the "how it works" development, let's jump ahead and see how we could build on the flavor we've already defined, `foo-flvr`.

2.12.1 Let's say that the Foo Facility (which is implemented by our `foo-flvr`) is a tried and true piece of software used for many years. Now you get the idea for a SuperFoo utility. This new program is very much like the old `foo-flvr` with the following differences:

- You need an additional method that (with a continuing lack of imagination) you call `:w`.
- This new method needs an additional instance variable, `d`, for its own private use. Outside code doesn't need to know about it.
- You still need an `:x` method that takes the same arguments as before, but your new version needs to do something completely different.
- The original `:y` method is okay as it stands, but you now need to attach a counter to it to see how many times it gets called.
- The original `:z` method is unchanged. However, when it internally calls `:x` and `:y`, then it needs to use your new versions rather than the ones written for `foo-flvr`.

Starting from `foo-flvr` and given the additional requirements listed above, then you could do the following:

```
(defflavor super-foo-flvr      ; flavor name
          (d)                ; instance variable
          (foo-flvr)         ; component flavor
          )
          ; no options

(defmethod (super-foo-flvr :w) (args...)
  ...body forms...)

(defmethod (super-foo-flvr :x) (args...)
  ...body forms...)

(defmethod (super-foo-flvr :before :y) (args...)
  ...increment counter...)
```

Here is what you are seeing:

- The definition of `super-foo-flvr` includes `foo-flvr` as a *component flavor* (a `defflavor` field we have ignored until now). `super-foo-flvr` will (unless overridden) automatically *inherit* all instance variables and methods of `foo-flvr`.

- In addition to what `super-foo-flvr` inherited from `foo-flvr`, it also defines a new instance variable, `d`, and a new method, `:w`.
- `super-foo-flvr`'s version of method `:x` replaces `foo-flvr`'s version that would normally have been inherited. (We'll see how in a moment.)
- `super-foo-flvr`'s version of method `:y` is called every time the original version of `:y` (known as the *primary* method) is called but `super-foo-flvr`'s version is called first.

Notice that there is no mention of the `:z` method here. The actual code body written for `foo-flvr` will be used for `super-foo-flvr` because of inheritance. Furthermore, whenever `:z` does a send self of `:x` or `:y`, it always gets the `:x` and `:y` handlers associated with its own instance—whatever they may be.

This definition of `super-foo-flavor` with `foo-flvr` as a component is equivalent to the following alternative definition where everything is written from scratch rather than being defined as a mixture:

```
(defflavor super-foo-flvr
  (a b c d)
  ()
  (:settable-instance-variables a b c)
  (:gettable-instance-variables a b c))
```

; flavor name
; instance variables
; component flavors
; options

This alternate notation emphasizes that instance variables `a`, `b`, and `c` are settable and gettable from the outside world because they were gettable and settable in `foo-flvr`. However, the new instance variable `d` is accessible only from within the methods themselves.

The above example also introduces additional `defflavor` syntax. `defflavor` options such as `:settable-instance-variables` that refer to instance variables can be written two ways. Written by themselves, they apply to all instance variables of the enclosing `defflavor` (but not of its component flavors). If written as the first element of a list, they apply only to the instance variables names in the remainder of that list. Therefore,

```
:settable-instance-variables
```

would have applied to all four instance variables: `a`, `b`, `c`, and `d`; but

```
(:settable-instance-variables a b c)
```

applies only to `a`, `b`, and `c`.

Furthermore, this alternate built-from-scratch flavor definition would have the following definitions for its methods:

- `:w` and `:z`—just as before
- `:x`—just as `super-foo-flvr`'s (`foo-flvr`'s version would be ignored)
- A new version of `:y` that would combine the `:y` bodies for `super-foo-flvr` and `foo-flvr` in the specified order:

```
(defmethod (super-foo-flvr :y) (args...)
  ...increment counter...
  ...original body forms...)
```

The implications of all of this are a little overwhelming to take in at once. Having `foo-flvr`'s instance variables and methods added to those of `super-foo-flvr` is understandable enough. The action of a component flavor declaration is very much like the traditional INCLUDE or INSERT statements in other languages. The fact that `super-foo-flvr`'s version of `:x` superseded `foo-flvr`'s original version is also understandable because such shadowing is frequently used in the link edit control files of conventional languages by controlling the order in which libraries are searched.

However, what happened to the `:y` method is strange. It was not ignored, it was not replaced, it was *modified*. Now when you send an instance of `super-foo-flvr` a `:y` message, a counter will be incremented *before* the original body of `:y` is executed. Incidentally, only instances of `super-foo-flvr` will see this change. New instances of the original `foo-flvr` will still execute the original `:y` method just as they have always done. `super-foo-flvr`'s use of `foo-flvr` does not change the operation of `foo-flvr` in the slightest.

In flavor nomenclature, `foo-flvr`'s `:y` method is called a *primary* method because it is always executed. `super-foo-flvr`'s `:y` method is called a *daemon* method because, like Maxwell's daemon, it is never called directly but does its thing automatically whenever its primary method is called. The principal flavor daemons are `:before`, `:after`, and `:around`.

Implementing super-foo-flvr

2.12.2 We now have enough background to thoroughly illustrate the flavor producing mechanism. Here is a quick review of what we know:

- `defflavor` declares a set of instance variables definitions and an initially empty method table to record all later methods that listed it as their parent flavor.
- A `make-instance` on a flavor name returns an instance object customized to know that flavor's set of instance variables and its method hash table.
- In a simple case, such as `foo-flvr`, this instance object serves as a simple dispatcher redirecting messages to their handlers based upon their operation name.

Now let's consider the difference in the instance object for the original `foo-flvr` and the one for the new `super-foo-flvr`.

- `super-foo-flvr`'s instance object knows about one extra instance variable and about one new method, but that seems straightforward enough.
- `super-foo-flvr`'s instance object knows about the same handler for the `:z` operation that `foo-flvr`'s instance function uses, so they share that piece of code.
- `super-foo-flvr`'s instance uses a different method definition for the `:x` operation.
- `super-foo-flvr`'s instance includes a special handler for the `:y` operation. This handler first calls the code for `super-foo-flvr`'s `:before :y` daemon method and then calls the code for `foo-flvr`'s primary `:y` method.

This last action is the most instructive. It is the heart of the flavor system and is called *method combination*.

In most programming languages, if you had wanted to add a counter to a subroutine, then you would have had either to modify the source of that subroutine or to find every call to that subroutine and insert a call to the counter function in front of it. That is, to make the change, you would have had to modify existing code to add a new function.

The flavor system, however, effectively did this modification for you by intercepting calls to `:y` and doing the right thing. We have two distinct pieces of code. `foo-flvr`'s primary `:y` method does the basic `:y` action, and `super-foo-flvr`'s `:before :y` daemon method implements the counter.

When a `:y` operation is sent to an instance of `foo-flvr`, then that instance function simply calls a handler that is `foo-flvr`'s primary `:y` method. When a `:y` operation is sent to an instance of `super-foo-flvr`, then that instance (which, remember, is unique to `super-foo-flvr`) calls a handler that first calls `super-foo-flvr`'s `:before :y` method that does the counting. Then it calls `foo-flvr`'s primary `:y` method to do the actual work.

compile-flavor-methods

2.13 There is a step between compiling `defflavors` and `defmethods` and executing the first `make-instance` on a flavor name that we have alluded to but glossed over. This step is called `compile-flavor-methods` even though it has nothing to do with compiling `defflavors` or `defmethods`.

When the compiler is given a `defflavor` to compile, most of the information eventually needed for execution is missing.

- None of this flavor's `defmethods` can be compiled yet because the method table created for this `defflavor` must exist first so there will be a place to remember associated methods.
- The component flavors of this flavor are known, but only by name. They have not necessarily been compiled yet.
- The instance variables contributed by this flavor and defined in this `defflavor` are known, of course, but none of the instance variables to be contributed by component flavors are known yet.
- None of the methods contributed by component flavors are known yet.

The compiler will presumably come across all missing information eventually, but it never knows when it has finished compiling everything needed by one flavor.

Therefore, the first time a flavor is instantiated, `make-instance` pauses to compile all the information that has been accumulating about this flavor into executable form. This last-minute fix up (after all normal compiling and loading is already done) is called compiling flavor methods because it creates the handlers for the *combined methods* that will actually be executed.

For example, now that all component flavors are known, then all instance variables from all sources are known. So a single master mapping table can be created and recorded in the flavor definition. All methods will access their instance variables indirectly through this mapping table.

Similarly, all methods are now known, along with details on who shadows whom and who gets combined with whom. So a handler function is created for each operation supported by this instance. Some handlers simply call the

appropriate method function. Handlers for combined methods actually call several method functions. For example, the handler for our `:y` method above would first call `super-foo-flvr`'s `:before :y` method function and then `foo-flvr`'s primary `:y` method function.

All these handlers are then put into a method *hash* table for fast access each time a message is sent to this instance. Notice that the simple method table created when the `defflavor` was compiled simply remembered the `defmethods` later compiled for this flavor. The compile flavor methods step takes the method tables from this flavor and all its component flavors and creates one coherent method hash table out of them.

Because the compile flavor methods step on a complex flavor can take a noticeable amount of time, you may wish to execute the `compile-flavor-methods` macro on your flavor at compile time. Thus, your user does not see any extra delay at run time.

There is no advantage to doing a `compile-flavor-methods` on a mixin (that is, a flavor that is used *only* as a component in some other flavor). Only flavors that are actually instantiated by name need the mapping table and method hash table.

Also, if you do your own `compile-flavor-methods`, then be sure you wait until everything has been defined. If, for example, another `defmethod` is compiled for the flavor before instantiation, then `make-instance` will have to do another `compile-flavor-methods` anyway.

Executing unnecessary `compile-flavor-methods` for mixin flavors or executing redundant `compile-flavor-methods` that will have to be redone later create no logical problems. However, they do create garbage in the environment that cannot be garbage-collected.

Data Hiding and Mixing Methods

2.14 Method combination, the unique feature of flavors, is what allows reusability of code and data hiding. If you need *new* code, then you define a new flavor with new methods. If your new flavor can reuse some of the functionality of existing flavors, then you can specify that your new flavor inherit the existing flavor's functionality (that is, its instance variable definitions and methods).

If unconditional inheritance is not what you want, then you can modify it by replacing an inherited method with one of your own (while still using all the others unchanged). Alternatively, you can use the inherited method in modified form through method combination.

Data Hiding

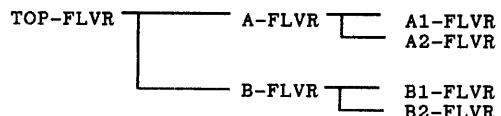
2.14.1 Data hiding is achieved because only the writer of the new flavor needs to know how the component flavors must be modified. The end user of a combined flavor never needs to know how a given operation came to be the way it is. To that end user, `:char-out` always writes a character to the device represented by whatever instance the message is sent to. Only the flavor writers for, say, the CRT I/O flavor, the TTY I/O flavor, the plotter I/O flavor, and so on need to know about the unique handshakes required to actually get the character written.

Furthermore, this data hiding capability compounds. If end users of the `:char-out` operation decide to write a higher level I/O flavor, they can use all

the device-dependent knowledge embodied (and therefore hidden) inside the :char-out operation to build whatever else they need.

Flavor Trees

2.14.2 You have seen the :before type of method combination in the previous examples, but you have had to intuit exactly what it does. Our examples so far have shown only one component flavor and a single level nesting of one component within another. In practice, one flavor can have several components, and each component may itself have components. Therefore, real life flavors are often the result of mixing a whole tree of subflavors together. For example, given the following flavor tree:



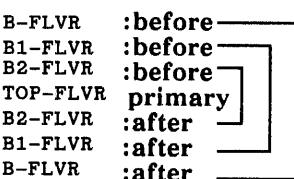
then the flavors would be ordered top-down, depth-first as shown below for the purposes of method combination:

Although this example does not show it, it is possible that a primitive flavor may appear several places in the tree as a subcomponent of different flavors. In such a case, only the first appearance of the duplicated flavor appears in the final ordering.

Primary methods (that is, those with no explicit type) that appear earlier in the flavor list normally shadow primary methods for the same operation contributed by flavors later in the list. However, **defflavor** options can force duplicate primary methods to be combined in other ways too.

Daemon methods such as :before and :after do not shadow each other. Instead, they are all concatenated. All :before methods for a given operation are executed in the order shown above, then the first primary method for that operation is executed, and finally all :after methods for it are executed in reverse order.

For example, assume that `top-flvr` and `a-flvr` define a certain primary method, and all the `B` flavors define `:before` and `:after` methods for that same operation. Then those methods are combined in the following order.



Notice the characteristic nesting of the `:before` and `:after` methods of one component flavor inside the `:before` and `:after` methods of component flavors earlier in the ordered list of flavors. The primary method for `a-flvr` doesn't appear because it is shadowed by the primary method for `top-flvr`.

The **:before** and **:after** daemon methods are used for this example because their names imply their actions. Other method combinations process the ordered list of flavors differently. But in all cases, the description of the method combination is referencing that same ordered list.

The following is a partial list of method combinations that are available in flavors. They are listed here just to get your imagination started.

NOTE: These are deliberately *not* rigorous definitions.

:before — All **:before** daemons are run before the primary method in the order specified by the component flavor nesting. They can see, but not modify, the arguments to the primary method. They can neither see nor modify their primary's return value.

:after — All **:after** daemons are run after the primary method in reverse order. They can see, but not modify, the input arguments to their primary. They can neither see nor modify their primary's return value.

:around — An **:around** daemon is called in place of its primary. It can both see *and* modify inputs to and return values from its primary. It may also decide if its primary is even to be called.

:or — All **:or** methods are called in order. When any of the methods being executed returns a non-nil value, execution stops, and that non-nil value is returned. Otherwise, all are executed and the combined method returns nil.

:and — All **:and** methods are called in order. When any of the methods being executed returns nil, execution stops, and nil is returned. Otherwise, the value returned by the last method is returned by the combined method.

:progn — All **:progn** methods are executed in order, and the value of the last one is returned.

:append — The values returned by all the methods are appended together to form the return value of the combined method.

:nconc — The values returned by all the methods are nconc'd together to form the return value of the combined method.

Method combination can be done by indicating the combination type in the **defmethod** itself (as we did with the **:before :y** method). The manner in which the methods of component flavors are to be combined can also be specified in one of the **defflavor** options of the parent flavor.

Other Features

2.14.3 So far, we have only talked about flavors depending upon component flavors. Actually, two component flavors (both are components of a higher level flavor) can communicate with each other if they define common instance variables.

For example, assume mixin-A defines instance variables I and J while mixin-B defines instance variables J and K. If mixin-A and mixin-B should both be declared as components of another flavor, then that other flavor inherits (among other things) instance variables I, J, and K. At run time, references to instance variable J by mixin-A and mixin-B are to the same instance variable in the higher level flavor.

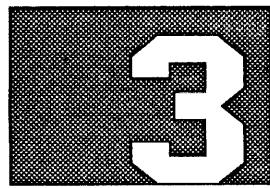
**What's in
a Name?**

2.15 It all started in an ice cream parlor.

Near the MIT campus, the ancestral home of Lisp machines, was Steve's® Ice Cream parlor. Whereas many ordinary ice cream emporiums provide toppings you can put on your ice cream, Steve's featured *mixins*. You would choose your base flavor and your mixins, and they would be thoroughly mixed together for you. The ice cream you were finally served was one new flavor just as though it had been created that way from scratch.

The flavors software system does not just borrow a clever set of names from an ice cream vendor. The software version also has the unique characteristic that a base flavor compiled with mixins is almost indistinguishable from a new flavor written that way from scratch. A complex flavor is not the layers and layers of software one might first think. Compilation causes a mixed flavor to be near enough to an original version that you do not have to be concerned on that account.

Steve's Ice Cream is a registered trademark of Steve's, Inc.



CONDITION SIGNALING AND HANDLING

Introduction

3.1 Lisp programs need a way to detect and handle special conditions, expected or unexpected, that can occur during processing. You can handle expected conditions with ordinary IF-THEN-ELSE style logic in the code because you know where to expect which condition and what to do about it. But what do you do about unexpected conditions? That is, how can you write your program to reasonably handle the following:

- Conditions that are expected but that almost never happen, so it seems burdensome to put explicit checks in the program (for example, putting error checks around a hundred divide operations on the off chance that one of them might some day be asked to divide by zero)
- Conditions that are detected easily enough by low-level routines but that cannot be handled by those routines because they do not have the big picture of what their callers were trying to do (for example, one math analysis program may want to treat floating point underflow as a result of zero while another such program may want to abort all processing)
- Conditions for which timing is totally unpredictable (for example, someone pressing the ABORT key while your program is running)

This report develops an intuitive explanation about what you need to do and how you might go about doing it. The Explorer documentation provides the detailed definitions of the Lisp forms you need. Since the condition signaling and handling system is built upon flavors, you need a general knowledge of flavor terms to make best use of this report.

Some General Terminology

3.2 The main terms you need to know are *condition*, *make-condition*, *signal*, *handler*, and *proceed type*. Unfortunately, experienced Lisp programmers have a bad habit of using each of these terms to refer to several distinct but related concepts. Eventually, you will probably fall into the same habits because the distinctions are not really important in practice. Rather than fight the popular usage, the following definitions deliberately explain each family of meanings together.

Condition

3.2.1 Lisp programmers use the one word *condition* to refer to condition events, condition data structure (that is, flavor) definitions, and condition instances.

Condition Events

3.2.1.1 A condition event is any event, situation, circumstance, state, or such that the Lisp software or system microcode finds interesting for some reason. For example, if you ask the file system to open a file that does not exist, then that is a *file not found* condition event.

Condition events are frequently used to indicate errors such as trying to open a nonexistent file, but you cannot assume that all condition events must be errors. For example, an *end of page* condition event simply means your

output has filled up the screen and you have to do something about it (such as wrapping around back to the top).

Furthermore, your idea of which condition events are errors may change depending upon what you are trying to do. For instance, if you ask the file system if a certain file exists, then the same condition event mentioned above, *file not found*, is equivalent to the simple answer, No.

Condition Data Structure Definition

3.2.1.2 If a condition event is going to trigger the processing described in this report, then certain information must be recorded, such as which condition event occurred and the circumstances of that particular occurrence. Notice that we are talking only about a data structure *definition* here. That is, this is just a template describing each condition event. When some specific event occurs later, this template will be used to create the actual data object.

Since the entire condition signaling and handling system is built on flavors, this data structure definition is really a flavor definition. For example, the file not found condition event mentioned above is represented by the `fs:file-lookup-error` flavor definition.

As a matter of convention, a condition event is commonly referred to by the name of the flavor that defines its template. So, someone speaking of a *condition name* is usually talking about the name of its defining flavor. Therefore, if you ever define some of your own conditions, use reasonable names for their flavors. In particular, if you give the `type-of` function a condition instance, it would return that instance's condition flavor name that, hopefully, describes the condition. Later we will see that a condition name can be something other than just a flavor name. But for now, we can treat them as the same.

Because interesting condition events are usually represented by flavors with matching names, and because occurrences of such a condition event are represented by instances of those flavors, Lisp programmers tend to further confuse newcomers by using the terms *condition* and *flavor* synonymously.

Condition Instance

3.2.1.3 When a particular condition event occurs, an instance of its associated flavor is created to record that occurrence. Notice the distinction: the condition flavor *definition* describes the generic condition event while the condition flavor *instance* contains both the generic information common to all such conditions and specific information about this particular occurrence.

If you do not understand this flavor terminology, never mind. The point is that some of the information is shared by all, some of it is unique to the occurrence, and none of it can ever be changed once it is created.

make-condition

3.2.2 A condition instance is actually created by the `make-condition` function, which is a slightly customized version of the `make-instance` function used for flavors. If you were to provide the extra arguments, you could use the `make-instance` function directly instead of `make-condition`, but that is a tedious thing to do and is prone to errors. You yourself may never need to use the `make-condition` function because it is usually done for you by higher level signaling functions.

Signal 3.2.3 A *signal* is the action that brings to the attention of the system a condition instance describing a particular condition event that has occurred. Simply making a condition instance does not, by itself, trigger any processing. The condition instance must be signaled before anything happens.

Once again, Lisp programmers are casual about their terminology. Strictly speaking, you are supposed to have a condition instance already available before you signal it. However, several of the most convenient signaling functions accept either an existing condition instance or just a condition name plus some initialization values. When these functions are called with a condition name, they go on and call the **make-condition** for you. So, when someone talks about *signaling a condition*, you do not really know where the signaled condition instance came from—but you do not usually care, either.

If you are familiar with the **catch** and **throw** functions in Lisp, then a signal is analogous to a **throw** while the handlers described next are part of the **catch** structure. The condition names are used as if they were **throw** tags.

Handler 3.2.4 The term *handler*, or *handling*, can also take on a number of related meanings. In the most basic sense, a handler is just the piece of Lisp code that does whatever is necessary to process a particular signal. By extension, Lisp programmers often loosely refer to the Lisp forms that contain the handler code (such as **condition-bind**, **condition-case**, and **condition-call**) as *condition handlers*, or just *handlers*.

Finally, the detailed function descriptions in the manuals draw a subtle distinction between the handler functions used in **condition-bind** and the simple list of forms that perform a similar role in **condition-case** and **condition-call**. This distinction is sometimes important because **condition-bind**'s handler functions are run in the same environment in which the condition occurred (and can therefore see all bindings local to the code that signaled the event). In contrast, the forms in **condition-case** and **condition-call** run in the environment in which they (not the condition event) were defined. An example of this difference is shown under paragraph 3.3.4, Handler Functions versus Handler Forms.

A Simple Handler Example

3.3 Rather than jumping into an explanation of all of the sophistication and features of the entire system, let's start with a simple example and build from there. The basic example is first fully described using **condition-bind** and is then briefly repeated with **condition-case** and **condition-call** to show their additional features.

One additional word of caution: each of these forms contains features other than what is used in this example. Therefore, do not artificially limit yourself. Read the detailed descriptions in the *Explorer Lisp Reference* manual to get the full power of this system.

A condition-bind Version

3.3.1 For this example, assume that there is a block of Lisp code containing a number of condition events that might be signaled. Of all these conditions, we are interested only in handling conditions W, X, Y, and Z, whatever they may be. (Remember, when we say we "handle condition name X", this means that the handler will receive an instance of the condition flavor X—or, as explained later, of a condition built on flavor X.)

Let's further assume that we have written several functions:

- A function `handler-w` to handle condition W
- A function `handler-xy` to handle both conditions X and Y (which are presumably closely related)
- A function `handler-z` to handle condition Z

If it was necessary for `handler-xy` to tell which condition, X or Y, was actually signaled, then it could have used, say,

```
(typep signaled-instance 'x)
```

which returns `true` if and only if its argument (called *signaled-instance* here) is an instance of flavor X, which defines condition event X. Actually, we would really want to use

```
(condition-typep signaled-instance 'x)
```

because, as alluded to before (but not explained yet), a condition name does not always have to be a flavor name. The `typep` predicate above would tell you if *signaled-instance* was built on the flavor X. The `condition-typep` predicate would tell you everything that `typep` does, plus it would tell you if X was a nonflavor condition name associated with *signaled-instance*.

```
(condition-bind (( w      'handler-w )
                  (( x y) 'handler-xy)
                  ( z      'handler-z ))
  . . .
  . . . forms to be protected that contain
  . . . signals of w, x, y, and z
  . . .)
```

Now, here is what you are seeing:

- `condition-bind` is *wrapped around* a block of forms (represented by the ellipses). This block of forms is referred to in these examples, for want of a better term, as the *protected forms*.
- `condition-bind` can only see signals that originate in these protected forms and responds only to signals of conditions w, x, y, and z. Signals of any other conditions are ignored and are passed through to the code that this `condition-bind` is nested within.
- The symbols `w`, `x`, `y`, and `z` are the condition names that identify the condition event this `condition-bind` handles.
- The symbols `handler-w`, `handler-xy`, and `handler-z` are the names of the functions that are called if their corresponding condition is signaled. Notice that `handler-xy` handles both condition `x` and condition `y`.
 - These handlers are called with at least one argument that is the condition instance that was signaled and that contains all pertinent information about this occurrence of the condition.
 - Although this example does not show it, you can also provide additional arguments after the handler name. If the handler is called, its first argument is the condition instance, and its remaining arguments are the ones you coded in the `condition-bind`.

- As elsewhere in Lisp, these handler function names could have been replaced by lambda expressions. Lambda expressions are typically used when the function is short and is used only in one place. For example, if the `handler-z` function was very simple, then we might have used:

```
#'(lambda (signaled-instance) ...handler-z forms... )
```

When a condition is signaled somewhere in the protected forms, the system searches the `condition-bind`'s handler list from the first one listed to the last. The fact that the handlers are searched in a known order can be useful, as shown in the example under paragraph 3.4.2, An Example.

If the system finds a match as it searches the handler list, then that handler is called. If it finds no match, then the search expands outward through the code containing this `condition-bind`. As with any Lisp forms, `condition-binds` (and related forms) can be nested to any depth. The innermost one that can handle the signaled condition gets called. Outer level `condition-binds` and so on never see signals that were handled lower down.

A condition-case Version

3.3.2 Now let's repeat the previous example using a `condition-case`. There are two differences. First, the order of the code to be protected and the handlers is swapped. The protected code now comes first, and the handler clauses matching what to do for each condition come second.

The second difference is that we now code the *body* of the handler functions directly into the `condition-case` rather than just providing the name of a function defined elsewhere. Remember that the `condition-bind` called its handler functions with one argument that was the particular condition instance that was signaled. The `condition-case`, however, binds that condition instance to a symbol that you provide (named *signaled-instance* in this example) for use by the handler forms.

The other change is that a `condition-case` protects only a single form. However, since a single Lisp form such as the `progn` shown in the example can contain an entire program if necessary, this single form restriction is not important.

```
(condition-case (signaled-instance)
  (progn
    . . .
    . . .
    . . .
    (w   - - - body forms of handler-w
      - - -)
    ((x y) - - - body forms of handler-xy
      - - -)
    (z   - - - body forms of handler-z
      - - -))
```

The one distinction between `condition-bind` and `condition-case` that does not show up in this example is that the handler function in `condition-bind` runs in the environment where the condition was signaled while the handler forms in `condition-case` run in the environment of the `condition-case`.

Remember that the block of code being protected may be quite elaborate with its own local bindings and declarations. A `condition-bind`'s handler function can see these local binding and modify them if necessary. A `condition-case`'s handler forms can only see and modify things that are global to

the condition-case form itself. These handler forms can never see things that were local to the code they were protecting. The further implications of this distinction are explained below in paragraph 3.3.4, Handler Functions Versus Handler Forms.

One other option to condition-case that sometimes comes in handy is that you can add a clause at the end with the pseudo-condition name of :no-error. The forms in this clause are called only if no other clause of the condition-case was executed.

When the :no-error forms are being executed, the list of symbols before the protected form (which, in this example, is just the symbol *signaled-instance*) takes on a different meaning. These symbols are bound to the values returned by the protected form so that they can be accessed by the form in the :no-error clause. For example,

```
(condition-case (A B C)
    ( ... protected form... )
    (w      ...   A = condition instance ... )
    ((x y) ...   A = condition instance. ... )
    (z      ...   A = condition instance ... )
    (:no-error ...   A,B, and C = return values ... ))
```

shows the same example as before, except that a :no-error clause has been added and several symbols are provided.

If a condition W, X, Y, or Z were signaled, then A—the first symbol in the list—would be bound to the condition instance, and the remaining symbols, B and C, would remain unbound. If, however, none of the listed conditions (W, X, Y, or Z) were signaled, then the symbols A, B, and C would be bound to the first three values returned by the protected form so that they can be accessed by the :no-error forms.

A condition-call Version

3.3.3 A condition-call is structurally identical to a condition-case. Actually, this example is not very good for showing off what a condition-call can do. The whole point of a condition-call is to allow a more complex case choice than simply matching the signaled instance to condition names. The following example shows predicate expressions that mimic the original condition flavor choices. Just remember that these predicates can be as elaborate as your program requires.

```
(condition-call (signaled-instance)
  (progn
    . . . forms to be protected that contain
    . . . signals of w, x, y, and z
    . . .)
  ((condition-typep signaled-instance 'w)
    - - - body forms of handler-w
    - - -)
  ((or (condition-typep signaled-instance 'x)
        (condition-typep signaled-instance 'y))
    - - - body forms of handler-xy
    - - -)
  ((condition-typep signaled-instance 'z)
    - - - body forms of handler-z
    - - -))
```

Handler Functions Versus Handler Forms

3.3.4 The previous text mentions several times that the handler function specified in **condition-bind** forms runs in the environment in which the condition was signaled. Let's see graphically what the difference is.

```
(condition-bind ((...handler...))
               (...handler...))

. . .

signal    <-- handler runs as though it
. . .
      were called from this point
. . .

)
```

A **condition-bind** is the only one in which a handler can fix up the error and proceed from the signal point. It can proceed because the **condition-bind** arranges things so it appears that the handler function was actually called by the signal. Therefore, *proceeding* is little more than simply returning from the handler (at least as it appears to the outside world).

```
(condition-case ()
  (progn
    . . .
    signal
    . . .
    (...handler...)
    (...handler...)
    )
. . .
      <-- handler runs as though it
      were called from this point
. . .
```

A **condition-call** (or **condition-case**) handler cannot proceed from the point that the event was signaled because all code that caused the signal has already been exited before the handler is called. **condition-call** arranges things such that execution appears to branch from the signal point to the end of the protected code block and then call the handler. Although the handler is given the condition instance that was signaled, it can do no more than any other inline code following the **condition-call**.

Now, what does this difference translate to in practice? If the same condition can be signaled from many places within the protected code block under a wide variety of circumstances, then there is very little that the handler code can assume about which variables are bound at this particular site. In this case, a handler function in a **condition-bind** cannot do much more than the simple handler forms in a **condition-case** or **condition-call**.

If, however, that condition was signaled from exactly one spot (as shown in the previous example), then the handler function could be written as though it was a piece of local error fixup code. That is, it is almost as if you could replace **IF error THEN signal-condition** with **IF error THEN call handler function**. Now the question arises, if such a handler function is equivalent to an ordinary piece of inline code, then why bother with writing a separate function and setting up a **condition-bind**?

Because, with this feature, the programmer who writes the code that detects the condition need not devise a universal error handler that would fit all future needs. That is, just because you can detect an error, that does not mean you have any idea what the caller would like to do about it.

In other words, if you're writing a relatively primitive function, you know what kinds of errors can occur (for example, a file doesn't exist, an array has the wrong dimensions, and so on). You signal a separate condition for each

kind of error. Then, when your colleague Smedley writes code that calls this primitive function, Smedley (not you) can decide the appropriate way to handle a particular condition (such as creating the non-existent file, truncating the array to the correct dimensions, and so on).

As a general rule, all system functions that are meant to be called by someone else would normally choose simply to signal the condition and let the caller decide what is the best way to handle it. If the programmer who signals an error knows a default to take if the caller had no better idea, then that programmer can use a *default handler* as described below.

Default condition-bind Handlers

3.3.5 Sometimes you want to handle a condition only if nobody else has a better idea. Therefore, we need a way of establishing a default handler to be used only if a real handler is not available.

The need for default handlers is not as rare as it might sound. Many conditions are defined such that if they are not handled in the program, then they will bring up the debugger on the terminal—often to the total confusion of the end user.

Therefore, programmers trying to write user friendly application programs may often find themselves saying, “I do not really know what to do with this, but anything is better than the debugger.” Then they create a default condition handler that will probably give some vague message about a system error and encourage the user to try again anyway.

condition-bind-default is the counterpart of **condition-bind**. When a condition is signaled, the system begins searching from the most recently bound condition handler outward. It is looking for any regular handler clause that lists one of the condition names that matches the signaled instance or one of the condition flavors the signaled instance was built upon. If the system does not find such a match or if none of the matches it finds will agree to handle the condition (see paragraph 3.6, Provisional Handlers), then it starts back at the signal site and performs another inside out search, this time for the default handlers.

Conditional Condition Handlers

3.3.6 Confusing as this name might sound, there is one last variant on condition handlers that allows you to decide at run time if you actually want the forms to be protected or not.

Regular Handlers	Conditional Handlers
condition-bind	condition-bind-if and condition-bind-default-if
condition-case	condition-case-if
condition-call	condition-call-if

These *if* variants have an additional predicate argument. If this predicate is true when the form is evaluated, then the enclosed forms are protected as described in this section. If this predicate is false, then the forms are executed without protection as though the condition handler had never been coded.

One use of these conditional condition handlers is to make a production program handle errors differently, depending upon whether the program is being

run in a production environment by an end user or run in a debug session by the program's author. For example, in a production run, the condition handlers might be in force to change internal errors into messages an end user can deal with. In a debug run, however, the condition handlers would be turned off so that the author can use the debugger. For example:

```
(condition-bind-if *production-mode-p*
  ( w      'handler-w )
  ((x y) 'handler-xy)
  ( z      'handler-z ))
  . . .   forms to be protected for the
  . . .   end user but not the author
  . . .)
```

Therefore, if `*production-mode-p*` is false, then the condition handler functions are not bound and the forms are not protected—just as though the `condition-bind` had not been wrapped around them. If, however, `*production-mode-p*` is true, then the forms are protected just as described for `condition-bind` above.

The Condition Hierarchy

3.4 The discussion so far has simply assumed that each condition event is represented by one flavor definition—which is true as far as it goes. But sometimes you want to handle each condition individually, and sometimes you want to handle related conditions as a group. Therefore, we need a way of establishing a hierarchy of conditions.

For example, suppose a condition was signaled (remember, *signaled* means that it was detected and formally presented to the system for handling). You might approach your task of establishing a hierarchy by asking some questions:

- What kind of condition is it? (Let's say it was an error condition.)
- What kind of error condition is it? (Let's say an arithmetic error.)
- What kind of arithmetic error? (Let's say a divide by zero.)

In other words, one situation (a divide operation that had a divisor of zero) can imply many different things (condition, error, arithmetic exception, or divide by zero) depending upon what the programmer was trying to do at the time. For example, if a programmer was trying to maintain an error log, then he would need a handler for an error condition that would intercept all errors whether they were arithmetic exceptions or not. If, however, you are programming an iterative algorithm, then you may want, say, to intercept only divide by zero arithmetic exceptions so that you will know when to terminate the iteration.

Component Flavors

3.4.1 How do you create a Lisp object (the condition instance signaled by the divide operation) that implies "I am a divide by zero, which is a kind of arithmetic exception, which is a kind of error, which is a kind of condition"? The Lisp flavor system provides an ideal answer because it allows you to build flavors on other flavors.

For example, there is a specific flavor for divide by zero (named `sys:divide-by-zero`), which, among other things, records the divide by zero error message. But remember that a flavor can be built upon another flavor. In this case,

- the Divide by Zero Flavor (and other arithmetic exceptions) is built upon
- the Arithmetic Error Flavor (named `sys:arithmetic-error`) which, in turn, is built upon
- the Error Flavor (named simply `error`), which (along with all conditions) is built upon
- the Condition Flavor (named `condition`)

A divide by zero operation causes the Divide By Zero flavor to be instantiated. Because of the nesting of flavor definitions, this one instance of the Divide By Zero flavor carries along with it its entire pedigree, so to speak, which includes all the information necessary to treat it as a specific divide by zero error, an arithmetic exception, an error, or just as a general condition—depending upon what you are interested in.

A word of caution: at this point, the nomenclature begins to break down again. Since all conditions are defined as flavors, Lisp programmers tend to use the terms *condition* and *flavor* interchangeably. When they speak of an error condition, they are probably talking about a condition built on the Error flavor or an instance of that condition rather than about an erroneous situation. Similarly, a *condition instance* usually means an instance of a flavor that happens to define a condition. The final confusion is that each condition is represented by a unique flavor (of the same name), and all these flavors are built upon the base condition flavor. Therefore, there is such a thing as a Condition condition.

An Example

3.4.2 Finally, consider an example of how this hierarchy might be used to sort out conditions. Assume that your application program is doing file I/O and you would like to shield the user from some of the more obscure error messages built into the system by providing your own more informative messages. After consulting your handy reference manual, you discover that

- The `fs:file-operation-failure` condition flavor represents all of the I/O errors a user is likely to cause by mistake.
- The `fs:file-lookup-error` condition flavor is one of several flavors built on `fs:file-operation-failure` and is the one that covers the sorts of errors you get from, say, a typo in a pathname.
- The `fs:file-not-found` condition name is build on `fs:file-lookup-error` and indicates that the directory exists, but the specified file is not there.

Therefore, you might wrap a `condition-case` around your I/O code with several handler clauses such as those in the following:

```
(condition-case (signaled-instance)
    (...form containing file i/o...)
    (fs:file-not-found      ...handler forms...)
    (fs:file-lookup-error   ...handler forms...)
    (fs:file-operation-failure ...handler forms...)
    (error                  ...handler forms...))
```

which means:

- If the file system cannot find a particular file in an existing directory, then the handler for `fs:file-not-found` is called. A typical error message might be, `your directory exists, but it does not contain the file you asked for.`
- If the file system has some other problem in finding the file, then the handler for `fs:file-lookup-error` is called. A typical error message might be, `I cannot find your directory from the pathname you gave me.`
- If there was an I/O error that did not involve identifying the file, then the `fs:file-operation-failure` handler is called. A typical error message might be, `I got an I/O error while trying to access your file.`
- If any other error occurred, then the `error` handler is called. A typical error message might be, `I got a system error while trying to access your file.` [Remember, `error` is the name of a condition flavor as well as of a function.]

When the system is searching for a handler for a signaled condition, it checks each `condition-case` clause. It looks to see if any of the condition names associated with the signaled condition match the condition name (or names). In the above example, if `error` had been listed first as shown in the following,

```
condition-case (signaled-instance)
  ( ...form containing file i/o... )
  (error      ...handler forms... )
  (fs:file-operation-failure  ...handler forms... )
  (fs:file-lookup-error       ...handler forms... )
  (fs:file-not-found         ...handler forms... ))
```

then all errors would have shared that same generic “you got a system error” handler, and the other handlers could never be reached. For example, assume that an `fs:file-not-found` error is signaled. When the system begins to search the handler list, it finds an immediate match in the first entry, `error`, because `fs:file-not-found` was built on `error` and therefore is considered to be a type of `error`. In contrast, the general error handler *followed* its children in the first version; they got first chance to be called with the generic message reserved for unanticipated system errors.

