VAX LISP/VMS User's Guide

Order No. AA-Y921B-TE

May 1986

This document contains information required by a LISP language programmer to interpret, compile, and debug VAX LISP programs.

Revision/Update Information: This manual contains Update Notice 1, AD-Y921B-T1.

Operating System and Version: VAX/VMS Version 4.4

Software Version: VAX LISP/VMS Version 2.2

Digital Equipment Corporation Maynard, Massachusetts

First Printing, June 1984
Revised, May 1986
Updated, July 1987

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright © 1984, 1986, 1987 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

AI VAXstation
DEC
DECnet
DECUS
MicroVAX
MicroVAX II
MicroVMS
PDP
ULTRIX
ULTRIX-11
ULTRIX-32
UNIBUS
VAX
VAXstation
VAXstation II
VMS
digital

CONTENTS

PREFACE

Part I
VAX LISP/VMS SYSTEM CONCEPTS AND FACILITIES

CHAPTER 1 INTRODUCTION TO VAX LISP

1.1	OVERVIEW OF VAX LISP	1-2
1.1.1	DCL LISP Command	1-3
1.1.1.1	Interpreter	1-3
1.1.1.2	Compiler	1-4
1.1.2	Editor	1-4
1.1.3	Error Handler	1-4
1.1.4	Debugging Facilities	1-4
1.1.5	Pretty Printer	1-5
1.1.6	Call-Out Facility	1-5
1.1.7	Alien Structure Facility	1-5
1.1.8	Interrupt Function Facility	1-6
1.1.9	VAXstation Graphics Interface	1-6
1.1.10	System-Building Utility	1-6
1.1.11	VAX LISP/VMS Function, Macro, and Variable Descriptions	1-6
1.2	HELP FACILITIES	1-7
1.2.1	DCL HELP	1-7
1.2.2	LISP HELP	1-8
1.3	VAX/VMS FILE SPECIFICATIONS	1-8
1.4	LOGICAL NAMES	1-10
1.5	ENTERING DCL COMMANDS	1-10

CHAPTER 2 USING VAX LISP

2.1	INVOKING LISP	2-1
2.2	EXITING LISP	2-2
2.3	ENTERING INPUT	2-2
2.4	DELETING AND EDITING INPUT	2-2
2.5	ENTERING THE DEBUGGER	2-3
2.6	USING CONTROL KEY CHARACTERS	2-4
2.7	CREATING PROGRAMS	2-5
2.8	LOADING FILES	2-5
2.9	COMPILING PROGRAMS	2-6
2.9.1	Compiling Individual Functions and Macros	2-7
2.9.2	Compiling Files	2-7
2.9.3	Advantages of Compiling LISP Expressions	2-9
2.9.4	Advantage of Not Compiling LISP Expressions	2-9
2.10	DCL LISP COMMAND QUALIFIERS	2-9
2.10.1	Three Ways to Use the DCL LISP Command	2-12
2.10.2	/COMPILE	2-13
2.10.3	/ERROR_ACTION	2-14

2.10.4 /[NO]INITIALIZE 2-15
2.10.5 /INTERACTIVE 2-16
2.10.6 /[NO]LIST 2-17
2.10.7 /[NO]MACHINE_CODE 2-17
2.10.8 /MEMORY 2-18
2.10.9 /[NO]OPTIMIZE 2-19
2.10.10 /[NO]OUTPUT_FILE 2-20
2.10.11 /RESUME 2-21
2.10.12 /[NO]VERBOSE 2-21
2.10.13 /[NO]WARNINGS 2-22
2.11 USING SUSPENDED SYSTEMS 2-23
2.11.1 Creating a Suspended System 2-24
2.11.2 Resuming a Suspended System 2-24

CHAPTER 3 USING THE VAX LISP EDITOR

3.1 INTRODUCTION TO THE EDITOR 3-2
3.1.1 Editing Cycle 3-3
3.1.2 Invoking the Editor 3-3
3.1.3 Interacting with the Editor 3-6
3.1.3.1 Getting Help 3-7
3.1.3.2 Input Completion and Alternatives 3-8
3.1.3.3 Errors and Other Problems 3-9
3.1.4 Moving Work Back to LISP 3-10
3.1.5 Returning to the LISP Interpreter 3-10
3.1.6 Summary of Commands 3-11
3.2 EDITING OPERATIONS 3-13
3.2.1 Keypad 3-14
3.2.2 Inserting and Formatting Text 3-14
3.2.2.1 Inserting Ordinary Text 3-14
3.2.2.2 Typing and Formatting LISP Code 3-15
3.2.2.3 Inserting Nongraphic Characters 3-16
3.2.3 Moving the Cursor 3-17
3.2.3.1 Moving with the Keypad and Arrow Keys 3-17
3.2.3.2 Moving in LISP Code 3-18
3.2.3.3 Moving with the Pointer (VAXstation Only) 3-18
3.2.4 Modifying Text 3-19
3.2.4.1 Deleting Text 3-19
3.2.4.2 Undeleting Text 3-20
3.2.4.3 Cutting and Pasting Text 3-20
3.2.4.4 Changing Case 3-21
3.2.4.5 Substituting Text 3-22
3.2.4.6 Inserting a File or Buffer 3-23
3.2.5 Repeating an Operation 3-23
3.2.6 Summary of Commands 3-24
3.3 USING MULTIPLE BUFFERS AND WINDOWS 3-30
3.3.1 Introduction to Buffers and Windows 3-30
3.3.2 Creating New Buffers from Within the Editor 3-33
3.3.3 Working with Buffers 3-33
3.3.3.1 Saving Buffer Contents 3-34
3.3.3.2 Deleting Buffers 3-34

3.3.3.3 Buffer Name Conflicts 3-34
3.3.4 Manipulating Windows 3-35
3.3.5 Moving Text Between Buffers 3-36
3.3.6 Summary of Commands 3-36
3.4 RECOVERING FROM PROBLEMS 3-37
3.5 CUSTOMIZING THE EDITOR 3-38
3.5.1 Binding Keys to Commands 3-38
3.5.1.1 Binding Within the Editor 3-39
3.5.1.2 Binding from the LISP Interpreter 3-40
3.5.1.3 Selecting a Key or Key Sequence 3-43
3.5.1.4 Key Binding Context and Shadowing 3-44
3.5.2 Keyboard Macros 3-45
3.5.3 Summary of Commands 3-46
3.6 USING THE EDITOR ON A VAXSTATION 3-46
3.6.1 Screen Appearance and Behavior 3-47
3.6.2 Editing with the Pointer 3-47
3.6.2.1 The Pointer Cursor 3-47
3.6.2.2 Selecting and Removing Windows 3-48
3.6.2.3 Moving the Text Insertion Cursor and Marking Text 3-48
3.6.2.4 Cutting and Pasting 3-48
3.6.2.5 Invoking the DESCRIBE Function And Matching Parentheses 3-49
3.6.2.6 Information About Pointer Effects 3-49
3.6.3 Binding Pointer Buttons to Commands 3-49

CHAPTER 4 ERROR HANDLING

4.1 ERROR HANDLER 4-1
4.2 VAX LISP ERROR TYPES 4-1
4.2.1 Fatal Errors 4-2
4.2.2 Continuable Errors 4-3
4.2.3 Warnings 4-4
4.3 CREATING AN ERROR HANDLER 4-5
4.3.1 Defining an Error Handler 4-5
4.3.1.1 Function Name 4-6
4.3.1.2 Error-Signaling Function 4-6
4.3.1.3 Arguments 4-7
4.3.2 Binding the *UNIVERSAL-ERROR-HANDLER* Variable 4-7

CHAPTER 5 DEBUGGING FACILITIES

5.1 CONTROL VARIABLES 5-3
5.2 CONTROL STACK 5-3
5.3 ACTIVE STACK FRAME 5-4
5.4 BREAK LOOP 5-4
5.4.1 Invoking the Break Loop 5-5
5.4.2 Exiting the Break Loop 5-5
5.4.3 Using the Break Loop 5-6
5.4.4 Break Loop Variables 5-7

5.5 DEBUGGER 5-7
5.5.1 Invoking the Debugger 5-8
5.5.2 Exiting the Debugger 5-9
5.5.3 Using Debugger Commands 5-9
5.5.3.1 Arguments 5-11
5.5.3.2 Debugger Commands 5-13
5.5.4 Using the DEBUG-CALL Function 5-18
5.5.5 Sample Debugging Sessions 5-18
5.6 STEPPER 5-20
5.6.1 Invoking the Stepper 5-20.1
5.6.2 Exiting the Stepper 5-21
5.6.3 Stepper Output 5-21
5.6.4 Using Stepper Commands 5-24
5.6.4.1 Arguments 5-25
5.6.4.2 Stepper Commands 5-26
5.6.5 Using Stepper Variables 5-28
5.6.5.1 *STEP-FORM* 5-28
5.6.5.2 *STEP-ENVIRONMENT* 5-28
5.6.5.3 Example Use of Stepper Variables 5-29
5.6.6 Sample Stepper Sessions 5-31
5.7 TRACER 5-32
5.7.1 Enabling the Tracer 5-33
5.7.2 Disabling the Tracer 5-33
5.7.3 Tracer Output 5-34
5.7.4 Tracer Options 5-35
5.7.4.1 Invoking the Debugger 5-36
5.7.4.2 Adding Information to Tracer Output 5-36
5.7.4.3 Invoking the Stepper 5-36
5.7.4.4 Removing Information from Tracer Output 5-37
5.7.4.5 Defining When a Function or Macro Is Traced 5-37
5.7.5 Tracer Variables 5-37
5.7.5.1 *TRACE-CALL* 5-37
5.7.5.2 *TRACE-VALUES* 5-38
5.8 THE EDITOR 5-40

CHAPTER 6 PRETTY PRINTING AND USING EXTENSIONS TO FORMAT

6.1 PRETTY PRINTING WITH DEFAULTS 6-2
6.2 HOW TO PRETTY-PRINT USING CONTROL VARIABLES 6-3
6.2.1 Explicitly Enabling Pretty Printing 6-3
6.2.2 Limiting Output by Lines 6-4
6.2.3 Controlling Margins 6-4
6.2.4 Conserving Space with Miser Mode 6-5
6.3 EXTENSIONS TO THE FORMAT FUNCTION 6-5
6.3.1 Using the WRITE FORMAT Directive 6-7
6.3.2 Controlling the Arrangement of Output 6-8
6.3.3 Controlling Where New Lines Begin 6-11
6.3.4 Controlling Indentation 6-13
6.3.5 Producing Prefixes and Suffixes 6-14
6.3.6 Using Tabs 6-15
6.3.7 Directives for Handling Lists 6-16

6.4 DEFINING YOUR OWN FORMAT DIRECTIVES 6-18
6.5 DEFINING PRINT FUNCTIONS FOR LISTS 6-19
6.6 DEFINING GENERALIZED PRINT FUNCTIONS 6-21
6.7 ABBREVIATING PRINTED OUTPUT 6-23
6.7.1 Abbreviating Output Length 6-24
6.7.2 Abbreviating Output Depth 6-24
6.7.3 Abbreviating Output by Lines 6-25
6.8 USING MISER MODE 6-26
6.9 HANDLING IMPROPERLY FORMED ARGUMENT LISTS 6-28

CHAPTER 7 VAX LISP/VMS IMPLEMENTATION NOTES

7.1 DATA REPRESENTATION 7-2
7.1.1 Numbers 7-2
7.1.1.1 Integers 7-2
7.1.1.2 Floating-Point Numbers 7-3
7.1.2 Characters 7-5
7.1.3 Arrays 7-7
7.1.4 Strings 7-7
7.2 PATHNAMES 7-7
7.2.1 Namestrings 7-8
7.2.2 Logical Names and Pathnames 7-8
7.2.3 When to Use Pathnames 7-9
7.2.4 Fields of a COMMON LISP Pathname 7-9
7.2.5 Field Values of a VAX LISP Pathname 7-10
7.2.6 Three Ways to Create Pathnames 7-11
7.2.7 Comparing Similar Pathnames 7-13
7.2.8 Converting Pathnames into Namestrings 7-14
7.2.9 Functions That Use Pathnames 7-15
7.2.10 Using the *DEFAULT-PATHNAME-DEFAULTS* Variable 7-16
7.3 GARBAGE COLLECTOR 7-17
7.3.1 Frequency of Garbage Collection 7-18
7.3.2 Static Space 7-18
7.3.3 LISP Processing 7-18
7.3.4 Messages 7-19
7.3.5 Available Space 7-19
7.3.6 Garbage Collection Failure 7-19
7.4 INPUT AND OUTPUT 7-20
7.4.1 Newline Character 7-20
7.4.2 Terminal Input 7-21
7.4.3 End-of-File Operations 7-22
7.4.4 Record Length 7-22
7.4.5 File Organization 7-23
7.4.6 Functions 7-23
7.4.6.1 FILE-LENGTH Function 7-23
7.4.6.2 FILE-POSITION Function 7-24
7.4.6.3 OPEN Function 7-24
7.4.6.4 WRITE-CHAR Function 7-25
7.5 INTERRUPT FUNCTIONS AND KEYBOARD FUNCTIONS 7-25
7.6 COMPILER 7-26
7.6.1 Compiler Restrictions 7-26