Ad Hoc Condition Names

3.5 All conditions are built on flavors, and the names of these flavors can be used as condition names—but that is not quite enough. For example, suppose you have a function that detects ten fatal error situations. You want to signal an error condition for each such situation, but you don’t see the need for any special error handling other than a unique error message.

Unique Messages

3.5.1 From what has been explained so far, you might be led to believe that you will be forced to define ten new but trivial flavors to handle the ten error cases. Fortunately, however, you don’t have to go to this extreme. Since the message text is the most common difference among signals, you can just signal one error and provide a pertinent message for each one.

The easiest way to signal a general purpose fatal error is with the `error` function, which takes a `format-string` and `format-args` in the manner of the `format` function. For example,

```
(error "~a READ AN ILLEGAL CHARACTER ~c." fun-name in-char)
```

would be a typical fatal error message reporting that a certain function (represented by the argument *fun-name*) read a certain illegal input (represented by the argument *in-char*).

Appropriate error message text solves the problem of *human* readable errors but not *software* readable errors. If the only thing the error handling system ever did were to print messages to the terminal, then you would never need anything more than the `format` function. But the whole point of condition signaling and handling is that your programming power is multiplied many times if the software itself can detect and rationally respond to a signaled error.

Unique Errors

3.5.2 Returning to our previous example of your function with ten error signals, external software and the operator at the terminal are both able to tell that you have signaled an error—but only the operator can tell which one from the message text. Although the message text is available to the software, the study of artificial intelligence has not yet progressed to the point that software can reliably infer the nature of an error from a programmer's one-line error message. How can the software tell which of the ten errors was signaled?

Once again, it seems that we must resort to writing ten trivial flavors differing only in name so that the condition handlers can tell them apart. But once again, the answer is, "Not necessarily". The problem of providing software readable distinctions among signals is so common that extra condition names can be attached to a condition flavor instance.

That is, when a condition is signaled, an instance of some flavor is instantiated as described above. And also as described above, this instance automatically carries with it all the names of its component flavors as condition names. However, at instantiation time, you are allowed to add extra ad hoc names as properties of that instance.

Common Examples

3.5.3 If you look through system code, you will see that most fatal errors are signaled with the `ferror` (that is, *fatal error*) function as shown below:

```
(ferror nil "...format string..." arg arg arg...)
```

Actually, the `error` function mentioned above is implemented as a call to `ferror` with the first argument as `nil`, as is shown here. These calls provide unique error messages to an anonymous error condition.

The `ferror` function allows simple but limited access to the ad hoc condition names feature. For example, you could signal your ten different fatal errors by putting something like the following at each signal point:

```
(ferror 'err1 "...error message 1 ..." arg arg ...)  
(ferror 'err2 "...error message 2 ..." arg arg ...)
```

```
.
```



```
(ferror 'err10 "...error message 10..." arg arg ...)
```

Of course, you would hopefully come up with more meaningful signal names than `err1`, `err2`, and so on—but the error handler software doesn't really care. Each signal will now have a unique condition name that can be detected. The condition handling software arranges things such that all

condition names are treated the same way regardless of whether they represent a component flavor name or just a last minute add-on name.

In the above example, the actual condition flavor being signaled is **ferror** in all ten cases. But each instance of **ferror** has a different condition name in its property list. Regardless of how many ad hoc condition names may be used, there *must* be at least one flavor underlying each condition.

A Confession

3.5.4 If you took the time to check the system software, you will find that I've lied to you (but all in a good cause). The earlier text described each condition name as being represented by a distinct flavor. Actually, some are flavors (for example, **fs:file-lookup-error**), and some are merely ad hoc condition names attached to instances of other flavors (for example, **fs:file-not-found** is really just a condition name on the property list of an instance of **fs:file-lookup-error**).

This deception was useful to simplify the initial explanation. The deception is also not too important because all relevant condition name accessing functions handle the two different sources of condition names transparently. If you pressed the point, you would find that even experienced Lisp programmers don't always know (and fewer care) which condition names are really flavors.

Ad Hoc Hierarchies

3.5.5 Calling **ferror** with a unique symbol solved the immediate problem of creating software distinguishable signals, but what if we also wanted to treat all ten errors signaled from your function as a group on some occasions. What we need to do is to attach two ad hoc condition names to each signal: one is a unique symbol such as we are now unimaginatively calling **err1**, **err2**, and so on; the other is the same for all, such as **my-func**.

Unfortunately, **ferror** itself accepts only one condition name. The easiest way to get several is to use **defsignal**. Updating that last example, you would have the following declarations at the front of the file containing your function:

```
(defsignal fe1  (ferror err1 my-func))
(defsignal fe2  (ferror err2 my-func))

.
.

(defsignal fe10 (ferror err10 my-func))
```

and your actual signals would look like this:

```
(signal 'fe1  "...error message 1 ..." arg arg ...)
(signal 'fe2  "...error message 2 ..." arg arg ...)

.
.

(signal 'fe10 "...error message 10..." arg arg ...)
```

Consider the first **defsignal**: it defines a *signal name*, **fe1**, which is to be built on the flavor **ferror** (that is, the first symbol in the list) and is to have the additional *condition names* **err1** and **my-func**. **defsignal** actually allows several other pieces of information to be associated with the signal name, but these are the ones that interest us now.

Once the signal names `fe1` through `fe10` have been defined, you can signal any of them as many times as you wish with the `signal` function, providing a different error message each time. If you were to signal, say `fe5`, then an instance of the flavor `ferror` would be instantiated with properties that recorded the extra condition names of `err5` and `my-func`.

This particular example uses distinct signal and condition names for clarity. In practice, the signal name can be—and often is—the same as one of the condition names.

The following is an example of how you might use both the specific condition names and the general one too:

```
(condition-case ()  
    (...protected form...)  
    (err3    ...handler forms for error 3...)  
    (err7    ...handler forms for error 7...)  
    (my-func ...default handler forms...))
```

and here is what you are seeing:

- None of the error handler forms in this particular example need to reference the condition instance that was signaled, so the list immediately before the protected form is empty.
- Error 3 and Error 7 are handled uniquely.
- All other errors are handled as a group by a default handler.

Provisional Handlers

3.6 Sometimes a handler, especially a general purpose handler, is not able to handle all of the conditions handed to it. For example, an Arithmetic Exception condition handler may know how to overcome all arithmetic exceptions *except* for Divide By Zero conditions. Therefore, it needs a way of gracefully declining a condition that it has already tentatively accepted.

When a condition is signaled, the system begins its inside out search of condition handlers, looking for a match. When it finds one, then it gives the condition instance to the matching handler. If the handler finally tells the system, "I took care of it," then condition processing is finished. If, however, the handler tells the system, "I cannot take care of this one after all," then the original search resumes, first in any clauses remaining in the current condition handler and then in the outer level statements. See paragraph 3.7.3, A Sketchy Example, for an explanation of how a handler accepts or rejects a signal.

Error Recovery

3.7 So far, we have discussed various ways of informing special code (the handlers) that some noteworthy situation (the condition events) has occurred. Since few conditions are actually catastrophic and force processing to stop, we need a way for handlers to continue processing, possibly after some fixup, if that is a reasonable thing to do.

**Throwing and
Restarting**

3.7.1 One option is always available to condition handlers (or any other Lisp code, for that matter): a handler can always throw to a tag in some higher level enclosing code. This action has the effect of making the program go back and start over. This form of error recovery is so common that there are functions, such as **error-restart**, that do it for you directly.

Notice that **error-restart** is another unfortunately named Lisp function because it does not necessarily have anything to do with errors. It would be more appropriate to call this the **conditional-restart** function.

Proceeding

3.7.2 A second, and far more powerful, feature of the condition handling system is *proceeding* with *proceed types*. The details on how to define and invoke these proceed types are quite elaborate and are described in the *Explorer Lisp Reference* manual. These paragraphs will just give you an intuitive feeling for what they do.

Some error condition events have an obvious fix that might work and allow processing to continue. For example, if your program tried to access another host on the network and received a timeout, then you would probably want to be able to try it again knowing that these things sometimes just fix themselves. Similarly, if your program collected a pathname from the user at the terminal only to find that there was no such file, then you would probably want to ask the user if he had any other ideas. This is the notion of proceed types.

Three things must be done before processing can proceed from the point at which the condition was signaled:

- The condition flavor must have been defined with whatever proceed types might be pertinent.
- The code that signaled the condition must indicate in the signal which, if any, of this condition's proceed types it is prepared to handle.
- The handler code (which *must* be in a **condition-bind**) must either decide for itself or ask the user at the terminal whether it wants to proceed, whereby it returns the appropriate proceed type to the system, along with any new information that is needed.

The system then returns the fixup information, if any, to the code that originally signaled the condition along with a note as to how it is to proceed.

Notice that just because a condition has a potential proceed type, this does not mean that the code at the site that detected the condition knows how to correct the problem even if it were given the right data. That is why each signal point can itemize which proceed types it can use.

A Sketchy Example

3.7.3 Let's suppose that you have signaled an error that has several proceed types, all of which you can use if someone in the outer world gives you replacement information. At the point where you detected the error, you would signal it just as described above. If the user at the terminal (or some intervening error handler) decides not to proceed, then the **signal** function never returns. This is what **ferror** would do, for example.

If someone (or something) *does* decide to proceed, then the `signal` function you called returns with a proceed type keyword as its value. You then use that keyword in, say, a `case` statement to decide exactly what must be done.

Some proceed types imply the need for new data. For example, the `sys:wrong-type-argument` error condition has `:argument-value` as a proceed type that indicates a new value is to be substituted for the bad one. If the user decides to proceed from this error, then the `signal` function returns two values: the `:argument-type` keyword and the new value. In general, if a given proceed type requires N arguments, then `signal` returns N+1 values (that is, the proceed type keyword plus its N arguments).

Let's look at this same example from the point of view of a `condition-bind` error handler for `sys:wrong-type-argument`. If the handler decides not to handle the condition, then it returns `nil`, and the search for an error handler resumes. If the handler does handle the condition, it returns the multiple values described above (proceed type keyword plus arguments), which are then returned by the `signal` function.

Resume Handlers

3.7.4 The basic notion of proceed types implies that the range of possible ways to proceed is inherent in the condition and that the selection of which proceed types can be handled in a given instance is stated in the signal itself. However, there may be other ways to proceed with the *overall* process that are independent of the particular signal.

For example, assume that a function tries to open a file only to find that there is no such file by that pathname. That function can signal `fs:file-not-found`, which, reasonably enough, has the proceed type `:new-pathname`. However, the outer level of software that called the function in the first place may understand that a nonexistent file may really mean something else. Therefore, this software may want to offer some additional proceed types such as "give up trying to open that file". This notion of non-local proceed types is the basis for *resume handlers*.

You use a resume handler when you—with your broader view of what you are trying to accomplish—know that a reasonable way of handling certain errors is to try something seemingly unrelated. Therefore, you wrap a resume handler around the code that is likely to signal the error that interests you.

A fairly common example of this try-something-else-instead strategy can be seen in most errors that are handled by the debugger. The first few proceed types listed are specific to the particular error that was signaled. At the bottom of the list, however, will be some extra proceed types such as reset the process, return to top level, and so on. These are the nonlocal proceed types that are more like alternatives to the error than like ways of handling the error proper.

Ignoring Errors

3.7.5 It may be pushing the definition a little to class the `ignore-errors` function as a form of error recovery, but it is quite useful anyway. In the simplest case, you wrap it around a form if you want to say "Do this if you can, but never mind if you can't". For example,

```
(ignore-errors (delete-file "temp-pathname"))
```

says, "Delete my temporary file if you can. Otherwise, forget it." In the more general case, you can use this function as a pass/fail condition handler. For example,

```
(multiple-value-setq (stream failed-p)
  (ignore-errors (open "tentative-pathname")))
(when failed-p
  (setq stream (open "fallback-pathname"))))
```

says, "If I cannot open the first pathname (*tentative-pathname*) for any reason, then use my alternate (*fallback-pathname*)."

It may not be a good idea to ignore errors as blatantly as the above example did. It does come in handy, though, to suppress errors while you are trying to clean up during an error exit. That is, some fatal error has occurred, and you would like to cleanup as much as possible before you exit even though that clean-up may trigger other errors. By judicious use of **ignore-errors**, you can assure that your user will not be faced with an embarrassing error in error situation.

Signal Processing Summary

3.8 When a Lisp program or the system microcode detects, say, a **foo** situation, it does the following:

1. It assumes that a **foo** condition flavor
 - Has previously been defined
 - Has been built on the flavor or flavors of more general conditions that logically include this one in some manner (it can be the direct descendent of several different conditions, if that makes sense.)
 - Has instance variables, if needed, that will hold information unique to the particular situation that might be useful in handling the condition
 - Has a list of proceed types, if needed, that represent all of the things that might be done to recover from this condition gracefully if anyone wants to try
2. It instantiates the **foo** condition flavor with its instance variables initialized. (For example, a File Not Found condition flavor is instantiated with the **pathname** instance variable initialized to the pathname that could not be found.)

Since a condition flavor may be signaled from more than one point under a variety of circumstances, all of the proceed types defined in the flavor may not be usable from every signal point. Therefore, the instantiation of the condition flavor may limit which proceed types can be accommodated.

Notice that merely instantiating this condition flavor does not do anything in itself. Nothing happens until this instance is signaled. Although condition instances are normally instantiated as the conditions are detected and signaled immediately, they can be instantiated ahead of time and saved until needed.

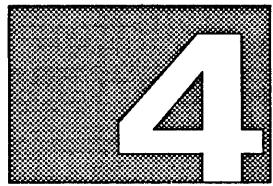
3. It signals this newly created **foo** condition instance, thereby officially informing the system that there is a condition that needs to be processed.

4. The system begins at the point where the **foo** condition was detected and searches the enclosing **condition-cases** (and related special forms) from the innermost to the outermost. Within a given **condition-case**, its handler clauses are searched in order.

If no regular handlers are found, the search returns to its original starting point and begins a new inside out search, this time for default handlers.

If no default handlers are found, the system goes to the handler of last resort. For conditions built on the **Error** condition, the handler of last resort is the debugger. For other conditions, the handler may do nothing more than resume the program as if nothing has happened; it all depends.

5. The search is looking for a handler to match the **foo** condition itself or *any* of the conditions upon which the **foo** condition is built.
6. When a handler is found that matches one of the component conditions in the **foo** condition, that handler is tentatively given that instance for processing. The handler must do one of three things after it has looked at the condition:
 - Change its mind and decline to handle the condition after all (in which case the original search resumes)
 - Do whatever processing it deems necessary (which may include prompting the user for suggestions based upon proceed types) and then **throw** to an outer level tag (which presumably resets the program to some earlier state before resuming execution)
 - Do whatever processing it deems necessary (which may include prompting the user for suggestions based upon proceed types) and then tell the system to proceed from the point where the **foo** condition was signaled



DEFSYSTEM AND MAKE-SYSTEM

Introduction

4.1 Let's assume you have an interpreted Lisp program composed of several files. What help would you like in managing this system? An aside: all the problems described here apply equally to a program of three files or to a system of 100 files. The only difference is that the more files you have, the harder it is for you to ignore the problem of managing the system.

- You want to load every file that needs to be loaded but you don't want to bother to load anything else.
- You want to load all files in the right order (if that happens to make a difference—as it sometimes does).

If you are worried about the time it takes to load your system, and you will be eventually, then you might also be interested in a related issue. You don't want to needlessly reload a file that is already satisfactorily loaded.

So far, we've been talking about the simple case of interpreted files—but what if you have compiled files? The list started above now gets more complicated. You can state the problem many ways, but it all comes down to making sure the program you are executing is the same as the source you are reading. The basic requirement is that you always want to load the compiled version (for the machine to execute) that matches the latest source (which you are reading).

This last requirement implies the final major set of requirements:

- You want to load the compiled version that matches the latest source, which may mean you have to recompile it *before* you load it if the compiled version is out of date.
- If you recompile, then you want the files presented to the compiler in the right order. (Before, you had to worry only about their being *loaded* in the right order.)

Our pair of simple sounding requirements at the top has grown to six not-so-simple statements of dependencies and conditions.

The Explorer's **defsystem** declaration and **make-system** function are intended to manage the problem described above. The **defsystem** defines a system by listing all of its parts and describing everything that needs to be done to compile and load the system.

make-system takes a **defsystem** description along with directives from you and manipulates your system for you. These manipulations range from a complete rebuild of the universe that causes everything to be recompiled and reloaded to a simple describe operation that tells you what the system has in it and what needs to be done to it.

Given the preceding introduction to the problem, what follows explains how **defsystem** and **make-system** work—after a lot of preliminaries.

Conditions and Dependencies

4.2 If it weren't for conditions and dependencies among files, there would be no reason for **defsystem**, for **make-system**, and for many gray hairs. Therefore, we must concentrate on what the required conditions and dependencies are and what causes them. Remember the difference between a dependency and a condition:

- A *dependency* is something like saying "I can't do *A* correctly until I know about *B*."
 - A *condition* is like saying "I need to do *A* only if *X* is true."
-

Common Conditions

4.2.1 Let's consider some of the obvious conditions and dependencies that you frequently run across. First, the conditions:

- The main compile condition is "Compile a source file if it's newer than its object version."
- A common supplemental compile condition is "Compile a source file if another file that defines code that must be *expanded inline* while compiling this source file (for example, a file containing **defmacros** and **defconstants**) is newer than this source file's object."
- The main load condition is "Load the version of a file on disk if it is newer than the version that is already installed in the system."

Although numerous other compile and load conditions can be used, these three are the obvious ones, and they cover just about everything you will normally need.

Common Dependencies

4.2.2 Load dependencies are usually the same as compile dependencies, so we won't treat load dependencies separately for now. Compile dependencies are common, and that's where all the fun is. The Lisp compiler has a significant set of do-this-before-you-do-that rules. If you could put everything in one file, then a lot of your problems would go away (and you certainly would not need a **make-system**), but you would still need to be careful of certain orderings (much more on that later).

If—as is most often the case—you must break your program into several files, then you must mentally analyze the ordering dependencies imposed by the compiler. Such analysis helps you know how to partition your code into separate files so you can finally state the code dependencies in terms of file ordering. Got that?

Or you might look at it this way: those heartless compiler writers have placed specific limitations on the order in which the Lisp forms in your program must be presented to their compiler. If everything were in one file, then you would just shuffle the source around until the compiler stopped complaining. If, however, things are in several files, then you must describe the necessary file ordering to a **defsystem** and then have a **make-system** handfeed the files to the compiler and loader according to what that **defsystem** says. If you have done the **defsystem** right, a hundred-file system compiles just as smoothly as a single-file system—if you have done the **defsystem** right.

On the other hand, the worst case is really bad. If you have been too casual about the way you broke your code up into files, then you may find that there is *no* order of compiling and loading that is unconditionally correct!

Summary of General Conditions and Dependencies

4.2.3 To recap our odyssey so far, we have listed six intuitive requirements for getting a system loaded and ready to execute, which can be boiled down to four guidelines as follows:

1. Compile any file whose object is out of date for any reason, but do not compile anything else.
2. Present the files to the compiler in the right order.
3. Load the latest version of all objects needed by the system, but do not load anything else.
4. Present the files to be loaded in the right order.

On closer inspection, those four requirements imply various conditions and dependencies for compiling and loading a system. Of these, the compiling dependences are the most detailed and hardest to get right. Finally those compiling dependences are imposed by the compiler, so before you even consider writing a **defsystem**, you need to figure out what the compiler expects of you.

However, if you try to wade through a stark list of DO this and DON'T DO that compiler ordering rules, then you will get discouraged sooner than necessary. Therefore, it is useful to understand the problems that Lisp presents to the compiler for which these rules are a solution.

Partitioning a Program Into Files

4.3 Figuring out how to partition a large program into multiple files is a small field of study in itself. Of course, you want to break your program into functional modules, but consider the following:

- If you really do it badly, there will be no way of compiling and loading your program and getting it right.
- If you don't do it well, then you'll find that **make-system** wastes a lot of time needlessly recompiling and reloading files.

The remainder of this section refers to macro-like definitions (or sometimes just macro definitions) that include **defmacros**, **defsubsts**, **defstructs**, **defflavors**, and (by a slight stretch of analogy) **defconstants**. These all have the common characteristic that at least part of what they define must be expanded inline by the compiler. Therefore, if their definition is changed, then not only must their definition be recompiled, but all source in which that definition was expanded inline must be recompiled too.

**Suppressing
Needless
Recompiles**

4.3.1 As mentioned above, the main compile condition is source newer than object. The need for a recompile in this case is obvious and always useful. It is the recompiles triggered by changes to macro-like source that are open to pitfalls. The chain of circumstances is as follows:

- **make-system** can only cause complete files to be compiled or loaded.
- Therefore, **defsystem** describes conditions and dependencies only among complete files.
- Therefore, if anything changes in a file, then everything in the file is recompiled (whether it was actually changed or not).
- Therefore, if a file is recompiled, then all other files that depend upon any macro definitions in the newly recompiled file must themselves be recompiled (whether their particular macro definitions were actually changed or not).

In an extreme case, changing one macro definition can trigger a recompile of the entire system. The rules of thumb to limit recompiles are as follows:

- If a macro-like feature is to be used only in one file, then place its definition at the top of that file.
- Otherwise, put macro-like definitions that tend to be used together into the same file. Therefore, changes to several macros still cause the same set of files to be recompiled.
- Avoid defining macros that use a lot of other macros, if at all possible. Otherwise, a cascade effect of a change to one macro file causing a change to all macro files can then cause all files to be recompiled—regardless of whether you need them to be.
- Put functions that use the same macro-like definitions in the same file or set of files while excluding other functions that do not use those definitions.

**It Worked Last
Night—Why Won’t
It Compile This
Morning?**

4.4 This is a good place to illustrate a delightful eccentricity of the Lisp environment: just because your program compiled cleanly one time does not mean that it will compile at all the next time. Why? Because almost everything you do modifies your environment.

Let’s suppose that you compile a file and get a warning something like `Error: use of X came before definition of x.` So, you innocently try compiling it again and it works! Your problem is solved (or so you think), and you continue happily on. But what almost certainly happened was that the second attempt to compile the use of X saw the definition of X left over in the environment from the previous compile, so no official warning was issued—but that does not necessarily mean you win.

What if between the two compiles you made a few changes, including a change to the definition of X. Now, during the second compile, the first uses of X will be compiled with the old definition while the later uses will get the new definitions. Since the compiler always had a definition available whenever it needed one, it never issued any warnings. You now have internal inconsistencies in the object as executed that do not appear in the source as

read. The compiler will try to do the right thing and warn you of inconsistencies like this that it sees. However, it can't see everything, so good luck.

The next day (that is, after the machine has been rebooted) you'll get the same "use before definition" warnings again. And again it will go away after a recompile. Think real hard. It's trying to tell you something.

Variation on a Theme

4.4.1 For plain vanilla files, doing a Compile Buffer command from the editor is supposed to have the same effect as a **compile-file** followed by a load from a Listener (which is what **make-system** essentially does). However, these two go about their job differently:

- Zmacs' Compile Buffer does an *incremental* compile and load. That is, each top level form in the buffer is first compiled and then loaded before the next top level form is compiled.
- **make-system**'s approach compiles the whole file in an environment that does not yet include any of the file's contents, and then it loads the completely compiled file.

So far, so good. These two alternatives behave as you would expect. However, if you take advantage of certain advanced programming features of the Explorer, then you might be in for a surprise.

In particular, you can bring problems on yourself if you deliberately force things to be done at compile time that are normally done at load or eval time. For example, wrapping an **eval-when (compile)** around a group of forms or prefixing a form with sharp-sign reader macro **#.** will cause it to be executed at compile time rather than just to be turned into code for later execution.

Of course, everything that appears in one of these forms must be known at compile time (not exactly the same as saying everything is compile time constants)—and that's the catch. If you are doing a Compile Buffer from Zmacs (incremental compile and load, remember), then the result of forms compiled and loaded at the top of the buffer are known at compile time for forms compiled at the bottom of the buffer. But! If you were to use a **compile-file** as **make-system** does, then nothing is known at compile time unless it is a true compile time constant.

For example, assume that you have a compile time calculation at the end of the file that relies on a variable defined by a **defparameter** at the beginning of the file. If you use Compile Buffer in Zmacs, then it will work because the **defparameter** is both compiled and loaded before the compiler continues. Therefore, the **defparameter**'s value is ready and waiting in the compile time environment when the later compile time calculation occurs.

But things are different for **make-system**'s **compile-file**. Again the **defparameter** is compiled, but its value is *not* loaded into the compile time environment. When the compiler reaches the later compile time calculation that attempts to use the **defparameter**'s value, you get a standard undefined variable error even though the definition for that variable (the **defparameter**) is sitting right there at the top of the file, as plain as day.

Therefore, if you use **eval-when** or **#.** to force extra calculations at compile time, then you will need an extra measure of embarrassment insurance.

Embarrassment Insurance

4.4.2 If you have gotten to a point where your program is finished and it works (or maybe it's not finished but you've got to turn it over to someone anyway); then you can buy some embarrassment insurance by doing the following:

1. A Precondition: if your program already permanently exists in the load band, then either try to kill it or use some other load band that does not have it installed for the following steps.
2. Starting from a cold boot, recompile your entire program without loading anything not needed by the compiles [for example, (`make-system 'foo :recompile :noload`)].
3. Starting from another cold boot, load your entire program without recompiling [for example, (`make-system 'foo`)].

If both the compile and the load went smoothly, then you can have a strong degree of confidence that your program can be ported and installed on other systems. On the other hand, if you've never done this before, you may be surprised and dismayed (I know I was).

Summary of Compile Conditions and Dependencies

4.5 The following represents the most important information that a `defsystem` typically must describe. These rules follow from one simple observation:

If the compiler uses some piece of information it found in File A to compile a piece of code from File B; then if that information in File A changes, File B must be recompiled.

The problem with that observation is figuring out which piece of information the compiler remembers from which form for later use in compiling other files.

For example, the compiler remembers only that a `defvar`'s symbol was proclaimed special—it does not need to remember that symbol's initial value after it has been compiled. Since most debugging changes you might make to a `defvar` are to its initial value, changing a `defvar` is not a reason to recompile another file that uses it.

In contrast, the compile *does* remember a `defconstant`'s value (after all, that's the purpose of the `defconstant`). Therefore, any change to a `defconstant` is reason to recompile all other files that use it.

When the following discussion refers to “placing X so that the compiler sees it before Y”, it is referring both to the ordering of X and Y within a single file and to the ordering of files presented to the compiler by `make-system` because of a `defsystem` declaration. Notice that the admonition that the compiler must see X before Y is confounded by back-to-back compiles. The second compile will always see X before Y even though some of the X's may be old definitions, and some may be new ones. Therefore, the consequences described below are true for only the first compile and load following a cold boot.

- For correct execution, place the **defsetf** so that the compiler sees it before it sees **setfs** of that object. Otherwise, the compiler will complain that it does not know how to **setf** the object.
- For efficiency, place **deftype** before you use the type it defines. However, execution is correct regardless of the order.
- The **defresource**, **defsignal**, and **defsignal-explicit** macros have *no* ordering constraints for either efficiency or correct execution.
- For correct execution of the special forms you define by using a " in a function's lambda list:
 - You must always place the special form definition so that the compiler sees it *before* it sees any calls to that special form. Otherwise, the function will be called at run time with its quoted argument(s) evaluated.
 - You must recompile all calls to that special form only if you change which arguments to the function are quoted. Otherwise, the function will be called at run time with the wrong arguments evaluated.
- For correct execution of macros you define by using **defmacro**:
 - You must always place the **defmacro** so that the compiler sees it *before* it sees any calls to that macro. Otherwise, code attempts to illegally **funcall** a macro at run time.
 - You must recompile all calls to that macro only if you change the definition such that it expands differently. Otherwise, unrecompiled code still uses the old macro definition.

NOTE: If, and only if, you have provided an **optimize** declaration with **safety** equal to 0 and **speed** greater than **size** and you have not provided a **proclaim notinline**, then the compiler can choose—at its discretion—to make short functions inline even though you did not **proclaim** them **inline**. Use **who-calls** to check whether this has happened to a particular function. If the function is listed as **used as a macro**, then this function was expanded inline (whether by a **defmacro**, **defsubst**, or **proclaim inline**).

- Of the functions that you have **proclaimed** to be **inline**:
 - For efficiency's sake only, you must place the **proclaim**, the **defun**, and the calls to that function so that the compiler sees them in that order. Otherwise, the calls are treated as ordinary function calls rather than as being expanded inline.
 - For correct execution, you must recompile all places where that function was expanded inline only if you change the function such that it expands differently. Otherwise, unrecompiled code still uses the old definition.

- Of substitutable functions you have defined with **defsubst**:
 - For efficiency's sake only, you must place the **defsubst** so that the compiler sees it *before* it sees any *simple* calls to that subst. Otherwise, the calls are treated as ordinary function calls rather than as being expanded inline.
 - For correct execution, you must place the **defsubst** so that the compiler sees it *before* it sees any nonsimple uses of that subst as a *generalized variable* (for example, as the place argument of a **setf**, **incf**, **push**, and so on). Otherwise, you will get a compiler warning that it does not know how to **setf** that subst.
- For functions defined automatically for structures you have defined using **defstruct**:
 - If you are in Common Lisp Mode and accept the defaults for callable constructors, for callable accessors, and for no alterant; then for correct execution treat the **defstruct** as a **defsubst**.
 - For correct execution in *all* other cases (including Zetalisp Mode), treat the **defstruct** as a **defmacro**.

NOTE: Recall that a **defstruct** actually defines a number of functions. If *struct* is the name of a structure and *slot* is the name of a slot in that structure, then **defstruct** creates all of the following:

- A funcallable and setfable accessor function *struct-slot*
 - A funcallable constructor function *make-struct*
 - A funcallable predicate *struct-p*
 - A funcallable copier function *copy-struct*
-

- For correct execution of variables you have proclaimed to be **special** or have defined with **defvar** or **defparameter**:
 - You must always place the **proclaim**, **defvar**, or **defparameter** so that the compiler sees it *before* it sees any uses of that variable. Otherwise, you get a compiler warning that it does not know what the variable is and has assumed that it is special.
 - If you should change your code so that a variable previously proclaimed or defined to the compiler as special is *no longer* special, then you must unbind that variable name with **makunbound** and must recompile all places that used that variable.
- Of constants you have defined by using **defconstant**:
 - For efficiency's sake only, you must place the **defconstant** so that the compiler sees it *before* it sees any uses of that constant. Otherwise, the value will be accessed as just another **defparameter** value at run time.
 - For correct execution, you must always recompile all places where a constant was expanded inline if you change that constant's value. Otherwise, the unrecompiled code uses the old value.

- Of functions you have defined by using **defun-method**, by placing a function definition inside a **declare-flavor-instance-variables** macro, or by placing a **declare :self-flavor** in a function definition (collectively referred to as *pseudo-methods* below):
 - For efficiency's sake only, you must place the pseudo-method so that the compiler sees it *before* it sees any calls to it. Otherwise, the function will suffer extra run-time entry overhead each time it is called.
 - For correct execution, you must call a pseudo-method only from within a method or another pseudo-method of the same parent flavor. Otherwise, the preinitialized environment the pseudo-method is expecting upon entry at run time does not exist, and you get arbitrarily weird results.
 - For correct execution, you must place the parent **defflavor** of the pseudo-method so that the compiler sees it *before* it sees the pseudo-method. Otherwise, the compiler mistakes references to the flavor's instance variables in the pseudo-method as references to free variables.
- For correct execution of functions you have defined using **defmethod** or **defwrapper**: you must place the parent **defflavor** of the **defmethod** or **defwrapper** so that the compiler sees the **defflavor** *before* it sees the **defmethod** or **defwrapper**. Otherwise, the compiler has no place to record the **defmethod** or **defwrapper** and will mistake references to the flavor's instance variables in the **defmethod** or **defwrapper** as references to free variables.
- For correct execution of **:select-method** function specifications: you must place the parent **defselect-incremental** so that the compiler sees it *before* it sees any of its **:select-method** function specifications. Otherwise, the compiler has no place to record the select method.

In general, the rule of thumb is **defs before refs**. That is, place definitions so that the compiler sees them *before* it sees any references to them. Typically, if the compiler sees improperly ordered forms during a compile of one file, it will warn you (that is, a **defmacro** follows a use of the macro). However, the compiler has a hard time seeing across compiles of different files, so you must accept that responsibility.

defsystem and make-system Tips

defsystem's :compile-load Transform

4.6 A number of features and implementation details of **defsystem** and **make-system** repeatedly cause confusion. This paragraph is an amorphous collection of things you should watch out for.

4.6.1 :compile-load is the single most common transform in **defsystem**. Unfortunately, the action implied by its name does not quite match what it does in real life. For example, if you had the following **defsystem** fragment:

```
(:compile-load a-mod)
(:compile-load b-mod)
```

then this is the action sequence you would intuitively expect:

```
compile a-mod
load   a-mod
compile b-mod
load   b-mod
```

whereas this is what you actually get:

```
compile a-mod
compile b-mod
load   a-mod
load   b-mod
```

That is, **make-system** will normally do *all* compiles in the order listed followed by *all* the loads in the same order.

In the above example, if the correct compilation of *b-mod* actually does depend upon *a-mod* already being loaded, then your **make-system** will fail. If *b-mod* has a compilation dependency on *a-mod*, then that dependency must be listed in the compilation dependency field of the **:compile-load**. For example,

```
(:compile-load a-mod)
(:compile-load b-mod (:fasload a-mod))
```

correctly states the dependency you are counting on, and your **make-system** will work.

A comment was made before in passing that load dependencies are usually the same as compile dependencies. The main reason is that the compiler routinely leaves certain classes of forms uncompiled in the object file so that they can be evaluated at load time or at run time. Since it is not always easy for you to discern which is which, it is a safe bet to assume that if the compiler needs to know something to compile the file correctly, then the loader may need to know the same thing. Therefore, we should rewrite that previous example:

```
(:compile-load a-mod)
(:compile-load b-mod (:fasload a-mod) ; compile dependencies
                     (:fasload a-mod)) ; load dependencies
```

**defsystem's
:compile-load-init
Transform**

4.6.2 In the documentation, the `:compile-load-init` transform is unfortunately listed as one of the esoteric transforms. Actually, this transform is possibly the second most common after `:compile-load`—or at least it should be.

Let's continue to develop the previous example by adding another module called `macros` that contains a bunch of `defmacros`. Further assume that both `a-mod` and `b-mod` use macros defined in the module `macros`. Now, remembering the compile dependencies, you might write the following:

```
(:compile-load macros)
(:compile-load a-mod (:fasload macros)
                     (:fasload macros))
(:compile-load b-mod (:fasload a-mod macros)
                     (:fasload a-mod macros))
```

which is correct as far as it goes. Whenever `a-mod` or `b-mod` needs to be compiled, the macro definitions in `macros` are loaded first.

But that's not the end of it. The macro bodies are actually expanded inline in the object files of `a-mod` and `b-mod`. If the `macros` module is changed and recompiled, then `a-mod` and `b-mod` are probably left with old versions of the macros. Every time we recompile `macros`, then we must also recompile `a-mod` and `b-mod`. Therefore, we need the following:

```
(:compile-load macros)
(:compile-load-init a-mod (macros) ()
                     (:fasload macros))
(:compile-load-init b-mod (macros) (:fasload a-mod)
                     (:fasload a-mod))
```

which adds an extra compile condition to both `a-mod` and `b-mod`. Notice that the presence of `macros` and an extra compile condition implies that it is also a compile dependency, so you don't have to list it again—but you could if you wanted.

The rule of thumb is that each module containing macros or any other forms that expand inline must be represented by `:compile-load-init` transforms. The extra compile conditions list of these transforms must include all other modules defining macros (and their friends) that the `:compile-load-init` module uses. Notice there is no `:fasload` or `:readfile` prefix to the module names in this list.

**make-system's
:compile Option**

4.6.3 Very large systems may take an hour or more to compile everything and then load it. However, if only a few files of the many files that make up the system were changed, then the `:compile` option of `make-system` attempts to do the minimum work necessary to get the system back up-to-date. In particular, the `:compile` option does the following:

1. Determines which compilable files in the `defsystem` have source newer than object and marks those to be compiled.
2. If any of these files marked to be compiled are mentioned as an extra compile condition of a `:compile-load-init`, then it also marks the files of that `:compile-load-init` for compilation.

3. When given the total list of files to be compiled, it collects all their dependencies (usually just **fasloads** of other files) and marks them to be done.

This technique is efficient and powerful, but it does assume that the **defsystem** declarations it was using were *fully constrained*. That is, it assumes that you have used **:compile-load-init**s wherever needed and that all compile and load dependencies are listed on each transform.

If you have *not* properly listed all the dependencies that exist among the files in your **defsystem**, then **make-system** still recompiles files whose source is newer than their object. However:

- It will not recompile other files that contain calls to macros that are being recompiled.
- It will not get all necessary files loaded into the environment before the compiling starts.

Therefore, if you have not taken the time to properly constrain your **defsystem**, then you should never use the **:compile** option on **make-system**. Instead, you should use **:recompile** to cause everything to recompile regardless of whether source or object is newer.

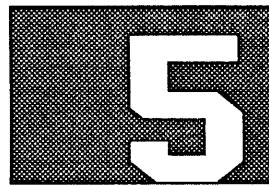
make-system's :reload Option

4.6.4 If you have not done any **make-systems** since the last reboot, then the **:reload** option does what you expect: causes all files to be reloaded regardless of whether they have been loaded before.

In other situations, however, **:reload** does something unexpected. If there has been a previous **make-system**, then **:reload** causes the transforms of that previous **make-system** to be reexecuted. Notice that this action is *not* the same as simply resubmitting the previous **make-system**.

For example, if you did a **make-system** with a **:compile** option, then only your modified files are recompiled. If you now do a **make-system** with a **:reload** option, then the same set of compiles and loads is done again. Contrast this with reentering the original **make-system :compile**, which would do nothing the second time because everything is freshly compiled and loaded.

As a rule of thumb, you should avoid **make-system's :reload** option because it seldom does what you think it is going to do. If you are in doubt about what any of these **make-system** options is going to do, then enter the **make-system** into a Lisp Listener along with whatever options you are interested in and then add the option **:print-only**. **make-system** will now list everything those other options would have caused it to do, but it will not actually compile or load anything.



LOADING AND PATCHING

Introduction

5.1 Most computers have a way of patching a program already installed. The Explorer system also supports patching—but in a form and with capabilities unfamiliar to the conventional world of assemblers, compilers, and link editors. Generally speaking:

- Ordinary patching involves permanently changing locations of a program's executable disk file.

On the Explorer system, you modify your active environment (that is, the copy of your load band in virtual memory). To make the change permanent, you must save this modified image to another load band. Therefore, you never directly modify a disk image—you create a new one.

- Conventional patching replaces existing locations in the executable disk file with new values so that the program size does not grow (but total patching is limited in size and functionality).

The Explorer system appends the changes to your active environment leaving the old code in place so that the load band size grows (and patching is essentially unlimited).

- Most patch writers try to distill the change down to as few locations as possible.

On the Explorer, an entire function is replaced if anything in that function needs to be changed.

Conventional systems have special utilities to apply their patches. In contrast, the Lisp Loader patches old code by doing ordinary Lisp loads of ordinary compiled Lisp source. Therefore, an introduction to patching must start with an introduction to loading.

Redefinition Versus Patching

5.2 The basic modification technique in the Lisp world is *redefinition*, whereas conventional machines use *replacement*. Both techniques are referred to as *patching* by their respective users, to the confusion of all.

Replacement simply means that one or more bytes of an executable image in memory or on disk are replaced with a different bit pattern. The replacement approach usually means that add-on code or data must be limited to space already reserved in the executable image.

Conventional patching utilities typically know nothing about an executable image except that it is an arbitrary byte array. These utilities cannot know program-specific information such as starting location of functions, locations of variables, dimensions of array, and so on. Therefore, patching tends to be in terms of specific bit patterns to specific addresses, which greatly discourages large replacements or the use of high level languages to define the patch. In fact, programmers sometimes equate the term *patch* with *hand-assembled hex code*.

Redefinition, on the other hand, means that alternate function bodies and data structures are appended to the current system, leaving the original versions intact. Then all existing references to these functions and structures are diverted to the new versions. Of course, if the only required change was to replace one data value with another, then Lisp just replaces it, since no definition is involved. That is, patching Lisp code (redefinition) is a little different from patching Lisp data (replacement). See paragraph 5.6, How to Patch the Environment.

Therefore, even if only one word in a function were wrong, a Lisp patch would create a whole new function body (with that word changed) and add it to the system. At first glance, the redefinition approach would seem to require an extremely complicated and time-consuming patch utility. Actually, this all happens to be done transparently by the Lisp Loader, as explained below.

In the following paragraphs, just remember that new code is always added to a Lisp system, and old code is modified through redefinition. When Lisp programmers talk about patches, they are really talking about a set of software maintenance conventions built on top of redefinition and *not* about bit-twiddling.

How the Loader Works

5.3 We can start with a small simplification for this discussion: it does not matter whether the Loader is loading Lisp source or XLD object files—they both behave the same way at the level we are discussing.

The Loader loads a file by executing each top level form (that is, each Lisp form not nested in another form) as it reads that form. For example, when the Loader reads a Lisp file, its action is very much as if you were rapidly typing that same Lisp source into a Lisp Listener. Each time you type a closing top-level parenthesis, the Listener executes that form then and there before looking at anything else you have typed.

We intuitively think of a loader as reading bits from a disk file and stuffing them into memory in some appropriate format, but not executing them. In contrast, the Lisp Loader simply acts like a Lisp Listener for disk files. Some Lisp forms have a side effect on virtual memory when executed, and some don't. Of those that have a side effect, some create definitions and some modify variables. For example,

Form Loaded	Side Effect on Memory
(+ 1 2)	None
(setf a (+ 1 2))	Set value of symbol a=3
(defun foo ...)	Define function foo

Executing a **defun** form, whether in the Listener or the Loader, acts to *define* a function body rather than to execute it. The act of definition includes the following major steps:

1. Allocate a *new* block of virtual memory and store the function's body in it.
2. If the function's name (in this case, **foo**) is not already interned as a symbol, then intern it; otherwise use the symbol already interned.

3. Place a pointer to the virtual memory where the function's body is stored in the definition cell of the symbol `foo`.

All later run-time calls to this function find the function code by using this definition cell of the function's name symbol. The implication of all of this is that the Lisp system maintains a permanent symbol table (actually several dozen of them, called *packages*) of everything ever loaded. The Loader updates these symbol tables, and all symbolic accesses use them. Therefore, the Lisp system implements something similar to a linking loader.

Notice that the function's name symbol is reused if it already exists (regardless of what it was previously used for), but new memory is always allocated for the function's body. We will later see that this characteristic is the foundation of the Lisp patching technique.

How Redefinition Works

5.4 What if you load a `defun` of `foo` and then load it (or perhaps a different `defun` of `foo`) again? After the first load, there exists in virtual memory a symbol `foo` with a definition cell that points to the code body of the first `foo`. After the second load, the same symbol `foo` is still there, but its definition cell now points to a new block of memory that holds the second definition of `foo`.

- New memory was allocated for the second body (even if the second body were identical to the first).
- The old memory that held the first body is still there undisturbed.
- The first body is remembered as the `:previous-definition` property of the symbol `foo`. A symbol's definitions cell and its `:previous-definition` property act as a two-entry pushdown stack. Therefore, the third `defun` of a symbol causes the first definition to fall off the bottom of this stack. The `undefine` function swaps these two stack entries.

In short, a simple reload of any function definition automatically patches the previous definition by diverting all later calls to that function to the new body.

More Than Just Patching

5.4.1 Part of Lisp's widely advertised programming development environment is built on this redefinition ability. You can enter a function definition into a Zmacs buffer, compile that function, and then execute it. If the test uncovers a problem, you can re-edit, recompile, and reexecute it as many times as necessary.

Actually, when you compile a Zmacs buffer with the `Compile Region` or `Compile Buffer` commands, you are in fact compiling *and* loading it. The traditional link edit step is not needed since the Loader is maintaining those global symbol tables in real time. Therefore, normal software development on a Lisp machine has little functional distinction from patching.

When Is It Really a Patch?

5.4.2 While Lisp's loading technique offers a transparent method of patching, such transparency can be a problem:

Given two function definitions with the same name, how does the Loader know if the second is meant to patch the first or if you have accidentally given a new function the same name as an old one?

To get around this problem, the Loader uses a simple heuristic that does the right thing most of the time.

Whenever the Loader loads a function definition (or any symbol definition, for that matter), it records on that symbol's :source-file-name property the pathname of the file the definition was in. If a later redefinition comes from the *same* place, the Loader does not complain because it assumes you are just debugging the original source file. However, if a function is redefined by a *different* file, then the Loader issues a warning and asks if you really wanted to redefine this function. Your responses can be any of the following:

Response	Description
Y	[also T or SPACE] Yes, redefine the current definition with the one just read.
N	[also RUBOUT] No, leave the current definition in place and discard the one just read.
E	Enter the error handler.
P	Proceed by redefining the current definition, and accept all later redefinitions between these two files without further queries.

The Proceed option is best explained with an example. If **foo** was originally defined in, say, File-A and then later another definition was found in File-B, then the Proceed option tells the Loader to automatically accept the current redefinition in question plus all redefinitions of other functions originally found in File-A and later also found in File-B. Redefinitions of File-A's definitions by any file other than File-B are still flagged.

You will find this Proceed option useful if you should ever have to rename a file. Without this option, the Loader would stop and ask you about every single definition in the whole file (and there could be dozens).

You can control the Loader's reaction to redefinitions by binding the global variable **inhibit-fdefine-warnings** to one of the following symbols:

Possible Symbol	Description
nil	Output the warning and then query the user about how to proceed. (This is the default.)
t	Ignore redefinitions: do not warn, do not query. (Note: this <i>must</i> be the symbol t , not just non-nil.)
:just-warn	Output the warning and then proceed without querying.

Similarly, if you add the attribute **patch-file:t** to a file's file attribute list (use META-X Set Patch File in Zmacs), then the Loader will effectively bind **inhibit-fdefine-warnings** to **t** for you while it loads that particular file. Patch files generated by Zmacs use this feature.

**The Down Side
of Patching**

5.4.3 When a file is initially loaded into the system, all of the definitions in it end up on adjacent virtual memory. That is, there tends to be good locality of reference among related functions. However, patching causes a new definition to be created in a remote piece of virtual memory, thereby destroying locality.

Therefore, although a patched definition is functionally indistinguishable from the original version, there is a potential performance penalty if that function is called from a time-critical block of code.

**How Creating a
Patch Works**

5.5 Since a patch file is an ordinary Lisp file, you can manually create a patch yourself—if you remember all of the details. Zmacs has several commands defined to make things easier by taking care of those details for you. Before getting into a discussion of how patching works from Zmacs, let's consider why those details are important.

Patch File Details

5.5.1 The extra details that go into a patch file can be divided into two groups: those required for correct compiles and those that provide a patch audit trail. First, let's consider the required details.

Normally when a function is compiled by **make-system**, it is actually in the middle of a file of many functions all being compiled at the same time. The file attribute list of a function's parent file effectively sets up a miniature environment in which the function is compiled. The main attributes of this environment are:

- Mode (Common Lisp or Zetalisp)
- Base (usually 8 or 10)
- Package
- Fonts (if this file includes fonts)

If you are going to be able to compile a function properly when it is isolated in a patch file, then you must reestablish at least this much of its original environment.

However, simply copying these attributes into the file attribute list of the patch file isn't good enough. A patch file may have several function redefinitions, each with its own requirements. Therefore, Zmacs first wraps a **compiler-let** special form around the function definition to establish things for the compiler, and then it prefixes the **compiler-let** with reader macros to establish things for the Lisp reader. In the end, we have a half dozen or so lines prefixing each function body in the patch file, looking something like this:

```
#!c #10r foo-package: ; reader macros
(compiler-let ((...    ...)) ; compile time bindings
               (...    ...))
               (...    ...))
(defun bar           ; patch for function BAR
  -- -
  -- -))
```

Since each function body now carries its own package declaration, the **package** attribute in the patch file's file attribute list is actually unused. Therefore, if you notice a progress message from the Loader claiming that your patch file is being loaded into the user package, ignore it. Each function in that patch file is going into whichever package it belongs in. During the load, you can watch the package name in the status line change as the Loader encounters each function body.

Now, let's consider the details that are added to a patch file to establish an audit trail. You may, of course, put anything in there you want such as, perhaps, bug report disposition information. If you create the patch using Zmacs commands, Zmacs adds the following comments to the front of the patch file for things like the following:

- Time, date, and login name of the programmer making the patch
- Machine name and load band on which the patch was made
- A copy of the patch documentation string (described below)
- A poorly formatted list of all loaded systems and their current version numbers

In addition, Zmacs inserts a comment in front of each function body specifying that function's parent file. It needs to record the source of each definition in the patch file individually because the fix being implemented by the patch file may touch several different functions in several different files. Therefore, a single comment at the top of the file would not be enough.

Using Zmacs To Create a Patch

Correct the Source

5.5.2 The following is one way to use Zmacs to create patches for you. Once you are familiar with this sequence, you can experiment with the related Zmacs commands.

5.5.2.1 Edit the source file(s) to make your correction(s) and then test it. This step is not really different from how you would go about debugging any problem. Saving the source file(s) when you are finished effectively modifies the source for the next build. However, modifying the source does nothing about getting the change to the field before the next build. Now you must mark the region.

Mark the Region

5.5.2.2 Mark a form you just modified in the source file (that is, point at its open parenthesis with the mouse and click once on the middle mouse button). If you don't mark a region, Zmacs takes the top level form surrounding the cursor.

Remember, Zmacs is going to blindly take whatever is in this region and make a standalone patch out of it. So make sure you mark something (such as an entire function definition) that can reasonably be compiled and loaded by itself.

Execute Add Patch

5.5.2.3 Execute the Zmacs command Meta-X Add Patch. The first time you do an Add Patch, Zmacs asks you what system you are patching. After you specify the system name, Zmacs then does the following:

1. Finds that system's defsystem and from it the patch directory

2. Warns you if you have not already loaded all patches for the system you are now patching
3. Determines what the pathname of the next patch file will be
4. Creates a new patch buffer with that pathname
5. Places the marked region in it along with all the other details discussed above

If all you needed to change was just the one form in this one file, then you are finished with this step. If more changes are needed for this fix, then iterate on the Mark Region and Add Patch steps. You may include forms from several source files in one patch file. All Add Patch commands after the first one merely append to the current patch file.

Execute Finish Patch

5.5.2.4 Until you execute the Zmacs command Meta-X Finish Patch, the patch file you are accumulating is available as an ordinary Lisp file in the Zmacs buffer. Therefore, you are able to add to it yourself manually or to edit it as you please. We will see later why you often must edit a patch file before doing a Finish Patch command.

When you execute Finish Patch, Zmacs prompts you for a documentation string (mentioned above) that briefly describes the purpose of this patch. This documentation string can be as long as you want, but you should choose your words carefully. This string is likely to be read by people, possibly customers at a remote site, where flippancy may not be fully appreciated. Furthermore, this comment should be phrased in terms of what benefit the user will get from this patch rather than in terms of what you fixed.

Once Zmacs has collected your documentation string, it saves the patch file to the associated system's patch directory and compiles it. When Zmacs finally returns, the patch is complete.

How To Avoid Common Mistakes

5.5.2.5 To avoid common mistakes, keep the following points in mind:

- Before you begin to debug any system, be sure that you have loaded all patches for that system. Otherwise, you will be executing in an environment that does not match the source you are patching. Any testing that you do in this situation is suspect.
- Be very sure that you have included in the patch every change you originally made to the source while debugging it. If all of the changes you made to the source to get it working are not reflected in patches, then a freshly booted system with your patch loaded will *still* not work.
- Zmacs saves off the patch buffer to the patch directory, but it does *not* save off the modified source files the patch was made from. If you forget to save off all of the source changes, then your patch may work fine, but the patch will disappear after the next build.
- Sometimes a project may choose to have patching done on a special file server. Sometimes project personnel don't pay attention to which file server they are using. So, after you think you've saved off everything, go look on the official file server to see if your newly modified source files are really there.

- If the patched file is known by a logical pathname, Zmacs tries to determine that logical pathname and record it in the patch buffer, regardless of what name you used to load the buffer for editing. Double-check that this pathname in the patch buffer is the right one.
- If your patch involves structure-defining forms such as `defstruct` or `defflavor`, then be aware that the patch will only affect *new* instances of these structures. If correct execution also requires that existing instances be changed, then you will also have to add environment patches (discussed below).
- A `defstruct` in Zetalisp mode with defaults or a `defstruct` without callable accessors and constructors in either mode creates hidden macros. Therefore, you must recompile every place that accessed or constructed one of these `defstructs`. The basic `defstruct` in Common Lisp mode doesn't cause this type of problem.
- If you edit the patch file in the patch directory *after* you have done a Finish Patch (that is, after Zmacs has compiled it), then remember to recompile your change. Otherwise, META-. shows you the newly modified Lisp source, but `load-patches` loads the old XLD object.
- When everything is finished, cold boot, load all of your system's patches, and run your tests one last time. Your original debugging session may have had several false starts that change your environment in subtle ways. Loading patches after a cold boot is the way the customers in the field will see it, so that's what counts.

How To Patch the Environment

5.6 So far, we have talked only about a patch redefining a function or function-like object (for example, a flavor method)—but that is only half the story. Step back for a moment to get a higher-level view: there is something wrong with the system, and the patch is supposed to fix it.

If that something is just a bad function definition, then we've already seen how to handle it. If, however, that something is a bad data structure, then changing its definition is necessary, but it isn't enough. New instances of that data structure created after the patch will be correct, but the original ones will still be there.

Patching a Data Structure Definition

5.6.1 For example, suppose your system contains the `*bar-record*` variable, which is an instance of the `bar` `defstruct`. The original source code might have looked something like this:

```
(defstruct bar ...)  
(defvar *bar-record* nil)
```

where `*bar-record*` is initialized to an instance of the `defstruct bar` later in the code.

Now let's suppose you find that the `defstruct bar` was wrong. Your natural inclination is to just patch the `defstruct`, but that is only half the answer. To make this patch both redefine the `defstruct bar` (for use by later instantiations) and reset the value of `*bar-record*`, we must create a patch file something like:

```
(defstruct bar ...)  
(setf *bar-record* (make-bar ...))
```

The new **defstruct** of **bar** corrects the definition (just as we have done before with functions) while the **setf** of ***bar-record*** corrects the existing data structure.

This new definition of the **defstruct** must be reflected in the source code so that it will be correct on the next build. However, the **setf** form is just a random piece of fixup code for the patch file only.

At the beginning of this section, it appeared that top level forms that did not define something were just so much wasted effort on the part of the Loader. Now we see that they have their own set of uses. However, creating a patch file with forms such as this **setf** is not quite as automatic as patching definitions.

If you were using Zmacs to create the above patch file, you could edit the **defstruct** in the source, mark it as a region, and then use an Add Patch command as before. But there may be nothing in the source file you can mark as a patch to accomplish the **setf**. This is the case alluded to above where you must manually edit Zmacs' patch buffer before you do the Finish Patch command to add the **setf** (or whatever fixup is needed).

Patching a Flavor Method

5.6.2 If you are patching a method of a flavor and that flavor has

- *no component flavors*
- *no combined methods* (for example, **:before** or **:after** methods, and so on)

then you can skip this section. Otherwise, you need to know how to compile flavor methods.

Despite what it sounds like, *compile flavor methods* does not refer to the compilation of either **defflavors** or **defmethods**. Even though a flavor and all of its methods have been compiled and loaded, several special tables must be created for a flavor before it can be instantiated. This extra table creation step (referred to as *compiling flavor methods*) is part of the magic that allows a flavor—which may have been defined as a complex hierarchy of component flavors—to execute at run time as if it were a simple flavor with a bunch of methods you wrote just for it.

Normally, your Lisp source code contains the **defflavor** followed by all of the **defmethods** for that flavor. For run time efficiency of flavors with component flavors or with combined methods, you should also include **compile-flavor-methods** for that flavor after its last **defmethod**—for the following reasons.

When the system instantiates a flavor for the first time, it checks to see if that flavor's tables are freshly compiled (that is, no new methods defined after the last flavor methods compile). If everything is already compiled, the instantiation is done immediately. However, if the compilation is out-of-date (for example, you've patched a method in the meantime), then the system stops then and there and does another flavor method compilation—which can cause a noticeable delay to your user.

To save your users a run-time wait, you can append a **compile-flavor-methods** form to the end of any patch file that includes a redefinition of a **defmethod**. If several different patches all patch methods for the same flavor, then you need the **compile-flavor-methods** form only after the last

one. Multiple calls to this form are okay, but all compiles except the last one become garbage taking up space in the load band.

There is a simple way to check whether you have forgotten a **compile-flavor-methods** form in your patch files (or in your source, for that matter).

1. Set the global variable **sys:*flavor-compilations*** to nil.
2. Load and patch your system.
3. Exercise your code (try to make sure you instantiate each of your flavors at least once).
4. Examine **sys:*flavor-compilations***. It will contain a list of all flavors that had their flavor methods compiled on the fly during run time.

Things You Can't Patch

5.6.3 Actually you can patch anything; it's just that patches for some things have no effect. These troublesome things are known collectively as *inlines*: **defmacros**, **defsubsts**, **defconstants**, and any **defuns** that have been proclaimed inline.

In addition, the compiler may optimize trivial **defuns** to be inline even though you have not explicitly proclaimed them to be inline. The compiler will consider this optimization only if you have explicitly included an **optimize** declaration that Safety is 0 and Speed is more important than Space.

Think of these inlines as special directives to the compiler for use in compiling *other* pieces of code. If you patch them, then while you have created new directives, your program is still executing those other pieces of code compiled under the original, incorrect directives. Net effect: your patched code executes just as badly as your unpatched code.

If you need to correct one of these inlines, then your patch must not only include the corrected definition of the inlines itself, but it must also include all other definitions that *use* that inline. This means that the patched source code for these other definitions will be identical to their original source—but their compiles will be new, and that's what counts in the case of inlines.

You can find out where an inline you are patching has been used with the **who-calls** function. Wherever **who-calls** says your patched function has been used as a macro (it uses this general term for any inline expansion), then you must recompile that function too. Furthermore, if you suspect that the compiler might have converted some of your simple functions to inlines, then use **who-calls** to see if they were ever recorded as being used like macros. Unfortunately, there is no current way to search for uses of **defconstants**.

How Installing a Patch Works

5.7 The basic way a user installs a patch in the field is simply to load the patch file—and that is it. However, there are some other considerations.

- Simply loading the patch file is sufficient to modify the environment for testing purposes, but to make the change permanent, you must do something more.
- Since the potential for having many different versions of a piece of software is always a problem, you need some way to formally record which patches have been loaded.

You solve the question of permanence by a manual procedure: cold boot, load the patch file(s), garbage collect, and then do a **disk-save**. Maintaining a formal record of which patches have been applied to which systems requires a small database (called a *patch directory*) and a bookkeeping function (done by the **load-patches** function).

A Little Nomenclature

5.7.1 How many different things does the word *system* mean? When used in a phrase such as *the Lisp system*, it usually means the entire Lisp environment and all software that comes with the machine. This usage is usually apparent from its context.

When used in relation to patching, it literally means any piece of software defined by a **defsystem** and built by **make-system**. Such a system logically corresponds to a program, a product, or an application on a conventional machine. The common connotation is that a system is a more or less stand-alone piece of software that can be optionally added to a *base system*.

Unfortunately, the clean distinction drawn by the two previous paragraphs blurs when you consider that the base system of Lisp software (the stuff that eventually becomes known as the *Lisp system*) is itself defined by a **defsystem** named **system**. So there is such a thing as the **system system**.

Patch Directory

5.7.2 While anyone can create and load a patch file by hand, formal patch maintenance is reserved for software defined by a **defsystem** with a **:patch-directory** declaration and built by **make-system**. The **:patch-directory** declaration makes a system a *patchable* system. The same program files—if compiled and loaded by some other means—execute the same way, of course, but they will not have the extra overhead information needed by the patch maintenance utilities.

First, all the patches for a system are kept in a directory specified in that system's **defsystem**. Therefore, before you do anything with patches, the software usually asks you to identify which system you are talking about so it can find the appropriate patch directory. If the **defsystem** simply says that the system is patchable without specifying a patch directory, the system defaults to a top-level directory on the sys-host with the same name as the **defsystem** name.

Within a patch directory, individual patch files are named according to a three-part standard filename:

1. A prefix string that may be:

- A string specified by the programmer as the second optional parameter of the **:patch-directory** declaration of the **defsystem**
- The string “PATCH” if the system has its own private patch directory (this is the most common case)
- The string “*defsystem-name-PATCH*” if the system shares a patch directory

2. A major version number (usually incremented each time you use **make-system** to recompile that system)
3. A sequential patch number (called a minor version number) reset for each release.

For example, "PATCH-2-12" would presumably be the twelfth patch written for the second build of the system that is the sole user of the patch directory containing this patch file.

The patch number is used for more than just keeping the filenames unique. Two patch files in one directory may or may not be independent of each other (that is, one patch may require that the other patch be loaded first). Therefore, system utilities such as **load-patches** always take the conservative approach and load patches in ascending order.

In addition to the patch files themselves, there is a header file in each patch directory (possibly with several versions) with a name in the form

prefix.LISP

where *prefix* is the prefix string mentioned above. This file exists solely for identifying the name of the system that owns this patch directory and the current major version number.

Finally, there is a log file in each patch directory (almost certainly with several versions) with a name in the form

prefix-r#.LISP

where *prefix* is the same as above and *r#* is the major version number. This file contains the documentation string for each patch file in the directory.

This header file and log file are initially created by **make-system** during the compilation step. These files, or later versions of them, are required by **make-system** during the load step if your **defsystem** indicates to **make-system** that your system is patchable. These requirements are transparent if you exclusively use **make-system** to compile and load the system. If, however, you try to do some of the compile steps by hand, you may trigger error messages from **make-system** if it thinks the patch directory is inconsistent.

load-patches

5.7.3 The **load-patches** function uses the patch directory database to apply patches to the system or systems you specify in its argument list. If you choose to unselectively patch a system, then **load-patches** does the following:

1. Reads a global data structure it maintains to find what is the highest numbered patch that has been applied to that system so far.
2. Reads the system's **defsystem** to find out where the system's patch directory is.
3. Starts loading patches, beginning with the first unapplied patch, and loading in ascending order.
4. Records the number of the last patch loaded back in that global data structure.

If you choose to load patches selectively, then **load-patches** pauses for each patch and displays that documentation string recorded in the patch log file and then asks if you want to load that particular patch or not.

Notice that although **load-patches** records the highest patch loaded, it does not record whether you told it to skip some patches. Therefore, if you choose to load patches selectively, then you assume the responsibility of making certain that dependencies between patches (that is, this patch must be loaded after that patch) are met.

Miscellaneous

5.8 The following are various things that come up during patching but that do not seem to fit in anywhere else.

Explorer System Patch Directories

5.8.1 Patch directories for Explorer system code have logical pathnames of the form "SYS:*system*-PATCH;" where *system* is a system name such as Compiler, Window, Zmacs, and so on. These directories are for the sole use of the officially distributed Explorer system software patches.

You should *not* add your own patches to Explorer system software to these directories even though that seems the logical place to put them. This restriction exists for the most practical of reasons: patches from the Explorer Technical Support Line are typically installed by replacing the old patch directory with a new one. Also, new releases are distributed with new patch directories—usually empty. Therefore, your site-specific patches on these system patch directories will be lost.

Your patches to Explorer system code should be kept on other directories. One site patching convention that allows you the convenience of the patching utilities is to define a trivial **defsystem** that only has a :patch-directory declaration such as:

```
(defsystem explorer-site
  (:name "explorer site pseudo-system")
  (:patch-directory "pathname"))
```

where *pathname* is pointing to anywhere except the Explorer's official patch directories. Also, be sure to create a .SYSTEM file for EXPLORER-SITE as explained in paragraph 5.8.2, How Do You Find **defsystems**.

Now, whenever you come up with something you want to patch in the Explorer system software, you tell Zmacs that you are patching the EXPLORER-SITE system. This is a lie, of course, but all that really interests Zmacs is finding a directory to put the patch in. From then on, whenever you load system patches, for whatever reason, also load the patches for the EXPLORER-SITE system.

How Do You Find **defsystems**

5.8.2 Previously, we saw how the Zmacs Add Patch command asks you for the name of the system you are patching so that it can find the **defsystem**. Then, from that **defsystem**, we saw how the declarations tell where everything else is. Question: How did it find the **defsystem** in the first place?

The global variable **sys:*systems-list*** is a list of the system objects of all systems that have actually been loaded along with the names (as keywords) of all other systems for which the location of their **defsystems** are known (but

not yet loaded). When *software* such as **make-system**, **load-patches**, **Zmacs**, and so on needs to find a system's **defsystem**, it first looks here.

- If it finds the system object it needs, it uses it.
- If it finds the system name it needs, it looks up the **defsystem** pathname on that symbol's **:source-file-name** property and loads it, thereby creating the system object.
- If the system is not on this list, then it falls back to the **SYS:SITE;** directory.

By convention, the **SYS:SITE;** directory holds files that are site specific. If you have written your program properly, then you can move it from site to site by changing nothing more than certain files in this directory at each site. The file **SYS:SITE;-READ-ME-.TEXT** explains several things this directory is used for. For the moment, we are interested in its files of type **SYSTEM**.

If there is no record of, say, system **FOO** on the **sys:*systems-list***, then the software looks for a file named **SYS:SITE;FOO.SYSTEM**. If it finds one, then it loads that file with the expectation that when the load is complete, system **FOO** will be on **sys:*systems-list***. This file should contain one of the following:

- The **defsystem** itself (not recommended)
- A **load** form that loads the file containing the **defsystem** (not much better)
- A **sys:set-system-source-file-name** form that maps a system name to a namestring containing that system's **defsystem** (this is preferred)

It is this **sys:set-system-source-file-name** form that puts those system names on **sys:*systems-list*** and attaches the **defsystem** pathname to that symbol's **:source-file-name** property.

The Strange Case of **defvar**

5.8.3 defvar and defparameter do the same thing—almost. The first time either one is loaded, each

1. Defines its respective symbols as global special variables
2. Evaluates its initial forms (if any)
3. Sets its symbols to those initial values

The second time they are loaded, **defparameter** does exactly what it did the first time—**defvar** does nothing. In particular, **defvar** does *not* reevaluate its initial form, and it does *not* set its symbol to a new value.

The key here is that on the first load, the symbols defined by both **defvar** and **defparameter** were unbound. On the second and all later loads, however, the symbols had already been bound. Therefore, **defparameter** always rebinds its symbol regardless of that symbol's bound state, while **defvar** binds only an unbound symbol. Why?

While debugging, it is not unusual to load a file or to compile a buffer (which quietly loads it too) many times. Is it reasonable to keep re-evaluating those initial forms over and over? Usually yes, sometimes no. For example,

- An initial form may instantiate a complex data structure that takes a long time to do and makes garbage out of the previous identical instantiation, thereby wasting time and memory.
- An initial form may have a significant side effect such as starting a process, establishing a network connection, triggering a file system transaction, and so on, each of which should be done only once.

Therefore, you could say that **defvar** and **defparameter** do the same thing except that **defvar** does the same thing only once. This subtle distinction is lost on a lot of people.

Even though a **defvar**'s symbol does not get reset by default on each load, there are times when you do want it reset regardless. So there are ways to force it to be reset (or not reset) on loading. For example,

- If you *mark* a region and then compile or evaluate it (that is, CTRL-SHIFT-C or CTRL-SHIFT-E) then the **defvar** value is *not* reset. (The assumption is that this **defvar** just happened to fall inside a region being compiled.)
- If you just point at the **defvar** with the cursor *without* marking it as a region and compile or evaluate it, then it *is* reset. (The assumption is that you were deliberately trying to change it.)

At this point, the options (and confusion) compound. If you mark a region and evaluate it with META-CTRL-SHIFT-E rather than just CTRL-SHIFT-E, then it will be reset. (The assumption is—obscure.)

Finally, the one that most interests us: if the file being loaded has a PATCH-FILE file attribute of non-nil (that is, a patch file), then the **defvars** are always reset.

Therefore, it is safe and reasonable to put **defvars** into patch files if you need to redefine their initial values.

6

PATHNAMES

Introduction

6.1 The original Lisp machine software developed at MIT was built on assumptions that have proven more farsighted than those implemented even today:

- The Lisp machine can be used as a backend to various mainframes.
- The Lisp machine is normally on a local area network with an unknown mixture of machines and operating systems.

By implication, the Lisp machine file system would have to routinely communicate with *foreign hosts*, as they were called, and the communication would have to be in terms the foreign host's operating system could understand.

The most visible problem of talking to a multitude of disparate operating systems is that each has its own idiosyncratic file-naming conventions. For example,

```
DS01.DIR.SUBDIR.OBJ.NAME  
/DIR/SUBDIR/NAME.O  
DIR.SUBDIR;NAME.OBJ#1  
SYS$DISK:[DIR.SUBDIR]NAME.OBJ;1  
>DIR>SUBDIR>NAME.OBJECT.1
```

are all different ways of identifying the same file on different operating systems. The problem is how to handle such diversity without creating masses of ad hoc code for each case.

Internal Versus External Names

6.2 The first step is to separate the external form of these file specifications that you see from the internal format that the Explorer software uses:

- A *pathname object* is the internal form that the software sees. On the Explorer, it is an instance of a flavor built on the `fs:pathname` flavor.
- A *namestring* is the external form that you see. It is a string with the internal syntax of the host holding that file.

Time out for confusion: Lisp programmers often use the terms *name*, *name-string*, *path*, *pathname*, *pathname object*, *file*, *filename*, *file spec*, *stream*, and *file stream* interchangeably. Part of this laxity is brought on by the fact that most Lisp functions needing to identify a file accept either an external namestring or an internal pathname object. It is easy for a function to coerce either one into the other.

To limit confusion, I always use the following terminology in this section:

- *Namestring* for the external user representation
- *Pathname object* for the internal software representation
- Just *pathname* for the general case of anything convertible to a pathname object

Notice that just about anything that accepts a namestring as an argument also accepts a symbol. In such a case, the symbol's print name (which is a string, see **symbol-name**) is used as the namestring.

Pathname Objects

6.3 Converting the namestrings of all foreign operating systems into one, standard internal pathname object sounds like a good idea, but how do you do it?

If you were to inspect namestrings from a variety of operating systems, then you would notice that many of them can be viewed as a hierarchy of components:

Device
Directory
Filename
File type
Version

NOTE: Those of you who already know something about pathnames will wonder why the host component is missing from the list. Right now, we are talking about namestrings as different operating systems see them. Later we will introduce the notion of a host to identify which namestring belongs to whom.

Device Component

6.3.1 The device component identifies the storage medium holding the file on the host. Depending upon the host, the device component could represent the name of any of the following:

- A physical disk drive (independent of the pack mounted on it)
- A physical disk pack (independent of the drive it is mounted on)
- A physical partition on a disk
- A logical partitioning of any online storage medium

On Explorers, the device component corresponds to a File Band—which requires a little explanation.

An Explorer *partition* is a sequence of one or more consecutive blocks on an Explorer disk. A disk is typically divided into several such physical partitions, which are used for various things. An Explorer *band* is a logical grouping of

one or more physical partitions. However, not all bands are created alike (to say the least). A few examples are as follows:

- There can be multiple Load Bands and Microload Bands per disk, but each must be exactly one partition (hence the common notion that *partition* and *band* are synonymous).
- There can be only one Page Band for the entire system, and it can be composed of any number of partitions on any number of disks.
- There can be multiple File Bands per system, and each File Band can be composed of any number of partitions on any number of disks.

The moral of this story is that each type of band has its own characteristics. Although an Explorer can have more than one File Band on its disk(s), only one of these may be booted at a time. Therefore, pathname processing assumes that it has no choice since there is only one File Band on an Explorer, so it ignores the device component.

Most hosts that allow multiple devices have the notion of a default device that they use if the device component is not specified. Similarly, if the host has only one device, then there is no need for this component. The Explorer device component in a pathname object is always coerced to `nil`. For other hosts that have no need for a device component, that component is coerced to `:unspecified`.

Directory Component

6.3.2 The notion of a directory is common enough. The directory component of a namestring actually includes the top level directory as well as any subdirectories under it. Different operating systems use different delimiters to separate directories and subdirectories in their namestrings.

The directory component is the one component of a pathname object that is normally *structured* (that is, it is composed of lower level subdirectories). Therefore, while the other components are normally represented as simple strings, the directory component is represented as a list of strings, one for each *subcomponent*. If there is only a single level of directory, then it is represented as a list of one string.

For example, the namestring fragment “FOO.BAR;BAZ” would be represented in a pathname object as

```
Directory Component => ("FOO" "BAR")
```

```
Name Component => "BAZ"
```

while “MUMBLE;BAZ” would be represented as

```
Directory Component => ("MUMBLE")
```

```
Name Component => "BAZ"
```

See paragraph 6.13, Absolute Versus Relative Directories, for other information that can be recorded in a pathname object’s directory component.

The highest level directory on a device is called the *root*. In functions like `make-pathname`, an Explorer extension to Common Lisp allows the root to be designated with the keyword `:root` since it does not have a name itself. If

you specify the directory component as being :wild, then—depending on the context—the system may assume that you really meant :root and use that.

Filename Component

6.3.3 The name component is the one traditionally thought of as *the* filename within a directory.

One minor point of confusion: many operating systems have generalized their file systems such that a directory is just one of several types of files.

- If a directory is being used as a directory (for example, to access things below it), then its name appears in the directory component of the name-string.
- If it is being used as a file (for example, to read its contents to, say, create a directory listing), then its name appears as the name component.

On the Explorer system, for instance, “HOST:DIR.SUBDIR;*.*#*” refers to the contents of the directory SUBDIR, which is itself a member of the directory DIR. Therefore, the form

```
(delete-file "HOST:DIR.SUBDIR;*.*#*")
```

would delete all files under the directory SUBDIR. A listing of the directory parent DIR would still show an entry for SUBDIR, but it would have zero length. In contrast, the namestring “HOST:DIR;SUBDIR.DIRECTORY#1” refers to the file under the directory DIR that records the contents of the directory SUBDIR. Therefore, the form

```
(delete-file "HOST:DIR;SUBDIR.DIRECTORY#1")
```

would delete the SUBDIR directory itself (assuming it was already empty). Another listing of the directory DIR would no longer show an entry for SUBDIR.

File Type Component

6.3.4 The file type component does not exist on all operating systems, and the ones that do support it sometimes view the type component as just an extension of the name component (it's even called the *file extension* in some cases).

However, even though the basic operating system may make no assumptions about the relationship of name and type components, many higher level functions use it for the convenience of the users. For example, unless you tell it otherwise, the Explorer's Lisp compiler assumes that its input source files are of type :lisp and that its output files are of type :xld (the reason for using keywords rather than strings is explained in paragraph 6.7, Canonical File Types). For example,

```
(compile-file "HOST:DIR;FOO")
```

actually compiles file “HOST:DIR;FOO.LISP” into file “HOST:DIR;FOO.XLD”. So, if you go against the type component naming conventions too much, you might find that a lot of otherwise convenient software is working against you. That is, you will find that you can no longer use all of those convenient defaults built into the system. Instead, you must specify everything completely.