7.6.1.1 COMPILE Function 7-26
7.6.1.2 COMPILE-FILE Function 7-27
7.6.2 Compiler Optimizations 7-27
7.7 FUNCTIONS AND MACROS 7-30

CHAPTER 8 VAX LISP I/O EXTENSIONS

8.1 DEFINING NEW TYPES OF STREAMS 8-1
8.1.1 Overview of VAX LISP I/O 8-2
8.1.2 Defining Stream Structures 8-2
8.1.3 Stream Dispatch Functions 8-3
8.2 GETTING INFORMATION ABOUT STREAMS 8-5
8.3 NEW I/O FUNCTIONS 8-7
IMMEDIATE-OUTPUT-P Function 8-7
LINE-POSITION Function 8-7
LISTEN2 Function 8-8
NREAD-LINE Function 8-8
OPEN-STREAM-P Function 8-9
RIGHT-MARGIN Function 8-10

Part II
VAX LISP/VMS FUNCTION, MACRO, AND VARIABLE DESCRIPTIONS

APROPOS Function 1
APROPOS-LIST Function 3
ATTACH Function 4
BIND-KEYBOARD-FUNCTION Function 6
BREAK Function 9
CANCEL-CHARACTER-TAG Tag 10
CHAR-NAME-TABLE Function 11
COMPILEDP Function 13
COMPILE-FILE Function 14
COMPILE-VERBOSE Variable 17
COMPILE-WARNINGS Variable 18
CONTINUE Function 20
DEBUG Function 21
DEBUG-CALL Function 22
DEBUG-PRINT-LENGTH Variable 23
DEBUG-PRINT-LEVEL Variable 24
DEFAULT-DIRECTORY Function 25
DEFINE-FORMAT-DIRECTIVE Macro 27
DEFINE-GENERALIZED-PRINT-FUNCTION Macro 30
DEFINE-LIST-PRINT-FUNCTION Macro 32
DELETE-PACKAGE Function 34
DESCRIBE Function 35
DIRECTORY Function 37
DRIBBLE Function 40
ED Function 41
ENLARGE-BINDING-STACK Function 42.1
ENLARGE-CONTROL-STACK Function 42.2

ERROR-ACTION Variable 43
EXIT Function 44
Format Directives Provided with VAX LISP 45
GC Function 48
GC-VERBOSE Variable 49
GENERALIZED-PRINT-FUNCTION-ENABLED-P Function 50
GET-DEVICE-INFORMATION Function 51
GET-FILE-INFORMATION Function 55
GET-GC-REAL-TIME Function 59
GET-GC-RUN-TIME Function 61
GET-INTERNAL-RUN-TIME Function 63
GET-KEYBOARD-FUNCTION Function 64
GET-PROCESS-INFORMATION Function 65
GET-TERMINAL-MODES Function 73
GET-VMS-MESSAGE Function 76
HASH-TABLE-REHASH-SIZE Function 77
HASH-TABLE-REHASH-THRESHOLD Function 78
HASH-TABLE-SIZE Function 79
HASH-TABLE-TEST Function 80
LOAD Function 81
LONG-SITE-NAME Function 83
MACHINE-INSTANCE Function 84
MACHINE-VERSION Function 85
MAKE-ARRAY Function 86
MODULE-DIRECTORY Variable 88
POST-GC-MESSAGE Variable 89
PPRINT-DEFINITION Function 90
PPRINT-PLIST Function 92
PRE-GC-MESSAGE Variable 95
PRINT-LINES Variable 96
PRINT-MISER-WIDTH Variable 97
PRINT-RIGHT-MARGIN Variable 98
PRINT-SIGNALED-ERROR Function 100
PRINT-SLOT-NAMES-AS-KEYWORDS Variable 102
REQUIRE Function 103
ROOM Function 105
ROOM-ALLOCATION Function 108
SET-TERMINAL-MODES Function 108.1
SHORT-SITE-NAME Function 111
SOFTWARE-VERSION-NUMBER Function 112
SOURCE-CODE Function 112.1
SPAWN Function 112.2
STEP Macro 115
STEP-ENVIRONMENT Variable 116
STEP-FORM Variable 117
SUSPEND Function 118
THROW-TO-COMMAND-LEVEL Function 121
TIME Macro 122
TOP-LEVEL-PROMPT Variable 123
TRACE Macro 124
TRACE-CALL Variable 135
TRACE-VALUES Variable 136

	TRANSLATE-LOGICAL-NAME Function	137
	UNBIND-KEYBOARD-FUNCTION Function	139
	UNCOMPILE Function	140
	UNDEFINE-LIST-PRINT-FUNCTION Macro	141
	UNIVERSAL-ERROR-HANDLER Function	142
	UNIVERSAL-ERROR-HANDLER Variable	143
	WARN Function	144
	WITH-GENERALIZED-PRINT-FUNCTION Macro	145
APPENDIX A	PERFORMANCE HINTS	
A.1	DATA STRUCTURES	A-1
A.1.1	Integers	A-2
A.1.2	Floating-Point Numbers	A-2
A.1.3	Ratios	A-2
A.1.4	Characters	A-3
A.1.5	Symbols	A-3
A.1.6	Lists and Vectors	A-4
A.1.7	Strings, General Vectors, and Bit Vectors	A-5
A.1.8	Hash Tables	A-6
A.1.9	Functions	A-6
A.2	DECLARATIONS	A-6
A.3	PROGRAM STRUCTURE	A-10
A.4	COMPILER REQUIREMENTS	A-12
APPENDIX B	USING THE "EMACS" EDITOR STYLE	
B.1	INTRODUCTION TO THE EDITOR	B-1
B.2	ACTIVATING THE "EMACS" STYLE	B-3
B.2.1	Activating "EMACS" as a Minor Style	B-3
B.2.2	Making "EMACS" the Major Style	B-4
B.3	"EMACS" STYLE KEY BINDINGS	B-4
APPENDIX C	EDITOR COMMANDS AND KEY BINDINGS	
C.1	EDITOR COMMAND DESCRIPTIONS	C-1
C.2	EDITOR KEY BINDINGS	C-14
INDEX		
FIGURES		
3-1	Numeric Keypad	3-15
6-1	Variables Governing Miser Mode	6-26

TABLES

1-1	File Specification Defaults	1-10
2-1	Keys Used In Line Editing	2-3
2-2	Control Characters	2-4
2-3	DCL LISP Command Qualifiers	2-10
2-4	DCL LISP Command Qualifier Modes	2-13
3-1	General-Purpose Commands and Key Bindings	3-11
3-2	Editing Commands And Key Bindings	3-24
3-3	Commands For Manipulating Buffers And Windows	3-36
3-4	Characters Generated by Keys	3-41
3-5	Commands For Customizing The Editor	3-46
4-1	Error-Signaling Functions	4-7
5-1	Debugging Functions and Macros	5-1
5-2	Debugger Commands	5-10
5-3	Debugger Command Modifiers	5-12
5-4	Stepper Commands	5-24
6-1	Format Directives Provided by VAX LISP	6-6
7-1	VAX LISP Floating-Point Numbers	7-4
7-2	Floating-Point Constants	7-5
7-3	VAX LISP Pathname Fields	7-10
7-4	Summary of Implementation-Dependent Functions and Macros	7-30
8-1	I/O Request Specifiers	8-4
8-2	Stream Data Types and Predicates	8-6
8-3	Stream Informational Functions	8-6
1	Format Directives Provided with VAX LISP	45
2	GET-DEVICE-INFORMATION Keywords	51
3	GET-FILE-INFORMATION Keywords	55
4	GET-PROCESS-INFORMATION Keywords	65
5	GET-TERMINAL-MODES Keywords	73
6	Data Type Headings	106
7	TRACE Options	125
B-1	Differences Between "EMACS" Key Bindings and Default Bindings	B-2
B-2	"EMACS" Style Key Bindings	B-4
C-1	Editor Commands And Key Bindings	C-2
C-2	Editor Key Bindings	C-14

PREFACE

Manual Objectives

The *VAX LISP/VMS User's Guide* is intended for use in developing and debugging LISP programs and for use in compiling and executing LISP programs on VAX/VMS systems. The VAX LISP language elements are described in *COMMON LISP: The Language*.*

Intended Audience

This manual is designed for programmers who have a working knowledge of LISP. Detailed knowledge of VAX/VMS is helpful but not essential; familiarity with the *Introduction to VAX/VMS* is recommended. However, some sections of this manual require more extensive understanding of the operating system. In such sections, you are directed to the appropriate manual(s) for additional information.

Structure of This Document

An outline of the organization and chapter content of this manual follows:

PART I: VAX LISP/VMS SYSTEM CONCEPTS AND FACILITIES

Part I consists of eight chapters, which explain VAX LISP concepts and describe the VAX LISP facilities.

- Chapter 1, Introduction to VAX LISP, provides an overview of VAX LISP, explains how to use the help facilities, describes VAX/VMS file specifications and the logical name mechanism, and provides hints on entering DCL commands. Chapter 1 also describes where in the VAX LISP documentation you can find information on each of the VAX LISP features.

---

* Guy L. Steele Jr., *COMMON LISP: The Language*, Digital Press (1984), Burlington, Massachusetts.

- Chapter 2, Using VAX LISP, explains how to invoke and exit from VAX LISP; use control key sequences; enter and delete input; create and compile programs; load files; and use suspended systems. In addition, Chapter 2 describes the DCL LISP command and its qualifiers.

- Chapter 3, Using the VAX LISP Editor, describes how to use the Editor provided with VAX LISP to create and edit LISP code.

- Chapter 4, Error Handling, describes the VAX LISP error-handling facility.

- Chapter 5, Debugging Facilities, explains how to use the VAX LISP debugging facilities.

- Chapter 6, The Pretty Printer, explains how to use the VAX LISP pretty printer.

- Chapter 7, VAX LISP Implementation Notes, describes the features of LISP that are defined by or are dependent on the VAX implementation of COMMON LISP.

- Chapter 8, VAX LISP I/O Extensions, describes VAX LISP extensions to the COMMON LISP I/O system.

PART II: VAX LISP/VMS FUNCTION, MACRO, AND VARIABLE DESCRIPTIONS

Part II describes functions, macros, and variables specific to VAX LISP and any COMMON LISP objects that have specific implementation characteristics in VAX LISP. Each function or macro description explains the function's or macro's use and shows its format, applicable arguments, return value, and examples of use. Each variable description explains the variable's use and provides examples of its use.

Associated Documents

The following documents are relevant to VAX LISP/VMS programming:

- *VAX LISP/VMS Installation Guide*

- *COMMON LISP: The Language*

- *VAX LISP/VMS Editor Programming Guide*

- *VAX LISP/VMS System Access Programming Guide*

- *VAX LISP/VMS Graphics Programming Guide*

- *VAX LISP/VMS System-Building Guide*

- *Introduction to VAX/VMS*
- *VAX/VMS DCL Dictionary*
- *VAX/VMS System Services Reference Manual*
- *VAX/VMS I/O User's Reference Manual: Part I*
- *VAX/VMS Run-Time Library Routines Reference Manual*
- *VAX Architecture Handbook*

For a complete list of VAX/VMS software documents, see the *VAX/VMS Information Directory and Index*.

Conventions Used in This Document

The following conventions are used in this manual:

Convention	**Meaning**
( )	Parentheses used in examples of LISP code indicate the beginning and end of a LISP form. For example: (SETQ NAME LISP)
[ ]	Square brackets enclose elements that are optional. For example: [doc-string] Square brackets do not indicate optional elements when they are used in the syntax of a directory name in a VAX/VMS file specification. Here, the square bracket characters must be included in the syntax.
UPPERCASE	DCL commands and qualifiers and defined LISP functions, macros, variables, and constants are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters.
lowercase italics	Lowercase italics in function and macro descriptions and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters.