Version Component

6.3.5 The version component is also not supported on some operating systems (for example, the UNIX® OS). The version is simply an incrementing number for each successive update of the file. Thus, Common Lisp functions such as `make-pathname` allow you to use the keyword :*newest* to represent the highest currently available version. The Explorer also allows you to use :*oldest* for the lowest available number.

Again, the average operating system makes few assumptions about the relationship of the version components to the name and type components—but almost every other piece of file handling software on the Explorer *does*. In particular, everybody defaults to working with the *newest* version (that is, the highest version number) of a file. If you have manipulated the file system so that the highest version number is not the newest, you lose.

A Re-Illustration

6.3.6 Now that we know what each component is supposed to be used for, let's look at those five diverse namestrings we saw on the first page again:

Device	Directory	Name	Type	Version
DS01.	DIR.SUBDIR.OBJ.	NAME		
	/DIR/SUBDIR/	NAME	.O	
	DIR.SUBDIR;	NAME	.OBJ	#1
SYS\$DISK:	[DIR.SUBDIR]	NAME	.OBJ	;1
	>DIR>SUBDIR>	NAME	.OBJECT	.1

As you can see, all namestrings don't have all the components.

Namestrings

6.4 The internal syntax of namestrings is simple enough because they are formally defined by each operating system. But how do you get such a wide variety of syntaxes parsed into one common pathname object?

Time out for a few assumptions: computers (called *hosts* in Explorer nomenclature) on a local area network are commonly identified by nicknames. Each host name represents a specific hardware type (for example, Explorer, VAX™, and so on) and a specific operating system (for example, Common Lisp, VMS™, UNIX®, and so on). Therefore, for our purposes, knowing the host name automatically tells us the operating system type and, by implication, the namestring syntax. Now, back to our story.

The convention developed for the Lisp machine is that a namestring must be parsed with respect to a specific host. That is, it has to know which operating system the namestring is for before it can tell how to parse the namestring. Part of the information recorded about each type of host known to the system is a namestring parsing function. Next question: where do you find which host you are supposed to parse the namestring with respect to?

UNIX is a registered trademark of AT&T.

VAX and VMS are trademarks of Digital Equipment Corporation.

There are three basic sources for determining a namestring's host. First, if the namestring is in the following form:

host:namestring

then the portion of the string before the *first* colon is considered to be the host name. There may be additional colons in the namestring itself, but the first one delimits the host name. This namestring format is said to have a *manifest* host name (that is, the host name is actually present in the namestring rather than just being available from some default). Furthermore, Lisp programmers do not bother to distinguish a namestring with a manifest host from a namestring without one. Therefore, they would have referred to the above format as just a namestring.

Since the manifest host name is used only by the Explorer parsing functions, its case is unimportant. That is, even if the rest of the host's namestring is case sensitive, its manifest host prefix is still case insensitive.

Second, the general-purpose **parse-pathname** function allows you to optionally specify a with-respect-to host. If you use this feature, you cannot use it to actually override a manifest host. If both a manifest host and a with-respect-to option are present, the two hosts must match or else you will get an error.

Third, if there is no manifest host in the namestring and no with-respect-to host option in the function call, the system looks at the available default pathnames and uses the first likely host it finds to parse the original namestring. Sometimes the default pathname is specified as an argument to the parsing function. Otherwise, the information in the ***default-pathname-defaults*** variable is used.

Since the system must always have a host in order to understand a namestring, it treats the host as a sixth component that actually comes first in the hierarchy:

Host
Device
Directory
Filename
File Type
Version

In addition, even after an external namestring has been parsed into an internal pathname object, its original host is remembered along with the other components. Among other things, recording the host allows special purpose external namestrings (with proper host-specific syntax) to be reconstructed from the standard internal representation on demand. These reconstructed namestrings are collectively called "string-fors" because they are created by methods named **:string-for-host**, **:string-for-dired**, **:string-for-printing**, and so on.

Parsing Functions

6.5 The following is a brief illustration of the functions used to convert among namestrings, pathname components, and pathname objects.

(pathname “namestring”)	=> pathname-object
(namestring pathname-object)	=> “namestring”
(make-pathname :host :device :directory :name :type :version)	=> pathname-object
:host “host”	
:device “device”	
:directory (“dir” “subdir”)	
:name “name”	
:type “type”	
:version number	
(pathname-host pathname-object)	=> host-object
(pathname-device pathname-object)	=> “device”
(pathname-directory pathname-object)	=> (“dir” “subdir”)
(pathname-name pathname-object)	=> “name”
(pathname-type pathname-object)	=> “type”
(pathname-version pathname-object)	=> number

Note the symmetry. **pathname** and **namestring** are complementary functions converting between namestrings and pathname objects while **make-pathname** and **pathname-host** and so on are complementary functions converting between individual pathname components and pathname objects. Some miscellaneous points:

- You can experiment with namestring parsing with the form (describe (pathname “namestring”)) where **pathname** does the parsing and **describe** lists each component of the pathname object. Notice that the string-fors are not filled in until someone actually asks to see one.
- Just because you have a pathname object doesn’t mean you can do anything with it. A pathname object *always* has a valid host component or it could not be a pathname object, but any and all other components may be **nil**.
- The **parse-namestring** function is similar to **pathname** except that it allows you more options and control over defaults (for example, **pathname** calls **parse-namestring** with fixed options).
- Components omitted from the end of the namestring usually default to **nil** in the pathname object.

Some things have been deliberately left out of these examples for simplicity. Look up the individual function descriptions before you try to use them.

Interned Pathnames

6.5.1 If you *intern* the external name of something (we’re not just talking pathnames now), then the system looks it up in a special table.

- If the external name is *not* already represented in the table, then a new internal data structure appropriate to whatever you are interning is created, stored in the table, and returned.
- If an internal version of the external name *is* already in the table, then that internal object is returned.

That is, the first time you intern something, you get a new object. Every time you intern the same external name from then on, you get the same object.

On the Explorer, symbols and pathname objects are the only objects that are interned. Symbols are interned by the `intern` function in tables called packages. Pathname objects are interned by the `parse-pathname` function in the `fs:*pathname-hash-table*` hash table. Therefore, if

```
(pathname "FOO.LISP") => <pathname-object>
```

then on the Explorer

```
(setf a (pathname "FOO.LISP"))
(setf b (pathname "FOO.LISP"))
(eq a b) => true
```

For the purpose of portability, Common Lisp recommends that `equal` be the preferred predicate for comparing pathnames. Since objects that are `eq` (such as identical Explorer pathname objects) are also `equal`, the Explorer's pathname interning does not invalidate the standard usage.

A namestring cannot be used until it has been converted into a pathname object by `parse-namestring` or one of its variants. Since the conversion causes the pathname object to be interned, then `fs:*pathname-hash-table*` becomes something like a pathname log for the system. That is, it should always have one copy of every pathname that has ever been used in the system.

Why Intern?

6.5.2 Why does the system go to all the trouble to intern pathnames? Well, if it didn't, then you could never reliably compare two pathname objects to see if they represent the same pathname.

The problem is that pathname objects are instances of flavors, and a property of flavor instances is that they can be compared *only* as `eq`. That is, two instances are considered to be the same only if they are actually the same data structure in memory. Simply pointing to two identical data structures is *not* sufficient. For example,

```
(setf a (make-instance 'my-flavor))
(setf a1 a)
(setf b (make-instance 'my-flavor))
(eq a a1) => true
(eq a b) => false
```

Even though the values of `a` and `b` are bit-for-bit identical, they are not `eq` because `a` and `b` themselves are pointers to two different addresses. You can't even fool the system by using the `eql`, `equal`, or `equalp` predicates. As soon as they recognize that they are comparing flavor instances, they all revert to `eq`.

Now consider the needs of a programmer manipulating Explorer namestrings. The two forms

```
(setf x (pathname "EX22:ABC;XYZ.LISP#4"))
(setf y (pathname "EX22:ABC;XYZ.LISP#4"))
```

both represent fully qualified namestrings. They are as specific as you can get, and they match component for component. By all standards, they should be considered as the same pathname; however, they won't compare as `eq` unless `x` and `y` actually point to the same flavor instance.

Interning Our Way Around the Problem

6.5.3 Interning solves our comparison problem. On the first call to `pathname` above, the pathname object representing "EX22:ABC;XYZ.LISP#4" is interned into `fs:*pathname-hash-table*` as described above. The newly interned instance is returned by `pathname` and stored in `x`. Next, the second call to `pathname` finds the first pathname object in the hash table and returns it again (without interning anything new). Therefore, the same instance is also stored in `y` so that `x` and `y` are `eq` (and all is well once again).

If comparing pathname instances is such a problem, how did the system ever manage to find a matching pathname object in the hash table? Fortunately, an Explorer extension to the Common Lisp `make-hash-table` function allows you to specify an arbitrary predicate for the hash table lookup comparisons. The Explorer uses a special predicate function that individually compares all six components for `equal` while either observing or ignoring case as the host's namestring syntax requires. This fairly elaborate predicate successfully implements a programmer's notion of which namestrings should be considered the same and which should be considered different.

By interning the namestring, the elaborate, time consuming comparisons are done only when a namestring is parsed. All later pathname object comparisons are done with the fastest available predicate: `eq`.

Defaulting Versus Merging

6.6 Defaulting refers to the general practice of the Explorer software prompting you for input while simultaneously offering you the input it thinks you want. In most cases outside the file system, you have only an all-or-nothing choice: you either accept the entire default as-is, or you supply a replacement for the entire default.

Inside the file system, however, the thing most frequently being defaulted is pathnames. Since pathnames have several components, the file system uses *merging* to create namestrings that are a mixture of some default components merged with some user-specified components. Since virtually all defaulting in the file system requires merging, Lisp programmers seldom bother to distinguish between the two and tend to use them interchangeably.

Defaulting

6.6.1 Defaulting refers to the general practice of the Explorer software prompting you for input while simultaneously offering you the input it thinks you want. That is, you specify part of a namestring, and the system fills in the rest.

A Good Example

6.6.1.1 One of the best examples of the convenience of defaulting is the Zmacs editor. If you are editing one file and then enter the Find File command, then Zmacs prompts you for the namestring of the file you want while displaying the current buffer's namestring as a default. Although newcomers might be tempted to type in the whole thing just to be sure, they really need to type in only those components of the new namestring that are different from the default. Given a default Explorer namestring of, say,

"host-a: dir-b; file-c.type-d#>"

the input on the left in the table shown below would be merged with the default shown here to create the namestring shown on the right in the table below:

Type-In	Merged Namestring
FILE-Y	=> host-a: dir-b; <u>FILE-Y.type-d#></u>
.TYPE-X	=> host-a: dir-b; file-c. <u>TYPE-X#></u>
HOST-W:	=> <u>HOST-W:</u> dir-b; file-c.type-d#>
#4	=> host-a: dir-b; file-c.type-d#4
HOST-W: FILE-Y#4	=> <u>HOSTA-W:</u> dir-b; <u>FILE-Y.type-d#4</u>

The capitalization in this example is just a way of emphasizing the changed components—Explorer namestrings are not case-sensitive (although the namestrings for some other hosts are).

This example also illustrates the advantage of having different delimiters for each component: if you input a namestring fragment, then the parsing function, `parse-namestring`, can figure out which components you specified and which you left out. For example, the host and directory components are identified by trailing punctuation whereas type and version have leading punctuation.

A namestring fragment containing *no* delimiters is treated as a name component. Therefore, if you want Zmacs to find another file in the same directory your current buffer came from (which is a common case), then you only have to type in the new filename. Defaulting takes care of filling in the host, directory, type, and version for you.

A Bad Example

6.6.1.2 Zmacs also offers a counter example of how a defaulting scheme that works 99% of the time can trip you up. To understand what is happening, remember that

- *Reading* the newest version of a file reads the highest existing version number.
- *Writing* the newest version creates a new version number one higher than the highest existing one.
- Reading or writing a specific version reads or writes that exact version.

Let's say the most recent version of the file `FOO.LISP` is version 4. If you ask Zmacs to edit "`FOO.LISP`", it does the right thing, so to speak, and defaults your input to "`FOO.LISP#>`", that is, the newest version, which, in this case, happens to be 4. In particular, the namestring "`FOO.LISP#>`" is now associated with the Zmacs buffer. When you save that buffer, it is saved to "`FOO.LISP#>`", which has the effect of creating a newer version, version 5, and all is well with the world.

But what if you had explicitly asked to edit "`FOO.LISP;#4`" (which is supposed to be just another name for the same file)? Now the namestring "`FOO.LISP;#4`" is associated with the buffer. When you ask to save it this time, Zmacs tries to save it literally as version 4—but a version 4 already exists and the file system complains. This is yet another case of the stupid computer doing *exactly* what you told it to.

An Aside

6.6.1.3 Although it's not properly a part of this section, you might be interested in how to recover from the above problem. Fortunately, your original state is left unchanged: the file system has not been modified, the buffer has not been modified, and it is still marked as needing saving. Therefore, you are free to enter new Zmacs commands.

You are either going to have to get rid of the conflicting version on disk by deleting it or renaming it, or you are going to have to store your buffer to a version that doesn't conflict. The simplest thing is to use Write File (CTRL-X CTRL-W) because it will prompt you for a new namestring while offering the buffer's associated namestring as a default.

Since the version number is the only problem, enter *just* a new version number into the minibuffer and let everything else default. In particular, entering, say, "#42" would cause it to be written out as version 42 while "#>" would write it out as the next higher version—whatever that may be.

Merging

6.6.2 You accomplish pathname defaulting by merging two pathnames (that is, symbols, namestrings, or pathname objects) with the `merge-pathnames` function.

`merge-pathnames` accepts two arguments: a target pathname to be defaulted and a source pathname from which to take the defaults. It converts both arguments into pathname objects (if they weren't already) and replaces `nil` components in the target pathname object with the corresponding components from the source pathname object. Components in the target pathname that are anything other than `nil` are left unchanged.

Therefore, merging means replacing `nil` components in one pathname object with non-`nil` components from another pathname object.

If you sneak a peek at the definition of `merge-pathnames`, then you'll notice that the default pathname source argument is actually optional, which means you can call `merge-pathnames` with only one pathname. But what does it mean to merge *one* pathname? This is beginning to sound like the Lisp version of the Zen *koan*, "What is the sound of one hand clapping?".

Defaulting Defaults

6.6.3 As it turns out, Lisp is quite resourceful. If you don't specify a default pathname argument, it goes and finds one for itself. Lisp uses a simple heuristic that allows it to do the right thing (from your point of view) by choosing the last pathname used for the host of the target pathname you are merging. If that host has not been accessed before, it uses some other previously accessed pathname.

These candidate default pathnames are remembered in the variable named `*default-pathname-defaults*`. That's right. These are the defaults for the optional default pathname arguments that you defaulted by not specifying them in the function calls.

Taking that again more slowly: a number of pathname functions require a default pathname to use as a source of missing components. These functions have an optional argument to let you specify that default pathname. But if you don't supply one yourself on the function call, it still needs to find a default somewhere—so it looks in `*default-pathname-defaults*`.

The Common Lisp Approach

6.6.3.1 In Common Lisp, ***default-pathname-defaults*** is just a single pathname. The basic assumption is that you have a choice either of binding this Common Lisp variable to a default value once or of specifying the default argument to each function that needs one. The Common Lisp usage of this variable is not supported on the Explorer—yet.

The Explorer's Approach

6.6.3.2 The Explorer uses Zetalisp's more general version of this variable, which is a small database that allows more intelligent guesses about which defaults would do you the most good. In the Explorer, this variable is an association list of specific file server hosts and the most recent pathname object used for each host. Therefore, if you are merging a pathname of, say, host EXPLR5, then the merging function searches this alist for the last pathname object you used for EXPLR5.

If no match is found in the alist, there is a catch-all entry (its host is listed as nil). This entry is roughly equivalent to Common Lisp's view of what its ***default-pathname-defaults*** should be. You might say this catch-all entry is the default for the default pathname defaults—got that?

The Explorer's use of that alist is controlled by the variable **fs:*defaults-are-per-host***. If this variable is true, then merging operates as discussed above. If it is false, then merging uses only the catch-all entry in the alist.

Problems With the UNIX OS

6.6.4 The UNIX operating system does not support version numbers on files. Therefore, pathname objects for UNIX hosts always have a version component of :unspecified so UNIX version numbers present no particular problem. As long as the pathname object is being manipulated internally, the :unspecified marker prevents internal functions from making use of the version number. Later, when a string-for namestring is created for use by UNIX itself, the :unspecified marker is dropped.

UNIX also does not support type components, but that does not stop UNIX programmers from developing programming conventions that use the notion of file types. UNIX itself recognizes no components below the level of the name component, but that name component may contain periods (which count as just another alphabetic character to UNIX).

Nevertheless, if the average UNIX programmer should see two UNIX files, FOO.C and FOO.O, then he would assume FOO.C is a C source file and that FOO.O is probably the compiled version of FOO.C. Similarly, if a UNIX C compiler was told to compile the file FOO, it would probably look for FOO.C to read for input and produce FOO.O for output. To UNIX itself, on the other hand, FOO.C and FOO.O are two unrelated files with unknown contents.

Implied UNIX File Types

6.6.4.1 When the Explorer's **parse-namestring** function is given a UNIX namestring with a name component of, say, FOO.BAR.BAZ, it uses the last period as the delimiter of the name component, FOO.BAR (note the embedded period) and the *implied* type component of BAZ. Notice that the delimiting period itself is not explicitly recorded in either component.

This interpretation of FOO.BAR.BAZ as a name/type combination may not match what the UNIX programmer had in mind. He may have considered it to be just one long filename with two periods in it. Fortunately, this potential misinterpretation is just a detail of the internal representation in the UNIX pathname object.

If the Explorer software is asked to reconstruct a string-for namestring from the UNIX pathname object, then the FOO.BAR name component and the BAZ type component are recombined into FOO.BAR.BAZ complete with the original delimiting period. Since `parse-namestring` (and its simplified variant, `pathname`) is symmetrical with `namestring`, no information is lost despite the occasional internal misrepresentation.

Unimplied UNIX File Types

6.6.4.2 Unfortunately, parsing problems *do* arise if the UNIX programmer did not imply a file type in his namestring. For example, UNIX programmers would commonly view filenames such as FOO, FOO., FOO.., .FOO, ..FOO, and so on as just filenames with no type.

`parse-namestring` understands this convention and records the undivided filename complete with all leading or trailing periods as the name component of the UNIX pathname object. But what should it record as the type component? As it turns out, there can be no *single* correct answer.

If a Lisp programmer inputs a file named FOO, he expects the system to provide appropriate default canonical types of, say, :lisp or :xld depending on how the file is being used. Such defaulting is a known feature of the system and is widely used—in Lisp.

If, on the other hand, a UNIX programmer inputs FOO, he expects it to stay FOO no matter how it is used. Outside of a few specific cases where certain programs, such as compilers, have their own naming conventions, UNIX programmers do *not* expect the system to manipulate filenames behind their backs.

A Compromise

6.6.4.3 What's a parser to do? If it parses FOO as a type of `nil`, then the type component in the pathname object is eligible for defaulting by `merge-pathnames` later. This is the right thing to do for Lisp programmers, but it is a bug to UNIX programmers. If it parses FOO as a type of `:unspecific`, then the type component in the pathname object cannot be defaulted—just the way UNIX programmers like it, but now it's a Lisp bug.

The Explorer's solution to this impasse is to let the users fight it out among themselves. The global variable `fs:*merge-unix-types*` determines what `parse-namestring` supplies as the type component for UNIX namestrings that do not contain an implied type.

Value of <code>fs:*merge- unix-types*</code>	Supplied Type Component	Description
true	<code>nil</code>	The real file type can be merged in later.
false	<code>:unspecific</code>	The file type will remain unchanged regardless of later attempts at merging.

Therefore, set `fs:*merge-unix-types*` as you like it.

The following are some examples of how UNIX namestrings would be parsed using these conventions. The phrase *as per flag* means that a type component of `nil` or `:unspecific` is substituted according to the `fs:*merge-unix-types*` flag as described above.

Namestring Fragment	Name Component	Type Component
"FOO"	FOO	<as per flag>
".FOO"	.FOO	<as per flag>
"..FOO"	..FOO	<as per flag>
"FOO."	FOO.	<as per flag>
"FOO.."	FOO..	<as per flag>
"FOO.BAR"	FOO	"BAR"
"FOO.BAR.BAZ"	FOO.BAR	"BAZ"

The following examples show how the default operation shown above can be overridden for specific cases by using special Explorer characters to force specific interpretation of individual components regardless of the value of `fs:*merge-unix-types*`. The Explorer character of the double left/right arrow (\leftrightarrow) is produced by pressing (SYMBOL-1). The Explorer character of the union operator (\cup) is produced by pressing (SYMBOL-r).

Namestring Fragment	Name Component	Type Component
"FOO. \leftrightarrow "	FOO	nil
"FOO.. \leftrightarrow "	FOO.	nil
"FOO. \cup "	FOO	:unspecific
"FOO.. \cup "	FOO.	:unspecific
\leftrightarrow .BAR"	nil	BAR
" \cup .BAR"	:unspecific	BAR

Notice that when the \leftrightarrow or \cup notation is used, it must stand for the entire component. Therefore, a namestring fragment such as " \leftrightarrow ..BAR" cannot be parsed. The ".BAR" suffix would become the type component of "BAR", leaving \leftrightarrow . (note the trailing period) to be interpreted as a name component of—what?—the symbol nil and the character #\.?

Canonical File Types

6.7 The purpose of many features of the Explorer namestring processing is to allow the free transfer and access of files between widely differing operating systems. There are certain obstacles, however, that even the most clever of software cannot overcome. For example, if the filename on one machine is longer than another machine will accept, then it is pretty much up to the programmer to rename the file to some mutually acceptable name before transferring it.

The Role of File Types

6.7.1 But when it comes to file types, the software can help out. Although, strictly speaking, you can specify anything you want for a namestring's type component, in practice, you usually stick to a handful of well-known types. You choose the host, device, directory, and name components to uniquely identify components within their respective levels of the pathname hierarchy. In contrast, you choose the type component mainly to *classify* a file. Such classification allows various functions to respond intelligently to a file, depending upon what kind of file (that is, what type) it is.

For example, the **readfile** function loads a Lisp *source* file into memory whereas its companion **fasload** function loads an *object* file. Lisp programmers seldom use these functions (and most newcomers have never heard of them) because of the intelligent use the general purpose **load** function makes of file types. When you give **load** the pathname of a file to load, it examines the file type to determine whether to call **readfile** or **fasload**. That is, **load** calls the right function on your behalf, depending upon the class of the file you want loaded.

Actually, **load** has a few other surprises, too. If you specify a pathname without a type, then **load** tries to do the right thing by first probing to see if there is an :**xld** version of that file. If so, it gets loaded. Otherwise, **load** falls back to loading the :**lisp** version. But be warned: **load** does not check version numbers. If the :**lisp** files are newer than the :**xld**, **load** still loads the :**xld**.

Portable File Types

6.7.2 Unfortunately, different operating systems have different ideas about what to name the same classes of files. Text files are of type TEXT on some systems and TXT on others. Lisp files may be l, LSP, or LISP, depending upon where your files are stored. When the **load** function mentioned above first probes for the Lisp object file and then for the Lisp source file, how does it figure out what exact type name to look for?

The idea behind *canonical types* is that you identify the class of a file to the system with a keyword, and the system chooses an appropriate type string to use in constructing a host-specific namestring. For example, the canonical file type :**lisp** would be converted into a type component of LISP on an Explorer, LSP on DEC's TOPS-20™OS, and l (note the lowercase) on the UNIX OS.

The function **fs:define-canonical-type** defines the mapping of a canonical type keyword to an OS-specific string, and the **make-pathname** function translates the keywords into the appropriate strings for you. The global variable **fs::canonical-types** is a property list of canonical type keywords paired with their mappings for each operating system. About three dozen canonical types are now defined, but only a handful are commonly used.

Unfortunately, canonical types can be used only with **make-pathname** and not with namestrings. The colon that introduces the canonical type keyword would be mistaken for a syntax delimiter in a namestring.

Rule of thumb: any pathname you must generate in your program (that is, not something the user gave you) should have a canonical file type.

Two-Way Mappings

6.7.3 Actually, this canonical mapping is two-way. If an Explorer namestring with a type component of LISP has to be converted into, say, a TOPS-20 namestring (which accepts only three character type strings), then the pathname software—if asked—can deduce that LISP is an Explorer-specific instance of the canonical type :**lisp** and that the canonical type :**lisp** translates to LSP on TOPS-20.

The `fs:define-canonical-type` function can establish a one-to-many mapping if need be. For example, the canonical type `:lisp` has a preferred mapping for UNIX of `l`. This mapping is preferred in the sense that UNIX programmers traditionally use terse, lowercase mnemonics to name things. However, UNIX itself will accept longer type names such as “`lisp`”. Therefore, the canonical type `:lisp` is translated to `l` on UNIX, but both the UNIX implied types `l` and `lisp` are recognized as instances of the one canonical type `:lisp`.

This one-to-many mapping is both a feature and a problem. While the canonical type software recognizes that the files `FOO.LSP` and `BAR.LISP` both represent Lisp source files, it also sees `FOO.LSP` and `FOO.LISP` as being the same file—which probably isn’t what the programmer intended.

Interchange Format

6.8 Namestring *interchange format* is an effort to maintain the aesthetics of namestring capitalization as namestrings are first parsed into pathname objects and then later reformed from pathname objects to be sent to foreign hosts. This is one of those topics you need to know just enough about to recognize it when you see it and to know how to look up specifics when you need them.

Even on operating systems that allow both cases in pathname components, programmers commonly develop preferred capitalization conventions. For example, UNIX namestrings are traditionally lowercase while VAX namestrings are traditionally uppercase. If a namestring or a namestring component is in interchange format, then its case (all upper, all lower, or mixed) tells something about how it should be recorded in a pathname object for a particular host:

- If the interchange component is all uppercase, then when it is stored into a pathname object, it should be given the preferred capitalization of that pathname object’s host.
- If the component is all lowercase, it should be stored in just the *opposite* of the host’s preferred capitalization.
- If the component is mixed case, then no assumptions are made about preferred capitalization as it is moved around.

A namestring that a user types in is considered to be in *raw* form. The rules of interchange format attempt to preserve any *significant* information that the capitalization of the original raw input implied no matter how individual components may be shuffled about.

Consider the following example of a UNIX namestring (where case is significant and lowercase is preferred) versus an Explorer namestring (where case is ignored and uppercase is preferred). The functions used in this example are the following:

- `pathname`, which parses a namestring and returns a pathname object which, as you’ll remember, is an instance of a flavor
- `symeval-in-instance`, which lets us directly see the value of an instance variable *without* going through any translation the accessor methods might sneak in

- **pathname-directory**, which returns the interchange form of the directory component of a pathname object
- **fs:pathname-raw-directory**, which returns the raw form of the directory component of a pathname object

pathname and **pathname-directory** are standard Common Lisp functions. The other two are Explorer extensions. We need **symeval-in-instance** because the **:directory** method of the **fs:pathname** flavor doesn't just return the contents of the **fs:directory** instance variable as it would for most flavors. Instead it returns the interchange format so we need a more primitive way to look at the value of the instance variable itself.

Now, if you assume that UHOST represents some UNIX host at your site and EHOST represents some Explorer host, then

```
(setf u-path (pathname "uhost:/aaa/bbb"))
(symeval-in-instance u-path 'fs:directory) => ("aaa")
(pathname-directory u-path)                  => ("AAA")
(fs:pathname-raw-directory u-path)          => ("aaa")
```

which shows that

- The raw input was in preferred capitalization.
- The components were stored in the object as raw components.
- Extracting the interchange format gives uppercase, indicating that the component should have preferred capitalization.

Now, looking at an uppercase UNIX pathname we see

```
(setf u-path (pathname "uhost:/AAA/BBB"))
(symeval-in-instance u-path 'fs:directory) => ("AAA")
(pathname-directory u-path)                  => ("aaa")
(fs:pathname-raw-directory u-path)          => ("AAA")
```

which shows that

- The raw input was the *opposite* of preferred capitalization.
- The components were still stored in the object as raw components.
- Extracting the interchange format gives lowercase, indicating that the component should have the opposite of preferred capitalization.

Now let's look at an Explorer pathname

```
(setf e-path (pathname "ehost:xxx;yyy"))
(symeval-in-instance e-path 'fs:directory) => ("XXX")
(pathname-directory e-path)                  => ("XXX")
(fs:pathname-raw-directory e-path)          => ("XXX")
```

Actually, if you try this for yourself, you'll find that it doesn't matter what capitalization you type in for Explorer pathnames, for everything comes out in uppercase. Since case isn't significant in Explorer pathnames, **pathname** sort of assumes that everything is equally preferred, so it stores everything as uppercase.

Now for a demonstration of the lengths that the Explorer's file system will go to to try to keep everybody happy:

```
(setf new-u-path (make-pathname :directory (pathname-directory e-path)
                                :defaults u-path))
(symeval-in-instance new-u-path 'fs:directory) => ("xxx")
(pathname-directory new-u-path)                  => ("XXX")
(fs:pathname-raw-directory new-u-path)          => ("xxx")
```

where the **make-pathname** function effectively replaces the directory component of the UNIX pathname with the directory component from the Explorer pathname. Since the **pathname-directory** function of **e-path** returned an uppercase string, **make-pathname** knew that it should be stored in the preferred form for the destination host. Therefore, the Explorer directory "xxx" became the UNIX directory "xxx".

If you were to try the inverse of this last experiment and use **make-pathname** to insert the UNIX directory into the Explorer pathname object, then you would see that the UNIX directory "aaa" becomes the Explorer directory "AAA".

Generic Pathnames

6.9 Let's say your XYZ program includes a source file FOO plus a compiled version of that file—what exactly do you name these two related files? Your friendly local file system (Explorer or otherwise) probably doesn't care, but most programmers do.

Most programmers develop some pattern for naming related files. On the Explorer, related files are traditionally distinguished by file types. Furthermore, the single pathname that is used to represent one of these families of related files is called a *generic pathname*. If you send a pathname object **:generic-pathname** message, then it will return another pathname object with the type and version components as **:unspecific**.

The **:generic-pathname** method actually goes a step further. It attempts to back translate physical pathnames into their logical pathname forms—if there is one. Unfortunately, the **back-translated-pathname** function is not fully deterministic. Since any number of logical pathnames can be defined to translate into the same physical pathname, back translating may result in several choices.

Generic pathnames are mainly used by programs such as **make-system**, which have their own conventions about how related files are named. In **make-system**'s case, the namestring in the **defsystem**'s **:module** declaration is first converted into a generic pathname. Then, individual namestrings with specific type names are created for input and output to the various **make-system** transforms according to the defaults defined for those transforms (for example, the **:compile** transform assumes its input is of type **:lisp** and its outputs of type **:xld** while the **:fasload** transform assumes its input is **:xld**).

As with most conventions on the Explorer, **defsystem** offers a specific syntax to override these defaults. It is something of a tribute to the planning behind the Lisp system that programmers seldom need to change these defaults and many don't even know that you can.

Logical Pathnames — Part I

A Program Development Scenario

6.10 Logical pathnames are one of the most important *production* software facilities on the Explorer. And yet they are so poorly understood by newcomers that they are usually counted as a hinderance rather than as a vital tool. The overall problem here is that inexperienced programmers (and some experienced ones who should know better) can't see the trouble that's in store for them when it's eventually time to release their work as a product.

6.10.1 Let's say that a new Explorer programmer, Smedley, is assigned to build a new expert system. He starts by creating a directory for the program on his system under his personal directory, "EX22:SMEDLEY.EXPERT;". So far so good—all the program's files will be in one place so that it will be easy to hand off the software when finished.

Part of this expert system includes a help file that Smedley displays to the user if the user presses the HELP key. He places this help file in

"EX22:SMEDLEY.EXPERT;HELPER-FILE.TEXT"

in the directory along with all the other program files, but this file is different: he must code a reference to this file inside his program. In a burst of cleverness, Smedley codes the namestring in his program as

"LM:SMEDLEY.EXPERT;HELPER-FILE.TEXT"

so that wherever his expert system is running, it will always get its help file off the local machine rather than his personal machine, EX22. The special host name LM always refers to the machine the software is running on. Perhaps you already see some problems with this choice, but let's itemize poor Smedley's problems.

Choice of Host

6.10.1.1 Smedley assumed that each machine on which his expert system will be installed in the load band would also have a copy of his program's files in its file band. Wrong:

- Some machines (and many delivery vehicles) do not have a local file band.
- Even if the machine does have a file band, sites often choose to conserve disk space by keeping common files on a central file server.

What Smedley really needs is some way to postpone the decision as to what host to use until installation time.

Furthermore, Smedley coded this pathname as a host-specific (in this case, Explorer-specific) namestring. Even if we found a way to change the host component to something else, the syntax in the rest of the namestring might be wrong (for example, VAX VMS requires square brackets around the directory, whereas UNIX requires slashes between everything).

- | | |
|----------------------------|--|
| <i>Choice of Device</i> | <p>6.10.1.2 Since an Explorer has only one file system per machine, it never needs to make use of the device component of the namestring syntax. Smedley did not code a device in the help file namestring, so he is going to get the default device on whichever host he is loaded on. On systems that <i>do</i> have multiple devices, the default device may or may not be the one the user wants to use (default devices have a way of becoming overcrowded). So Smedley needs to allow a host-specific device to be specified once the host is decided upon.</p> |
| <i>Choice of Directory</i> | <p>6.10.1.3 It is reasonable to assume that all of the expert system files that were in Smedley's expert system directory on his machine are still in one directory at the user's site. It is probably even reasonable to assume (but not guaranteed) that this dedicated directory at the user's site is named EXPERT (that is, it's short and doesn't use special characters). But it is a wild flight of fancy to assume that there will be a SMEDLEY directory at each site under which to place the EXPERT subdirectory. In fact, he is not guaranteed that the host operating system even supports multilevel directories, regardless of their names.</p> <p>Smedley has committed the programming equivalent of the man who built a boat in his basement only to discover when he finished that he had no way to get it out. Smedley is going to have to explain to each site that not only are they going to have to create a directory named EXPERT, but also they are going to have to create a dummy top-level directory named SMEDLEY to put it in—programmer immortality the hard way.</p> <p>What Smedley needs is a way of changing the name of his directory as his program is moved from site to site and of allowing his directory to be attached at any level in an existing hierarchy of directories at a given site.</p> |
| <i>Choice of Name</i> | <p>6.10.1.4 While the file name HELPER-FILE is nicely descriptive, it is too long for some popular operating systems (8-9 characters limit is common), and it contains a special character, the hyphen, that other operating systems disallow. Unless Smedley is willing to name his files to fit the worst case operating system (five uppercase letters?), then he needs a way to map filenames as his program is moved from site to site.</p> |
| <i>Choice of Type</i> | <p>6.10.1.5 Some systems don't support file types, but as long as Smedley doesn't have any other file named HELPER-FILE (that is, he doesn't need the type component to distinguish the file), then he should just be able to safely drop the type. Of those systems that do allow file types, virtually all have a text file type, so he is on safe ground there. But! All systems with text file types don't necessarily name that type TEXT. Several popular systems name it TXT.</p> |

We have already seen the answer for this particular part of the problem: canonical types. Smedley needs to specify the file type as :text and then let the system figure out what type name string is appropriate. That is, this problem of type name and the problem of Explorer-specific syntax mentioned at the start could have been solved if Smedley had used **make-pathname** to assemble the pathname components rather than hardcoding a namestring. Instead of

```
"LM:SMEDLEY.EXPERT;HELPER-FILE.TEXT"
```

he should have used

```
(make-pathname :host      "LM"
              :directory ('("SMEDLEY" "EXPERT")
              :name       "HELPER-FILE"
              :type       :text)
```

which would have taken care of the host-specific syntax and type name problems but not the general name mapping problem. This example might suggest some other alternatives to you. For example,

```
(make-pathname :host      *expert-file-server*
              :device    *expert-device*
              :directory *expert-directory*
              :name      *expert-help-file*
              :type       :text)
```

allows the troublesome host, device, directory, and name components to be filled in at run time from global variables that can be set at each site.

Although such use of **make-pathname** with variable arguments solves Smedley's problem, widespread use of this technique would create havoc at the user's site. Each new program would potentially require dozens of meaningless variables to be initialized to obscure values before the program would work. The real solution is to use logical pathnames in the program and then to provide a single translation table at each site mapping those logical pathnames to site-specific physical pathnames.

By the way, if you think I'm ridiculing inexperienced programmers like our hapless Smedley, then please understand that I am speaking from the collective experience of my friends and me. One of the reasons I was asked to rewrite my Load Distribution Tape utility for the Explorer is that the original version assumed that the local machine would always have a file system. Wrong.

Logical Pathnames — Part II

6.11 Question: What makes a logical pathname different from a physical pathname? **Answer:** a logical host rather than a physical host.

We saw in our original discussion of namestrings that the characters before the first colon are assumed to be the name of the physical host the namestring pertains to. The namestring parsing software looks up that physical host name in system tables to find out how to parse the remainder of the namestring following that colon.

Logical namestrings are the same: the characters before the colon still identify the host. However, when the namestring parsing software looks up the host name, it discovers that this is a *logical* rather than physical host. (That is, the host object it finds is built on the **fs:logical-pathname** flavor, as physical pathnames are not.) Knowing that this namestring has a logical host tells

the parser how to parse the remainder of the logical namestring into components (just as before). However, those components cannot be used directly. Instead, they must be mapped, or translated, according to a translation table associated with that particular logical host.

There are two logical namestring syntaxes. The original syntax should be portable across Zetalisp-derived Lisp machines, and it uses a minimum number of delimiters:

```
"HOST:DIRECTORY;NAME TYPE VERSION"
```

The newer syntax looks just like Explorer namestring syntax but, of course, is not portable:

```
"HOST:DIRECTORY;NAME.TYPE#VERSION"
```

The Explorer's `parse-namestring` function accepts either syntax and parses the logical namestring into a pathname object with these logical (untranslated) components. When it comes time to actually access the physical file, you can send a `:translated-pathname` message to the pathname object, and it will return the corresponding physical pathname object. If this message is sent to a physical pathname object (one that requires no translation), then that physical pathname object is returned unchanged.