Convention	Meaning
...	In a DCL command description, a horizontal ellipsis indicates that the element preceding the ellipsis can be repeated. For example: *function-name ...* In LISP examples, a horizontal ellipsis indicates code not pertinent to the example and not shown.
. . .	A vertical ellipsis indicates that all the information that the system would display in response to the particular function call is not shown; or, that all the information a user is to enter is not shown.
{ }	In function and macro format specifications, braces enclose elements that are considered to be one unit of code. For example: {*keyword value*}
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered to be one unit of code, which can be repeated zero or more times. For example: {*keyword value*}*
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments after it are defined to be optional. For example: PPRINT *object* &OPTIONAL *package* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.
&REST	In function and macro format specifications, the word &REST indicates that an indefinite number of arguments may appear. For example: BREAK &OPTIONAL *format-string* &REST *args* Do not specify &REST when you invoke the function or macro whose definition includes &REST.

Convention	Meaning
&KEY	In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: COMPILE-FILE *input-pathname* &KEY {*keyword value*}* Do not specify &KEY when you invoke the function or macro whose definition includes &KEY.
<RET>	A symbol with a 1- to 3-character abbreviation indicates that you press a key on the terminal. For example: <RET> or <ESC> In examples, carriage returns are implied at the end of each line. However, the <RET> symbol is used in some examples to emphasize carriage returns.
CTRL/x	CTRL/x indicates a control key sequence where you hold down the CTRL key while you press another key. For example: CTRL/C or CTRL/Y
Black print	In examples, output lines and prompting characters that the system displays are in black print. For example: $ LISP/COMPILE $_File(s): MYPROG.LSP
Red print	In examples, user input is shown in red print. For example: $ LISP/COMPILE $_File(s): MYPROG.LSP

SUMMARY OF NEW AND CHANGED INFORMATION

This manual contains the following new and changed information:

- You can now define new stream types. VAX LISP now provides functions that return information about streams and also new I/O functions. Chapter 8 describes these facilities.
- The /INSTALL and /REMOVE qualifiers to the DCL LISP command are no longer needed and no longer exist. Chapter 2 reflects this.
- The VAX LISP debugger, stepper, and tracer have different or enhanced information displays. Examples in Chapter 5 and in Part II have been updated to show these new displays.
- The following new functions have been added to VAX LISP and are described in Part II:

  ENLARGE-BINDING-STACK
  ENLARGE-CONTROL-STACK
  ROOM-ALLOCATION
  SOFTWARE-VERSION-NUMBER
  SOURCE-CODE

CHAPTER 1

INTRODUCTION TO VAX LISP

LISP is a general purpose programming language. The language has been used extensively in the field of artificial intelligence for research and development of robotics, expert systems, natural-language processing, game playing, and theorem proving. The LISP language is characterized by:

- Computation with symbolic expressions and numbers
- Simple syntax
- Representation of data by symbolic expressions or multilevel lists
- Representation of LISP programs as LISP data, which enables data structures to execute as programs and programs to be analyzed as data
- A function named EVAL, which is the language's definition and interpreter
- Automatic storage allocation and garbage collection

VAX LISP is implemented on both the VMS and the ULTRIX-32 operating systems. VAX LISP as implemented on the VMS operating system is formally named VAX LISP/VMS. VAX LISP as implemented on the ULTRIX operating system is formally named VAX LISP/ULTRIX. Both VAX LISP/ULTRIX and VAX LISP/VMS are the same language but with some differences. For the differences, see the *VAX LISP/VMS Release Notes*. These are on-line in the file SYS$HELP:LISP*nnn*.RELEASE_NOTES, where *nnn* is the VAX LISP version number. For example, LISP020.RELEASE_NOTES is the file containing the release notes for Version 2.0.

This manual describes VAX LISP/VMS, but refers to VAX LISP/VMS as VAX LISP, where practicable.

This chapter provides an overview of the VAX LISP language. The overview is arranged so that it parallels the structure of this manual

and the remaining VAX LISP documentation. In addition to the overview, the chapter explains how to get on-line help at the DCL and the LISP language levels of operation and describes:

- VAX/VMS file specifications
- Logical names
- Hints for entering DCL commands

1.1 OVERVIEW OF VAX LISP

The VAX LISP language is an extended implementation of the COMMON LISP language defined in *COMMON LISP: The Language*. In addition to the features supported by COMMON LISP, VAX LISP provides the following extensions:

- DCL (DIGITAL Command Language) LISP command
- Extensible editor
- Error handler
- Debugging facilities
- Extensible pretty printer
- Facility for calling out to external routines
- Facility for defining non-LISP data structures (alien structures)
- Facility for defining interrupt functions (that is, functions that execute asynchronously)
- Window and graphics support for the VAXstation II workstation
- Utility for creating custom LISP systems

These extensions are described in Sections 1.1.1 through 1.1.10.

VAX LISP does not support complex numbers. However, you can manipulate complex numbers by using the alien structure and call-out facilities.

Some of the functions, macros, and facilities defined by COMMON LISP are modified for the VAX LISP implementation. Chapter 7 provides implementation-dependent information about the following topics:

- Data representation

- Pathnames
- Garbage collector
- Input and output
- Asynchronous functions
- Compiler
- Functions and macros

The implementation-dependent functions and macros mentioned in *COMMON LISP: The Language* are defined in Part II.

VAX LISP also supplies a number of functions that are extensions of the I/O system defined in *COMMON LISP: The Language*, as well as a means of defining new types of streams. These extensions are described in Chapter 8.

1.1.1 DCL LISP Command

The DCL LISP command invokes VAX LISP from the VMS command level. Depending on the qualifier you use with the LISP command, you can start the LISP interpreter or the LISP compiler. Chapter 2 describes the LISP command and the qualifiers you can use with it. Chapter 2 also explains how to:

- Invoke LISP
- Exit LISP
- Create programs
- Load files
- Compile programs
- Use suspended systems

1.1.1.1 **Interpreter** - The VAX LISP interpreter reads an expression, evaluates the expression, and prints the results. You interact with the interpreter in line-by-line input.

While in the interpreter, you can create LISP programs. You can also use programs that are stored in files if you load the files into the interpreter. Chapter 2 explains how to create LISP programs and how to load files into the VAX LISP interpreter.

1.1.1.2 **Compiler** - The VAX LISP compiler is a LISP program that translates LISP code from text to machine code. Because of the translation, compiled programs run faster than interpreted programs.

You can use the compiler to compile a single function or macro or to compile a LISP source file. If you are in the LISP interpreter, you can compile a single function or macro with the COMPILE function (see Chapter 2).

You can compile a source file either at the VMS command level or the LISP level of operation. If you are at the VMS command level, you must specify the LISP DCL command with the /COMPILE qualifier; if you are in the LISP interpreter, you must invoke the COMPILE-FILE function. Chapter 2 explains how to compile LISP programs that are stored in files.

1.1.2 Editor

VAX LISP includes a screen-oriented editor. You can use it to edit text files, and functions and macros that are defined in the LISP system. The Editor provides specialized commands to help you edit LISP code; they balance parentheses, properly indent text, and evaluate LISP text. Chapter 3 describes how to use the Editor to write and edit LISP code.

The Editor is written in LISP, so you can extend and customize it for your needs. The Editor provides predefined commands and several functions, macros, and data structures, which you can use to create Editor commands. After you create an Editor command, you can bind it to a key on your terminal keyboard. In this way, you can build up alternative editing systems or complete applications based on the Editor. See the *VAX LISP Editor Programming Guide* for more information on programming the Editor.

1.1.3 Error Handler

VAX LISP contains an error handler, which is invoked when errors occur during the evaluation of a LISP program. Chapter 4 describes the error handler and explains how you can create your own error handler.

1.1.4 Debugging Facilities

VAX LISP provides several functions and macros that return or display information you can use when you are debugging a program. VAX LISP also provides four debugging facilities: the break loop, debugger, stepper, and tracer.

The functions that return debugging information and the break loop, stepper, and tracer facilities are defined in COMMON LISP and are extended in VAX LISP. The break loop lets you interrupt the evaluation of a program, the stepper lets you use commands to step through the evaluation of each form in a program, and the tracer lets you examine the evaluation of a program.

The debugger is a VAX LISP facility. The facility provides commands that let you examine and modify the information in the LISP system's control stack frames.

Chapter 5 explains how to use the debugging facilities.

1.1.5 Pretty Printer

VAX LISP provides a pretty printer facility. You can use the facility to control the format in which LISP objects are printed. The pretty printer can be helpful in making objects easier to understand by means of indentation and spacing. You can use the pretty printer with the existing defaults, control it with control variables, or extend it by using special directives with the FORMAT function. Chapter 6 explains how to use the pretty printer in each of these ways.

1.1.6 Call-Out Facility

VAX LISP includes a call-out facility, which lets you call programs written in other VAX/VMS programming languages and programs that include run-time library (RTL) routines and VMS and RMS system services. Chapter 2 of the *VAX LISP/VMS System Access Programming Guide* describes the call-out process and explains how to use the call-out facility.

1.1.7 Alien Structure Facility

VAX LISP supplies an alien structure facility. It lets you define, create, and access VAX data structures that are used to communicate between the VAX LISP language and other VAX/VMS languages or system services. Chapter 3 of the *VAX LISP/VMS System Access Programming Guide* describes the alien structure facility and explains how to use it.

1.1.8 Interrupt Function Facility

VAX LISP allows you to define functions that can execute at arbitrary and unpredictable points in your program, usually as the result of an event in the operating system. Such functions are called interrupt functions, because they interrupt the normal flow of program execution. Chapter 4 of the *VAX LISP/VMS System Access Programming Guide* describes how to define and use interrupt functions.

1.1.9 VAXstation Graphics Interface

VAX LISP/VMS provides access to the graphics capabilities of the VAXstation II family of workstations. You can create windows on the screen, draw lines and write text in the windows, track the workstation's pointing device and react to pointer buttons, and create LISP streams to windows. The *VAX LISP/VMS Graphics Programming Guide* describes this interface.

1.1.10 System-Building Utility

The VAX LISP System-Building Utility lets you create custom VAX LISP systems. A custom VAX LISP system has the following potential advantages:

- It can exclude various components of VAX LISP, thereby reducing the size of LISP.
- It can include code that you write.
- It can start execution by calling a function that you specify.
- It can be used as a delivery vehicle for a VAX LISP-based application.

The *VAX LISP/VMS System-Building Guide* describes the System-Building Utility.

1.1.11 VAX LISP/VMS Function, Macro, and Variable Descriptions

VAX LISP/VMS contains many functions, macros, and variables that are either not mentioned or are mentioned but not fully defined in the COMMON LISP language. These functions, macros, and variables are divided into the following categories:

- Implementation-dependent objects mentioned but not fully defined in *COMMON LISP: The Language*

- VAX LISP objects that implement the parts of VAX LISP that are described in this manual
- VAX LISP extensions to the COMMON LISP I/O system
- Editor-specific objects
- System access-specific objects (pertaining to the call-out, alien structure, interrupt function, and program synchronization facilities)
- Graphics-specific objects
- Objects that implement the VAX LISP System-Building Utility

These LISP objects let you use the VAX LISP facilities and some VMS facilities without exiting or calling out from the LISP system.

The LISP objects in the first two categories listed above are described in Part II. VAX LISP extensions to COMMON LISP I/O are described in Chapter 8. Editor-specific objects are described in Part III of the *VAX LISP/VMS Editor Programming Guide*. System access-specific objects are described in Part II of the *VAX LISP/VMS System Access Programming Guide*. Graphics-specific objects are described in Part II of the *VAX LISP/VMS Graphics Programming Guide*. The VAX LISP System-Building Utility is described in the *VAX LISP/VMS System-Building Guide*.

1.2 HELP FACILITIES

When using VAX LISP, you can get help at both the DCL and the LISP levels of operation.

1.2.1 DCL HELP

The VAX/VMS help facility lets you obtain on-line information about a DCL command, its parameters, and its qualifiers. Invoke the help facility by entering the HELP command. When the HELP command is executed, the facility displays the information available.

To obtain information about VAX LISP, enter the following command:

```
$ HELP LISP
```

1.2.2 LISP HELP

VAX LISP provides two functions you can use to obtain help during a LISP session: DESCRIBE and APROPOS. The DESCRIBE function displays information about a specified LISP object. The type of information the function displays depends on the object you specify as its argument. You can use the APROPOS function to search through a package for symbols whose print names contain a specified string. See *COMMON LISP: The Language* for information about packages. Descriptions of the DESCRIBE and APROPOS functions are provided in Part II.

1.3 VAX/VMS FILE SPECIFICATIONS

A VAX/VMS file specification indicates the input file to be processed or the output file to be produced. File specifications have the following format:

node::*device*:[*directory*]*filename*.*filetype*;*version*

A file specification has the following components:

node

The name of a network node. The name can be either an integer or a string and can include an access control string. The following node name includes an access control string:

```
MIAMI"SMITH MYPASSWORD"::
```

This component applies only to systems that support DECnet-VAX.

device

The name of the device on which the file is stored or is to be written.

directory

The name of the directory under which the file is cataloged. The name must be a string. You can delimit the directory name with either square brackets ([ ]) or angle brackets (< >).

You can specify a sequence of directory names where each name represents a directory level. For example:

```
[SMITH.EXAMPLES]
```

In the preceding directory specification, EXAMPLES represents a subdirectory.

filename

The name of the file.

filetype

An abbreviation that usually describes the type of data in the file.

version

An integer that specifies which version of the file is desired. The version number is incremented by one each time you create a new version of the file. You can use either a semicolon (;) or a period (.) to separate the file type and version.

The punctuation marks (colons, brackets, period, and semicolon) in the file specification format are required. The marks separate the components of the file specification.

You do not have to supply all the components of a file specification each time you compile a file, load an initialization file, or resume a suspended system. The only component you must specify is the file name; the operating system supplies default values for the components that you do not specify. Table 1-1 summarizes the default values. The special variable *DEFAULT-PATHNAME-DEFAULTS* contains the default values for the *node*, *device*, and *directory* elements.

The way the system fills in default values depends on the operation being performed. For example, if you specify only a file name, the compiler processes the source program if it finds a file with the specified file name that is stored on the default device, is cataloged under the default directory name, and has an LSP file type. If more than one file meets these conditions, the compiler processes the file with the highest version number. Suppose you pass the following file specification to the compiler:

```
$ LISP/COMPILE DBA1:[SMITH]CIRCLE.LSP
```

The compiler searches directory SMITH on device DBA1, seeking the highest version of CIRCLE.LSP. If you do not specify an output file, the compiler generates the file CIRCLE.FAS, stores it in directory SMITH on device DBA1, and assigns it a version number that is one higher than any version of CIRCLE.FAS cataloged in directory SMITH on device DBA1.

Table 1-1: File Specification Defaults

Optional Element	Default Value
node	Local network node
device	User's current default device
directory	User's current default directory
filename	Input -- None Output -- Same as input file; if no input file is specified, there is no default
filetype	Depends on usage: FAS -- Fast-loading file (output from compiler) LIS -- Error listing (output from compiler) *LSC -- Editor checkpointing file LSP -- Source file (input to LISP reader or compiler) SUS -- Suspended system
version	Input -- Highest existing version number Output -- If no existing version, 1 If existing version, highest version number plus 1

1.4 LOGICAL NAMES

The VAX/VMS operating system provides a logical name mechanism that allows programs to be device and file independent. Programs do not have to specify the device on which a file resides or the name of the file that contains data if you use logical names. Logical names provide great flexibility, because you can associate them not only with a device or a complete file specification but also with a directory or another logical name.

For more information on logical names, see the *Guide to Using DCL and Command Procedures on VAX/VMS*.

1.5 ENTERING DCL COMMANDS

This section lists hints for entering DCL commands.

- You can abbreviate command and qualifier names to four characters. You can use fewer than four characters if the abbreviation is unambiguous.

- You must precede each qualifier name with a slash (/).
- If you omit a required parameter (for example, a file specification), the DCL command interpreter prompts you for the parameter.
- You can enter a command on more than one line if you end each continued line with a hyphen (-).
- You must press the RETURN key after you enter a command; pressing the RETURN key passes the command to the system for processing.
- You can delete the current command line by typing CTRL/U.
- You can interrupt command execution by typing CTRL/Y. If you do not enter a command that executes another image, you can resume the interrupted command by entering the DCL CONTINUE command. To stop processing completely after typing CTRL/Y, enter the DCL STOP command.

CHAPTER 2

USING VAX LISP

This chapter describes the DCL LISP command and its qualifiers and explains the following:

- Invoking LISP
- Exiting LISP
- Using command levels
- Controlling input
- Creating programs
- Loading files
- Compiling programs
- Using suspended systems

2.1 INVOKING LISP

You invoke an interactive VAX LISP session by typing the DCL command LISP. When it is executed, a message identifying the VAX LISP system appears, and then the LISP prompt (Lisp>) is displayed. For example:

```
$ LISP

Welcome to VAX LISP, Version 1.0

Lisp>
```

2.2 EXITING LISP

You can exit from LISP by using the LISP EXIT function. For example:

```
Lisp> (EXIT)
$
```

When you exit the LISP system, you are returned to the DCL level of operation. If you have used the Editor, modified buffers are not saved on exiting LISP. See the _VAX LISP Editor Manual_ for information on how to save modified buffers before exiting LISP.

2.3 USING COMMAND LEVELS

VAX LISP provides various facilities with which you can interact. The most apparent facility is the top-level read-eval-print loop, which provides the basic means by which you write programs and execute them. The break-loop facility enables you to temporarily suspend the top-level loop and establish another read-eval-print loop. The debugger and stepper facilities provide support for testing and debugging programs. Chapter 4 describes these facilities.

When the break loop, the debugger, or the stepper is invoked by means of a function call, an error, or some other event, the facility establishes a "command level." A command level represents a point of interaction with the user, and each such level is assigned a number. The highest-numbered level represents the current level of interaction, while lower-numbered levels represent interactions that have been temporarily suspended. When a facility prompts you for input, the prompt includes the facility's name and the command level number. The exception to this rule is the top-level read-eval-print loop, which is always at level zero and therefore does not include the number in its prompt.

Nothing prevents the same facility from being invoked more than once, so that there can be multiple command levels representing the same facility. For example:

```
Lisp> (BREAK)
Break 1> (+ *COUNTER* 1)

Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Symbol has no value: *COUNTER*

Control Stack Debugger
Frame #7: (EVAL (+ *COUNTER* 1))
Debug 2> (BREAK)
Break 3> (DESCRIBE *COUNTER*)

Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Symbol has no value: *COUNTER*

Control Stack Debugger
Frame #19: (EVAL (DESCRIBE *COUNTER*))
Debug 4> ...
```

In this example, the user invokes a break loop and makes an attempt to use the special variable *COUNTER*, which has no value, causing the debugger to be invoked. Then the user invokes another break loop and accidentally makes the same mistake again, causing another debugger level to be invoked. This example is not particularly realistic, but is only meant as an illustration of command levels and their numbering.

The THROW-TO-COMMAND-LEVEL function can be used to cancel one or more command levels and return control to a previous one. Typing CTRL/C always cancels all command levels and returns control to the top-level loop. The THROW-TO-COMMAND-LEVEL function is described in Part II.

2.4 CONTROLLING INPUT

You enter input into the VAX LISP system a line at a time. Once you move to a new line, you cannot go back to the previous line. However, you can recover an input expression or an output value by using the following 10 unique variables: /, //, ///, *, **, ***, +, ++, +++, -. The variables are described in COMMON LISP: The Language. The following example illustrates the use of the plus sign (+) variable that is bound to the expression most recently evaluated:

```
Lisp> (CDR '(A B C))
(B C)
Lisp> +
(CDR (QUOTE (A B C)))
Lisp>
```

You can use the DELETE key and several control characters on your terminal keyboard to control input. The DELETE key enables you to delete characters that are to the left of the cursor on the current line of input. Table 2-1 lists the control characters you can use and their functions. The first control character in the list, CTRL/C, is the only one whose listed function is specific to LISP. The other control characters perform standard VMS functions.

Table 2-1
Control Characters

Control Character	Function
CTRL/C	Returns you to the top-level loop from any other command level. In LISP, CTRL/C is bound to the form (THROW-TO-COMMAND-LEVEL :TOP). Pressing CTRL/C is also a quick way to recover from an error without using the VAX LISP debugger. If you want to recover from an error by discarding the expression you typed in and starting over, type CTRL/C.
CTRL/O	Discards output being sent to the terminal until you type another CTRL/O.
CTRL/Q	Resumes terminal output that had been halted with CTRL/S.
CTRL/R	Redisplays what is on a line.
CTRL/S	Stops output to the terminal until a CTRL/Q is typed.
CTRL/T	Displays process information. This is useful during a computation to watch the resources used.
CTRL/U	Deletes the current input line. The prompt is not echoed in LISP.
CTRL/X	Deletes all input that has not yet been read from the type-ahead buffer.
CTRL/Y	Returns you to DCL level of control and purges the type-ahead buffer.

NOTE

The preceding control characters do not work in the VAX LISP Editor. For additional information on control characters, see the VAX/VMS I/O User's Guide (Volume 1).

2.5 CREATING PROGRAMS

The most common way of creating a LISP program is to compose it with a text editor. In this way, the program exists in a source file that can be loaded into the LISP environment by means of the LISP LOAD function.

Although you can compose source programs with any text editor, the VAX LISP Editor provides facilities that help you enter and edit LISP source code. For example, the Editor helps you balance parentheses and maintain proper indentation. Furthermore, this editor, being integrated into the LISP environment, can be extended with various features that fit your own style of editing. See the VAX LISP Editor Manual for a complete description of the Editor.

Another way to create LISP programs is to define them using the interpreter in an interactive LISP session. If you define functions with the DEFUN macro or macros with the DEFMACRO macro, the definitions become a part of the interpreted LISP environment. You can then invoke your defined functions and macros. However, since these definitions are not in a permanent text file, your work is stored only temporarily and disappears when you exit VAX LISP. Entering programs by means of the interpreter is really only useful for experimenting with small functions and macros.

The following LISP definition of the FACTORIAL function is an example of a LISP program. It can be written in the following format in a file or in an interactive LISP session:

```
(DEFUN FACTORIAL (N)
  (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
```

DEFUN indicates that this is a function definition. FACTORIAL is the name of the function. (N) is the argument list; that is, FACTORIAL has one argument, N. When FACTORIAL is called, the code following the argument list is evaluated and the last result computed is returned as the value of the function.

2.6 LOADING FILES

Before you can use a file in interactive LISP, you must load the file into the LISP system. You can load a file into the LISP system in several ways:

- Load the file by specifying the DCL LISP /INITIALIZE qualifier.

Table 2-2 (cont.)

Control Character	Function
CTRL/S	Stops output to the terminal until a CTRL/Q is typed.
CTRL/T	Displays process information. This is useful during a computation to watch the resources used.
CTRL/U	Deletes the current input line. The prompt is not echoed in LISP.
CTRL/X	Deletes all input that has not yet been read from the type-ahead buffer.
CTRL/Y	Returns you to the DCL level of control and purges the type-ahead buffer.

2.7 CREATING PROGRAMS

The most common way to create a LISP program is by using a text editor. In this way, the program exists in a source file that can be loaded into the LISP environment by the LISP LOAD function.

Although you can compose source programs with any text editor, the VAX LISP Editor provides facilities that help you enter and edit LISP source code. For example, the Editor helps you balance parentheses and maintain proper indentation. Furthermore, this editor, being integrated into the LISP environment, can be extended with features that fit your own style of editing. See Chapter 3 for a description of how to use the Editor.

Another way to create LISP programs is to define them using the interpreter in an interactive LISP session. If you define functions with the DEFUN macro or macros with the DEFMACRO macro, the definitions become a part of the interpreted LISP environment. You can then invoke your defined functions and macros. However, since these definitions are not in a permanent text file, your work is stored only temporarily and disappears when you exit VAX LISP. Entering programs by typing to the interpreter is really useful only for experimenting with small functions and macros.

2.8 LOADING FILES

Before you can use a file in interactive LISP, you must load the file into the LISP system. The file can be compiled or interpreted;

compiled files load more quickly. You can load a file into the LISP system in three ways:

- Load the file by specifying the DCL LISP INITIALIZE qualifier. For example:

  ```
  $ LISP/INITIALIZE=MYINIT.LSP

  Welcome to VAX LISP, Version V2.0

  Lisp>
  ```

  The LISP prompt indicates the file has been successfully loaded. If the file is not successfully loaded, an error message indicating the reason appears on your terminal screen. Include the /VERBOSE qualifier to cause the names of functions loaded in an initialization file to be listed at the terminal. For more information on the /VERBOSE qualifier, see Section 2.10.12.