The `translated-pathname` function is a little more general-purpose. It accepts either a namestring or a pathname object and returns a physical pathname object. As with the `:translated-pathname` message, a physical namestring is returned as a physical pathname object, and a physical pathname object is returned as is.

The mapping that defines how to translate each namestring for a given logical host is specified by the `fs:set-logical-pathname-host` function. By convention, the `fs:set-logical-pathname-host` function that defines logical *lhost* is the sole form in the file

```
"SYS:SITE;lhost.TRANSLATIONS"
```

so that loading this file automatically defines the logical translations for logical *lhost*. Creating this file and defining the `fs:set-logical-pathname-host` form that goes in it is typically done by the programmer whose program needs that host.

A Better Program Development Scenario

Choice of Logical Host

6.11.1 Now that we know something about logical pathnames, how should Smedley have named his files?

6.11.1.1 The first question is: "How many logical hosts does he need?" This immediately brings up a second question: "Why do we care?"

We care because one logical host translates into one physical host. Therefore, if your end user has a practical need to place your files on more than one file server, then you need more than one logical host.

For most Explorer software products, the only files involved are the product object files and, if a source license was included, the source files. Because of assumptions built into the Explorer system software, source and object files should always be in the same directory if at all possible. Therefore, if source

and object files are the only files associated with the product, then one logical host will do.

Once a product has been installed in a load band, there is no need to access its source or object files during run time. However, some products also include run-time files such as help files (like Smedley's), configuration files, database files, and so on, that are needed in the day-to-day operation of the product. Furthermore, some of these run-time files may have been designed to be customized by the user so that one site-wide copy is not appropriate. Even if personal copies of run-time files are not needed, software license restrictions often require that access to source files be restricted, whereas the run-time files are treated as public information.

Considering all of this, Smedley would probably do well to define two logical hosts: one for his expert system's source and object files and one for his run-time files such as the help file we talked about before.

Choice of Device

6.11.1.2 Logical pathnames don't have device components, so Smedley is spared making a decision about what to code in his program for this one. However, his translations file can specify a physical device for each logical directory if need be.

Choice of Directory

6.11.1.3 Again, Smedley doesn't have a problem here when coding his program. He can use anything he wants because it is going to be mapped into a specific physical directory or subdirectory before it is used.

It is common practice for programmers to define their logical directories to have the same names as the physical directories on their home machines. Newcomers are occasionally confused when they translate a logical name-string only to discover that they still have the same namestring but now with a physical host. Not to worry. The presence of the physical host shows that the namestring really was translated.

Then again, maybe you should worry. A misfeature of our current logical pathname translation software is that an unknown logical directory is translated as being itself without flagging an error. Therefore, if all your logical directories have the same names as your physical directories, then you'll still need a translation file to map the logical host to a physical host, but you won't need any directory translations. All directory names default to themselves. Of course, if there is a typo in your logical directory name, instead of getting some sort of undefined directory error you quietly get a typo in your physical pathname.

Choice of Name

6.11.1.4 Smedley's two main concerns here are the length of the name component and special characters in the name. For the most part, logical pathname translations aren't going to be of any help to Smedley. He's going to have to pick something mutually acceptable to all operating systems that are likely to host his program.

Choice of Type

6.11.1.5 Smedley should *always* use canonical types.

Choice of Version

6.11.1.6 Smedley should *always* use the newest version of any file. Therefore, he should not mention the version and let it default to newest, or—if he is paranoid (as it pays to be)—then he should explicitly code the newest version indicator into each pathname.

Miscellaneous

6.11.1.7 Smedley should never hardcode a namestring—even a logical namestring—in his program. Instead, he should use **make-pathname** to assemble the individual logical components into a pathname object. The **make-pathname** function is required because logical namestrings do not accept canonical types.

If Smedley needs a namestring to display to his users, then he should use the **namestring** function on the pathname object returned by **make-pathname**.

Tying It All Together

6.11.2 Now that Smedley has converted his program to use logical pathnames, there are a few odds and ends left.

First, for each logical host (generically shown as *lhost* below), he needs to create a file named

SYS:SITE; lhost .TRANSLATIONS

which contains a single **fs:set-logical-pathname-host** form. Aside from comments, there should be nothing else in this file. For details, see the file

SYS:SITE;-READ-ME-.TEXT

Second, he should create a file named

SYS:SITE; system .SYSTEM

where *system* is the name he has chosen for this system (that is, the name defined by **defsystem**). The read-me file mentioned above also explains the contents of this file. A detail the read-me file doesn't mention is that Smedley should include a **load** form to load the translations file(s) ahead of the **defsystem** form in the **.SYSTEM** file. Furthermore, the namestrings used in the **defsystem** declarations should be logical namestrings.

Thus, the *only* place that the site-dependent physical pathnames associated with Smedley's program are mentioned is in the translation files(s).

SYS-Host

6.12 The logical host named **SYS** (and conversationally referred to as the **SYS-host**) is the single most important logical host on the Explorer. **SYS-host** has two main uses:

- It holds the **SITE** directory which, in turn, holds all site-dependent information needed by Explorer software products.
- It holds the system source files such as those accessed by **META-**, in **Zmacs**.

The Site Directory

6.12.1 If you write your code correctly, then you can move your program from site to site without changing anything except files in the **SYS:SITE;** directory. The idea behind the **SITE** directory is that all the types of information that are likely to change among sites have been broken down into several classes. Each class is then represented by a file or a group of files on the **SITE** directory.

For example, each logical host on the system must have its own set of logical-to-physical translations. Therefore, the two namestrings

```
SYS:SITE;FOO.TRANSLATIONS
SYS:SITE;BAR.TRANSLATIONS
```

represent the translations files for logical hosts FOO and BAR, respectively.

The previous example illustrates the naming convention for files in the SITE directory:

```
SYS:SITE; instance-name.class-name
```

That is, even though these files are ordinarily :lisp or :xld files, their file type component is used to identify what class of site-dependent information the file holds.

The two most common types of files in the SITE directory are :translations mentioned above and :system, which tells make-system where to find the defsystem form that defines the system indicated by the name component. For example, consider some files associated with the Grasper toolkit:

```
SYS:SITE;GRASPER.SYSTEM
SYS:SITE;GRASPER.TRANSLATIONS
```

The first file, when loaded, either contains the defsystem for Grasper or it tells make-system where it can find the defsystem (see the file SYS:SITE;-READ-ME-.TEXT for details). The second file contains the translations for the Grasper logical host, which contains the Grasper source and object files.

This example also illustrates the distinction between system conventions and programmer conventions. The system understands the implications of the file types :translations and :system when found in the SYS:SITE; directory. If, for some obscure reason, you decide *not* to follow these system conventions, then you must be careful not to use any system defaults. Instead, you must fully specify everything you want done (and you'll probably still get caught).

The example also shows the programming convention of using the product name, Grasper, as the system name, GRASPER.SYSTEM, and as the principal logical host associated with the product, GRASPER.TRANSLATIONS. While this naming convention might seem eminently sensible, the system couldn't care less. As far as the system software is concerned, there is no relationship between the name of the product and its defsystem name or between its defsystem name and any logical hosts it may use.

System Source Files

6.12.2 The other principal use of SYS-host is to hold the Explorer system source files. That is, it contains the source files for everything that comes with the basic Explorer system. Optional Explorer software products usually have their own logical host rather than sharing SYS-host.

Why SYS-Host Causes Problems

6.12.3 Newcomers to the Explorer often have the system fail for obscure reasons "because the SYS-host was set wrong". Why would anyone reset SYS-host to a different machine? What breaks if you do reset SYS-host? The main culprit is usually development software.

Let's say you have *two* copies of the system source at your site. Perhaps one is for the current release and one is for the previous release. Or perhaps one is for the official release, and one has a bunch of experimental software added. The META-. facility in Zmacs is so handy that programmers will not part with it easily. If they are working with something other than the current, official source, then they will frequently reset SYS-host to point to the machine that has their development source on it.

Okay, so much for why you might want to change SYS-host; now what can go wrong? The problem is that just as no man can serve two masters, neither can SYS-host. When you switch SYS-host so that you can META-. your development source, you also switch the place make-system looks to find its :SYSTEM files.

If everybody were thorough and conscientious, then there would be no problem. If you make sure that every potential SYS-host on your network starts out with a *complete* copy of the system source for META-. and a *complete* SITE directory, then you can just change the individual files you are interested in, and everything else works fine.

But! SYS-host encompasses a large amount of data. It takes up a lot of valuable disk space to keep multiple copies of it; it takes a lot of time to copy it; and it takes a lot of vigilance to keep everything up-to-date. In short, no one bothers. Instead, they create a partial SYS-host with just the files they are interested in and then proceed to use the system as they normally do. Good luck.

Programmers' habits of changing SYS-host to this machine or that machine for one reason or another is what leads to the admonition in the product installation instructions "make sure your SYS-host is set correctly". Otherwise, you can never be sure where your new software will end up.

You can find out where your SYS-host is set to with the form

`(translated-pathname "sys:")`

and you can change your SYS-host with the form

`(sys:set-sys-host "host-name")`

where *host-name* is the name of a network host written as a string.

Absolute Versus Relative Directories

6.13 Some operating systems (such as, UNIX, Symbolics™, VAX VMS) support the concept of relative directory specifications in their namestring syntax. The Explorer system does not have such a feature (it supports only absolute directories), but it must still be able to parse and manipulate pathname objects for all hosts it touches.

- An *absolute* directory specification assumes you are starting at the root directory of the device. Such a specification always identifies the same directory regardless of the current execution environment.
- A *relative* directory specification assumes that there is some current working directory known in the execution environment and that this relative directory is really just a suffix specification. Such a specification identifies different directories at different times.

Lisp machines don't have the formal concept of a working directory to use when parsing relative directories, but they do always have a default pathname handy. Therefore, relative directory parsing on the Explorer system is always relative to the directory component of the current default pathname.

Simple Relative Directories

6.13.1 Let's assume that your UNIX working directory is `/x/y/z/`. Now, if you specify an absolute UNIX directory of `/a/b/c/d/` (note the leading `/`), then you get exactly that; `/a/b/c/d/`, the working directory is ignored. However, if you specify a relative directory of `c/d/` (note the absence of a leading `/`), then you get `/x/y/z/c/d/`. That is, the relative directory is concatenated onto the working directory.

The relative directory feature is often loosely referred to as *relative pathnames* even though only the directory component is involved. Typical.

If you view a file system as a logic tree of directory nodes with files as its leaves, then an absolute directory—such as our working directory in the example above—uniquely identifies a node in that tree. Given that you are standing at that node in the directory tree, a relative directory tells you how to walk away from the working directory node and go one or more levels down the tree towards the leaves.

General Relative Directories

6.13.2 If one notation can tell you how to walk *down* the tree, then another notation can tell you to walk *up* the tree towards the root.

Let's return to our previous UNIX example where our working directory (that is, the starting location in the tree) is `/x/y/z/`. We saw that the relative directory `c/d/` was equivalent to the tree walk instructions:

1. Start at `/x/y/z/`
2. Go down to `z`'s child named `c`
3. Go down to `c`'s child named `d`

Now we see that the relative directory `../c/d/` where the leading `..` refers to the parent directory means to

1. Start at `/x/y/z/` (as before)
2. Go *up* to the parent of `z` which is `y`
3. Now go *down* to `y`'s child named `c`
4. Go *down* to `c`'s child named `d`

In a similar vein, `../../c/d/` would mean go up to `z`'s parent `y` and then go up to `y`'s parent `x` before starting down to `x`'s child named `c` and so on.

If this relative directory syntax were carried to the extreme, then one overly elaborate namestring could specify a single-directory walk up and down and up and down the tree. In practice, most operating systems put a stop to such exuberance by limiting the up components to the beginning of the relative directory specification. Once you finally start down the tree, you can never go back (to coin a cliche).

**Pathname Object
Notation**

6.13.3 We saw before that the internal representation of a directory component is a list of strings where each string is the name of a directory subcomponent. For example, the namestring directory `/a/b/c/d/` would be represented in a pathname object as

```
(“a” “b” “c” “d”)
```

In the case of a simple relative directory, such as `c/d/`, the list starts with the keyword `:relative` as in

```
(:relative "c" "d")
```

In the case of a general relative directory with up components, it starts with `:relative`, and each initial up component is represented by the keyword `:up`. In this notation, a string by itself implies down even though there is no explicit `:down` keyword as a counterpart to `:up`. For example, `../../c/d/` would be

```
(:relative :up :up "c" "d")
```



PROCESSES AND SCHEDULING

Introduction

7.1 The Explorer is a uniprocessor with a multiprocessing operating system. That is, it has one CPU that is time-shared among several independent *processes*. The time-sharing is controlled by the *Scheduler*. The current Common Lisp standard does not include the notion of processes and therefore effectively assumes that everything always executes in the same process.

In many operating systems, processes and their schedulers are system primitives and may even be linked directly to hardware features. On the Explorer, both processes and their Scheduler are simply coroutines the same as any Explorer programmer might create for his or her own use in a program. Corouting, of course, is that well-known programming concept you learned in computer science class and have never used since.

Some Terminology

7.2 Before we get mired in details, let's sort out some terms that we need and see how they are related to each other.

Stack Group

7.2.1 At the lowest level we have the *stack group*, which is a primitive Explorer data type. Stack groups are primitive only in the sense that they are implemented directly in microcode. Actually, their internal structure is quite complex because they must be able to record the complete processing state of the Explorer hardware and software in case of a context switch.

Coroutines

7.2.2 A stack group, by itself, is just a passive processing environment. If you give a stack group a function to execute in that environment, then you have a *coroutine*. The distinguishing characteristic of a coroutine is that it allows you to return from it part-way through its execution. Later, when you call it again, it picks up execution in the middle of wherever it left off rather than starting over at the beginning again as a subroutine would do.

Processes

7.2.3 A coroutine matches our general notion of what a software process should be (independent execution, independent data), but practical processes require a lot of bureaucratic overhead: what's ready to run, what's not; what process has priority over which other; what needs to be killed; and so on.

On the Explorer, a *process* is an instance of the `sys:process` flavor, which associates various overhead information with exactly one stack group that records that process's execution state. Notice that while a programmer may include coroutines in the implementation of his or her process, the process itself is represented by one unique stack group.

Scheduler

7.2.4 The Scheduler is another coroutine (*not* a process) that knows about all active processes. The Scheduler, in turn, is uniquely known by certain system initialization code that kicks it off.

Why Do I Need Processes?

7.3 Actually, if your life's goal is to write functions that someone else will call, then you are neither interested in processes nor care how they are scheduled. If, on the other hand, you think of the program you are writing as a *standalone application* or as an *asynchronous server*, then you are probably going to need your own process.

The key word here is *asynchronous*. For pure functionality, you can do just about everything you want with one (possibly elaborate) function running in one process, but it can do only one thing at a time. For example, if you called a long-running function from a Lisp Listener, then you've lost the use of that Listener until the function returns because the function is using the Listener's process.

For a more elaborate example, let's assume that your function puts two windows on the screen: one for display, one for queries. If both windows are controlled by the same process, then only one can be doing its thing at a given time.

- If your function is scrolling a volume of information in the display window, the query window appears to be dead.
- If the process has queried you and is waiting for your response, then the display window is frozen.

If these two windows had been controlled by separate processes, then instantaneously only one would have been active at a time (because the Explorer is a uniprocessor), but activity would have regularly alternated between them independently of what the user happened to do. The data scrolling in the output window becomes jerky when you respond to a query, and the query window is sluggish if the display window is in use; but neither comes to a complete halt waiting on the other to finish. There is the illusion of simultaneous execution.

Why Does the System Need Processes?

7.4 The previous example also illustrates why coroutines by themselves are not enough. A pair of query/display coroutines would have given independent, and *relatively* asynchronous execution. For example, simple coroutines would have allowed the program logic of the query code and the display code to deliberately alternate between a little querying and a little displaying.

However, if users were slow in answering a query, then the display would have had to wait for them to finish their type-in. Alternately, a long display might lock up the query window for many seconds.

In other words, if your goal is to create the illusion of parallel processing through coroutines, then your coroutines need to be well-behaved and never hog the processor. Of course, simply giving up the process periodically isn't necessarily the right thing to do either. There is no particular value in, say, the display coroutine courteously giving up the processor in the middle of a long display if the query coroutine is still waiting for the user to type something.

Rule of Thumb: Coroutines should be designed to switch among themselves only as their program logic requires. Coroutine switching for the purpose of processor sharing should be done by some third party who has a more global view of what is going on.

Explorer Coroutines

7.5 The notion of calling a subroutine is deeply ingrained in our thinking; a call to a function causes execution to do the following:

1. To jump to the logical beginning of that function
2. To run to the logical end of that function
3. To return

Notice that there is something of a master/slave relationship implied by a subroutine call: master calls slave, slave returns to master.

Call/Return Versus Resume

7.5.1 Our faithful terms, call and return, fail us when we talk about coroutines. Coroutines are equals. An executing coroutine can transfer to another coroutine, which causes that other coroutine to be *resumed* (we'll see how coroutines are resumed later). A transfer out of the executing coroutine is much like calling an ordinary function, but resuming means just what it says: execution picks up in the middle where it left off—not at the beginning.

Furthermore, in coroutining, there is no analogy to the subroutine return. Instead, Coroutine A resumes Coroutine B, which runs a while and then resumes another coroutine. Coroutine B *may* have resumed Coroutine A, but it may have resumed some other coroutine. While an Explorer coroutine can blindly resume its Resumer just as a subroutine can blindly return to its Caller, this capability is just a convenience detail.

If you were to characterize subroutine-like execution as call, return, call, return; then coroutine execution would be resume, resume, resume, resume. With coroutines, you've got to stop thinking about what happens when the function you have called returns.

Argument Passing

7.5.2 Instead of talking about arguments and return values, which implies that master/slave relationship again, we need to think in terms of transferring a Lisp object to the coroutine being resumed. Therefore, when Coroutine A resumes Coroutine B, A *may* optionally transfer an object to B. If Coroutine B executes a while and then resumes A, it *may* transfer an object to A.

From the point of view of Coroutine A, it transferred a single argument to Coroutine B and then B eventually returned a value. Sounds like our familiar call/return relationship, doesn't it? But there is a surprise lurking here.

Let's say Coroutine A resumed Coroutine B. Next, B resumed C and finally C resumed A with an object transferred on each resume. Again, from Coroutine A's viewpoint, it transferred an object to Coroutine B and, at some later point, Coroutine B returned to A—passing the object that came from Coroutine C!

Beware: if you insist on thinking of coroutine transfers in terms of the traditional call and return, then remember that the coroutine that returns to you is not necessarily the one you called.

**Keeping Track
of Coroutines**

7.5.3 The description of Explorer coroutines so far makes it sound as if to use coroutines is to lose control of execution. Actually, it is not as bad as all that. The important point is that coroutines allow, but do not require, a call/return execution pattern.

For example, if Coroutine A resumes Coroutine B, then the microcode updates **current-stack-group-resumer**, a special variable in B's environment, to record B's Resumer (which, in this case, is Coroutine A). Therefore, any coroutine can return to its caller (without knowing who that caller is) by resuming the value of **current-stack-group-resumer**—but it is usually simpler just to execute the **stack-group-return** function that does the resume for you.

On the other hand, a strict call/return pattern implies that everything was planned out ahead of time. What do you do when something unplanned happens?

For example, the Scheduler normally resumes a process and then that process later resumes the Scheduler (that is, the call-return pattern). However, if that process should get an error, then the microcode forces the process to resume the Error Handler coroutine. If the Error Handler can fix up the problem, then it resumes the interrupted process (call/return again). If, however, the error is fatal, the Error Handler resumes a different process. In the end, the Scheduler will once more be resumed, but not by the same process the Scheduler thought it was running.

An Aside

7.5.4 The fact that the Error Handler runs in a separate process explains why newcomers trying to use the Error Handler to examine their code sometimes complain, “Hey! Where did my data go?” Actually, the data is right where you left it in its own process—but that isn’t the environment the Error Handler is in.

Normally, you’ll never notice any problem because the Error Handler’s examination commands make a point of examining the process that called the Error Handler and not the Error Handler’s environment itself. Of course, even the Error Handler can be fooled on occasions.

If you ever get more than one right pointing arrow as an Error Handler prompt, then your Error Handler has gotten an error and called itself recursively. Now the Error Handler’s examination functions will show you the previous Error Handler and not your original process. Surprise!

**Coroutine
Programming**

7.5.5 If you want to use the Explorer’s coroutining facility for traditional coroutine implementations, then the simplest way to do a call/return is to funcall the stack group that represents the coroutine you want to call with one argument that is the object you want transmitted. That coroutine later returns by executing the **stack-group-return** function, again with the object to be transmitted as its single argument.

On the other hand, if you need the anybody-can-resume-anybody feature of Explorer coroutines, then you can use the **stack-group-resume** function,

which takes two arguments: the stack group (that is, the coroutine) to be resumed and the object to be transmitted.

Notice that even though you can **funcall** a stack group as though it were a function, stack groups and functions are two different beasts. You'll remember that stack groups are a primitive data type, which means they can be recognized on sight by the microcode. Therefore, if the object you **funcall** is a **symbol** data type, then the microcode does a traditional function call. If it is a **stack group** data type, then the microcode does a stack group switch. (If it is an **instance** data type, then the microcode sends a message—but that's another section.)

Stack Groups

7.6 Coroutines and stack groups don't come up in ordinary conversation much; but when they do, the two words tend to be used interchangeably. One term reflects the programmer's viewpoint; one reflects the microcode's.

There is a distinction, however. A stack group as created by the function **make-stack-group** is merely a passive environment in which a function can execute. The **stack-group-preset** function initializes an existing stack group so that a specified function (your coroutine) starts executing the first time that stack group is resumed.

Stack Group Contents

7.6.1 A stack group is a data structure containing three principal pieces of information:

- Register save area — This is the traditional save area for the hardware registers when the stack group is not executing in the CPU.
- Control stack — This is the traditional call/return stack that holds the function-calling history, arguments, and local variables. This stack is also called the *regular PDL*. PDL stands for *push down list*, the old name for a push down stack, and is pronounced *piddle*.
- Dynamic environment stack — This is the stack that records which special variables have been shadowed by which bindings so far in this stack group. This stack is also called the *special PDL*.

This pair of stacks, the regular PDL and special PDL, is why this object is called a *stack group*.

Global Versus Private Variables

7.6.2 Now that we know that a stack group is used to record the private environment of a process, it would be useful to distinguish private from global variables.

- Special variables declared at the top level (in other words, **proclaimed special** or defined in a **defvar**, **defparameter**, or **defconstant**) are *global special variables* and are equally accessible to all stack groups.
- Global special variables that are later bound within a stack group are *shadowed* such that stack group code inside the binding sees only these newly bound values while other stack groups still see the original global values.
- Variables declared **special** within a stack group are accessible only within that stack group.

Do not confuse special variables that are accessible only within one stack group with *local* variables (that is, non-special variables) that are accessible only within the lexical scope of a block of source code. For example,

```
(defvar *a* 11)
xxxx
xxxx
(let ((*a* 99)
      (b 22))
  (declare (special b))
yyyy
yyyy)
xxxx
xxxx
```

Let's call the code represented by the xxxx lines in the above example the *x-code* and that represented by yyyy lines the *y-code*.

The variable ***a*** with the value 11 is a global special variable and is therefore available to all code of all stack groups. In particular, the *x-code* of this example (both before and after the `let`) sees `*a*=11`. This `defvar` definition of `*a*` illustrates the first bullet above.

The `let` in this example binds `*a*` to 99 so that the *y-code* sees `*a*=99` even though other stack groups still see `*a*=11` (assuming, of course, that they haven't bound `*a*` to something else as this example did). Furthermore, if the *y-code* should setf `*a*` to some other value, then it is this `let`'s binding (which can be seen only by the *y-code*) that is set, not the global value seen by the other stack groups. In contrast, if the *x-code* (which is outside the `let`) should setf `*a*`, then it *would* be seen in other stack groups. This `let` binding of `*a*` illustrates the second bullet.

The `let` binding of `b` and the `declare` make `b` available to the *y-code* and to everything called by the *y-code* as a special variable. Neither the *x-code* in this stack group nor any code in any other stack group can see `b`. This `let` binding of `b` illustrates the third bullet.

Initial Process Bindings

7.6.3 When a new process is started, the system binds the global special variables associated with Lisp Listener type-in and type-out (that is, *, **, ***, +, ++, +++, /, //, ///, -, and `*values*`) to themselves. The rationale is that each process starts out with the same values that its parent process had. However, later type-in and type-out in that new process's Listener will not affect any other Listener.

The important thing to notice here is that these automatic bindings are relatively unimportant because those variables are relatively unimportant to most code. They deal only with an interactive feature of Lisp Listeners, and most processes aren't used as Listeners. Few people will ever care whether these variables are bound or not. Pity.

On the other hand, the most critical global special variables are *not* automatically bound at process initialization time. Therefore, if you were to setf one of these variables, then your process and everybody else's would see the change. If that is what you want—fine. However, if you do not want to

interfere with other processes, then bind that global variable within your process before you setf it. For example,

```
(let ((*global-var* *global-var*))  
...your code...)
```

Why Does Anyone Care?

7.6.4 If two Explorer processes need to share a piece of information, then all you have to do is define a global special variable with a **defvar** or **defparameter** and let both processes access it. Data sharing is as easy as that. This simplicity is in stark contrast to conventional operating systems that were designed to keep processes separate at all costs. Data sharing on these systems is tedious and not necessarily efficient.

Then again, sometimes you don't want to share data. For example, the global special variable ***print-base*** determines how integers are printed by default. ***print-base*** is normally set for base 10, but let's say you need your printouts in base 16. A setf of ***print-base*** to 16 would solve your problem, but it would change the print base of *all* processes (much to their surprise). Here is where you need to bind ***print-base*** within your process so that your changes to ***print-base*** do *not* become global changes.

Rule of Thumb: Never setf a global special variable. *Bind* it.

There is one final problem to consider: just because your process is independent of other processes in the system, will it be independent of itself? That is, if you have defined a couple of global special variables for your process, then what happens if several copies of your process all start using those variables at the same times?

If any two processes need to update the same global special variable, then there is always the chance that one process will be time-sliced out of the processor with the update only half completed. The simplest way to prevent incomplete updates is to wrap a **without-interrupts** form around the update code. This form prevents the Scheduler from being run while the code in the body of the form is executing.

For example, let's suppose ***print-que*** is a global special variable of outstanding print requests. Various processes put their requests onto this queue, and the print server takes them off. If a requesting process were to simply use

```
(setf *print-que* (cons my-request *print-que*))
```

then another process could potentially be time-sliced in between the time the **cons** reads ***print-que*** and **setf** writes it. The proper way of doing this operation would be

```
(without-interrupts  
  (setf *print-que* (cons my-request *print-que*)))
```

Now we can see that data sharing is a two-edged sword. Operating systems such as UNIX pride themselves on process isolation and make sharing hard. At the other extreme, the Explorer allows such simple data sharing that

you've got to plan your code carefully so you don't step on someone else's numbers. Nothing comes for free.

Bypassing Bindings

7.6.5 Binding of global special variables such as shown above usually provides a solid bulkhead separating the code inside the `let` from code outside the `let`. However, there are ways of tunneling under the bulkhead.

The `set-globally` function bypasses any intervening bindings of a special variable and permanently sets the global value of that variable for all to see. The macro `let-globally` temporarily sets the value of one or more global special variables while its body executes. Upon exit, the `let-globally` sets those variables back to their original value. For example,

```
(defvar *x* 11)
wwww
wwww
(let ((*x* *x*))
  xxxx
  xxxx
  (set-globally '*x* 99)
  yyyy
  yyyy)
zzzz
zzzz
```

If the above code were the entire program, then

- While the *w-code* and later the *x-code* is executing, everyone in the system sees $*x*=11$ (even though the *x-code* is seeing a local binding of $*x*$ with that value).
- While the *y-code* is executing, the *y-code* still sees $*x*=11$ because the local `let` binding of $*x*$ is remembering the original global value, but everyone else in the system sees $*x*=99$ (unless they have bound $*x*$ themselves) because of the `set-globally`.
- While the *z-code* is executing (notice that the `let` binding of $*x*$ has expired), everybody in the system including the *z-code* once again sees the same value of $*x*$, which is now 99 .

Processes

7.7 Even if you did not know that Explorer processes were implemented as coroutines, it would be obvious that processes and traditional coroutines have something in common. The most important similarity is that they must be able to execute independently of each other and of each other's data.

The most important difference is that a set of coroutines normally knows about one another, and each resumes the other as needed to accomplish their common goal. In contrast, processes normally know nothing of each other and have no idea of who should be resumed next. Therefore, coroutines used as processes need a third party mediator who has a more global view of what is going on in the system.

On the Explorer, this mediator is yet another coroutine called the *Scheduler*. The normal operating pattern is for the Scheduler to resume a waiting process, the process runs for a while, and then it resumes its Resumer (that is, the Scheduler). Once resumed, the principal job of the Scheduler is deciding who to run next. Notice that most of the information recorded in a process

(described next) goes towards providing the Scheduler with information to make intelligent “Who’s next?” decisions.

An Explorer process is an instance of a flavor built on the `sys:process` flavor. The main information recorded in the instance variables of this flavor is:

- Initial Function and Arguments — The Scheduler applies this function to these arguments the first time a new process starts (that is, the first time it is resumed).
- Stack Group — This is the private environment in which the process’s Initial Function executes.
- Wait Function and Arguments — The Scheduler applies this function to these arguments for each process that is a candidate for being resumed. If this form returns true, this process is resumed.
- Miscellaneous memory allocation parameters.
- Miscellaneous scheduling parameters.

Process Initial Function

7.7.1 This function and any functions it calls do the work of your process. The process is temporary or permanent depending on whether this Initial Function returns or not. For a temporary process, the Initial Function eventually returns and the process terminates normally. For a permanent process, the Initial Function usually contains an infinite loop around the main processing body.

This Initial Function can do anything you want, but good programming practice suggests that this function establish all process private bindings and then call the main processing body. A starter set of bindings that all processes should have is:

```
(let ((*terminal-io* *terminal-io*)
      (*standard-output* (make-synonym-stream '*terminal-io*))
      (*standard-input* (make-synonym-stream '**terminal-io*))
      (*error-outout* (make-synonym-stream '**terminal-io*))
      (*trace-output* (make-synonym-stream '**terminal-io*))
      (*query-io* (make-synonym-stream '**terminal-io*))
      (*debug-io* (make-synonym-stream '**terminal-io*)))
  ...your Initial Function Body...)
```

These bindings assure that the standard I/O streams operate in the expected way, regardless of whatever strange state they may have been in while inside the process that made yours. Furthermore, if you suspect that code called by your Initial Function might be imprudently setting shared global variables such as `*print-base*` mentioned above, then add bindings of those variables to themselves such as

```
(*print-base* *print-base*)
```

to guarantee that local modifications to those variables can never escape to raise havoc in the outside world.

Once a process starts executing, it can stop for one of four reasons:

- A process may voluntarily pause by calling the function **process-allow-schedule**. It will automatically be resumed the next time through the Scheduler's polling loop.
- A process may deliberately wait by executing the function **process-wait** and leaving a wake-up call in the form of a Wait Function and arguments (explained below).
- A continuously running process is involuntarily suspended about once a second by a hardware interrupt called a *sequence break*. The process is automatically resumed the next time through the polling loop.
- In case of error, the microcode forces the process to resume the Error Handler process (for its own good, you understand).

Even though the Initial Function executes in the process's private stack group environment, its arguments (if any) are evaluated in the Scheduler's environment. Therefore, these Initial Function arguments should be forms that reference only constants and global special variables. See paragraph 7.7.6, Errors Inside the Scheduler, for some caveats.

Wait Function and Arguments

7.7.2 The way the Scheduler determines whether a waiting process is ready to run yet is to apply that process's Wait Function to that Wait Function's arguments, if any. If the function returns true, the process is resumed. Otherwise, the process continues to wait at least until the next time through the polling loop.

Both the arguments to the Wait Function and the Wait Function itself are evaluated in the Scheduler's environment. Therefore, as before in the case of the Initial Function arguments, this function and its argument forms should reference only constants or global special variables. In particular, a process's Wait Function *cannot* reference any of the process's private variables. See paragraph 7.7.6, Errors Inside the Scheduler.

You should design your Wait Functions so that if they are going to return false, they do so as quickly as possible. If that sounds like an odd requirement, look at things from the Scheduler's point of view as it goes around the polling loop. Once around the loop means the Scheduler has executed every Wait Function once, and usually all but one or two have returned false. Therefore, inefficiently written Wait Functions do little for their processes, but they do slow down the polling loop for everyone.

Here is one more important caveat about what you can put into a Wait Function: a Wait Function should *never* have side effects. You are not guaranteed when it will be called, what order it will be called in, or how many times it may be called as the Scheduler goes through the polling loop.

process-wait Versus Wait Function

7.7.3 **process-wait** is a system supplied function that a process calls when it wants to suspend execution temporarily. The arguments to **process-wait** are a function and some arguments for that function that are handed to the Scheduler to be used as that suspended process's Wait Function.

There are several variants on **process-wait**. The function **process-sleep** is like a **process-wait** in which the wait function automatically checks for an

elapsed time measured in 60ths of a second. The Common Lisp `sleep` function and `process-sleep` are similar, except that `sleep`'s argument is in whole seconds.

Finally, `process-wait-with-timeout` combines the `process-wait` and the `process-sleep`. You specify both a Wait Function and a maximum wait time. If the time limit expires without the Wait Function coming true, then the process is resumed anyway. The return value allows you to distinguish why you were resumed.

`process-wait-with-timeout` is frequently used by long-running functions that may occasionally need to query the user about some special situation. There is always the possibility that the user got bored and wandered away from the terminal. If the query has a reasonable default, then the process might issue the query and use `process-wait-with-timeout` to wait for the reply. If the process resumed because of timeout, then it continues as though the user had chosen the default.

More About Initial and Wait Functions

7.7.4 We intuitively think of creating a process to perform some specific job. Therefore, it is natural to think of each process as having one well-defined Initial Function that gets the job done. However, just because one job is completed, there is no particular reason to turn its associated process instance into garbage.

You can think of a process object as an execution container complete with an empty environment just waiting for an Initial Function to run. In fact, the system maintains a resource of reusable processes (see the functions `process-run-function` and `process-run-restartable-function`).

Therefore, although `make-process` can create a new process instance with a predefined Initial Function, we also need a way of modifying a process's Initial Function. Changing the Initial Function is called *presetting* the process.

Actually, the `:preset` operation on a process instance does more than just update the Initial Function and its arguments in the instance. The system assumes that if you have reason to change the Initial Function, then you are no longer interested in whatever the old Initial Function was doing, so it *resets* the process.

Resetting a running process causes it to gracefully stop whatever it was doing and start over at the beginning by re-executing its Initial Function. Of course, if you have preset the process's Initial Function in the meantime, then the process will start over by doing something entirely different. Resetting is the basic way of starting a new process.

A process's Wait Function is even more temporary than its Initial Function. A new Wait Function is reestablished each time a process voluntarily waits by calling the function `process-wait`. Later, you will see that the Scheduler freely modifies the Wait Function for its own purposes under certain circumstances.

Simple Processes

7.7.5 Even though the stack switching that occurs when one coroutine resumes another has been optimized, there is still overhead involved. If a process is going to execute for only a short time before waiting again, then the time it takes to switch stacks from Scheduler to process and then back to Scheduler is significant. Therefore, the system provides *simple processes* that are instances of the flavor sys:simple-process rather than of sys:process.

If the simple process's Wait Function returns true, then instead of doing a context switch to the process's stack group, the Scheduler simply calls the process's Initial Function directly. Therefore, a simple process runs inside the Scheduler as though it were just another function in the Scheduler (see paragraph 7.7.6, Errors Inside the Scheduler). Other implications of simple processes are:

- No Stack Group — A simple process has no stack group of its own, so it can have no process private variables. If it needs to remember information across calls, it must use global special variables.
- No Automatic Timeout — A simple process must be short. A runaway simple process inside the Scheduler is out of reach of the one-second sequence breaks triggered by the watchdog timer, and the system will hang.

Simple processes are often used to implement fast dispatching functions that do no real work themselves but just initialize something else.

For example, the network hardware has only limited buffer space available for incoming packets, so they must be moved out of the hardware buffer quickly. However, full software processing of a given packet may be quite lengthy. Therefore, a dispatcher function built as a simple process can be set to watching the network board. When a packet arrives, this processing copies the hardware buffer onto a software buffer queue in virtual memory, thereby freeing the network board. The *real* network process now watches this software queue.

Errors Inside the Scheduler

7.7.6 This is the paragraph you've heard so much about. We've previously seen several things that were evaluated inside the Scheduler. We've already seen that you are restricted to forms that reference only constants or global special variables. Now consider what happens if something the Scheduler is evaluating on your behalf should signal an error.

The Scheduler is low man on the totem pole. There is no one for it to fall back on. If it gets an error—either one of its own or one of yours—it goes into the infamous Cold Load Stream—the user interface of last resort. Your best bet in this situation is to choose the option to restart the Scheduler.

Getting thrown into the Cold Load Stream and then getting restarted annoys the Scheduler to no end. It reacts by bashing the process that caused the trouble so that the process cannot run again without outside intervention. All in all, it is considered very bad form for your software to cause an error inside the Scheduler.

Of course, there is a very small, but finite, possibility that a hardware error unrelated to any software activity (for example, a NuBus™ Timeout) might occur while the Scheduler is executing. In such a situation, the Scheduler retaliates by bashing an innocent process. In at least one known instance, the Scheduler imprudently bashed itself, necessitating a reboot.