- Load the file by using the LISP LOAD function when in an interactive LISP session. For example:

  ```
  Lisp> (LOAD "TESTPROG.LSP")
  ; Loading contents of file DBA1:[JONES]TESTPROG.LSP;1
  ;   FACTORIAL
  ;   FACTORS-OF
  ; Finished loading DBA1:[JONES]TESTPROG.LSP;1
  T
  Lisp>
  ```

  The file name ("TESTPROG.LSP" in the example) can be a string, symbol, stream, or pathname. FACTORIAL and FACTORS-OF are the functions contained in the file TESTPROG.LSP. The final T indicates that the file has been successfully loaded. For more information on the LOAD function, see Part II.

- Evaluate the contents of a buffer in the Editor when that buffer contains a file. See Chapter 3 for more information on this topic.

With the /INITIALIZE qualifier, you can load more than one file at a time. With the LOAD function, however, you can specify only one file at a time.

2.9 COMPILING PROGRAMS

You compile LISP programs by compiling the LISP expressions that make up the programs. You can compile LISP expressions in two ways: individually, by using the LISP COMPILE function; or in a file, by

2.7.3 The Advantages of Compiling LISP Expressions

You can use both compiled and uncompiled (interpreted) files and functions during a LISP session. Both compiled and uncompiled LISP expressions have their advantages. The advantages of compiling a file, a macro, or a function follow:

- Compiling a function or a macro is a good initial debugging tool, since the compilation does static error checking, such as checking the number of arguments to a function or a macro. For example, consider the following function definition:

  ```
  (DEFUN TEST (X)
    (IF (> X 0)
        (+ 1 X)
        (TEST (TRY X) X)))
  ```

  In the definition of the function TEST, the alternate consequent (the false part) of the IF condition has two arguments, while the function definition of TEST calls for only one argument. Despite this error, this function might work correctly as an interpreted (uncompiled) function if the argument given is a positive number, since it uses only the first consequent (the true part); so you may not detect the error. But if you compiled the function, the compiler would detect the error in the second consequent and issue a warning.

- A compiled file not only loads much faster, but the compiled code executes significantly faster than the corresponding interpreted code.

2.7.4 The Advantage of Not Compiling LISP Expressions

You can debug run-time errors in an interpreted function more easily than you can debug them in a compiled file or function. For example, if the debugger is invoked because an error occurred in an uncompiled function, you can use the debugger to find out what code caused the error. If the debugger is invoked because an error occurred in a compiled function, the code surrounding the form that caused an error to be signaled may not be accessible. The stepper facility is also more informative with interpreted than with compiled functions. See Sections 4.4 and 4.5, respectively, for information on the debugger and the stepper.

2.8 DCL LISP COMMAND QUALIFIERS

The LISP command can be specified with several qualifiers according to the standard VMS conventions. The format of the LISP command with qualifiers follows:

```
LISP[/qualifier...]
```

Some qualifiers have a corresponding negative form, /NOqualifier, which negates the specified action. Other qualifiers accept values. To specify a qualifier value, type the qualifier name followed by an equal sign (=) and the value. For example:

```
/INITIALIZE=MYPROG.LSP
```

Qualifier values are surrounded by braces ({ }) when you can choose only one value from a list. For example:

```
/ERROR_ACTION={EXIT or DEBUG}
```

To specify a list of qualifier values, enclose the values in parentheses. For example:

```
/INITIALIZE=(MYPROG1.LSP,MYPROG2.LSP)
```

Table 2-2 summarizes the qualifiers you can use with the LISP command. Sections 2.8.2 through 2.8.15 describe each qualifier in detail.

Table 2-2
DCL LISP Command Qualifiers

Qualifier	Function
/COMPILE	Invokes the VAX LISP compiler to compile one or more source files.
/ERROR_ACTION={EXIT or DEBUG}	EXIT causes your program to exit LISP when an error occurs. EXIT is the default in batch mode jobs and in compile mode. DEBUG invokes the VAX LISP debugger when an error occurs. DEBUG is the default in interactive mode.
/INITIALIZE=(file-spec,...)	Causes the LISP system to load an initialization file(s). The default file type for an initialization file is LSP or FAS.
/INTERACTIVE	Starts an interactive LISP session. This is the default.
/INSTALL=suspended-system-spec	Causes the read-only code in the LISP suspended system to be shareable. The default file type for a suspended system file is SUS.
/[NO]LIST=[file-spec]	Specifies that a listing file be made. A listing consists of the file name, date of compilation, names of the LISP expressions compiled, and warning and error messages. The default file type for a listing file is LIS. /NOLIST suppresses a listing file and is the default except in batch mode. /LIST is the default for batch mode operations.
/[NO]MACHINE_CODE	Includes VAX LISP machine code in the listing file. /NOMACHINE_CODE suppresses a listing of the machine code and is the default. If /MACHINE_CODE and /NOLIST are both specified, /NOLIST is ignored.

(Continued on next page)

2.9.3 Advantages of Compiling LISP Expressions

You can use both compiled and uncompiled (interpreted) files and functions during a LISP session. Both compiled and uncompiled LISP expressions have their advantages. The advantages of compiling a file, a macro, or a function follow:

- Compiling a function or a macro is a good initial debugging tool, since the compilation does static error checking, such as checking the number of arguments to a function or a macro. For example, consider the following function definition:

  ```
  (DEFUN TEST (X)
    (IF (> X 0)
        (+ 1 X)
        (TEST (TRY X) X)))
  ```

  In the definition of the function TEST, the alternate consequent (the false part) of the IF condition invokes TEST with two arguments, (TRY X) and X, while the function definition of TEST calls for only one argument. Despite this error, TEST might work correctly as an interpreted (uncompiled) function if the argument given is a positive number, since it uses only the first consequent (the true part); so you may not detect the error. But if you compiled the function, the compiler would detect the error in the second consequent and issue a warning.

- A compiled file not only loads much faster, but the compiled code executes significantly faster than the corresponding interpreted code.

2.9.4 Advantage of Not Compiling LISP Expressions

You can debug run-time errors in an interpreted function more easily than you can debug them in a compiled file or function. For example, if the debugger is invoked because an error occurred in an uncompiled function, you can use the debugger to find out what code caused the error. If the debugger is invoked because an error occurred in a compiled function, the code surrounding the form that caused an error to be signaled may not be accessible. The stepper facility is also more informative with interpreted than with compiled functions. See Chapter 5 for information on the debugger and the stepper.

2.10 DCL LISP COMMAND QUALIFIERS

The LISP command can be specified with several qualifiers according to the standard VMS conventions. The format of the LISP command with

qualifiers follows:

```
LISP[/qualifier...]
```

Some qualifiers have a corresponding negative form, /NOqualifier, which negates the specified action. Other qualifiers accept values. To specify a qualifier value, type the qualifier name followed by an equal sign (=) and the value. For example:

```
/INITIALIZE=MYPROG.LSP
```

Qualifier values are surrounded by braces ({ }) when you can choose only one value from a list. For example:

```
/ERROR_ACTION={EXIT or DEBUG}
```

To specify a list of qualifier values, enclose the values in parentheses. For example:

```
/INITIALIZE=(MYPROG1.LSP,MYPROG2.LSP)
```

You can define DCL symbols to represent LISP command lines that you use frequently. For example:

```
$ BIGLISP :== LISP/INITIALIZE=SYS$LOGIN:LISPINIT/MEMORY=10000
```

Following this command, the DCL symbol BIGLISP, when typed at the DCL prompt, results in execution of the LISP command line shown.

Table 2-3 summarizes the qualifiers you can use with the LISP command. Sections 2.10.2 through 2.10.13 describe each qualifier in detail.

Table 2-3: DCL LISP Command Qualifiers

Qualifier	Function
/COMPILE	Invokes the VAX LISP compiler to compile one or more source files (input arguments that default to the file type LSP).
/ERROR_ACTION={EXIT or DEBUG}	EXIT causes your program to exit LISP when an error occurs. EXIT is the default in batch mode jobs and in compile mode (with the /COMPILE qualifier). DEBUG invokes the VAX LISP debugger when an error occurs. DEBUG is the default in an interactive LISP session.

Table 2-3 (cont.)

Qualifier	Function
/[NO]INITIALIZE=(file-spec,...)	Causes the LISP system to load an initialization file(s). The default file type for an initialization file is LSP or FAS. NOINITIALIZE suppresses the loading of initialization files.
/INTERACTIVE	Starts an interactive LISP session. /INTERACTIVE is the default qualifier for the LISP command.
/[NO]LIST=[file-spec]	Specifies that a listing file be created during compilation. A listing consists of the file name, date of compilation, names of the LISP expressions compiled (if the /VERBOSE qualifier is specified), and warning and error messages. The default file type for a listing file is LIS. /NOLIST suppresses a listing file and is the default except in batch mode. In such jobs, /LIST is the default.
/[NO]MACHINE_CODE	Includes VAX LISP machine code in the listing file. /NOMACHINE_CODE suppresses a listing of the machine code and is the default.
/MEMORY=number	Specifies the amount of dynamic virtual memory LISP allocates in 512-byte pages.
/[NO]OPTIMIZE=(SPEED:*n*,SPACE:*n*, SAFETY:*n*,COMPILATION_SPEED:*n*)	Tells the compiler that each quality has the corresponding value. SPEED is the speed at which the object code runs, SPACE is the space occupied or used by the code, SAFETY is the run-time error checking of the code, and COMPILATION_SPEED is the speed of the compilation process. *n* is an integer in the range 0 to 3. The value 0 is the lowest priority value; the value 3 is the highest. The default value for *n* is 1. See Chapter 7 for a description of optimization declarations.

Table 2-3 (cont.)

Qualifier	Function
/[NO]OUTPUT_FILE=[file-spec]	Causes the name of the compiled file to be the specified name. The default output file type is FAS. /NOOUTPUT prevents compiled code from being written to a file. /OUTPUT_FILE is the default.
/RESUME=file	Resumes a suspended LISP system. The default file type for a suspended LISP system is SUS. See Section 2.11 on Using Suspended Systems.
/[NO]VERBOSE	Lists on the output device and the listing file, if any, the names of functions and macros defined in a file. /NOVERBOSE suppresses a listing of function and macro names defined in a file. /NOVERBOSE is the default.
/[NO]WARNINGS	Specifies that the compiler is to produce warning messages. /NOWARNINGS suppresses warning messages. /WARNINGS is the default.

2.10.1 Three Ways to Use the DCL LISP Command

Depending on the qualifier modifying it, you can use the DCL LISP command in one of the following three ways called modes:

- INTERACTIVE -- to invoke an interactive LISP session (the default)
- COMPILE -- to compile LISP files
- RESUME -- to resume a suspended LISP system

Table 2-4 lists the LISP command qualifiers that apply to each mode. Without a qualifier, the DCL LISP command puts you in an interactive session (the default).

Table 2-4: DCL LISP Command Qualifier Modes

Qualifier	Mode
/COMPILE	COMPILE
/ERROR_ACTION	INTERACTIVE or COMPILE or RESUME
/[NO]INITIALIZE	INTERACTIVE or COMPILE
/INTERACTIVE	INTERACTIVE
/[NO]LIST	COMPILE
/[NO]MACHINE_CODE	COMPILE
/MEMORY	INTERACTIVE or COMPILE or RESUME
/[NO]OPTIMIZE	COMPILE
/[NO]OUTPUT_FILE	COMPILE
/RESUME	RESUME
/[NO]VERBOSE	INTERACTIVE or COMPILE
/[NO]WARNINGS	COMPILE

2.10.2 /COMPILE

The /COMPILE qualifier invokes the VAX LISP compiler to compile one or more source files. The compiler creates a fast-loading (FAS) file from each source file. Unlike other compilers, such as those for BASIC and COBOL, the LISP compiler does not generate VMS object modules. Consequently, the LISP compiler does not have an object file type. FAS is the default file type for a LISP compiled file. If the /COMPILE qualifier is used with the /NOOUTPUT_FILE qualifier, the compiler compiles the source file but does not put the compilation in a file. That method is helpful if your purpose in compiling the file is to check for errors. See Section 2.10.10 for more information on the /[NO]OUTPUT_FILE qualifier.

By default, the compiler gives your newly compiled file the same name as your source file with a FAS file type, puts the new file in your source file's directory, and returns you to DCL command level when the compiler is finished. If you want functions to be listed on your output device as they are compiled, you must specify the /VERBOSE qualifier (see Section 2.10.12). If you want to compile files with the aid of initialization files, use the /INITIALIZE qualifier (see

Section 2.10.4). For information on how to load files, see Section 2.8.