Process Activity States

7.8 There are many instance variables associated with a process so it technically has many states. However, there are only three major classes of states:

- Inactive — For debugging purposes, the process is recorded on the sys:all-processes list. But the Scheduler cannot see it since the Scheduler polls only the sys:active-processes list. The **process-disable** function is the most common way of inactivating a process.
- Active — The process is recorded on both lists. Its Wait Function is evaluated each time through the polling loop. There may be many active processes.
- Executing — Of course, no more than one process can actually be executing in the hardware at one time, and there will be no process in this state in an idle system.

We've already seen how a process moves from the Active state to Executing: when the Wait Function of an active process evaluates to true, the process executes. Now, how does an Inactive process become Active?

Simple Is Not Good Enough

7.8.1 The simple expedient of having something like, say, an Active Flag that you can turn on or off as needed is not good enough. The problem is that in a system with multiple software processes *and* a human user, there can be conflicting, but independent, requirements. A simple Boolean flag is just not enough to record all possible conditions.

An Example of the Problem

7.8.1.1 Let's consider a simple physical analogy of a line printer that has only an Active Flag to turn it on or off. If a user somewhere on the network wants to print a file, he or she waits until the Active Flag is off, then turns the Flag on and transmits the file to be printed. Once the file is transmitted, the user turns the printer's Active Flag off. So far, so good.

Now let's look at another independent agent, the computer operator who occasionally notices that the paper in the printer is bunching up and about to jam. This operator needs to be able to turn the Active Flag off temporarily while he or she straightens the paper and then to turn the Flag back on.

The problem is that we have two agents trying to independently control the same printer—one to temporarily turn it on, one to temporarily turn it off—but neither agent has direct knowledge of the other. How can the network user tell if the printer's Active Flag is off because the printer is idle or because the operator is clearing a paper jam? When the operator finishes clearing a paper jam, how does he or she know if there is more to be printed or if the user has canceled the print request in the meantime? A single on/off Active Flag just isn't enough.

Our problem is this: there may be one or more independent reasons for running an Explorer process. At the same time, there may be one or more independent reasons for arresting a running process. Therefore, the `sys:process` flavor records a list of Run Reasons and a list of Arrest Reasons for each process. A process is placed on the `sys:active-processes` list (that is, ready to run whenever its Wait Function returns true when polled) if

- There is at least one item on the Run Reasons list.
- There are no items on the Arrest Reasons list.

Otherwise, the process is inactive, is *not* on the active processes list, and will not be polled.

The Example Revisited

7.8.1.2 Now, let's see how our user and operator would have controlled the printer using this Run/Arrest Reason list convention. Assume the global special variable `*printer*` holds the printer process. When the user wants to start the printer, he sends the following message

```
(send *printer* :run-reason 'print-a-file)
```

This message adds the symbol `print-a-file` to the Run Reasons list of the printer process. This symbol was chosen simply for its mnemonic value. The system does not care what kind of Lisp object you push onto the Run Reasons list as long as you push something. However, as we will see in a moment, there is sometimes an advantage in using a unique symbol for your Run Reason.

After the `:run-reason` operation has pushed `print-a-file` onto the Run Reasons list, it then checks to see if the conditions are now right for this to be an active process (that is, any Run Reason but no Arrest Reason). If so, then the process is moved to the `sys:active-processes` list if it is not already there.

When the user is finished printing the file, he sends the following message:

```
(send *printer* :revoke-run-reason 'print-a-file)
```

This message removes the symbol `print-a-file` from the process's Run Reasons list if it is present. Any other items on the list are left undisturbed. The reason for using a unique symbol as mentioned above is that you can assert and revoke your own reasons without stepping on other outstanding run and arrest reasons.

As before, after the `:revoke-run-reason` operation has deleted `print-a-file` from the Run Reasons list, it checks to see if this process needs to be removed from `sys:active-processes`.

The operator who sees a paper jam can send the message

```
(send *printer* :arrest-reason 'paper-jam)
```

This message, of course, pushes the symbol `paper-jam` onto the process's Arrest Reasons list and removes the process from the `sys:active-processes` list. When the paper jam has been cleared, the operator sends

```
(send *printer* :revoke-arrest-reason 'paper-jam)
```

After all of this build up for the justification and analysis of dual Run Reasons and Arrest Reasons lists, the truth is that most Explorer processes do not

need such elaborate control. The usual way to make a process active is to use the **process-enable** function, which first removes all Run and Arrest Reasons and then gives the process a single Run Reason of the keyword :enable.

All Things Considered

7.9 If you have been paying close attention (possible) and have been thinking about what you have read (doubtful, this is tedious stuff), then you will have realized that we have set the stage for some really confusing processing states. The following terms are used to describe the simultaneous states of a process's Run Reasons list, Arrest Reasons list, and Wait Function return value:

Stopped — All Run Reasons and all Arrest Reasons are revoked. Wait Function is unchanged, but it is not being polled by the Scheduler since no Run Reasons means it is not on the **sys:active-processes** list. The function **process-disable** is normally used to stop a process.

Active (or Runnable) — There is at least one Run Reason and no Arrest Reasons. The process is on the **sys:active-processes** list and being polled, so execution depends upon the Wait Function.

Executable — Same as Active above except that the Scheduler has forced the Wait Function to be #'true (that is, always to return true). The implication is that this process either has called **process-allow-schedule** or is a long-running active process that got time-sliced by a sequence break. It will run for its full allotted second each time through the polling loop.

Waiting Forever — Same as Active above except that the system has forced the Wait Function to be #'false. Even though this Wait Function can never return true, the process is still on the **sys:active-processes** list and being polled. It will not be able to execute again until someone Presets or Resets it.

Flushed — Same as Waiting Forever above except that the Wait Function is #'**sys:flushed-process** (which always returns false, but is distinguishable from #'false).

Arrested — Any process not on the **sys:active-processes** list because it has no Run Reason or has at least one Arrest Reason.

Bashing a Process

7.9.1 *Bashing* is a colorful term with the general meaning of inflicting your will on a process from the outside. A process can be bashed in several ways:

Reset — If the process is running, it stops the process gracefully by forcing it to throw out of all computations (that is, **unwind-protects** in the process are honored). The Run Reasons and Arrest Reasons lists are not modified, but the process's Wait Function is set to #'true. The next time the Scheduler polls this process, it will reapply its Initial Function to the Initial Function's arguments, and the process takes off.

Flush — Same as Reset except that the Wait Function is replaced with #'**sys:flushed-process**, which always returns false but is detectably different from the #'false function. Even if this process is officially active, it will wait forever until someone Presets or Resets it.

Kill — Similar to Reset except that once the process is stopped, it is removed from both the **sys:active-processes** list and the **sys:all-processes** list. If

the process instance originally came from a resource, it is returned. If no one is remembering this process, it becomes garbage.

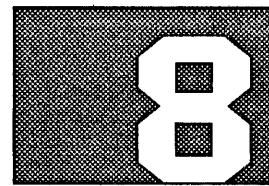
Scheduler

7.10 Most of what you need to know about the Scheduler has already been said in passing. The Scheduler is an Explorer coroutine whose job it is to poll the processes on the `sys:active-processes` list. For each process on that list, it applies that process's Wait Function to the Wait Function's arguments. This evaluation takes place in the Scheduler's stack group, and if the evaluation returns true, the process is run.

In an idle system, (that is, only the Scheduler is running) the Scheduler polls `sys:active-processes` once every 60th of a second. Of course, if it takes more than one 60th of a second to go around the loop, then the processes are polled less frequently.

A continuously running process is usually allowed to execute no more than one second at a time (this default can be changed) before a hardware watchdog timer forces a sequence break. These long-running processes were originally started when their Wait Function returned true to the Scheduler. Once interrupted by a sequence break, they need to be unconditionally run on each subsequent poll regardless of what their original Wait Function might later return.

Therefore, a process interrupted by a sequence break is given a Wait Function of #'true so that it will continue to execute each time through the polling loop until it calls some variant of `process-wait` and provides its own process-specific Wait Function.



HINTS TO MACRO WRITERS

Introduction

8.1 There are many useful techniques for writing macros. Over the years, Lisp programmers have discovered techniques that most programmers find useful, and they have identified pitfalls that must be avoided.

The most important thing to keep in mind as you learn to write macros: the first thing you should do is figure out what the macro form is supposed to expand into, and *only* then should you start to actually write the code of the macro. If you have a firm grasp of what the generated Lisp program is supposed to look like, you will find the macro much easier to write.

In general, any macro that can be written as an inline function should be written as one, not as a macro, for several reasons: inline functions are easier to write and to read; they can be passed as functional arguments (for example, you can pass them to `mapcar`); and there are some subtleties that can occur in macro definitions that you need not worry about with inline functions. A macro can be an inline function only if it has exactly the same semantics as a function, rather than those of a special form. The macros discussed in this section are not semantically like functions; they must be written as macros.

Name Conflicts

8.2 One of the most common errors in writing macros is best illustrated by example. Suppose you want to write `dolist` as a macro that expands into a `do`. The syntax of the `dolist` is as follows:

```
(dolist (element (append a b))
  (push element *big-list*)
  (foo element 3))
```

From this macro, you decide the expanded code should look like the following:

```
(do ((list (append a b) (cdr list))
      (element))
    ((null list))
    (setf element (car list))
    (push element *big-list*)
    (foo element 3))
```

Now you could start writing the macro that would generate this code and, in general, convert any `dolist` into a `do`. However, there is a problem with the above scheme for expanding the `dolist`. The above example's expansion works fine unless a user happened to code the following:

```
(dolist (list (append a b))
  (push list *big-list*)
  (foo list 3))
```

This is exactly like the form above, except that the programmer decided to name the looping variable `list` rather than `element`. The corresponding expansion would be as follows:

```
(do ((list (append a b) (cdr list))
      (list)
      ((null list))
      (setf list (car list)))
      (push list *big-list*)
      (foo list 3))
```

This form does not work at all. In fact, this is not even a valid form because it contains a `do` that uses the same variable in two different iteration clauses.

The following is another example that causes trouble:

```
(let ((list nil))
  (dolist (element (append a b))
    (push element list)
    (foo list 3)))
```

If you work out the expansion of this form, you see that there are two variables named `list` and that the programmer meant to refer to the outer one, but the generated code for the `push` actually uses the inner one.

The problem here is an accidental name conflict. This can happen in any macro that has to create a new variable. If that variable ever appears in a context in which user code might access it, then you have to worry that it might conflict with another name that the user has defined for his own program.

One way to avoid this problem is to choose a name that is very unlikely to be picked by the user, simply by choosing an unusual name in a package that only you will write code in. This strategy will probably work, but it is inelegant because there is no guarantee that the user will not coincidentally choose the same name. The way to reliably avoid name conflicts is to use an uninterned symbol as the variable in the generated code. The function `gensym` is useful for creating such symbols.

The following is the expansion of the original form, using an uninterned symbol created by `gensym`:

```
(do ((#:G4005 (append a b) (cdr #:G4005))
      (element))
      ((null #:G4005))
      (setq element (car #:G4005))
      (push element *big-list*)
      (foo element 3))
```

This is the right kind of form to expand into. Now that you understand how the expansion works, you can actually write the macro:

```
(defmacro dolist ((var form) . body)
  (let ((dummy (gensym)))
    `(do ((,dummy ,form (cdr ,dummy))
          (,var))
         ((null ,dummy))
         (setf ,var (car ,dummy))
         . ,body)))
```

Many system macros don't use `gensym` for the internal variables in their expansions. Instead, they use symbols whose print names begin and end with a dot. This convention provides meaningful names for these variables when you are looking at the generated code and when you are looking at the state of a computation in the error handler. These symbols are in the `SYS` package; as a result, a name conflict is possible only in code that uses variables in the `SYS` package. These name conflicts don't normally happen in user code, which resides in other packages.

Block Name Conflicts

8.3 A related problem occurs when you write a macro that expands into a `prog` or `do` (or anything equivalent) unexpectedly (unlike `dolist`, which is documented to be like `do`). Suppose you want to implement `error-restart` as a macro that expands into a `loop`, which becomes a `prog`. In this case, the following (contrived) Lisp program would not behave correctly:

```
(dolist (a list)
  (error-restart ((sys:abort error) "Return from FOO.")
    (cond ((> a 10) (return 5))
          ((> a 4) (ferror 'lose "You lose.")))))
```

The problem is that the `return` returns from the `error-restart` instead of the `dolist`.

The best solution is to make the expanded code use only explicit `block` with obscure block names or block names processed by `gensym` and never by a `prog` or `do`.

Macros such as `dolist` specifically should expand into an ordinary `do`, because you expect to be able to exit them with `return`.

Macros Expanding Into Many Forms

8.4 Sometimes a macro is supposed to do several different things when its expansion is evaluated. In other words, sometimes a macro should expand into several things, all of which should happen sequentially at run time (not macro-expand time). For example, since `defparameter` is implemented as a macro, it must do two things: declare the variable to be special and set the variable to its initial value. (Here a simplified `defparameter` is implemented; it does only these two things, without any options.) What should a `defparameter` form expand into? Ideally, the appearance of `(defparameter a (+ 4 b))` in a file should be treated as the same as the following two forms:

```
(proclaim '(special a))
(setf a (+ 4 b))
```

However, because of the way that macros work, they expand only into one form, not two. So a `defparameter` form must expand into one form that is exactly like having two forms in the file. The following is such a form:

```
(progn (proclaim '(special a))
      (setf a (+ 4 b)))
```

In interpreted Lisp, it is easy to see what happens here. This is a `progn` special form, so all its subforms are evaluated in turn. The `proclaim` form and the `setf` form are evaluated. The compiler recognizes `progn` specially and treats each argument of the `progn` form as if it were encountered at top level.

The following is the macro definition to produce one form to represent the two forms in the file:

```
(defmacro defparameter (variable init-form)
  `(progn (proclaim '(special ,variable))
          (setf ,variable ,init-form)))
```

Next is another example of a form that should expand into several things. A special form called `define-command` is implemented, which is intended to be used to define commands in an interactive user subsystem. For each command, there are two things provided by the `define-command` form: a function that executes the command and a character that should invoke the function in this subsystem. Suppose that, in this subsystem, commands are always functions of no arguments, and characters are used to index a vector called `dispatch-table` to find the function to use. A typical call to `define-command` would look like the following:

```
(define-command move-to-top #\control-<
  (do () ((at-the-top-p))
    (move-up-one)))

; Expands into:

(progn (setf (aref dispatch-table (char-int #\control-<))
             'move-to-top)
       (push 'move-to-top *command-name-list*)
       (defun move-to-top ()
         (do () ((at-the-top-p))
           (move-up-one))))
```

The `define-command` expands into three forms. The first one sets up the specified character to invoke this command. The second one puts the command name onto the list of all command names. The third one is the `defun` that actually defines the function itself. Note that the `setf` and `push` are executed at load time (when the file is loaded); the function, of course, is also defined at load time.

When you write a large system in Lisp, frequently you can make it much more convenient and clear by using macros to extend Lisp into a customized language for your application. In the above example, a small language extension has been created: a new special form that defines commands for the system. It lets you put documentation strings right next to the code that they document so that the two can be updated and maintained together. Because the Lisp environment allows load-time evaluation to build data structures, the documentation database and the list of commands can be constructed automatically.

Macros That Surround Code

8.5 A particular kind of macro very useful for many applications is one that you place *around* Lisp code so that this code is evaluated in a modified context. For a very simple example, you can define a macro called `with-output-in-base` that executes the forms within its body such that any output of numbers defaults to a specified base:

```
(defmacro with-output-in-base ((base-form) &body body)
  `(let ((*print-base* ,base-form))
    . ,body))
```

A typical use of this macro might look like the following:

```
(with-output-in-base (*default-base*)
  (print x)
  (print y))
```

The preceding form expands into the following:

```
(let ((*print-base* *default-base*))
  (print x)
  (print y))
```

This example is too trivial to be very useful; it is intended to demonstrate some stylistic issues. However, there are standard Explorer constructs that are similar to this macro, such as `with-open-file` and `with-input-from-string`. Most importantly, of course, you can define similar constructs for your applications.

One very powerful application of this technique was used in a system that manipulates and solves the Rubik's cube® puzzle. The system heavily uses a construct called `with-front-and-top`, which translates to the following: evaluate this code in a context in which one specified face of the cube is considered the front face, and another specified face is considered the top face.

When you write this sort of macro, keep in mind that you can make your macro much clearer to people who might read your program if you conform to a set of loose standards of syntactic style. By convention, the names of such constructs start with `with-`. This seems to be a clear way of expressing the concept that a context is being established; the meaning of the construct is: perform these operations *with* the following conditions true. Another convention is that any parameters to the construct are to appear in a list that is the first subform of the construct and that the rest of the elements are to make up a body of forms that are evaluated sequentially with the last one returned.

All of the examples cited above work this way. In the `with-output-in-base` example, one parameter (the base) appears as the first (and only) element of a list that is the first subform of the construct. The extra level of parentheses in the printed representation serves to separate the parameter forms from the body forms so that these two kinds of forms remain textually distinct; it also provides a convenient way to specify default parameters (a good example is the `with-input-from-string` construct, which takes two required and two optional parameters). Another technique is to use the `&body` keyword in the `defmacro` to tell the editor how to indent the elements of the body.

Also keep in mind that the construct can relinquish control either by the last form's returning or by a nonlocal exit (`go`, `return`, or `throw`). You should write the definition in such a way that everything is cleaned up appropriately no matter how control exits. The `with-output-in-base` example presents no problem because nonlocal exits release lambda bindings. However, in even slightly more complicated cases, an `unwind-protect` form is needed: the macro must expand into an `unwind-protect` that surrounds the body, with

cleanup forms that deactivate the context that the macro set up. For example:

```
(using-resource (window menu-resource) body...)
;;; Expands into
(let ((window nil)
      (unwind-protect
        (progn (setf window (allocate-resource 'menu-resource))
               body...)
        (when (not (null window))
          (deallocate-resource 'menu-resource window))))
```

In this way, the allocated resource item is deallocated whenever control leaves the **using-resource** macro.

Multiple and Out-of-Order Evaluation

8.6 In any macro, you should always pay attention to the problem of multiple or out-of-order evaluation of user subforms. Consider the **test** macro, which defines a special form with two subforms. The first is a reference, and the second is a form. The macro is defined first to create a cons whose car and cdr are both the value of the second subform and then to set the reference to be that cons. The following is a possible definition:

```
(defmacro test (reference form)
  `(setf ,reference (cons ,form ,form)))
```

Simple cases using this definition work correctly:

```
(test foo 3) ==> (setf foo (cons 3 3))
```

But a more complex example, in which the subform has side effects, can produce surprising results:

```
(test foo (setf x (1+ x)))
==> (setf foo (cons (setf x (1+ x))
                     (setf x (1+ x))))
```

The resulting code evaluates the **setf** form twice, and so **x** is increased by 2 instead of by 1. A better definition of **test** that avoids this problem is the following:

```
(defmacro test (reference form)
  (let ((value (gensym)))
    `(let ((,value ,form))
       (setf ,reference (cons ,value ,value)))))
```

With this definition, the expansion works as follows:

```
(test foo (setf x (1+ x))) ==>
(let ((#:G4005 (setf x (1+ x))))
  (setf foo (cons #:G4005 #:G4005)))
```

In general, when you define a new construct that contains one or more argument forms, you must be careful that the expansion evaluates the argument forms the proper number of times and in the proper order. There is nothing fundamentally wrong with multiple or out-of-order evaluation if that is really what you want and if it is what you document your macro to do. But if this happens unexpectedly, it can make invocations fail to work as they appear they should.

The **once-only** macro can be used to avoid multiple evaluation. It is most easily explained by example. You write `test` using **once-only** as follows:

```
(defmacro test (reference form)
  (once-only (form)
    `(setf ,reference (cons ,form ,form))))
```

This form defines `test` in such a way that the `form` is only evaluated once, and any references to `form` inside the macro body refer to that value. The **once-only** macro automatically introduces a lambda binding of a generated symbol to hold the value of the `form`. Actually, this macro is more clever than that; it avoids introducing the lambda binding for forms whose evaluation is trivial and can be repeated without harm or cost, such as numbers, symbols, and quoted structure. This lambda binding is merely an optimization that helps produce more efficient code.

The **once-only** macro makes it easier to follow the principle but does not completely or automatically solve the problems of multiple and out-of-order evaluation. It is merely a tool that can solve some of the problems some of the time; it is not a panacea.

Nesting Macros

8.7 A useful technique for building language extensions is to define programming constructs that employ two macros, one of which is used inside the body of the other. The following is a simple example with two macros. The outer one is called `with-collection`, and the inner one is called `collect`. The `collect` macro takes one subform, which it evaluates; `with-collection` only has a body, whose forms it evaluates sequentially. The `with-collection` macro returns a list of all the values that were given to `collect` during the evaluation of the body of `with-collection`. For example:

```
(with-collection (dotimes (i 5) (collect i)))
=> (1 2 3 4 5)
```

Remembering the first step in designing macros, you next determine what the expansion looks like. The following shows how the above example can expand:

```
(let ((#:G4005 nil))
  (dotimes (i 5)
    (push i #:G4005))
  (nreverse #:G4005))
```

Now, you write the definition of `with-collection`, which is fairly easy:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       ,@body
       (nreverse ,var))))
```

Writing `collect`, however, is more difficult:

```
(defmacro collect (argument) `(push ,argument ,var))
```

Note that `collect` uses the variable `var` freely. It depends on the binding that takes place in the body of `with-collection` in order to get access to the value of `var`. Unfortunately, that binding takes place when `with-collection` is expanded; the expander function of `with-collection` binds `var`, and it is unbound when the expander function is executed. By the time the `collect` form is expanded, `var` has long since been unbound. The macro definitions

above do not work. Somehow the expander function of `with-collection` must communicate with the expander function of `collect` to pass over the generated symbol.

The only way for `with-collection` to convey information to the expander function of `collect` is for it to expand into something that passes that information. You can define a special variable (called `*collect-variable*`) and have `with-collection` expand into a form that binds this variable to the name of the variable that `collect` should use:

```
(defvar *collect-variable*)

(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((*collect-variable* ',var))
       (,var nil)
       ,@body
       (nreverse ,var))))
(defmacro collect (argument)
  `,(push ,argument ,*collect-variable*))
```

But this is still incorrect: it works in the evaluator but not in the compiler. To understand this problem, consider how it works in the evaluator. The evaluator first sees the `with-collection` form and calls the expander function to expand it. The expander function creates the expansion and returns to the evaluator, which then evaluates the expansion. The expansion includes in it a `let` form to bind `*collect-variable*` to the generated symbol. When the evaluator sees this `let` form during the evaluation of the expansion of the `with-collection` form, it sets up the binding and recursively evaluates the body of the `let`. Now, during the evaluation of the body of the `let`, the special variable is bound, and if the expander function of `collect` is run, it is able to see the value of `*collection-variable*` and to incorporate the generated symbol into its own expansion.

When compiling, however, the `let` binding for `*collect-variable*` causes that variable to be bound only when the compiled code is executed. It does not cause `*collect-variable*` to be bound at any time during compilation, including the time when `collect` must be expanded.

You can fix your definitions by using `compiler-let` instead of `let`. The `compiler-let` special form exists specifically to do the sort of thing you are trying to do in this case. The `compiler-let` special form is identical to `let` as far as the interpreter is concerned, so the macro continues to work in the interpreter with this change. When the compiler encounters a `compiler-let`, however, it actually performs the bindings that the `compiler-let` specifies and proceeds to compile the body of the `compiler-let` with all of those bindings in effect. In other words, it acts as the interpreter would.

The following is the correct way to write these macros in this fashion:

```
(defvar *collect-variable*)

(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       (compiler-let ((*collect-variable* ',var))
                     ,body)
       (nreverse ,var))))
(defmacro collect (argument)
  `,(push ,argument ,*collect-variable*))
```

Another correct way to write these macros is using **macrolet**:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(macrolet ((collect (argument)
                         `,(push ,argument ',var)))
        (let ((,var nil))
          ,@body
          (nreverse ,var))))
```

In this example, **with-collection** expands into code that defines **collect** specially to know about which variable to collect into. The `,`` form causes the value of **var** to be substituted when the outer backquote, the one around the **macrolet**, is executed. The **argument**, however, is substituted when the inner backquote is executed, which happens when **collect** is expanded.

This technique has the interesting consequence that **collect** is defined only within the body of a **with-collection**. It simply would not be recognized elsewhere, or it could have another definition, for some other purpose, globally. This has both advantages and disadvantages; for example, it might be preferable, when using **collect** outside of any **with-collection**, to give a specific error message rather than only a warning that an undefined function named **collect** was called.

Functions Used During Expansion

8.8 The technique of defining functions to be used during macro expansion deserves explicit mention here. Actually, a macro expansion function is a Lisp program like any other Lisp program, and it can benefit in all the usual ways by being broken down into a collection of functions that perform various parts of its work. Usually, macro expansion functions are relatively simple Lisp programs that take things apart and put them together slightly differently, but some macros are quite complex and perform a great deal of work. Several Explorer features, including flavors, the **loop** macro, and **defstruct**, are implemented using very complex macros, which, like any complex, well-written Lisp program, are broken down into modular functions. You should keep this in mind if you ever invent an advanced language extension or ever find yourself writing a five-page expansion function.

Note especially that any functions used by macro expansion functions must be available at compile time. You can make a function available at compile time by surrounding its defining form with an `(eval-when (compile load eval) ...)`. Doing this means that at compile time, the definition of the function is interpreted, not compiled, and hence runs more slowly.

Another approach is to separate macro definitions and the functions they call during expansion into a separate file, often called a **defs** (definitions) file. This file defines all the macros as well as all functions that the macros call. It can be separately compiled and loaded before compiling the main part of the program that uses the macros. The **make-system** facility (described in the *Explorer Lisp Reference* manual) helps keep these various files distinct, compiling and loading things in the proper order.

You can use the **mexp** function to help you debug macros; it prints the expansion form of an evaluated macro.

Suppose that you type the following:

```
(mexp)
```

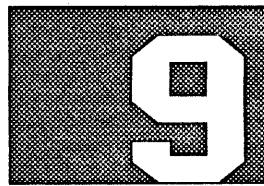
Next, you type the following:

```
(rest (first x))
```

Then, **mexp** prints the following:

```
(cdr (first x)) →  
(cdr (car x)))
```

You then press ABORT to exit **mexp**.



PREPARING A PROGRAM PRODUCT FOR DELIVERY

Introduction

9.1 Designing, implementing, and debugging a new program product is trouble enough in itself. Preparing that product for delivery to customers in the field is something else again. The goals for delivery to a customer's site include:

- The product can be installed by the customer with no previous knowledge of the product or its internal structure beyond prompts and instructions he receives during installation.
- The customer should be able to install all products from all vendors by the same basic procedure. Differences in installation should reflect a unique product requirement—not gratuitous inconsistencies.
- A newly installed product should be smoothly integrated with that portion of the system it interfaces with while remaining isolated from unrelated portions of the system and other products.

Each item represents one or more details that you must consider in the structuring of your product and in its preparation for delivery to your customer.

Checklist

9.2 Specifically, to accomplish the aforementioned details:

- Each product should be in its own *package*, and each of the product's Lisp files should have a file attribute list identifying its package.
- Independent of how the product was developed, the product should have a **defsystem** capable of loading the product at the customer's site.
- Each product should have
 - a translations file defining the mapping of the logical pathnames used by the product to physical pathnames at the customer's site
 - a system file identifying the location of the product's **defsystem** file
- Each product should have its own *logical host*, and all pathnames used to build or run the product should be logical pathnames.
- Each product that must manipulate the file types at run time should specify *canonical* file types.
- Each product should have a top-level function with sufficient bindings to ensure that it does not interfere with other products.

These elements are described further in the following paragraphs.

Choose a Package

9.3 Each product should be in its own package to avoid symbol-name conflicts with other products and with the Explorer system code. You should not put your code in any system-provided package such as LISP, SYSTEM, or USER or in the package of another product that you don't have full knowledge of.

That is, whatever package you use must contain the same thing on your development system as it will contain on the customer's system. Otherwise, you won't see the conflicts your user will see. If you create a package dedicated to your product, there will be no problem. If you use some other package that you don't control, then you have no idea what may be in that package at the customer's site when your product arrives to be installed.

Standard System Packages

9.3.1 On the Explorer, all standard Common Lisp symbols are in the LISP package. The symbols for all TI extensions are in the TICL package. The USER package uses both the LISP and the TICL packages, so it has access to everything commonly used. The combination of the LISP and TICL packages is roughly equivalent to the GLOBAL package of Zetalisp but without the obsolete symbols and without the symbols that conflict with Common Lisp symbols.

What does all this mean? If you create your package and define it to use the LISP package, then your code will automatically be restricted to Common Lisp standard functions plus whatever functions you write. If you wish to use any of the Explorer extensions, you must explicitly prefix those function symbols with TICL:. When you have completed coding your product, if you have used no package prefixes anywhere, then you have taken a big step towards Common Lisp portability.

On the other hand, if you define your package to use the LISP and TICL packages, your code will have access to all Common Lisp standard functions as well as all TI extension functions *without* having to worry about package prefixes in your code (but probably at the cost of Common Lisp portability).

What's An Extension?

9.3.2 The Common Lisp standard is in two parts: a standard language syntax and a standard function/variable library. These standard functions (and other standard symbols) are in the LISP package, which itself is part of the standard. Normally, the symbols in the LISP package are the *only* symbols that can be used everywhere without a package prefix.

In addition to this standard function library, the Explorer comes with an even larger library of functions written in Common Lisp syntax. These functions are things that you could write yourself if you had a *lot* of time. Most of these TI-supplied, Common Lisp-conforming functions are in their own special purpose packages and normally have to be written with their package prefixes (that is, FS for the file system, W for the window system, EH for the error handler, and so on). These functions are no different from any other user-supplied functions.

There is another group of TI-supplied functions in the TICL package that are as widely used on the Explorer as the Common Lisp standard functions. These functions are so frequently used that it is a decided annoyance to always have to write the TICL prefix. Therefore, system-supplied packages (including the USER package of the Common Lisp standard) all use the TICL package as well as the standard LISP package. So, it is possible to reference symbols in the TICL package without a package prefix, thereby giving the outward appearance that TI has *extended* the Common Lisp standard function library in the LISP package.

Exporting Symbols

9.3.3 Of all the symbols used in your product, you should decide which will need to be commonly called by users and programs outside your package. These symbols should be listed in an **export** statement. As a rule of thumb, if you have documented a symbol to your users, it should be exported.

According to the Common Lisp standard, an outsider can access a symbol that has been exported from its package as *package:symbol* (note the single colon). If a symbol has not been exported, the standard says it must be accessed as *package::symbol* (note the double colon). The Explorer system is sophisticated enough to allow any symbol to be accessed by either notation—but other Common Lisp implementations might not be so thoughtful.

For some products, only the top-level function that the user calls to start the product will need to be exported. At the other extreme, a product such as a math library should have all its advertised function names exported.

What is the practical benefit of exporting a symbol? If another package uses your package, only your officially exported symbols will be inherited by that new package. The users of this new package can now access your exported symbols without bothering with cumbersome package prefixes. At the same time, all of the unexported internal symbols you happened to use to implement your exported functions are still safely within your package—they will not accidentally interfere with similar symbols in the new package. If that user wants to deliberately reach inside your package, Common Lisp says to use the double colon notation, while the Explorer accepts either one or two colons.

Placement of the Package Declaration

9.3.4 Because the existence of your package is basic to the compilation and loading of all your product's code, the **make-package**, **in-package** (the Common Lisp standard), or **defpackage** (the TI extension) form should probably be in the first file loaded. Because the first file loaded whenever a system is compiled or loaded is the DEFSYSTEM file, one likely place for your package definition is at the beginning of this file (see the example DEFSYSTEM file below).

Although it may seem like a frill, you should design your product so that it can be loaded on top of itself. The most common reasons why a product gets loaded on top of itself are

- A glitch in the first installation attempt (real or imagined) causes the customer to try to reinstall it.
- The customer upgrades an existing product to a newer version.

The **make-package** function by itself will thwart reinstallation attempts by signaling an error if asked to make a package that already exists. The **defpackage** macro is more accommodating. It will simply modify an existing package as necessary to make it conform to the **defpackage** macro's argument. If you wish to stick to the Common Lisp standard, simply use **in-package** instead:

```
(in-package "XYZ" . . .)
```

Place the **export** statement anywhere after the package has been created.

A Naming Convention

9.3.5 Time out for a small, but useful, naming convention. In theory, Lisp lets you name most things anything you want no matter how long, how fanciful, or how inappropriate. However, we can use the Explorer's TCP/IP program product as an example of a practical (though unimaginative) naming convention—everything is named **IP**:

- The product's system name (as defined by its **defsystem**) is **IP**.
- The product's package name is **IP**.
- The product's logical host name is **IP**.
- The logical directory holding the TCP/IP files is named **IP** (that is, the logical pathname of the **IP** directory is **IP:IP:**).
- When the customer is installing the IP Distribution Tape, the prompts he sees always mention the **IP** product.

While all of this naming consistency is of little interest to the system itself, it is a blessing to the programmer because just knowing the product's common name tells the programmer where to find a lot of information about that product. In following examples, we will use **xyz** as our product's common name.

Problems with Package Names

9.3.6 One small caveat about package names: Don't put spaces in them even though spaces technically are allowed. Certain places in the system software (such as the patch file generation logic) do not expect such freedom of choice and will output package names *without* the necessary escape characters.

For example, if you were to try to use the Zmacs editor to write a patch for a function in the "PRODUCT ABC" package (note the embedded blank), then Zmacs will construct a patch file containing the following:

```
PRODUCT ABC:  
(defun . . .)
```

which Zmacs thinks is telling the compiler that everything in the **defun** form is in package **PRODUCT ABC**. But because the space between **PRODUCT** and **ABC** is *not* escaped with a backslash (that is, **PRODUCT\ ABC**), the compiler will actually see an undefined top-level symbol **PRODUCT** (which it objects to) followed by a function definition in the package **ABC** (which it hasn't heard of either). If you had named your package **PRODUCT-ABC**, all would have been well. (Of course, if you follow the naming convention suggested above, you would have product **XYZ**'s package to be simply **xyz**.)

A second caveat: don't use lowercase letters in package names even though they are allowed. Why? Because Common Lisp package names are case sensitive. Actually, a lot of things are case sensitive, but no one ever notices because the Lisp Reader automatically uppercases all symbols before the case-sensitive code ever gets to see them. For example, the code fragment

```
(in-package "xyz")
(setf xyz:count 0)
```

would fail when the Lisp Reader objects that the package `xyz` does not exist, which is true. The package `xyz` is the one you defined; the package `xyz` does not exist. To make the `setf` work, you would have had to code the symbol `xyz:count` as `|xyz|:count` to keep the Lisp Reader from uppercasing the package prefix and then complaining that the uppercase package doesn't exist. All this is more trouble than it is worth. Uppercase your package names.

A Catch 22

9.3.7 If the purpose of packages is to avoid symbol name conflicts among independent products, how do you guarantee that your package name is unique? You don't. Basically, you choose a package name that doesn't conflict with the four dozen or so packages in the Explorer system software itself and that reflects your product closely enough that other vendors are unlikely to use it. So far, this problem has never come up.

**File Attribute
List**

9.4 Every Lisp source file should begin with a *file attribute list*. A file attribute list is defined as the text between the pair of `-*-` delimiters in the first nonblank line of the file. While a file attribute list might visually wrap onto more than one line when viewed in the editor, no RETURN characters are used before the closing `-*-` delimiter. For example, a basic file attribute list might look like this:

```
; ; ; -*- Mode:Common-Lisp; Package:XYZ -*-
```

This file attribute list informs the editor, the compiler, the loader, and anyone else who cares to ask that this file is written in Common Lisp syntax and all symbols are in the `XYZ` package unless explicitly overridden by a package prefix. If your code were actually in Zetalisp syntax, your `Mode` attribute would have looked like `Mode:Zetalisp`; instead.

You should *always* include the package attribute for each Lisp source file. Otherwise, the code will be loaded into whatever package happens to be in the global special variable `*package*` at the time the file is loaded. A product that does not have package attributes in its files will exhibit a certain disconcerting randomness in execution.

For example, whenever you bring one of these files into a Zmacs buffer for editing, the buffer will assume the same package as the previous Zmacs buffer had. Therefore, each time you edit one of these files, it could appear to be in a different package.

One last point: Notice the leading semicolons in the file attribute list example above. As far as the Lisp Reader is concerned, that line is a comment and is ignored. The contents of a file attribute list are read by system utilities independent of language processors such as the Lisp compiler. Therefore, file attribute lists can be placed in any sort of file that includes the notion of a

comment line. For example, a C source file might have the following file attribute list:

```
/* -*- Mode:C -*- */
```

which would be invisible to the C compiler but would tell the Zmacs editor to switch to its C editing mode when editing this file.

The Common Lisp Standard

9.4.1 Common Lisp does not have a convention that provides the type of general purpose information to system utilities that the file attribute list does. But the Common Lisp standard can still peacefully coexist alongside the Explorer conventions if you follow certain guidelines:

- All the code in one file has the same default package.
- The default package is recorded both in the file's file attribute list (for Explorer utilities) and in an **in-package** form at the beginning of the file (for Common Lisp systems).

If you follow these guidelines, you can have Common Lisp-conforming code and still get the full benefit of the Explorer environment. If, however, you were to try to mix several default packages in one file by inserting **in-package** forms in several places, then several situations might arise:

- Functions that actually process the Lisp forms in the file from start to finish (for example, **compile-file**, **load**, and the Zmacs Compile Buffer command) should see each package change correctly.
- Functions that process only fragments of Lisp code (for example, the Zmacs Compile Region and Add Patch commands) see only the package in the file attribute list.
- Functions that examine or manipulate the Lisp code without actually processing the forms (for example, most Zmacs commands) will probably see only the file attribute list.

Logical Pathnames

9.5 One of the most persistent problems in moving a product from your system to your customer's system is what to do about pathnames. In the simplest case, you have to worry only about finding a place for your product's object files on the customer's system just long enough to get your product loaded. In more complicated cases, a product must also access permanent auxiliary files (for example, help files, configuration files, and so on) during run time.