If you do not specify a file name with the /COMPILE qualifier, DCL prompts you for a file name. If you use the qualifiers /[NO]LIST, /[NO]MACHINE_CODE, /OPTIMIZE, /[NO]OUTPUT, /[NO]VERBOSE, and /[NO]WARNINGS with the /COMPILE qualifier and you specify them before the files to be compiled, the qualifiers apply to all the files to be compiled. If you use the preceding qualifiers with the /COMPILE qualifier, but you specify them after a file name, the qualifiers apply only to the immediately preceding file. If you specify qualifiers for all the files and a conflicting qualifier for a particular file, the LISP system uses the qualifier specified for the particular file.

Format

LISP/COMPILE *file-spec*[,...]

Example

```
$ LISP/COMPILE FACTORIAL.LSP
$
```

Mode

Compile

2.10.3 /ERROR_ACTION

The /ERROR_ACTION qualifier has two values: EXIT and DEBUG.

- EXIT causes the evaluation of your program to stop and exits LISP if a fatal or a continuable error occurs (for a complete description of errors and warnings, see Chapter 4). EXIT is the default in batch mode and in compile mode, that is, with the /COMPILE qualifier.

- DEBUG calls the VAX LISP debugger if an error occurs. Once you are in the VAX LISP debugger, you can look at your error, inspect the control stack, and continue your program from the point at which it stopped. DEBUG is the default in an interactive session. See Chapter 5 for more information on the debugger.

You can use the /ERROR_ACTION qualifier when invoking an interactive LISP session or when compiling files with the /COMPILE qualifier. The /ERROR_ACTION qualifier is mainly useful for batch jobs. It is equivalent to the VAX LISP *ERROR-ACTION* variable (see Part II).

Format

```
LISP/ERROR_ACTION=value
```

Example

```
$ LISP/COMPILE/ERROR_ACTION=DEBUG MYPROG.LSP
```

Mode

Interactive, Compile, or Resume

2.10.4 /[NO]INITIALIZE

The /INITIALIZE qualifier causes the LISP system to load one or more initialization files containing LISP source code or compiled code. An initialization file's purpose is to predefine functions you might want to use in a LISP session. The default is to have no initialization file.

If the initialization files contain calls to exiting functions or if these files contain errors and the /ERROR_ACTION qualifier is set to EXIT (/ERROR_ACTION=EXIT), the LISP system returns to the DCL level without prompting for interactive input. If the initialization files contain errors and the /ERROR_ACTION qualifier is set to DEBUG (/ERROR_ACTION=DEBUG), the LISP system puts you into the debugger. See Section 2.10.3 for more information on the /ERROR_ACTION qualifier.

The /INITIALIZE qualifier uses the LISP LOAD function to default the proper type, directory, and other parts of a file specification. For example, you do not have to specify the file type if your initialization file has a FAS or a LSP file type. If your directory contains a file name with both a FAS and a LSP file type, the LISP system selects the most recently created file as the initialization file. If only a LSP type file or only a FAS type file of a given name and directory exists, the LISP system selects the type file that exists.

Use the /VERBOSE qualifier (see Section 2.10.12) to display on the terminal screen the names of any functions or macros in the initialization file.

You can use the /INITIALIZE qualifier when invoking an interactive LISP session or when compiling files with the /COMPILE qualifier. You cannot use the /INITIALIZE qualifier with the /RESUME qualifier; if you do so, the /INITIALIZE qualifier is disregarded.

Format

LISP/INITIALIZE=(*file-spec,...*)

or

LISP/COMPILE/INITIALIZE=(*file-spec,...*) *file-spec*

Example

```
$ LISP/INITIALIZE=MYINIT/VERBOSE

Welcome to VAX LISP, Version V2.0

; Loading contents of file DBA1:[JONES]MYINIT.LSP;1
;   FACTORIAL
;   FACTORS-OF
; Finished loading DBA1:[JONES]MYINIT.LSP;1
*
```

In the preceding example, the file type defaults to LSP. FACTORIAL and FACTORS-OF are functions that are loaded into the LISP system from Jones's initialization file. The form (SETF *TOP-LEVEL-PROMPT* "*") in the initialization file changes the Lisp> prompt to an asterisk (*). The *TOP-LEVEL-PROMPT* variable is described in Part II.

The SETF form and the prompt variable are not listed on an output device when the file is loaded, because the /VERBOSE qualifier lists only functions and macros defined in the file.

Mode

Interactive or Compile

2.10.5 /INTERACTIVE

The /INTERACTIVE qualifier, the default, starts an interactive LISP session.

Mode

Interactive

2.10.6 /[NO]LIST

The /LIST qualifier is meaningful only if it is specified with the /COMPILE qualifier. The /LIST qualifier specifies that the compiler generate a listing file during compilation. You must specify this qualifier if you want a listing file. A listing includes the name of the file compiled, the date it was compiled, warning or error messages produced during compilation, and a summary of warning and error messages. If you specify the /VERBOSE qualifier with the /LIST qualifier, the listing also includes the names of the functions compiled.

Specify the /LIST qualifier with a file name value only when you want the listing file name to be different from the name of the source file. If you specify the /LIST qualifier without a file name, the LISP system produces a listing file with a LIS file type and the same name as the source file.

The /NOLIST qualifier suppresses a listing and is the default except in batch mode. The /LIST qualifier is the default for batch mode operations.

Format

LISP/COMPILE/LIST[=*file-spec*] *file-spec*

Example

```
$ LISP/COMPILE/LIST=FACTORIAL.LIS/VERBOSE MYPROG.LSP
```

Sample Listing File

```
Listing output for file DBA1:[JONES.LIS]MYPROG.LSP;1
Compiled at 10:33:30 on Friday, 20 December 1985 by JONES
Lisp Version V2.0

Starting compilation of file  "DBA1:[JONES.LIS]MYPROG.LSP;1".
FACTORIAL compiled.

Finished compilation of file  "DBA1:[JONES.LIS]MYPROG.LSP;1".
0 Errors, 0 Warnings
```

Mode

Compile

2.10.7 /[NO]MACHINE_CODE

The /MACHINE_CODE qualifier is meaningful only if it is specified with the /COMPILE qualifier. The /MACHINE_CODE qualifier requests the

compiler to put a listing of the VAX LISP machine code in a file separate from the FAS file the compiler generates. The compiler also puts anything usually included in a listing file in this file (see Section 2.10.6 for a description of a listing file).

VAX LISP machine code is similar to a standard assembly language code. However, compiling LISP source code does not generate object modules that must be linked.

The /MACHINE_CODE qualifier has no effect on the production of machine code; the qualifier produces only a machine-code listing file. The machine-code listing file generated when you use the /MACHINE_CODE qualifier has the same name as your source file and has a LIS file type (unless you also used the /LIST qualifier to specify a different name).

The /NOMACHINE_CODE qualifier, the default, suppresses a listing of LISP machine code.

Format

LISP/COMPILE/MACHINE_CODE *file-spec*

Example

```
$ LISP/COMPILE/MACHINE_CODE MYPROG.LSP
```

Mode

Compile

2.10.8 /MEMORY

The /MEMORY qualifier lets you specify the amount of dynamic virtual memory the LISP system allocates in 512-byte pages. This system requires a minimum of 4000 pages of dynamic virtual memory to function. This memory is in addition to the read-only and static memory. Consequently, the default page size for the dynamic virtual memory is 4000 pages. If you specify fewer than 4000 pages with the /MEMORY qualifier, the system disregards the requested page size and uses the default page size. You do not need the /MEMORY qualifier if you intend to use no more than 4000 pages of dynamic memory.

To see how many pages of memory are available at any point while you are in LISP, use the LISP ROOM function. If you discover that you need more memory, save your work by creating a suspended system, and exit LISP. Then reenter LISP with the /RESUME and the /MEMORY qualifiers. Use the /MEMORY qualifier to specify a larger number of pages than you had previously specified. For information on creating a suspended system, see Section 2.11.1; for descriptions of the

/RESUME qualifier and the ROOM function, see Section 2.10.11 and Part II, respectively.

Format

LISP/MEMORY=*number-of-pages*

or

LISP/COMPILE/MEMORY=*number-of-pages file-spec*

Example

```
$ LISP/MEMORY=15000

Welcome to VAX LISP, Version V2.0

Lisp>
```

Mode

Interactive or Compile or Resume

2.10.9 /[NO]OPTIMIZE

The /OPTIMIZE qualifier lets you optimize the results of compilation of your program according to the following qualities:

- SPEED (execution speed of the code)
- SPACE (space occupied by the code)
- SAFETY (run-time error checking of the code)
- COMPILATION_SPEED (speed of the compilation process)

You can optimize your program by setting a priority value for each quality. That value must be an integer in the range of 0 to 3. The value 0 means the quality has the lowest priority in relationship to the other qualities; the value 3 means the quality has the highest priority in relationship to the other qualities. When you do not specify the /OPTIMIZE qualifier, the qualities each take the default value of 1. To suppress optimization, use the /NOOPTIMIZE form of this qualifier.

The /OPTIMIZE qualifier is meaningful only if it is specified with the /COMPILE qualifier. The /OPTIMIZE qualifier affects only the compiler, and does nothing to the interpreter, the debugger, or any other VAX LISP facility. See Chapter 7, Appendix A, and *COMMON*

LISP: The Language for information on specifying optimization declarations.

Format

LISP/COMPILE/OPTIMIZE=(*quality*:*value*[,...]) *file-spec*

Example

```
$ LISP/COMPILE/OPTIMIZE=(SPEED:3,SAFETY:2) MYPROG.LSP
```

or

```
$ LISP/COMPILE/OPTIMIZE=SPEED:3 MYPROG.LSP
```

Mode

Compile

2.10.10 /[NO]OUTPUT_FILE

The /OUTPUT_FILE qualifier is meaningful only when it is specified with the /COMPILE qualifier. The /OUTPUT_FILE qualifier tells the compiler to write the compiled code to a specific file. If you specify the /OUTPUT_FILE qualifier with a file name, the LISP system puts the compiled code in a file with that specified name. Use the /OUTPUT_FILE qualifier only when you want to change the name of the compiled file so that the source file and the compiled file have different names.

The /OUTPUT_FILE qualifier does not specify a listing file, only a compiled file. See the /LIST qualifier (Section 2.10.6) for an explanation of a listing file.

If this qualifier is not specified, the compiler produces a file with the same name as the source file and a type of FAS.

The /NOOUTPUT_FILE qualifier prevents compiled code from being written to a file. If you want only to check a file for errors, use this qualifier with the /COMPILE qualifier.

Format

LISP/COMPILE/OUTPUT_FILE[=*file-spec*] *file-spec*

Example

```
$ LISP/COMPILE/OUTPUT_FILE=TEST.FAS FACTORIAL.LSP
```

Format

LISP/COMPILE/NOOUTPUT_FILE *file-spec*

Example

```
$ LISP/COMPILE/NOOUTPUT_FILE MYPROG.LSP
```

Mode

Compile

2.10.11 /RESUME

The /RESUME qualifier resumes a suspended LISP system where the suspension occurred. See Section 2.11 for an explanation of suspended systems. The /RESUME and the /INITIALIZE qualifiers cannot be used together.

Format

LISP/RESUME=*file-spec*

Example

```
$ LISP/RESUME=MYPROG.SUS
T
Lisp>
```

Mode

Resume

2.10.12 /[NO]VERBOSE

The /VERBOSE qualifier lists on the output device and in the listing file the names of the functions defined or loaded in an initialization file, and the names of functions in a file as they are compiled. The /VERBOSE qualifier applies only to files loaded with /INITIALIZE qualifier or compiled with the /COMPILE qualifier.

The /NOVERBOSE qualifier (the default) prevents the names of functions compiled with the /COMPILE qualifier or loaded with the /INITIALIZE qualifier from being listed in a file or at the terminal.

Format

LISP/VERBOSE/INITIALIZE=*file-spec*

or

LISP/COMPILE/VERBOSE *file-spec*

Examples

1. $ LISP/VERBOSE/INITIALIZE=MYINIT.LSP

```
Welcome to VAX LISP, Version V2.0

; Loading contents of file DBA1:[JONES]MYINIT.LSP;1
;   FACTORIAL
;   FACTORS-OF
; Finished loading DBA1:[JONES]MYINIT.LSP;1
Lisp>
```

FACTORIAL and FACTORS-OF are functions that are loaded into the LISP system from Jones's initialization file.

2. $ LISP/VERBOSE/COMPILE MYPROG.LSP

```
Starting compilation of file DBA1:[JONES]MYPROG.LSP;1

MULT compiled.
SUB compiled.
DIV compiled.

Finished compilation of file DBA1:[JONES]MYPROG.LSP;1
0 Errors, 0 Warnings
$
```

MULT, SUB, and DIV are functions compiled in the file, MYPROG.LSP. The compiled definitions of these functions are written to the file, MYPROG.FAS.

Mode

Interactive or Compile

2.10.13 /[NO]WARNINGS

The /WARNINGS qualifier specifies that the LISP system is to produce warning messages. Warning messages are the default when you use the /COMPILE qualifier.

A warning message indicates that the LISP system has detected

something that is likely to be wrong. If warnings are signaled while a file is being compiled and the value of the *BREAK-ON-WARNINGS* variable is NIL (the default), the compilation continues. But, if errors are signaled, compilation of the expression causing the error is not continued though the rest of the file is compiled. See Chapter 4 for more information on the differences between warnings and errors.

The /NOWARNINGS qualifier suppresses warning messages.

The following example of a warning message is the message the compiler displays for the TEST function defined in Section 2.9.3.

```
$ LISP/COMPILE TEST.LSP
Warning in TEST
  TEST earlier called with 2 args, wants at most 1.
$
```

Format

LISP/COMPILE/NOWARNINGS *file-spec*

Example

```
$ LISP/COMPILE/NOWARNINGS MYPROG.LSP
```

Mode

Compile

2.11 USING SUSPENDED SYSTEMS

A suspended system is a binary file that is a copy of the LISP memory in use during an interactive LISP session up to the point at which you create the suspended system. The purpose of a suspended system is to save the state of an interactive LISP session. You might want to do this if your work is incomplete. By resuming LISP from a suspended system, you can continue your work from the point at which you stopped.

NOTE

A suspended system can be resumed only by the VAX LISP system from which it was suspended. The VAX LISP system that resumes a suspended system must meet these criteria:

1. The VAX LISP system must be the same version of VAX LISP as the suspending system.

2. A custom VAX LISP system created with the VAX LISP System-Building Utility must be the same system as the suspending system or a copy of the suspending system. (See the *VAX LISP/VMS System-Building Guide* for a description of the System-Building Utility.)

2.11.1 Creating a Suspended System

The VAX LISP SUSPEND function puts in a file the LISP memory in use during an interactive LISP session, enabling you to resume the same LISP session at a later time. The SUSPEND function does not stop the current LISP session; you can continue to use the LISP session after the SUSPEND function has put a copy of memory into a file. The SUSPEND function also automatically invokes a garbage collection of dynamic memory space. See Chapter 7 for information on garbage collections.

In the following example, the file FILEX.SUS is created and a copy of the memory in a LISP session is put into that file. The file name can be a string, symbol, or pathname. See Chapter 7 and *COMMON LISP: The Language* for a description of pathnames.

```
Lisp> (SUSPEND "FILEX.SUS")
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
; Starting garbage collection due to SUSPEND function.
; Finished garbage collection due to SUSPEND function.
NIL
Lisp>
```

After your file is created, the system returns to your interactive LISP session. You can exit LISP when you see the LISP prompt. Your suspended system file is placed either in your default directory or in the directory you specified in the file specification. The file is usable only in an interactive LISP session.

If you use the Editor before using the SUSPEND function, Editor buffers that are associated with files are deleted in the resumed system. Consequently, if you want to save any material in a buffer, put that material in a file. For a description of the VAX LISP Editor, see Chapter 3. For a description of the SUSPEND function, see Part II.

2.11.2 Resuming a Suspended System

To resume a suspended system, use the LISP command with the /RESUME qualifier and the name of the file containing the suspended system.

Program execution continues from the point at which you called the SUSPEND function. See Section 2.10.11 for an explanation of the /RESUME qualifier.

After it creates a suspended system, the SUSPEND function returns NIL and execution continues with the LISP environment exactly as it was before the call to SUSPEND. However, when execution resumes as a result of using the /RESUME qualifier, the SUSPEND function returns T. Therefore, a program can use the return value of SUSPEND to determine if execution is resuming as the result of the /RESUME qualifier, and take action if necessary. See the SUSPEND function in Part II for a description of the effects of suspending a system.

When resuming a suspended system, VAX LISP checks to make sure that the resuming system matches the suspending system. The resuming system must be the same system that suspended or a copy of the file containing the system that suspended.

CHAPTER 3

ERROR HANDLING

The LISP system invokes the VAX LISP error handler when errors are signaled during program evaluation. This chapter explains what the error handler does when an error is signaled. Because the system's error handler might not meet your programming needs, VAX LISP allows you to create your own error handler. The procedure for creating an error handler is also explained in this chapter.

3.1 THE ERROR HANDLER

The VAX LISP error handler is a function named UNIVERSAL-ERROR-HANDLER, which performs four sequential steps.

1. Checks the number of nested errors that have occurred. If three nested errors have occurred, the error handler aborts your program, displays a message, and returns you to the top-level read-eval-print loop; otherwise, the handler continues to the next step.

2. Checks the type of the error.

3. Displays an error message that provides you with information about the error.

4. Performs the appropriate operation for the type of error that was signaled.

3.2 VAX LISP ERROR TYPES

Three types of errors can occur during the evaluation of a LISP program:

- Fatal error
- Continuable error
- Warning

When an error is signaled, the VAX LISP system displays an error message that provides you with the following information:

- The type of error that was signaled -- fatal error, continuable error, or warning

- The name of the function that caused the error -- ERROR, CERROR, or WARN
- The name of the function that was used to signal the error
- A description of the error
- If a continuable error, an explanation of what will happen if you continue the program's evaluation from the point where the error occurred

The format of an error message and the information a message provides depend on the type of the error. The next three sections describe the types of errors; each description includes the error type's message format and the operation the error handler performs.

3.2.1 Fatal Errors

When a fatal error is signaled, the error handler displays a message in the following format:

```
Fatal error in function function-name (signaled with ERROR).
Error description.
```

In the preceding format description, function-name is the name of the function that caused the error, and ERROR is the name of the function that was used to signal the error (see Table 3-1). The error description is a message that describes the error. The message is generated from the format string and the arguments in the call to the ERROR function; the message can be displayed on more than one line.

An example of a fatal error message follows:

```
Fatal error in function MAKE-ARRAY (signaled with ERROR).
Only vectors can have fill pointers.
```

After the message is displayed, the error handler checks the value of the VAX LISP *ERROR-ACTION* variable. Its value can be either the :EXIT or the :DEBUG keyword. The /ERROR_ACTION DCL command qualifier sets the value of the *ERROR-ACTION* variable when the LISP system is invoked (see Section 2.8.3). When the value is :EXIT, the error handler causes the LISP system to exit; when the value is :DEBUG, the handler invokes the VAX LISP debugger.

If the debugger is invoked, you can use it to locate the error in your program. After you locate the error, you can correct it and restart your program's evaluation.

NOTE

You cannot continue your program's evaluation from the point where a fatal error occurred.

The *ERROR-ACTION* variable is described in Part II and the debugger is described in Section 4.4.

3.2.2 Continuable Errors

When a continuable error is signaled, the error handler displays a message in the following format:

```
Continuable error in function function-name (signaled with CERROR).
Error description.
If continued: Continue explanation.
```

In the preceding format description, function-name is the name of the function that caused the error, and CERROR is the name of the function that was used to signal the error (see Table 3-1). The error description is a message that describes the error. The message is generated from the format string and the arguments in the call to the CERROR function; the message can be displayed on more than one line. A line of text that explains what will happen if you continue your program's evaluation follows the error description.

An example of a continuable error message follows:

```
Continuable error in function ENTER-NAME (signaled with CERROR).
The value you specified is not a string.
If continued: You will be prompted for a new value.
```

After the message is displayed, the error handler checks the value of the VAX LISP *ERROR-ACTION* variable. Its value can be either the :EXIT or the :DEBUG keyword. The /ERROR ACTION DCL command qualifier sets the value of the *ERROR-ACTION* variable when the LISP system is invoked (see Section 2.8.3). When the value is :EXIT, the error handler causes the LISP system to exit; when the value is :DEBUG, the handler invokes the VAX LISP debugger.

If the debugger is invoked, you can do one of the following:

- Continue from the error; the CERROR function performs the corrective action that is specified in the error message.
- Locate the error in your program. After you locate the error, you can correct it and restart your program's evaluation.

The *ERROR-ACTION* variable is described in Part II and the debugger is described in Section 4.4.

3.2.3 Warnings

A warning is an error condition that exists in your program, which does not affect your program's evaluation. When this type of error occurs, the system displays a message for the following reasons:

- You might want to correct the error later.
- Your program might correct the error, but you should know that the error occurred.

When a warning is signaled, the error handler displays a message in the following format:

```
Warning in function function-name (signaled with WARN).
Error description.
```

In the preceding format description, function-name is the name of the function that caused the error, and WARN is the name of the function that was used to signal the error (see Table 3-1). The error description is a message that describes the error. The message is generated from the format string and the arguments in the call to the WARN function; the message can be displayed on more than one line.

An example of a warning error message follows:

```
Warning in function ADD-TWO-NUMBERS (signaled with WARN).
The function produced a value greater than 10.
```

After the message is displayed, the error handler checks the value of the *BREAK-ON-WARNINGS* variable. When the value of this variable is NIL, the error handler returns NIL; when the value is not NIL, the error handler checks the value of the VAX LISP *ERROR-ACTION* variable. The value of the *ERROR-ACTION* variable can be either the :EXIT or the :DEBUG keyword. The /ERROR_ACTION DCL command qualifier sets the value of the *ERROR-ACTION* variable when the LISP system is invoked (see Section 2.8.3). When the value is :EXIT, the error handler causes the LISP system to exit; when the value is :DEBUG, the handler invokes the VAX LISP debugger.

If the debugger is invoked, you can use it to locate the error in your program. After you locate the error, you can correct it, exit the debugger, and then continue your program's evaluation from the point where the error occurred.

The *BREAK-ON-WARNINGS* variable is described in COMMON LISP: The Language, the *ERROR-ACTION* variable is described in Part II, and the debugger is described in Section 4.4.

3.3 CREATING AN ERROR HANDLER

The VAX LISP *UNIVERSAL-ERROR-HANDLER* variable is bound to the system's error handler. This binding provides you with a way to create your own error handler if the system's handler does not meet your programming needs. To create an error handler you must perform the following:

1. Define the error handler.

2. Bind the *UNIVERSAL-ERROR-HANDLER* variable to the defined handler.

The *UNIVERSAL-ERROR-HANDLER* variable is described in Part II.

3.3.1 Defining an Error Handler

The LISP system passes at least two arguments to the error handler each time an error occurs in a program. Therefore, when you define an error handler, the handler must be able to accept two or more arguments. Specify the arguments in an error-handler definition in the following format:

```
function-name error-signaling-function &REST args
```

The arguments provide the error handler with the following information:

- The name of the function that called the error-signaling function
- The name of the error-signaling function
- The arguments that were passed to the error-signaling function

An example of an error handler definition follows:

```
Lisp> (DEFUN CRITICAL-ERROR-HANDLER (FUNCTION-NAME
                                     ERROR-SIGNALING-FUNCTION
                                     &REST ARGS)
        (WHEN (OR (EQ ERROR-SIGNALING-FUNCTION 'ERROR)
                  (EQ ERROR-SIGNALING-FUNCTION 'CERROR))
              (FLASH-ALARM-LIGHT))
        (APPLY #'UNIVERSAL-ERROR-HANDLER
               FUNCTION-NAME
               ERROR-SIGNALING-FUNCTION
               ARGS))
CRITICAL-ERROR-HANDLER
```

The preceding error handler checks whether a fatal or continuable error is signaled. If either type of error is signaled, the handler calls the function FLASH-ALARM-LIGHT and then passes the error signal information to the VAX LISP error handler.

When you define an error handler, the definition can include a call to the UNIVERSAL-ERROR-HANDLER function. If the definition does not include a call to this function and you want the handler to check the value of the *ERROR-ACTION* or *BREAK-ON-WARNINGS* variable, you must include a check on the variable in the handler's definition.

If you want an error handler to display error messages in the formats described in Sections 3.2.1 to 3.2.3, include a call to either the UNIVERSAL-ERROR-HANDLER or PRINT-SIGNALED-ERROR function. Descriptions of these functions are provided in Part II.

The next three sections describe the arguments an error handler must be able to accept.

3.3.1.1 **Function Name** - The function-name argument is the name of the function that calls an error-signaling function. This argument enables the error handler to include the function's name in the error message it displays.