Common Lisp defines a pathname to have six components: host, device, directory (a general term including subdirectories), name, type, and version. When product files are moved from your site to the customer's site:

- The host and device names obviously must change, and the customer's host is not necessarily an Explorer.
- Whereas you may have used a top-level directory on your system, the customer's file server may have other naming conventions (for example, system administrators often frown on top-level directories).

- You are more or less free to pick the filenames as long as they are acceptable to your customer's file server OS (for example, some don't like hyphens, many have a length limit, and so on).
- You are not really free to choose arbitrary file type names because these names are used to *classify* the contents of a file (that is, if you give a Lisp source file a type that identifies it as a C object file, funny things might happen).
- You cannot count on the customer's file server supporting version numbers, so your code must assume that it is always working with the *newest* version of a file (which avoids the question of whether other versions are being remembered or not).

Your problems of choosing a host, device, and directory name are solved with logical pathnames explained here. The problem of choosing the correct file type is handled by canonical types which are explained later.

Pathname Translations

9.5.1 The basic idea is simple. You write all your code in terms of logical pathnames. Then at each customer's site, an `fs:set-logical-pathname-host` form is executed that maps the logical host and directory names you used to the customer's physical host and directory. By convention, a single `fs:set-logical-pathname-host` form that defines logical host *foo* is the sole content of the file `SYS:SITE;foo.TRANSLATIONS` (except for comments, of course). Simply loading this "translations" file causes the logical host to be defined. In practice, a `load` form loading this translations file is usually placed in the "system" file (explained below).

Why should there be only one `fs:set-logical-pathname-host` form per translations file? Some pieces of system code know that if they should need the translations for logical host *foo*, they can look for the file `SYS:SITE;foo.TRANSLATIONS` and load it if it exists. If you had also placed the translations for, say, logical host *bar* in this file, then they would effectively be hidden from the system software.

Customer-Specific Translations

9.5.2 Of course the problem now becomes: How do you get your customer to modify your translations file to fit his needs? Actually, the Explorer's built-in Load Distribution Tape utility can help here. The restore phase of a Distribution Tape typically starts with a call to a special utility function that presents your generic translations file to the customer and offers him the following alternatives:

- Accept your generic translations file as is.

NOTE: These generic files automatically detect what physical host the customer is using as the SYS-host and use that host as their physical host.

- Have the installation software automatically rewrite your generic translations file to use the same directories on a different physical host at the customer's site.

- Have the installation software automatically rewrite your file to place your product directory under some physical subdirectory specified by the customer.
- As a last resort, place your translations file in a Zmacs buffer and let the customer do as he wishes.

Whatever the customer chooses, the Load Distribution Tape utility writes the final version to the SYS:SITE; directory. The customized translations file is then loaded and all the rest of the installation is done in terms of those translations. In a properly designed product, the translation files are the *only* place that physical pathnames exist, and therefore, only the translations files must be changed as a product is moved from system to system.

For example, our XYZ example product would store its files on a logical host named XYZ. The generic translations for this XYZ logical host would be kept in the SYS:SITE;XYZ.TRANSLATIONS file. The contents of this file would look like the following:

```
; ;-- Mode:Common-Lisp; Package:USER -*-  
(fs:set-logical-pathname-host "XYZ"  
  :PHYSICAL-HOST (send (sys:parse-host "SYS") :name)  
  :TRANSLATIONS `(("XYZ" "XYZ;"")  
    ("PATCH" "XYZ-PATCH;")))
```

These translations are generic in the sense that their physical host will default to whatever the customer is using for his SYS-host. Furthermore, the translations assume top-level physical directories using Explorer syntax—a requirement for the Load Distribution Tape utility to be able to automatically manipulate the translations for the customer.

Patchable Products

9.5.3 The preceding example translation file also implies that the XYZ product is patchable. There are a number of implications to a patchable system on the Explorer:

- The product must be represented by a **defsystem** with a **:patchable** declaration.
- The product must have been compiled by **make-system** (that is, you can't make a system patchable after it has been compiled).
- You are able to write *patches* for products in the field that, for the most part, replace whole functions (although environment changes are allowed, too).
- Patches can be sent to customers either as periodic updates or as needed. These patches can be installed using the same Load Distribution Tape utility that was used to install the original product.
- The **print-herald** utility function can display a listing of all products in the system including their release number plus their *patch* level (for example, xyz 2.4 would mean that patch number four has been applied to release two of xyz).

The details of how to use **defsystems** and how to make a product patchable are explained below.

The patch directory mentioned in the translations file example would be used to hold the patch files themselves along with a brief description of what each patch is intended to accomplish. This patch directory was deliberately placed in a separate directory from the main XYZ source and object files.

Multiple Logical Directories

9.5.4 The rationale for a separate logical directory for the patch directory is that many customers with limited disk space move source and object files offline as soon as a product is installed. Nevertheless, the patch directory must be present if the customer intends to load new patches. By separating the patch directory, the customer can usually afford to leave the patch directory online even if the main directory has been deleted to save space.

This same line of argument applies to any auxiliary files your product may need at run time (for example, help files). Files that must be kept online at run time should always be in a separate logical directory from the main source and object directories. In this way, a customer strapped for disk space can easily pare the product's online files down to the essentials. The release notes for your product should indicate which directories must be kept online and which can be deleted after installation.

For more information on logical pathnames and on Explorer pathnames in general (including canonical types), see Section 6, Pathnames, in this manual.

Creating and Manipulating Internal Pathnames

9.6 One of the side effects of the Explorer's transparent file I/O is that you can never be sure of what kind of host your customer is using. Therefore, you must take particular care to avoid OS-specific pathname notation in your code. There are two main rules of thumb:

- Never hardcode a namestring in your product. Instead, specify the individual pathname components to **make-pathname** and let it create the pathname for you.
- Never hardcode a file type—even to **make-pathname**—if you can help it. Instead, specify a canonical file type if one is available.

The use of **make-pathname** to assemble a host-dependent pathname from host-independent components is a standard feature of Common Lisp. Canonical types, however, is an Explorer extension.

Common Lisp pathnames with their six components don't just represent a six-level hierarchy. Rather, different components represent different kinds of information. For example,

- The host and device components provide physical identification of a machine on the network and a disk (or file system) on that machine.
- The directory and name components represent the programmer's unique multilevel name for the file.
- The type component provides file content identification to system utilities.
- The version component provides a simple change identification.

The type component of a pathname represents a simplistic communication interface between the programmer and the system. To some extent, the programmer must constrain himself to type names the system understands if he wants the system's defaults to work for him.

The problem is, of course, that all operating systems don't agree on which file types means what. A text file may be designated as `TEXT` or as `txt`. A Lisp source file might be `LISP`, `LSP`, or `l` depending on the OS. The Common Lisp `make-pathname` function allows you to specify pathname components independent of OS namestring syntax; but what do you do about an OS-specific component?

On the Explorer, a canonical type is named by a keyword. Each canonical type has a set of OS-specific mappings associated with it. When you specify a canonical type keyword as the `:type` argument to `make-pathname`, a TI extension chooses a type component that matches the host of the pathname being formed.

For example, the Explorer Lisp compiler writes out an object file with the same pathname as the Lisp source file except that the file type represents a Lisp object file. Assuming that the compiler has just compiled the Lisp source file represented by `input-pathname`, the compiler might generate the output filename like this:

```
(make-pathname :type      :XLD
               :version   (pathname-version input-pathname)
               :defaults  input-pathname)
```

where `:XLD` is the canonical type for Lisp object files on Explorer Release 3. Although the point was simply to replace the type component of `input-pathname`, the new version had to be explicitly specified as being the same as the old version because `make-pathname` is defined to default its version to `:newest`.

defsystem and make-system

9.7 In theory, a defsystem specifies the order in which files must be compiled and loaded

- To transform the source files on disk into object files on disk at the developer's site
- To load object files on disk into memory at the customer's site

In practice, you are free to create the product as you wish, but you should still make an effort to give the customer a common interface across all products. Therefore, a product to be delivered on an Explorer should have at least a dummy `defsystem` that is capable of loading the product at the customer's site.

Ideally, a customer should never have to do anything more than enter the form

```
(make-system 'xyz :noconfirm)
```

to a Lisp Listener to get the XYZ product loaded. This form tells `make-system` to load the product with a `defsystem` named `xyz`. If the XYZ `defsystem` has already been loaded, then `make-system` uses the information in the `defsystem` to load the product.

The System File

9.7.1 If the **defsystem** for XYZ has not been seen yet, then **make-system** will load the file **SYS:SITE;XYZ.SYSTEM#>** if it exists (this is the *system* file mentioned above). Once this file is loaded, **make-system** is now supposed to know either what the **defsystem** is or at least where to find it. In the case of our XYZ product, this file would look something like the following:

```
; ;-- Mode:Common-Lisp; Package:USER -*- (load  
"SYS:SITE;XYZ.TRANSLATIONS") (sys:set-system-source-file 'xyz  
"XYZ:XYZ;DEFSYSTEM")
```

Remember, this system file is normally the first product-specific file loaded. Therefore, starting this file with a load of the XYZ translations file means that everything that follows can be written in terms of logical pathnames. The second form, **sys:set-system-source-file**, effectively tells **make-system**, "If you ever need the **defsystem** for xyz, it can be found in the XYZ:XYZ;DEF-SYSTEM file."

Another convention that we are trying to foster on the Explorer is that a product's system file is the primary point of reference for a customer wanting to install or use the product. Therefore, you might add comments to the file to cover such things as

- Official title and a one-line product description
- Copyright, trademark, and proprietary notices
- Hotline numbers, mailing address, and bug reporting procedure
- How to start the system (SYSTEM key? System menu? Lisp function?)
- Where to find out about the product (that is, name of the user's manual)
- And any other information

The DEFSYSTEM File

9.8 In the preceding example, the **sys:set-system-source-file** form in the **SYS:SITE;XYZ.SYSTEM** file told **make-system** where to look for the **defsystem** file. That file might look something like the following:

```
; ;-- Mode:Common-Lisp; Package:USER -*-  
(when (null (find-package "XYZ")) ;define the product package  
  (make-package "XYZ")); (once and only once)  
  (export '( . . . )) ;export "public" symbols  
  (defsystem xyz ;define the XYZ system  
    (:name      "XYZ Analysis Program")  
    (:short-name "XYZ")  
    (:readfile  "XYZ:XYZ;main.lisp"))
```

Actually, this is an example of a trivial **defsystem** that loads a single Lisp file, which (presumably) then loads the rest of the system. This sort of **defsystem** will interface the Explorer's **make-system** convention with the sort of products that instruct the customer to "load file X to load the product." Of course, if you have built your product using a **defsystem**, then you would put

that **defsystem** in this file. But in a pinch, the following simplified **defsystem** should be enough to allow the form

```
(make-system 'xyz :noconfirm)
```

to load the product. All your customer has to remember is your product's system name. The customer is not bothered with remembering which file you put your load-it-all forms in.

If you build your product using a **defsystem** with a **:patchable** declaration, then—at the end of the customer's **make-system** to install your product—the string from the **:name** declaration will appear in the **print-herald** display along with its version number and patch level. This publicly displayed version information is particularly handy if the customer calls you about a problem. Nonpatchable systems are not shown by **print-herald**. Following the conventions we've suggested here, the **:patchable** declaration in the **defsystem** should look like the following:

```
(:patchable "XYZ:PATCH;" PATCH)
```

In the normal course of events, it is possible for your DEF SYSTEM file to be loaded (and reloaded) even though the system is not being recompiled or reloaded. Therefore, you should not put your **defsystem** in a general-purpose source file with a bunch of other code. Try to limit the DEF SYSTEM file to what you see in the example above. Otherwise, the act of loading your **defsystem** will inadvertently load part of your system before its time.

For more information on **defsystem** and **make-system**, see Section 4, **defsystem** and **make-system**. For more information about patching, see Section 5, Loading and Patching.

Inter- and Intra- Product Independence

9.9 There is a crucial distinction between multiuser operating systems (such as workstations and mainframes have) and single user multiprocessing operating systems (such as an Explorer system has). A mainframe more or less assumes that each process is working for a different user, while the Explorer system assumes that all processes are working for the benefit of the same user. As a consequence, mainframe operating systems pride themselves on isolating individual processes, while the Explorer prides itself in the ease of sharing data among processes.

On the Explorer system, all global special variables (that is, variables defined by **defvar** or **defparameter**) are shared by all processes. Variables declared within a process and bindings of global special variables within that process are private to that process. The problem you face in designing your product is making certain that you share what you want to share and keep everything else to yourself. If the user is running one copy of your product, does it keep running if the user starts running a second copy? You must look at this problem from two angles:

- You must make certain your product does not interfere with any other product or the Explorer system code.
- You must make certain that multiple copies of your product (if such a thing is reasonable) do not interfere with each other.

For example, the global special variable ***print-base*** controls the default printing radix for integers (usually 10). If you wanted some numbers printed in hex, you might do the following:

```
(setf *print-base* 16.)  
. . . code to print the integers . . .  
(setf *print-base* 10.)
```

which is *not* good. Because you have modified the global value of ***print-base***, not only will your code print in hex, but *everyone* will print in hex. As a rule of thumb, *never* set a global variable, *bind* it. For example,

```
(let ((*print-base* 16.))  
. . . code to print the integers . . .  
)
```

causes your code that prints the integers and all code that it calls directly or indirectly to see a ***print-base*** of 16 while the rest of the world sees the original value...whatever it was.

Your Top-Level Form

9.9.1 If your product is something like a set of math utility functions, then you have few concerns here. Your functions will simply run in whatever environment they were called. Your precautions can be limited to not setting global variables.

If, on the other hand, your customer invokes your product by entering some top-level function to a Lisp Listener, then you will need to make sure your product runs in its own little world. For example, let's assume that your product has defined two global special variables, ***foo*** and ***bar***, and that each running copy of your product requires its own private copy of these variables. Then your top-level function should look something like this:

```
(defun xyz ( . . . )  
(let ((*foo* *foo*)  
      (*bar* *bar*))  
    . . . main body of the top-level function . . .  
)
```

Binding a global special variable to itself effectively makes a working copy of it. The initial values of ***foo*** and ***bar*** as seen by the body of the function will be the same as their values in the environment that called **xyz**. However, any attempt to set one of these variables during the execution of your product will modify only the values seen by this particular instance of the product. No changes will be seen by any other copies of the product that happen to be executing.

Furthermore, if you suspect that something in your code is setting a global special variable such as ***print-array*** rather than binding it (despite all admonitions to the contrary), then you can protect yourself by binding that global variable to itself in this outermost let. This binding ensures that any modification to that global variable will be local to your product, no matter what your code may do.

Running in Your Own Process

9.9.2 If the user calls your top-level function from a Lisp Listener, then your function will effectively take over that Listener's process until your function returns. In the meantime, the user has lost the use of that Listener. Sometimes this is fine. Sometimes, however, you want your product to run in its own process free and independent of whomever initiated it.

It is simple to run a function in its own process on the Explorer: just call **make-process** or **process-run-function** (which calls **make-process** for you) by passing it the name of your top-level function along with the arguments for that function, and your function is off and running in its own process. This process will terminate normally when your function returns (if ever). In process terminology, your top-level function and its arguments are known as the process's *initial function* and *initial arguments*.

When your top-level function is running in its own process, however, it has to worry about some extra bindings. When your function starts, the value of ***terminal-io*** will be bound to the bidirectional window stream that your function is supposed to use. Even if your function runs in background without a window, this value of ***terminal-io*** is important because—at a minimum—this stream is where errors occurring in your background process will be reported. The beginning of your process's initial function would now look like this:

```
(defun xyz ( . . . )
  (let ((*terminal-io*      *terminal-io*)
        (*standard-output* (make-synonym-stream '*terminal-io*))
        (*standard-input* (make-synonym-stream '*terminal-io*))
        (*error-output*  (make-synonym-stream '*terminal-io*))
        (*trace-output*  (make-synonym-stream '*terminal-io*))
        (*debug-io*      (make-synonym-stream '*terminal-io*))
        (*query-io*      (make-synonym-stream '*terminal-io*))
        (*foo* *foo*)
        (*bar* *bar*))
    . . .))
```

The binding of ***terminal-io*** to itself was done for the same reasons as before. The binding of the other standard streams (that is, ***standard-output*** and friends) to synonym streams of ***terminal-io*** is done to establish the default operation Explorer users have come to expect; that is, all standard streams default to the same place unless explicitly redirected elsewhere. Without these bindings, your defaults will match those of the process that created your process—regardless of how they might have been changed.

The other standard streams are bound to a synonym stream of ***terminal-io*** so that redirecting ***terminal-io*** redirects everybody. That is, ***terminal-io*** becomes a collective name for all the standard streams. The relationship between windows and ***terminal-io*** is explained in Section 1, Conventional Use of the Standard Streams.

Running in Your Own Window

9.9.3 Products that run in their own process frequently have their own window and vice versa. This close association is so common that the system has made special provisions for establishing this relationship. When you define your window flavor, simply mix in the **w:process-mixin** flavor. Now, anything that creates your window will automatically cause a new process to be created with your initial function running in it. Similarly, if someone kills the window, its associated process will be killed also.

There are several ways of specifying your top-level function to **w:process-mixin**. One option is to construct your product as a function of one argument, which is the newly created window it is to be associated with the process. Therefore, you will have to change the opening portion of your function as follows:

```
(defun xyz (window)
  (let* ((*terminal-io*      window)
         (*standard-output* (make-synonym-stream '*terminal-io*))
         . . .)
    . . .))
```

That is, everything is the same except that you must bind your copy of ***terminal-io*** to your new window (the function argument) rather than to itself. The other standard streams are still bound to synonym streams of ***terminal-io***.

For more information on Explorer processes, see Section 7, Processes and Scheduling.

Initializing Your Product

9.10 From the time your customer receives the tape with your product on it until the time he is running that product, there are several points at which initializations must be made only one time. For example,

- Tape Restore Time: These initializations are things such as creating directories and are usually done automatically during the restore phase of the Load Distribution Tape utility.
- Product Load Time: These initializations are things that might accompany the main **make-system** and are usually done automatically during the load phase of the Load Distribution Tape utility.
- Before Disk Save Time: These initializations are things that must be done to the current environment before it is saved as a new load band. These initializations get rid of temporary processes and data you don't want carried over to the new band and force active processes into a state that *will* carry over.
- After Cold Boot Time: These initializations are done only at cold boot time and not at warm boot. Only the most primitive systems functions usually need these sort of initializations.
- After Warm Boot Time: These initializations are done after every cold boot and on every warm boot. These initializations are widely used to start (or reset and restart) background servers.
- Before Garbage Collect Time: These initializations are done before a full garbage collection step. The idea is to kill (and thereby turn into garbage) all temporary data, buffers, caches, windows, processes, etc. so that the garbage collected load band will be as compact as possible.

The first two sets of initializations are handled automatically by the product specific restore and install files carried on the product's Distribution Tape. The remaining initializations are handled by the Explorer's initialization list facility.

An *initialization list* is a list of forms that are executed sequentially at certain specified times. If your product needs to have something special done at one of these times, then you can use the **add-initialization** form to have your initializations run.

For example, if our XYZ product were a network server for the customer's site, then we would want to be sure it gets started every time the system is booted—even if the user doesn't think to start it himself. Then you could add the following top-level form at the end of the last file loaded by **make-system**:

```
(add-initialization "Start XYZ Server"
  '(xyz:start-xyz-server)
  '(:warm :first))
```

where

"Start XYZ Server" is a comment that will appear with the entry on the initialization list.

'(xyz:start-xyz-server) is the form that will actually be executed at warm boot so it should do all the work of starting the server.

'(:warm ...) identifies this as a warm boot initialization (you will need one **add-initialization** form for each initialization list you want your form on).

'(... :first) indicates that the above form is to be executed first before it is added to the initialization list (this allows the server to start immediately after the **make-system** rather than having to wait for the next boot).

Notice that a warm boot by itself usually means that the system got so wedged that the customer was forced to warm boot to regain control. Therefore, your warm boot form should take into account that it may have been called under less than ideal circumstances. If your product was already running when the warm boot occurred, your warm boot form should make every effort to reset, recreate, and reinitialize everything about your product short of actually destroying the customer's work in progress.

Even if your product does not need any of the initialization lists for its own correct operation, you might still consider whether you can help out your customer when he does a full GC in preparation for saving a new load band. If your product has any history lists, caches, data buffers, temporary windows, and so on that are going to be thrown away by the next cold boot anyway, then you can use an initialization list to clean up your product. For example:

```
(add-initialization "Cleanup XYZ Server"
  '(xyz:cleanup-xyz-server)
  '(:full-gc))
```

Now whenever your customer does a **full-gc** in preparation for saving a new load band, your product will oblige him by turning data structures it no longer needs into garbage so that they can be collected. The tacit assumption, by the way, is that the call to **full-gc** will be immediately followed by a call to **disk-save** so that your product will not be accessed after the full GC initialization list is run.

A Helping Hand For copy-file

9.11 If your product uses only files with common file types (for example, LISP, XLD, TEXT, and so on), then this paragraph is of no interest to you. However, if your product does create its own brand of files, there is something you can do to allow your custom file types to *blend in*.

Let's begin by considering two details of the Explorer world. First, the Explorer uses a modified 7-bit ASCII code internally. The first 32 codes (which are normally ASCII control characters) have been replaced with additional printable characters, and the control functions have been moved to codes beginning at 128. The Explorer system cleverly masks its special character set from the outside world by performing character translation on character files. Therefore, you always have the ASCII standard on the network and the Lisp machine standard on the Explorer's disk.

Second, Common Lisp defines a *byte* to mean roughly what most languages call a bit-string. That is, byte does not necessarily mean eight bits. In particular, the Explorer supports files with 1-, 2-, 4-, 8-, 16-, and (in some cases) 32-bit bytes. The Common Lisp functions **read-byte** and **write-byte** work in terms of the file's defined byte-size.

Now, let's consider one of the consequences of these two details: before an Explorer can copy a file from one place to another, it must determine—in some manner—whether the file contains ASCII characters that need translating and the byte size of the file. The copy file logic usually manages to derive the information it needs from a heuristic based on file properties and file type name. If this heuristic fails, the software stops what it is doing and asks the user what kind of file he or she is copying. This query can become annoying if the user happens to be copying a whole directory of these funny files.

The quickest way to see if any of your private file types will cause problems at copy time is to use **copy-file** on one of them and see what happens. If it queries you about characters and byte-size, and if you want to be nice to your customers, then you can identify your private file types to the system so that **copy-file** won't have to ask next time.

Three global special variable are described below; each contains a list of file types. Push your private file type name (written as a string) onto the appropriate list:

- **fs:*copy-file-known-text-types*** records file types for 8-bit character files. The source file types for all languages as well as text and document file types are on this list.
- **fs:*copy-file-known-binary-types*** records file types for 16-bit non-character files. These file types are usually restricted to the special case of Lisp machine object files.
- **fs:*copy-file-known-short-binary-types*** record file types for 8-bit non-character files. These files fit the conventional notion of a *binary* file.

10

COLOR CONCEPTS

Introduction

10.1 This section discusses the issues involved in color programming and then presents the basics of programming the Color Explorer. References for further reading are provided at the end of this section.

NOTE: Color programming on the Explorer system requires optional color hardware.

Basics of Color Perception

10.2

Nature of Light

10.2.1 *Light* is that range of electromagnetic radiation that our eyes can detect and react to. Light radiation, like radio waves, can be characterized by its frequency. However, light is traditionally measured by its wavelength, which is an equivalent but irrelevant measure. Suffice it to say that light at the blue end of the visible spectrum has a wavelength shorter than light at the red end. Thus, one often sees references to short versus long wavelengths of light. We perceive light of different wavelengths as different colors.

All light we see is composed of lightwaves of various lengths. As we shall see, this is important when it comes to generating colors with a computer display.

Nature of the Eye

10.2.2 There are several important aspects of the interaction between light of various wavelengths and our eyes. Perhaps the simplest is the fact that our eyes cannot focus on all colors at the same time. If the lens of the eye has been shaped by the muscles to focus on red, green will be somewhat out of focus and blue will be badly out of focus. Thus, a display with a wide range of colors will place severe demands on the eye, leading to fatigue as the eye continuously refocuses on different portions.

Another aspect of the eye is that its sensitivity to light is due to the wavelength of the light and the part of the eye where the light lands. Similar intensities of red or green are harder to perceive when the light lands in the periphery of vision than in the center of vision. The eye is more sensitive to a yellowish-green, falling off on either side of a sensitivity-versus-wavelength plot.

Finally, and most importantly, color perception is not purely a function of wavelength. Color is a matter of the brain interpreting stimuli from the eye. In essence, the eye has three different receptors, each “tuned” to a portion of the visible spectrum. When light lands on the eye, each of the receptors sends the brain a message indicating how “strong” a signal is being received. The brain combines all these inputs and synthesizes a response, such as emerald green or royal blue. The same color sensation can be invoked by more than one combination of wavelengths and intensity.

For example, suppose the color x is generated by light consisting of 3 units at 4100 Angstroms, 12 units at 5350 Angstroms, and 7 units at 6442 Angstroms. Suppose this light is being projected on a screen. Now someone comes along and sets up a second projector and proceeds to “tune” it until the observer declares the two spots of light are the same color. When the “tuner settings” are examined, the results are 9 units at 4223 Angstroms, 8 units at 5290 Angstroms, and 9 units at 6511 Angstroms. The eye cannot tell the difference — the resulting response is identical. The brain reacts to a “net” stimulus, not individual wavelengths. If it were not for this marvelous property of the eye, generation of color displays would be much more difficult!

The Analytic Model

10.2.3 Because the brain reacts to a “net” stimulus and not individual wavelengths, one color can be generated by combining other colors. The most common way to demonstrate this is to shine light of different colors onto a screen. In the region where the two colors overlap on the screen, we perceive a different color. Shining red and green light onto a screen results in yellow. So naturally we can ask, what colors do we need to generate all the other colors?

There is no unique answer. Suppose we use three colors A, B, and C. Let the intensity of A be a , the intensity of B be b , etc. Then, a color X might be

$$X = a*A + b*B + c*C$$

But, we could choose three other colors, L, M, and N, such that

$$X = l*L + m*M + n*N$$

The same color is represented using 3 other colors! In either case, the set of colors used to generate a new color are called the *primary colors*, and the amount of each color required are *color coefficients*.

Suppose we use three colors L, M, and N to try to match the color Q. We shine a source of Q-colored light on the screen. Next to it we shine some L, some M, and some N. We fiddle and adjust but never come close to matching Q. What is wrong? Well, the answer is that the amount of one of the colors may be negative. STOP, you shout, how can you have “negative light”? Well, of course, you can’t. But let’s suppose that m equals -7.2. We can then write the equation as

$$Q = l*L - 7.2*M + n*N$$

This is the same as

$$Q + 7.2*M = l*L + n*N$$

This means that we can't match Q, but we can shine 7.2 units of M onto the the Q-colored spot. The resulting color can then be matched by a mixture of L and N. Since we are going to generate colors by having a CRT glow with the light landing in our eye, we cannot "subtract" colors to get a desired color. In other words, a CRT *cannot* generate every color that our eye can perceive. But there are three colors that, with all positive coefficients, maximize the range of colors we can generate with a CRT. These three colors are red, green, and blue.

Basics of Computer-Generated Color	10.3
---	-------------

Color CRT Operation	10.3.1
----------------------------	---------------

All we need to know to program a color display is that, for each picture element (pixel) on the display, there is a little triad of color phosphor. The three colors of the triad are the three primaries mentioned above, red, green, and blue. At the back of the CRT are three electron guns that each produce a stream of electrons (beam) that sweep across the inside face of the CRT. As the beams sweep back and forth, the intensity of each beam changes so that the pixels glow in the desired colors. On the Explorer system, each beam can take on one of 256 different intensity levels, with 0 being off and 255 being full strength. Since the magnitude of each of the primary colors is independent of the other two, there are a total of 256 times 256 times 256 possible combinations of red/green/blue intensity. That is, there are over 16 million possible colors that can be generated.

Color Models	10.3.2
---------------------	---------------

When controlling a device that has multiple independently controlled parameters, it is helpful to have a model that shows what happens when any one of the parameters is changed. This is certainly true of color programming, since there are three independent parameters (the amount of red, green, and blue). Color does not mix the way sound does. If one listens to three separate notes and then listens to these three notes in a chord, there is a new sensation (the harmony of the chord), but the three notes are still distinguishable. With light, *one* color results when three colors are mixed, and it is not always easy to determine the end result of a small change in one of the source colors. So, *color models* have been developed to help the user anticipate the effect of changing the parameters.

The most common (but least useful for many) is the RGB model. In this model the three parameters are the amount of red, the amount of green, and the amount of blue. Of course, as already stated, it's hard to predict the result of combining these three values. But the model is pervasive because it corresponds directly to how the hardware works.

A much more useful model is the HSV model. This model is more perceptually oriented than the RGB model. H stands for *hue*, S for *saturation*, and V for *value*. We will define these in more detail shortly, but first a few general observations about the model. With the HSV model, colors can be manipulated in a more natural way. As any of the three parameters is changed, the underlying software translates the change into the appropriate values of R, G, and B. The change is particularly easy to observe using the Explorer Color Map Editor. The editor provides six slide bars. Three slide bars provide the user with control over the values of the R, G, and B components of the color

being edited. The other three slide bars provide control over the values of H, S, and V. As you change the value of H using the slider, you can see how the R, G, and B sliders all change to create the new color.

Now, for a more detailed definition of HSV. Recall that color sensation is the result of the brain processing stimuli from the eye as light of various wavelengths. Wavelengths in the range of 450 to 480 nanometers (nm) evoke a sensation of blue, 500 to 550 nm evoke a sensation of green, 570 to 580 nm evoke a sensation of yellow, and 610 nm and higher evoke a sensation of red. This sensation of color is called *hue*. In the HSV model, hue takes on a value between 0 and 360 degrees, i.e., a circle that spans the spectrum around the circumference: red to yellow to green to cyan to blue to magenta and back to red again.

In the real world, light reaching the eye rarely consists of a single wavelength or a narrow range of wavelengths. Instead, a number of different wavelengths are usually present. A color produced by a very narrow range of wavelengths is called a *dominant hue*. The color of a dominant hue is considered to be pure or highly *saturated*. Adding different wavelengths to a dominant hue causes the color to become less saturated and therefore less pure.

The *value* parameter applies to a color's intensity. The best way to remember the relationships of value and saturation is this:

Value	Saturation	Yield
0	0	No hue
1	0	White
0	1	Black
1	1	Pure colors

Value and saturation can be any number from 0 to 1. Decreasing saturation without changing value is like adding white to a color. Decreasing value without changing saturation is like adding black to a color.

Guidelines for Effective Use of Color

General Principles

10.4.1 Why use color anyhow? In recent years, researchers have been asking questions about information processing. The researchers wanted to know how information should be coded and formatted, how quickly users can process information, how much information can be displayed at once, and how quickly the information can be updated. The results of the research have shown that color plays an important role in increasing human productivity when it is applied to interactive information displays. For example, suppose the task at hand was spotting and correcting words in a text file that have been flagged by a program as potentially misspelled. Color has been shown to be superior to reverse video, blinking, or underlining.

But one cannot just splash color across an interface indiscriminately. The following paragraphs lists guidelines for using color for Explorer color interfaces.

Designing the Color User Interface

10.4.2 The Explorer is an open system that any application developer can customize to a great extent. But research on the use of color in user interfaces has shown that the effectiveness of color coding can be impaired when the coding varies from application to application. Therefore, we have provided guidelines for developing Explorer-based applications that meet most user interface needs, and encourage everyone to model their user interfaces on these guidelines. To provide this consistency as simply as possible, the Explorer system features several Profile variables that specify the color of common parts of the interface, such as menu background color, menu foreground color, scroll bar shading, and so on. These variables are described in Section 19, Using Color, of the *Explorer Window System Reference* manual.

The following recommendations resulted from two areas of research. One area has focused on how the eye detects color and how the brain reacts to color. The other area has studied how color affects information processing tasks. If you would like to know more about the bases of these recommendations, please refer to the reference documents listed in paragraph 10.6. The numbers in brackets that follow an item refer to this reference list.

- Before you try to incorporate color into an application, design an effective user interface and screen layout in monochrome. Adding color to an effective design can enhance the usability of the application; adding color to a poor design simply adds color rather than makes the application easier to use. Adding color to a poor design may, in fact, emphasize its deficiencies.

Probably the most important recommendation resulting from the research is *don't overuse color*. The brain can only track about 5 to 7 independent concepts at once. If each color in a display has a specific meaning, more than 5 to 7 colors can lead to situations where the user must react to a color stimulus, but cannot recall the meaning associated with the color. This can be annoying in, say, an editor. It can be disastrous in a process control environment. [1]

- Avoid using too many bright colors on a single, frequently used application screen. As a rule of thumb:
 - Use only four colors (preferably the ones used in the Explorer convention) on a screen where each color signifies a different type of highlighting (such as a mouse blinker, text, marked items, and so on).
 - Use a maximum of seven colors for the screen.

NOTE: This guideline does *not* apply to colors used to highlight power-up screens, login screens, copyright screens, and the like. These types of screens often benefit from flashy displays of color that attract attention.

- When possible, use redundant coding for shape and color. [3]
- Use brightness and saturation to draw attention. [1]

- Use similar colors for similar meanings. This is the assumption most users make, whether true or not. [1]
- Group related elements by a common background color. [1]
- Use monochrome tones (that is, black, white, and the various shades of gray) for backgrounds, desktops, menus, and other nonactive elements of the user interface. Using a monochrome, or achromatic, appearance for nonactive elements increases the effectiveness of colors used for highlighting.
- Use primary (that is, named) colors for highlighting errors, marked text, selected objects, and so on. These colors provide good contrast when used with an achromatic background (a background composed of black, white, and/or shades of gray). In addition, if you use only named colors for highlighting, you can use the same colors from machine to machine and keep consistency regardless of brand of monitor.
- Avoid large areas of a high-intensity white color. Instead, use a light shade of gray for backgrounds, menus, and so on. Using a light gray rather than pure white reduces the halation effect that makes text and small objects displayed on a white area difficult to see. You can use small amounts of white to increase contrast and legibility—for example, to display text on a dark background.
- Avoid red and green at the outer edges of large displays. The peripheral color vision of the eye is poor for red and green. [1]
- Avoid pure blues for text, thin lines, and small shapes. The eye muscles cannot contour the lens enough to completely focus short wavelength light. For the same reason, blues are good for large background areas. [1]
- Avoid the simultaneous display of highly saturated, spectrally extreme colors. When the eye muscles focus the lens for long wavelength light (such as yellow or red), short wavelength light (such as blue or green) is no longer in focus. [1]
- Use brightness contrast to convey hierarchies, depth, and dimensionality. Use light colors to give the appearance of nearness or to suggest importance; use darker colors (ones with less contrast) to give the appearance of fading into the distance or to suggest relatively less importance. For example, a menu that includes several subheadings and subordinated entries could display the subheadings in white on a dark gray background with the entries displayed in light gray on the same dark gray background.
- The magnitude of a detectable change in color varies across the spectrum. Small changes in extreme reds, purples, and greens are more difficult to detect than the same amount of change in a yellow or blue-green color. [1]
- Use higher brightness levels for users with vision impairments. They need higher brightness levels to distinguish colors. [1]

- Keep in mind cultural considerations. Beyond the obvious RED = stop or alert while GREEN = go or save, specific disciplines have certain meanings associated with color. For example, in medical analysis of CAT scans, danger areas are often coded in greens or purples while safe areas are coded in red or yellow. Also, blue, (as in CODE BLUE) is associated with death. [2]
- Keep in mind the physiological response to color. For example, warm colors refer to action, required response, and spatial nearness. Cool colors refer to status, background information, and spatial separation. [3]

Basics of Programming the Color Explorer

Considerations for Programming Applications

10.5 Now that we have the basic ideas of how users perceive color, how the computer generates color, and how one might use these colors to create user interfaces, we can discuss how to program color on the Explorer system. The following paragraphs discuss adding color at the application level and how to incorporate color in your specialized system functions if you are working at a lower level (such as building tools for creating user interfaces).

10.5.1 This paragraph describes how color is implemented on the Color Explorer, *not* in terms of how the hardware works, but in terms of what an applications programmer needs to know in building an application on top of the window system.

The Explorer color option brings to the window system a hardware capability of 8 bits of information per picture element (pixel), compared with the 1 bit per pixel capability of the monochrome system. The hardware can generate 24 bits of color information to drive the color monitor. These 24 bits are composed of 8 bits for red, 8 bits for green, and 8 bits for blue. So, each of the three colors (R, G, and B) can take on one of 256 values, with 0 being completely off and 255 being full strength. Taking in all possible combinations, there are over 16 million possible colors. But, you may ask, how can 8 bits per pixel represent 16 million colors? The answer is, they can't! Each pixel can take on one of 256 possible values. Each of these values is translated by the hardware into a 24-bit value (8 bits of Red, 8 bits of Green, 8 bits of Blue). The translation is performed using a data table called the color look-up table (LUT). So, at any one time only 256 different colors can be displayed on the Explorer color monitor. Later on we will describe how to modify the contents of this LUT. For now we will rely upon the fact that the table is initialized by the window system with a factory-preset color LUT.

Although the addition of color would, at first glance, seem to be a simple extension to the window system, it does have some immediate consequences that affect how you might program the window system. Let us introduce these new concepts by way of contrast with the monochrome system you are already familiar with.

Remember that each pixel in the monochrome system can have only one of two values, either on or off. When running the window system on color hardware, each pixel can take on any one of 256 values, ranging from 0 to 255. Thus, a pixel is no longer just on or off, black or white, 0 or 1, but rather it is an integer with a range of values. Furthermore, since the integer value is translated into screen colors via a LUT, the same integer value may represent different colors, depending on the currently installed LUT. This integer value is referred to as either the *pixel value* or the *logical color* of a pixel. The

value generated by the LUT that corresponds to the logical color is the *physical color*.

For many operations such as writing text or drawing a simple line, there are only two colors of interest. These colors are called the *foreground color* and the *background color*. Perhaps the best way to understand the meaning of foreground and background is by comparison with the monochrome system. When the screen is cleared, all pixels are set to the off state. One can also think of this as setting all the pixels to their background color. When text is drawn to the screen, you can think of pixels as being turned on, or as pixels set to the foreground color. For example, clearing the screen on a color system may result in the screen being turned completely green (the background color) with text appearing on it in white (the foreground color). In general, any operation that in a monochrome system turns a pixel off, in a color system sets a pixel to the window's background color. Any operation that turns a pixel on in a monochrome system will, on a color system, set the pixel to the foreground color.

Each window has its own foreground and background color. In the simplest case, every window has the same foreground and background color. By default, any text or graphics operation is drawn in the window's foreground color. Thus, any application transferred from a monochrome system will run with black and white replaced by the window's current foreground and background colors.

As described in the *Explorer Window System Reference* manual, there are a number of ways that a pixel on the screen can be changed, based on the new value to be written and the ALU operation to be performed. For example, a common operation on a monochrome system is to perform an exclusive OR (XOR) operation between the pixel value to be written and the existing pixel value. After the XOR operation is performed, the resulting on or off state is written to the memory location corresponding to the desired pixel. The same ALU operations are available for the color environment, as are some additional operations.

However, some of the same operations that were provided for monochrome systems are less valuable in a color system. The main reason for the decreased usefulness is that these operations are binary operations. Their meaning is clear when applied to 1-bit pixels but not so when applied to the 8-bit pixels of a color system. 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? For color, more meaningful operations are needed.

These additional operations are add, subtract, max, min, average, add with saturation, subtract with clamping, transparency, and background. The first two are pretty obvious. Combining a source value of 3 with an existing pixel of 27 yields a resulting pixel of value 30. Now, what color is this? It depends on the LUT. For example, if the table was set up so that the values 20 to 40 corresponded to dark blue to light blue, then the pixel will have changed from one shade of blue to a slightly lighter shade of blue. But, 27 could have been green and 30 could have been orange. The lesson here is that, unlike the monochrome environment in which the only thing that determined how a pixel looked was the values and the ALU operation, in a color environment the same values combined using the same ALUs can result in different colors depending on the color LUT.

The add and subtract operations are performed modulo 256. That is, if a pixel's current value is 254 and a value of 4 is added, the result is 258. But 258 is greater than the largest available value, so the result is $258 - 256 = 2$. One can think of the addition as wrapping around to the front of the table. Similarly, a value of 3 minus a value of 7 yields a pixel value of 252.

One place the add and subtract ALUs are used is in blinkers. In a monochrome environment, blinkers are made to appear on the screen using XOR and removed using a second XOR. For the color environment, the blinker is added to make it appear and subtracted to make it disappear. The wrap-around feature is important, for example, in adding an offset of 5 to color 255 to yield color 4, and in restoring the contents to color 255 when 5 is subtracted.

For some applications this wraparound behavior is not desirable. For such applications the *add with saturate* and *subtract with clamp* are the proper operations to use. Suppose the saturation value is set to 150 and the clamp value to 10. Then $128 + 10$ yields 138 as expected. But $128 + 50$ yields 150, since the result is constrained to not exceed the saturation value. Similarly, $128 - 20$ yields 108, but $128 - 126$ yields 10. Section 19 of the *Explorer Window System Reference* manual describes how to set the saturate and clamp values.

The *max* and *min* operations are also straightforward. For example, if 20 is combined with 30 using max, the resultant pixel is 30. If 50 is combined with 30 using max, the resultant pixel is 50. The *average* operation performs the arithmetic average of the source and destination pixel, truncating any fractional remainder. For example, combining 20 with an existing pixel of value 40 yields $(20 + 40) / 2 = 30$. Once again, the results of these operations only make sense if the color map is properly prepared.

The other two operations are *background* and *transparency*. The background ALU simply forces the destination pixel to the window's current background color regardless of the value of the source or destination pixel value. Transparency is harder to understand, but very valuable. Let us discuss transparency by using text, since that is the most common application. The definition of each letter displayed in the screen consists of a pattern of on and off bits. As stated before, on bits result in pixels being set to the foreground color and off bits setting pixels to the background color. With text, all we are really concerned with are the on bits. When the transparency operation is used, on bits still set pixels to the foreground color, but off bits are *ignored*—the corresponding pixel is left as is. Since the result is that existing colors “show through” wherever a letter has off bits, the off bits are “transparent.”

NOTE: If you are converting applications to color by writing your applications to use the general ALU functions provided by the window system (i.e., `w:char-aluf` and `w:erase-aluf`), your application will work in either color or monochrome, since the window system always sets these to the appropriate value. If, however, you have placed specific ALU operations in your code (for example, ALU-XOR or ALU-IOR) your application will run in color—but it may have cosmetic problems. You can fix such problems by changing to the new ALU functions. In general, use the new ALU functions for all applications. They are defined to work equally well in either a color or a monochrome system.

Another extension added is the *texture pattern*, used in `w:graphics-rmixin` methods. Texture (or dither) patterns are an important supplement to color. In the monochrome window environment, the `w:graphics-mixin` methods have an argument somewhat misnamed *color*. In the monochrome environment, the *color* argument is used to draw in one of eight dither patterns, simulating various shades of gray. In the color environment this same argument actually does specify the drawing color. For some applications it is still desirable to draw a pattern, or texture, in color. Therefore, the `w:graphics-mixin` methods have a new optional parameter to provide a pattern.

The last but most important concept is that of the *color map*. We've already described LUTs. The LUT is used during each refresh cycle of the monitor to translate 8-bit pixel values into 24 bits of physical color information. The LUT is part of the color option hardware. There is only *one* LUT on the color board. Yet, you may need different color definitions for each of several windows.

Therefore, the window system has been extended to associate each window with a data structure called a *color map*. The name color map was chosen to distinguish it from the hardware LUT. A programmer works with the color map, not directly with the color LUT buffers. Functions to access and modify any of the elements of this structure are documented in the *Explorer Window System Reference* manual. For now, the important points are:

- Each window has its own color map.
- The color map contains the clamp and saturate values for the window.
- The color map contains a *color map table* of R, G, and B values for each of the 256 possible colors.

When a window is created, the default action is to give the window a pointer to its superior's color map. This default action is important for some applications. For example, panes in a constraint frame logically share the same color map. When a window is created, the default can be overridden with an initialization option. A window's color map can also be changed at any time by sending the window a method to set a new color map and then selecting the window to load the new color map. You can also modify the existing color map, either through programmer control or through user control via the color map editor.

Whenever a window is exposed or selected, its color map table is automatically loaded into one of the LUT buffers, and the LUT is then loaded from the freshly updated buffer. So, as different windows are selected, the hardware is always using a copy of the currently selected window's color map table. At the same time that the color map table is loaded into the LUT, the clamp and saturate values of the color map are placed where microcode can access them during add with saturate or subtract with clamp operations.

Although the user can modify any location in the color map table, *it is strongly recommended that the reserved colors be left alone*. The reserved colors are listed in the reserved field of the color map. Currently, the first 32 slots are reserved. These 32 slots contain 8 shades of gray (for compatibility with the 8 dither patterns in the monochrome environment) plus commonly named colors such as red, green, yellow, orange, and so on. All system software is defined to use only these reserved colors for such things as the default foreground and background of a window, border colors, highlighting

colors, and so on. By confining the specialized colors you need for your application to the rest of the color map table, you can ensure that all your windows will maintain a consistent appearance. For example, your application may require a set of pastel colors, some shades of brown, and several shades of green. Before exposing your application window, you would use the color map functions to, for example, define the pastel colors in locations 40 through 64, the shades of brown in 70 through 90, and the shades of green in locations 200 to 215.

If, for some application you must modify and use *all* 256 locations of the color map table, keep in mind that some system-defined features may look somewhat odd when your color map is installed and active. By the same token, popping up a window over your application may result in a different color map table being loaded to the LUT, making your application look wrong. As soon as the pop-up window is removed, however, your window's color map table is loaded and your application once again returns to normal.

The remaining locations of the color map table are loaded with values for a red color ramp, a green color ramp, and a blue color ramp. (A *ramp* is a sequence of colors. These ramps vary in intensity from very dark shades to very light shades of the same color.)

To summarize, Table 10-1 lists the important terms defined in this section, plus a few additional useful terms.

Table 10-1 Glossary of Color Terms

Term	Definition
background color	The color corresponding to an off or 0-bit pixel in a monochrome environment.
color	Generic term meaning pixel value, RGB value, HSV value, or some other abstract quality whose exact meaning is determined by context.
color environment	A configuration of optional hardware and software that allows the Explorer system to display 256 colors at one time.
color map	The software data structure (a defstruct) that contains, among other things, the color map table. Each window contains an instance of a color map. When a window is exposed or selected, its color map becomes the active color map, which means that the color map's table is downloaded to the hardware.
color map table	The software data structure (an array) that specifies a mapping of pixel values (logical colors) to RGB values (physical colors). The values in the color map table are downloaded to the hardware LUT buffer when a window is exposed or selected.
color model	The representation used to define a color. Providing the values for red, green, and blue is just one of many ways to define a color. The Explorer system uses the RGB model to define a color. Other models such as HSV (hue/saturation/value), HSI (hue/saturation/intensity), and YIQ (used for commercial TV broadcasts) can be found in popular books on color.
foreground color	The color corresponding to an on or 1-bit pixel in a monochrome environment.
frame buffer	The actual hardware memory array that contains the values of pixels displayed on the physical screen. The Explorer frame buffer is 1024 by 1024 by 8 bits. This is the equivalent of a screen array. Note that only 1024 by 808 is used for displaying an image on the screen.
logical color	The integer value that corresponds to an address in the LUT. Also see pixel value.
LUT (Look-Up Table)	The actual hardware table holding the currently available RGB values. This table <i>cannot</i> be read by the software. The LUT is the only table that the hardware uses to translate 8-bit color data into 24 bits of RGB data.
LUT buffer	The two hardware buffers from which the LUT can be loaded. These buffers can be read from and written to by software.
pixel	A single picture element.
pixel value	A value stored in the frame buffer that represents the <i>logical color</i> of its associated pixel. A pixel value is used as an index into the LUT, which transforms the pixel value into a <i>physical color</i> for driving the display. Pixel values are 8-bit quantities.
RGB value	A value describing a physical color. It is divided into three fields that specify the intensity of each of the additive primary colors (red, green, and blue) in a single color.

Considerations for Programming the Explorer System

10.5.2 This paragraph describes how the color system works and how the programmer interacts with the color system. The following paragraphs describe how the hardware works and how to use the functions and methods provided so you don't have to worry about manipulating the hardware.

As with the monochrome system interface board (SIB), the Color SIB (CSIB) has a section of memory called the frame buffer that is part of the address space of the Explorer system. This memory is scanned by the hardware on the CSIB to generate the display you see on the color monitor. The memory is arranged so that each 8-bit byte corresponds to one pixel on the display monitor. As the hardware scans this memory, each 8-bit value is read and used to look up the pixel's red, green, and blue intensity values from the LUT.

All color programming boils down to manipulating one of two quantities. The first is the contents of the color LUT, which contain, in essence, a definition of the color for each of the 256 possible values that an 8-bit pixel can have. For example, the LUT may contain values of red, green, and blue for each pixel value such that the resultant display is seen in shades of gray. Or the values in the LUT could be such that the pixel values 0 to 255 correspond to different slices of a rainbow. Functions are provided to directly manipulate the LUT buffers from which the LUT is loaded, and to manipulate the contents of the color map, which is an instance variable of windows.

The second quantity that can be manipulated is the contents of the frame buffer. If a pixel value of 99 results in a shade of blue appearing on the screen, and 100 results in a shade of green on the screen, then for each location in the frame buffer that you cause to be set to the value 99, a blue pixel will appear on the monitor, while for each location that you set to a value of 100, a green pixel appears on the monitor. As simple as this sounds, there are a number of factors that affect what the final contents of the frame buffer will be.

First, we must digress and discuss the concept of expansion. Expansion is used whenever the basic information known about an object is its shape. The classic example of this situation is text. For each letter, the pattern of 0s and 1s in a font merely describe the shape of the letter, they do not describe the *color* of the letter. In other words, fonts are essentially 1-bit deep sources of information. Yet, to cause color to appear on the monitor, an 8-bit quantity is required in each location of the frame buffer. The interface between the 1-bit deep source (the font) and the 8-bit deep destination (the frame buffer) is *color expansion*. Expansion works as follows: The hardware is loaded with an 8-bit value in a location called the *foreground color register*. Another 8-bit value is loaded in a second register called the *background color register*. Now, when we write a 1-bit deep pattern of 1's and 0's to the frame buffer, expansion kicks in. For each 1 written, the value written in the frame buffer is the value found in the foreground register. Similarly, for each 0 written, the value written in the frame buffer is the value found in the background register. For example, suppose the contents of the LUT are such that color 200 is dark-purple and color 76 is yellow. By loading 76 into the foreground color register and 200 into the background color register, text drawn on the screen will appear in yellow against a dark-purple background. This happens because the frame buffer has a 76 written for each 1 in the font copied to the screen, while a 200 is written in the frame buffer for each 0 in the font.

Now, this may all seem a bit messy, and we're going to muddy the waters a bit more here, but soon you will see that you rarely have to work this close to the hardware for color programming. The one thing that we haven't discussed yet that complicates the picture is ALU operations. On a monochrome system, values written to the screen memory are combined with the current contents using a specified operation. For example, the value to be written can be ANDed with the existing pixel value. The value can also be Exclusive Or'ed (XORed) with the existing pixel value. All these familiar operations are available in color, but in some cases their use is more limited than in a monochrome environment.

The classic example of these operations is the XOR. XOR is widely used in a monochrome environment because of one property: XORing the same pattern onto the screen twice in a row leaves the screen unmodified. The first time object X is drawn onto the screen using an XOR operation, it appears because wherever there is a pixel in the source, the destination is forced into the opposite state, (on pixels switch to off and vice versa). Thus, regardless of what is on the screen, the source pattern appears because it changes the pixel to its opposite state. Performing the operation a second time switches all the pixels again, thus restoring the original screen. When operating in a color environment, XOR behaves *logically* as it should, but the result on the screen can be more difficult to predict, since the resultant color is determined by the contents of the color LUT.

For color systems, however, there are some additional ALUs. For example, there are ADD and SUBTRACT ALUs. When the source and destination are combined using the ADD ALU, the resultant pixel value is the sum of the source and destination pixel values. Similarly, the SUBTRACT ALU causes the new pixel value to be the destination pixel minus the source pixel. By combining an object with the screen using first an ADD, then a SUBTRACT, the net change to the screen (as with two successive XORs) is no change at all. Of course, the result still depends on the contents of the LUT. For example, red plus yellow could result in a shade of pink, a shade of blue, or any other of the 16 million possible colors, depending on the color definitions stored in the LUT at the location specified by the pixel value resulting from the sum.

To summarize to this point, color programming involves defining colors by putting values for red, green, and blue intensity into the LUT, and by causing locations in the frame buffer to take on one of 256 possible values. But, it sure looks like a lot of work! So far, we have been talking about what happens at the low level of the window system. Now let's see how simple things can really be.

In the simplest case, you need do nothing to run the window system in color. Every window has an instance variable that contains the COLOR-MAP as well as instance variables that contain the FOREGROUND-COLOR and BACKGROUND-COLOR. When you turn on your Color Explorer, these instance variables are initialized to predetermined defaults. Whenever a window is exposed or selected, its color map is copied into the LUT, and the FOREGROUND-COLOR and BACKGROUND-COLOR instance variables are copied into the hardware. So, all of the programming described above is performed for you automatically using system defaults.

You can change the appearance of windows several ways. One way is to use the System menu to access the Edit Attributes menu to change the foreground and background colors of the window. After doing so, any operation performed is done using the new values of the instance variables. Thus, you can easily change a black and gray window to a green and yellow window, or whatever two colors you choose. Another way to change the color of windows is to use the Profile utility to change the window default colors. Then, every window you create will use your new defaults rather than the factory-set defaults.

The most common way to program the system in color is to use the optional color parameters added to the window system's functions and methods that perform screen output. For example, you can draw a pink line with

```
(send my-window :draw-line start-x start-y end-x end-y thickness  
w:pink)
```

Or you can write a blue letter using

```
(send my-window :tyo #\A `cptfont w:blue)
```

The *Explorer Window System Reference* manual lists all of the modified functions that indicate the additional color parameter.

If you need to develop your own output routines and want to have control over the output color, the following discussion describes how the current window system software works, and can serve as a model for your efforts.

As you probably are aware, all output ultimately is performed by a small set of microcoded routines. These routines are described in Section 12, Graphics, of the *Explorer Window System Reference* manual. These functions perform identically on monochrome or color systems with the exact same parameter lists. How, then, do we get color on color systems? There are two answers, which we will call the direct method and the indirect method. Before defining these two terms, let us take a quick detour.

Prior to the development of the color system, the microcode routines were designed to deal with either 1-bit or 8-bit quantities in matched sets. For example, one could BITBLT a 1-bit source to a 1-bit destination, or an 8-bit source to an 8-bit destination. But a 1-bit source to an 8-bit destination was illegal. With the addition of color capability and the concept of expansion, the microcode was enhanced to understand 1-bit sources (such as a font or a pattern) and 8-bit destinations (such as a screen array). In this case, the microcode ensures that the 1-bit source is expanded into 8-bit quantities as described above. Sending an 8-bit source to a 1-bit destination is still an illegal operation.

Now, back to the main road. *Direct* color programming means that you provide the microcode with 8-bit source information that is used to write into the 8-bit locations of the destination array. For example, suppose that you create an array called X with a repeating pattern of 0, 9, 27, and 33. Suppose also that these values correspond to black, dark-green, medium-green, and light-green. Then, if you were to use %draw-shaded-triangle with X as the PATTERN parameter, the resultant triangle on the screen would be painted with a repeating pattern of green intensities.

Indirect programming means that you load the foreground color register and the background color register, and then provide a 1-bit source array. When the drawing occurs, the source array is expanded into the frame buffer using the contents of the foreground and background registers. By changing the register contents, the same array can be drawn in a variety of colors. Because the resultant display depends on the register contents for color rather than upon the source array, this form of programming is called indirect programming.

NOTE: Direct programming overrides indirect programming. For example, if you set up the registers for a green foreground and a pink background, then supply an 8-bit deep source array whose values produce shades of blue, the result on screen will be shades of blue. Why? Because the source array is 8-bits deep, which means expansion is not required, which means the contents of the registers are not used.

With the above preparatory remarks, we can describe how the window system handles color. (Remember, all of this is handled automatically if you use the window system methods and functions.) First of all, the prepare-sheet macro has been modified so that everytime a prepare-sheet is performed, the foreground and background color registers are loaded using the values found in the window's foreground and background color instance variables. Thus, the hardware tracks the window colors automatically. When you provide one of the optional color parameters (for example, for drawing a line in a color other than the current foreground color), one of two things happens. If the internal software of the window system can make a change before a prepare-sheet is called, a macro called **prepare-color**, is used. For example:

```
(prepare-color (window COLOR)
  (progn
    .... ;;; misc code here
    ....
    (prepare-sheet (window)
      .... ;;; code here that draws
    )
  )
)
```

The **prepare-color** macro saves the window's current foreground color, and then sets the foreground color to the value **COLOR**. Within the **prepare-sheet** code this color is loaded into the hardware. The final part of the **prepare-color** executes after the body of its code, and restores the window's instance variable to the value saved earlier.

If the code is already within a **prepare-sheet**, a macro called **prepare-color-register** is used. This macro is very similar to **prepare-color**, except that it saves, loads, and restores the hardware register directly rather than via the instance variable.

Whenever possible, manipulate window instance variables. Each window has its own instances, so each window remembers what is in the hardware. If you manipulate the hardware directly, another window may come along and change the contents of the hardware.

Just as the window's foreground and background instance variables are loaded into the hardware automatically, so is a window's color map table. Whenever a window is exposed or selected and is visible on the screen, its color map table is downloaded into the hardware. Functions are provided for altering the contents of the hardware LUT and the color map table, for loading the LUT from the color map table. Again, the safest approach is to manipulate the window's color map table, since then the window remembers its color definitions. If you directly manipulate the hardware and then another window is exposed, your changes will be overwritten by the exposed window's values.

If you are developing your own system code, that is, code that does not rely on the window system, you can use the `prepare-color` and `prepare-color-register` macros just described. But what if you are developing general-purpose code you expect to work properly on either a monochrome system or a color equipped system? There are just a few simple rules.

First, always use the new ALU operations. The microcode automatically does the right thing on a monochrome or color system. But if you hardcode your routine to use `alu-xor`, then the microcode can't turn it into `alu-add` or `alu-sub`. If you use `alu-transp` anywhere you would draw with `alu-ior`, you get `IOR` on monochrome and `alu-transp` on color-equipped systems. Similarly, if you use `alu-back`, you get `alu-andca` on monochrome for erasing and `alu-back` in color for erasing. If you examine the truth table of a 1-bit `alu-add` or `alu-sub` with no carry, you discover that both `alu-add` and `alu-sub` have the same truth table as `alu-xor`. This means that, if you write your code to put items on the screen with `alu-add` and remove them with `alu-sub`, in a monochrome system you actually get two successive `alu-xors` as desired, and in color you get the color add and subtract as desired. So, the code does not need to know specifics of the system configuration.

Secondly, changing the `foreground-color` and `background-color` instance variables does no harm in a monochrome system, since the hardware registers are loaded with 255 and 0 respectively and never changed, even when the instance variable changes. But the same code running on a color system will change color as you change your instance variable values.

Conversely, if you do change the contents of the `foreground-color-register` and the `background-color-register` on a monochrome system, unexpected results can occur since the normal value of 255 (hexadecimal FF) ensures that upon expansion all 8-bit planes are active. If you cannot avoid writing code that reaches down to the hardware level, you should conditionalize the code to change the hardware only when the screen and its windows are color windows. This can be determined either by the general-purpose function

```
(get-display-type window-or-sheet)
```

which returns `:MONOCHROME` on a monochrome window or `:8-BIT-COLOR` on a color window. This function is intended to be general-purpose and can be expected to return other keyword values when new display products are developed and introduced.

If all you need to know is if the window is color or not, then you can use the macro

```
(color-system-p sheet)
```

which returns `nil` if `sheet` is on a monochrome system; otherwise, the macro returns the type of system, for example `:8-bit-color`.

References

10.6 There are many good sources of information about color and color programming available. Much of the information in this section was derived from the following sources. These references (especially "Human Factors of Color Displays") have extensive bibliographies providing you with additional sources of information.

1. "Human Factors of Color Displays," Gerald M. Murch, *Color in Computer Graphics Tutorial Course Notes*, SIGGRAPH, ACM, 1987.
2. "The Ergonomics of Enhancing User Performance with Color Displays," Wanda J. Smith, Joyce E. Farrell, *Color in Computer Graphics Tutorial Course Notes*, SIGGRAPH, ACM, 1987.
3. "Users Must Establish Own Rules for Color," Aaron Marcus, *Computer Graphics Today*, Volume 2, No. 9, Sept 1985.
4. *The Feynman Lectures on Physics*, Volume 1, Chapter 35, Richard P. Feynman, Robert B. Leighton, and Matthew Sands, Addison-Wesley Publishing Company, 1966.
5. *Fundamentals of Interactive Computer Graphics*, James D. Foley, Andries Van Dam, Addison-Wesley Publishing Company, 1982.
6. *Raster Graphics Handbook*, Conrac Division, Conrac Corporation, 1980.

INDEX

Alphabetization Scheme

The alphabetization scheme used in this index ignores package names and nonalphabetic symbol prefixes for the purposes of sorting. For example, the `rpc:*callrpc-retrys*` variable is sorted under the entries for the letter C rather than under the letter R.

Hyphens are sorted after spaces. Consequently, the multiple menus entry precedes the multiple-choice facility entry.

Underscore characters are sorted after hyphens. Consequently, the `xdr-io` macro precedes the `xdr_destroy` macro.

General

A

accessor methods, 2-9
action-oriented programming. *See* programming, action-oriented
Active processing state, 7-15
application program. *See* program product
arrest reasons, 7-14
Arrested processing state, 7-15

B

background stream
 helper function, 1-6
 typeout window, 1-6
band, 6-2
bashing processes, 7-15—7-16
bindings
 bypassing, 7-8
 initial values for a process, 7-6—7-7
block names, conflicts in macros, 8-3

C

canonical types, 6-15
code
 adding versus modifying, 2-4
 redefinition versus replacement, 5-1—5-2
 reusability of, 2-17
colons, indications of whether a symbol is exported, 9-3
color concepts, 10-1—10-18
 add ALU operation, 10-9
 add with saturate ALU operation, 10-9
 ALU operations, 10-8, 10-14, 10-17
 average ALU operation, 10-9
 background ALU operation, 10-9
 background color, 10-8
 background color instance variable, 10-16
 background color register, 10-13, 10-17
 basics of computer-generated color, 10-3—10-4
 color argument of w:graphics-mixin methods, 10-10
 color coefficient, 10-2
 color CRT operation, 10-3
 color, definition, 10-12
 color environment, 10-12
 color map, 10-10
 color map table, 10-10
 color models, 10-3
 color perception
 analytic model, 10-2
 basics, 10-1—10-3
Color SIB (CSIB), 10-13
color-system-p macro, 10-17

designing the color user interface, 10-5
direct color programming, 10-15
Edit Attributes menu for changing the window colors, 10-15
expansion, 10-13
foreground color, 10-8
foreground color instance variable, 10-16
foreground color register, 10-13, 10-17
frame buffer, 10-12
get-display-type function, 10-17
glossary of color terms, 10-12
guidelines for effective use of color, 10-4—10-7
HSV model, 10-3
hue, 10-3
indirect color programming, 10-16
light, 10-1
logical color, 10-7
Look-Up Table (LUT), 10-7
LUT buffer, 10-10
max ALU operation, 10-9
min ALU operation, 10-9
nature of the eye, 10-1
physical color, 10-8
pixel, 10-3
pixel value, 10-7
prepare-color macro, 10-16
prepare-color-register macro, 10-16
prepare-sheet macro, 10-16
primary color, 10-2
programming the color Explorer, 10-7—10-17
purity, 10-4
ramp, 10-11
references of color programming information, 10-18
RGB model, 10-3
RGB value, 10-12
saturation, 10-3
subtract ALU operation, 10-9
subtract with clamp ALU operation, 10-9
texture pattern, 10-10
transparency ALU operation, 10-9
value, 10-3
Common Lisp
 extensions to, 9-2
 portability of programs, 9-2
compile conditions, 4-1
 summary, 4-6—4-9
compile-flavor-methods function, 2-16
compiling
 back-to-back, 4-6
 commonly used functions for, 4-10—4-12
 incremental, 4-5

- component flavor, 2-13
 condition handlers, conditional, 3-8—3-9
 condition-bind macro
 example of a handler, 3-3—3-5
 handlers default, 3-8
 condition-call macro, example of a handler, 3-6
 condition-case macro
 example of a handler, 3-5—3-6
 :no-error clause, 3-6
 condition-typep predicate, 3-4
 conditions
 definition of, 3-1
 events, 3-1—3-2
 handlers, 3-3
 hierarchy of, 3-9—3-11
 ad hoc, 3-13—3-14
 instances of, 3-2
 names of, 3-11—3-14
 proceeding from, 3-7
 copy-file macro, testing file types with, 9-17
 coroutines, 7-3—7-5
 programming, 7-4—7-5
 resuming, 7-3
 customer sites and customer-specific
 translations, 9-7
- D**
- data hiding, 2-17—2-18
 definition, 2-6
 data sharing among processes, 7-7—7-8
 data structure patching, 5-8—5-9
 default handlers, 3-8
 default streams, 1-3
 defaulting pathnames, 6-9—6-11
 definition, 6-9
 specifying standard defaults, 6-11—6-12
 defflavor macro, 2-8
 defmethod macro, 2-8
 defparameter macro, compared to defvar, 5-14—5-15
 .DEFSYSTEM file, 9-3
 defsystem macro
 :compile-load transform, example of, 4-10
 :compile-load-init transform, example of, 4-11
 fully constrained declarations, 4-12
 defvar macro, compared to defparameter, 5-14—5-15
 dependencies, 4-1
 load, 4-10
 summary, 4-6—4-9
 device component of a pathname, 6-2—6-3
 directories
 characteristics
 as a directory, 6-4
 as a file, 6-4
 relative, 6-26—6-28
 root, 6-3
 site, 6-24—6-25
 directory component of a pathname, 6-3—6-4
 disk space, logical directories, reducing, 9-9
 dribble files, 1-10—1-11
- E**
- embarrassment insurance, 4-6
 environment
 modified, 4-4—4-5
 patch, 5-5
 patching the, 5-8—5-10
 error handler process, 7-4
 errors
 ignoring, 3-16—3-17
 proceeding from, 3-15
 software-readable, 3-12
 evaluations
 multiple, 8-6—8-7
 out-of-order, 8-6—8-7
 Executable processing state, 7-15
 exporting symbols, 9-3
- F**
- file attribute list, 9-5
 as a comment line, 9-5
 Common Lisp standard, 9-6
 file attributes, 5-5
 filename component of a pathname, 6-4
 files, translations, 9-7
 flavors, 2-1
 adding to software with flavors, 2-5
 instance, 2-12—2-13
 mixing, 2-13—2-16
 programming, 2-5—2-6
 trees, 2-18—2-19
 Flushed processing state, 7-15
 flushing processes, 7-15—7-16
 function, defining steps in, 5-2—5-3
- G**
- GLOBAL package, 9-2
 global special variables, providing program
 isolation with bindings, 9-13
 global variables, 7-5
- H**
- handler, 3-3
 default, 3-8
 definition of, 2-7
 functions versus handler forms, 3-7
 provisional, 3-14
 host name, manifest, 6-6

I

I/O, kinds of I/O a program may need, 1-1—1-2
incremental compile, 4-5
inheritance, 2-1
initialization list, 9-16
instance variables
 accessing, 2-9
 common, 2-20
 contrasted with operation and message, 2-6
 :gettable, 2-9
 naming, 2-10
 :settable, 2-9
interchange format of namestrings, 6-16—6-18
internal pathnames, 9-9
interned pathnames, 6-7—6-8

K

KEYWORD package, and operator names, 2-9
killing processes, 7-15—7-16

L

LISP package, 9-2
load dependencies, 4-10
local variables, 7-6
logical hosts, program products and, 9-6

M

macros
 block name conflicts, 8-3
 functions used during expansion of, 8-9—8-10
 multiform bodies, 8-3—8-4
 nesting, 8-7—8-9
 surrounding code, 8-4—8-6
 with-naming convention, 8-5
make-condition function, 3-2
make-system function
 :compile option, 4-11—4-12
 :reload option, 4-12
merging, pathnames, 6-11
merging pathnames, definition, 6-9
message, contrasted with operation and instance, 2-6
message passing, 2-1, 2-4—2-5
 send notation, 2-4
method combinations
 order of execution, 2-18
 partial list of, 2-19
method hash table, 2-17
method table, 2-8
methods
 combined, 2-16
 daemon, 2-15
 patching, 5-9—5-10
 primary, 2-15
mixing flavors, 2-13—2-16

modified environment, 4-4—4-5

modifying code
 redefinition versus replacement, 5-1—5-2
 versus adding code, 2-4

N

namestrings, 6-1
 interchange format, 6-16—6-18
 reconstructed by :string-for- methods, 6-6
nested macros, 8-7—8-9

O

object-oriented programming. *See* programming, object-oriented
operation, contrasted with instance and message, 2-6
operator names, 2-9

P

package declaration, placement of, 9-3
package names, unique, 9-5
packages
 naming, 9-4
 standard system, 9-2
partition, 6-2
patch directories, 5-11—5-12
 reason for separate, 9-9
 system-defined, 5-13
patch environment, 5-5
patch file, naming, 5-12
patch numbers, 9-8
patchable program products, 9-8
patchable system, print-herald and, 9-12
patches
 directory of, 5-11
 filenames of, 5-12
 loading, 5-12—5-13
 using Zmacs to create, 5-6—5-8
patching
 avoiding common mistakes, 5-7—5-8
 data structure definitions, 5-8—5-9
 editing a patch buffer manually, 5-9
 existing instances (the environment), 5-8—5-10
 installing a patch, explanation, 5-10—5-13
 methods, 5-9—5-10
 requirements for a patchable system, 9-8
 what you cannot patch, 5-10—5-11
patching the environments, 5-8—5-10
pathname component
 device, 6-2—6-3
 directory, 6-3—6-4
 filename, 6-4
 host, 6-6
 type
pathname component (continued)
 canonical, 6-15

- from UNIX, 6-12—6-14
 - role of, 6-14—6-15
 - type, 6-4
 - version, 6-5
 - pathname object, 6-1
 - pathnames
 - defaulting, 6-9—6-11
 - merging versus, 6-9
 - specifying standard defaults, 6-11—6-12
 - generic, 6-18
 - internal, 9-9
 - interned, 6-7—6-8
 - logical
 - defining translations, 6-22
 - example of use, 6-19—6-21
 - merging, 6-11
 - definition, 6-9
 - translations, 9-7
 - physical pathnames, translation files and, 9-7
 - presetting a process, 7-11
 - print names of macro internal variables, 8-3
 - proceed types
 - definition, 3-15
 - nonlocal, 3-16
 - processes, 7-8—7-13
 - active list, 7-14
 - activity states, 7-13—7-15
 - arrest reasons, 7-14
 - bashing, 7-15—7-16
 - error handler, 7-4
 - initial function, 7-9—7-10
 - killing, 7-15—7-16
 - presetting, 7-11
 - resetting, 7-11, 7-15—7-16
 - run reasons, 7-14
 - sequence break, 7-10
 - sharing data among, 7-7—7-8
 - simple, 7-12
 - standard bindings of, 7-9
 - states (specific terms), 7-15
 - stopping, 7-10
 - synonym streams and, 9-14
 - wait function, 7-10
 - waiting voluntarily, 7-10—7-11
 - why use them, 7-2
 - program
 - alternative ways of informing, 1-3
 - noting a situation, 1-3
 - program product
 - checklist for delivery, 9-1
 - initializations, 9-15
 - isolation from other code, 9-12
 - loading with a make-system, 9-10
 - logical host and, 9-6
 - naming convention for, 9-4
 - patchable, 9-8
 - processes and, 9-14
 - reinstalling software, 9-3
 - verifying that file types port, 9-17
 - windows and processes, 9-14
 - programming
 - action-oriented, 2-1—2-2
 - flavors and, 2-5—2-6
 - object-oriented, 2-1, 2-2—2-4
 - provisional handlers, 3-14
- R**
- recompiling, suppressing unnecessary, 4-4
 - redefinition
 - as a technique for modifying code, 5-1
 - explanation of, 5-3—5-5
 - responses to warnings, 5-4
 - relative directories, 6-26—6-28
 - release numbers, 9-8
 - replacement, as a technique for modifying code, 5-1
 - resetting a process, 7-11, 7-15—7-16
 - resume handlers, 3-16
 - resuming a coroutine, 7-3
 - reusability of code, 2-17
 - root directory, 6-3
 - rule of thumb
 - asking for information, 1-2—1-3
 - avoiding OS-specific pathname notation, 9-9
 - binding special global variables, 7-7
 - coroutine switching, 7-3
 - exporting symbols, 9-3
 - instance variable naming, 2-10
 - macros and the :compile-load-init transform, 4-11
 - order of declarations in a file, 4-9
 - pathnames and canonical types, 6-15
 - *query-io* similar to y-or-n-p, 1-3
 - :reload option of make-systems, 4-12
 - when to use error or ferror, 1-4
 - run reasons, 7-14
 - Runnable processing state, 7-15
- S**
- Scheduler, 7-8
 - errors, 7-12—7-13
 - signal, 3-3
 - site directory, 6-24—6-25
 - software
 - reinstalling reasons for, 9-3
 - using flavors to add to, 2-5
 - software-readable errors, 3-12
 - special global variables, binding versus setting, 7-7
 - stack groups, 7-5—7-8
 - Steve's Ice Cream, 2-20
 - Stopped processing state, 7-15
 - stream predicate, heuristic, 1-10
 - streams
 - checking for, 1-10

sys:cold-load-stream, supported messages, 1-9
default, 1-3
specific
 error-output, 1-3—1-4
 sys:cold-load-stream, 1-8—1-10
 debug-io, 1-5
 sys: *null-stream*, 1-10
 query-io, 1-3
 standard-input, 1-2—1-3
 standard-output, 1-2—1-3
 terminal-io, 1-5—1-8
 as a troublesome argument, 1-7
 initial value, 1-1
 trace-output, 1-5
synonym, 1-7
 guaranteeing the correct defaults for a process, 1-8
 processes and, 9-14
t as an argument, 1-3
variable versus, 1-10
structured component of a pathname, 6-3
symbols, exporting, 9-3
SYS-host
 manipulating, 6-26
 problems, 6-25—6-26
system file, as a reference source, 9-11
.SYSTEM files, 5-13—5-14

T

TCL package, 9-2
.TRANSLATIONS file, 9-7
translations files, 9-7
type component of a pathname, 6-4
 from UNIX, 6-12—6-14
 canonical, 6-15
 role of, 6-14—6-15

U

USER package, 9-2

V

variables
 binding versus setting global special, 7-7
 global versus private, 7-5—7-6
 local, 7-6
 name conflicts, 8-1—8-3
 shadowing, 7-5
 special, 7-5
 streams versus, 1-10
version component of a pathname, 6-5

W

Waiting Forever processing state, 7-15
window lock status, 1-5

Data Systems Group - Austin Documentation Questionnaire

Programming Concepts

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 _____

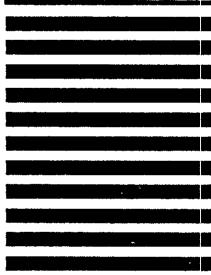
Telephone _____ Date _____

TAPE EDGE TO SEAL

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

A standard POSTNET barcode is positioned below the horizontal bars, consisting of ten vertical bars of varying widths.

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