3.3.1.2 **Error-Signaling Function** - The error-signaling-function argument is the name of the error-signaling function that is called to generate the error signal. Depending on which function is called, a fatal error, continuable error, or warning is signaled.

The error handler uses the error-signaling-function argument to determine the contents of the args argument.

Table 3-1 lists the functions that can be passed as the error-signaling-function argument and provides a brief description of each function.

Table 3-1
Error-Signaling Functions

Function	Description
CERROR Function	Signals a continuable error
ERROR Function	Signals a fatal error
WARN Function	Signals a warning

See COMMON LISP: The Language for detailed descriptions of the CERROR and ERROR functions. See Section 8.7 for a description of the WARN function.

3.3.1.3 **Arguments** - The args argument is the list of arguments that are passed to the error-signaling function when the error-signaling function is invoked. The contents of the list depends on which function is invoked. The list can include one or two format strings and their corresponding arguments. The format strings and arguments are passed to the FORMAT function, which produces the correct error message.

3.3.2 Binding the *UNIVERSAL-ERROR-HANDLER* Variable

Once you define an error-handling function, you must bind the *UNIVERSAL-ERROR-HANDLER* variable to it. The following example shows how to bind the variable to a function:

```
Lisp> (LET ((*UNIVERSAL-ERROR-HANDLER*
              #'CRITICAL-ERROR-HANDLER))
          (PERFORM-CRITICAL-OPERATION))
```

The LET special form binds the *UNIVERSAL-ERROR-HANDLER* variable to the CRITICAL-ERROR-HANDLER function that was defined in Section 3.3.1, and calls a function named PERFORM-CRITICAL-OPERATION. When the form is exited because the evaluation finished or the THROW function is called, the *UNIVERSAL-ERROR-HANDLER* variable is restored to its previous value.

CHAPTER 4

DEBUGGING FACILITIES

Debugging is the process of locating and correcting programming errors. When an error is signaled, the VAX LISP error handler displays a message, which provides you with your initial debugging information: the error type, the name of the function that caused the error, the name of the function the LISP system used to signal the error, and a description of the error.

Once you know the name of the function that caused an error, you can use the VAX LISP debugging functions and macros to locate and to correct the programming error. Table 4-1 lists the debugging functions and macros with a brief description of each.

Table 4-1
Debugging Functions and Macros

Name	Function or Macro	Description
APROPOS	Function	Locates symbols whose print names contain a specified string argument as a substring and displays information about each symbol it locates.
APROPOS-LIST	Function	Locates symbols whose print names contain a specified string argument as a substring and returns a list of the symbols it locates.
BREAK	Function	Invokes the break loop.
DEBUG	Function	Invokes the VAX LISP debugger.
DESCRIBE	Function	Displays detailed information about a specified object.
DRIBBLE	Function	Sends the input and the output of an interactive LISP session to a specified file.
ED	Function	Invokes the VAX LISP Editor.
ROOM	Function	Displays information about the state of internal storage and its management.
STEP	Macro	Invokes the stepper.

(Continued on next page)

Table 4-1 (Cont.)
Debugging Functions and Macros

Name	Function or Macro	Description
TIME	Macro	Displays timing information about the evaluation of a specified form.
TRACE	Macro	Enables the tracer for functions and macros.
UNTRACE	Macro	Disables the tracer for functions and macros.

This chapter provides the following:

- A list of the functions and the macro that provide you with debugging information
- Descriptions of two variables that control the output of the debugger, the stepper, and the tracer facilities
- A description of the VAX LISP control stack
- Explanations of how to use the following debugging facilities:
  - Break loop -- A read-eval-print loop you can invoke while the LISP system is evaluating a program.
  - Debugger -- A control stack debugger you can use interactively to inspect and modify the LISP system's control stack frames.
  - Stepper -- A facility you can use to interactively step through a forms evaluation.
  - Tracer -- A facility you can use to inspect a program's evaluation.
  - Editor -- An extensible editor that enables you to edit programs and data structures.

The following functions and macro display information that you can use to debug programs:

- APROPOS function
- APROPOS-LIST function
- DESCRIBE function
- DRIBBLE function
- ROOM function
- TIME macro

Descriptions of the preceding functions and macro are provided in Section 8.7.

4.1 CONTROL VARIABLES

VAX LISP provides two variables that control the output of the debugger, the stepper, and the tracer facilities: *DEBUG-PRINT-LENGTH* and *DEBUG-PRINT-LEVEL*.

DEBUG-PRINT-LENGTH — Controls the number of displayed elements at each level of a nested data object. The variable's value must be either an integer or NIL. The default value is NIL (no limit).

DEBUG-PRINT-LEVEL — Controls the number of displayed levels of a nested data object. The variable's value must be either an integer or NIL. The default value is NIL (no limit).

4.2 CONTROL STACK

The control stack is the part of LISP memory that stores calls to functions, macros, and special forms. The stack consists of stack frames. Each time you call a function, macro, or special form, the VAX LISP system does the following:

1. Opens a stack frame
2. Pushes the name of the function associated with the function, macro, or special form that was called onto the stack frame
3. Pushes the function's arguments onto the stack frame
4. Closes the stack frame when all the function's arguments are on the stack frame
5. Evaluates the function

The LISP system can open several stack frames at a time because the arguments used by LISP functions are frequently LISP expressions.

Each control stack frame has a frame number, which is displayed as part of the stack frame's output. Stack frame numbers are displayed in the output of the debugger, the stepper, and the tracer.

The control stack consists of two types of stack frames: open and active. Open and active stack frames can be either significant or insignificant. Significant stack frames are those that invoke documented and user-created functions. Debugger commands show only significant stack frames unless you specify the ALL modifier (see Section 4.4.3.1). Significant stack frames store one of the following calls:

- A call to a function named by a symbol that is in the current package
- A call to a function that is accessible in the current package and is explicitly or implicitly called by another function that is in the current package

See COMMON LISP: The Language for information on packages.

Many stack frames in the control stack store internal, undocumented functions. These stack frames are insignificant to most users; therefore, by default, the debugger does not display their representation. However, if you are using the debugger and you want to examine these stack frames, you can specify the ALL modifier with debugger commands.

4.2.1 Open Stack Frame

An open stack frame is a stack frame that is under construction. Open stack frames store functions the LISP system cannot invoke because the system has not evaluated all their arguments. The stack frames that are above an open stack frame store calls to other functions. The values those functions return are the arguments of the function in the open stack frame.

4.2.2 Active Stack Frame

The active stack frame is a stack frame that stores a call to a function the LISP system is evaluating. The system can evaluate a function call in the active stack frame because the frame contains all the function's argument values. Only one stack frame is active at a time and an active stack frame can exist anywhere on the control stack.

The active stack frame can have a previous active stack frame and/or it can have a next active stack frame. The previous active stack frame represents the caller of the function in the current active stack frame.

4.3 BREAK LOOP

The break loop is a read-eval-print loop that you can invoke to debug a program. You can invoke the break loop while a program is being evaluated. If you do, the evaluation is interrupted and you are placed in the loop.

NOTE

> If the value of the *BREAK-ON-WARNINGS* variable is T, the debugger is invoked rather than the break loop when a warning is signaled.

4.3.1 Invoking the Break Loop

You can invoke the break loop by calling the BREAK function. The two ways of using the BREAK function to debug a program are the following:

- Use the VAX LISP BIND-KEYBOARD-FUNCTION function to bind an ASCII keyboard control character to the BREAK function. Then use the control character to invoke the BREAK function directly while your program is being evaluated (see Part II for a description of the BIND-KEYBOARD-FUNCTION function)

- Put the BREAK function in specific places in your program

In either case, the BREAK function displays a message (if you specified one) and enters a read-eval-print loop. If you specified a message, the BREAK function displays the message in the following format:

```
Break in function function-name (signaled with BREAK).
Description.
```

In the preceding format description, function-name represents the name of the function the LISP system was evaluating when you entered the break loop. BREAK is the name of the function that caused the LISP system to invoke the break loop. The description is optional and can be printed on more than one line. A description usually provides the reason the break loop was invoked.

An example of a break loop message follows:

```
Break in function INTERRUPT-INPUT (signaled with BREAK).
Values are too high.
```

After the message is displayed, a prompt is displayed at the left margin of your terminal. The prompt looks like the following:

```
Break n>
```

The n in the prompt represents an integer, which indicates the number of the nested command level you are in. The value of n increases by one each time the level of the break loop increases. For example, the following prompt is displayed if you are in the third nested loop:

```
Break 3>
```

4.3.2 Exiting the Break Loop

When you are ready to exit the break loop and continue your program's evaluation, invoke the VAX LISP CONTINUE function.

```
Break 1> (CONTINUE)
```

The CONTINUE function causes the evaluation of your program to continue from the point where the LISP system encountered the BREAK function.

A description of the CONTINUE function is provided in Part II.

4.3.3 Using the Break Loop

Once you are in the break loop, you can check what your program is doing by interacting with the LISP system as though you were in the top-level loop. For example, suppose you define a variable named *FIRST* and a function named COUNTER, which uses the variable.

```
Lisp> (DEFVAR *FIRST* 0)
*FIRST*
Lisp> (DEFUN COUNTER NIL
        (IF (< *FIRST* 100)
            (PROGN (INCF *FIRST*) (COUNTER))
            *FIRST*))
COUNTER
```

If you bind the BREAK function to a control character, you can interrupt the function's evaluation by typing the control character. For example:

```
Lisp> (BIND-KEYBOARD-FUNCTION #\^B #'BREAK)
T
Lisp> (COUNTER) RET
CTRL/B
Break 1>
```

Once you are in the break loop, you can check the value of the variable *FIRST*.

```
Break 1> *FIRST*
16
Break 1>
```

If you call the CONTINUE function, the evaluation of the function COUNTER continues.

```
Lisp> (CONTINUE)
```

After you call the CONTINUE function, you can see that the evaluation was continued by invoking the break loop again and rechecking the value of the variable *FIRST*.

```
CTRL/B
Break 1> *FIRST*
93
Break 1>
```

Use the CONTINUE function again to complete the function's evaluation.

```
Break 1> (CONTINUE)
100
```

Changes that you make to global variables and global definitions while you are in the break loop remain in effect after you exit the loop and your program continues. For example, if you are in the break loop and you find that the value of the variable named *FIRST* has an incorrect value, you can change the variable's value. The change remains in effect after you exit the break loop and continue your program's evaluation.

NOTE

> The forms you type while you are in the break loop are evaluated in a null lexical environment, as though they are evaluated at top level. Therefore, you cannot examine the lexical variables of a program that you interrupt with the break loop. To examime such lexical variables, invoke the debugger (see Section 4.4). For information on lexical environments, see COMMON LISP: The Language.

4.3.4 Break Loop Variables

The break loop uses a copy of the top-level-loop variables (plus (+), hyphen (-), asterisk (*), slash (/), and so on) the same way the top-level loop uses them (see COMMON LISP: The Language). These variables preserve the input expressions you specify and the output values the VAX LISP system returns while you are in the break loop.

4.4 DEBUGGER

The VAX LISP debugger is a control stack debugger. You can use it interactively to inspect and modify the LISP system's control stack frames. The debugger has a pointer that points to the current stack frame. The current stack frame is the last frame for which the debugger displayed information. The debugger provides several commands that perform the following:

- Display help
- Evaluate a form or reevaluate a function call a stack frame stores
- Handle errors
- Move the pointer from one stack frame to another
- Inspect or modify the function call in a stack frame
- Display a summary of the control stack

The debugger reads its input and prints its output to the stream bound to the *DEBUG-IO* variable.

NOTE

> The stack frames the debugger displays are no longer active.

Before you use the debugger, you should be familiar with the VAX LISP control stack. The control stack is described in Section 4.2.

4.4.1 Invoking the Debugger

The VAX LISP system invokes the debugger when errors occur. You can invoke the debugger by calling the VAX LISP DEBUG function. For example:

```
Lisp> (DEBUG)
```

When the debugger is invoked, a message that identifies the debugger, a message that identifies the current stack frame, and the command prompt are displayed at the left margin of your terminal in the following format:

```
Control Stack Debugger
Frame #5: (DEBUG)
Debug n>
```

The letter n in the prompt represents an integer, which indicates the number of the nested command level you are in. The value of n increases by one each time the level of the debugger increases. For example, the following prompt might be displayed if a serious error was found in an expression you used:

```
Debug 3>
```

After the debugger is invoked, you can use the debugger commands to inspect and modify the contents of the system's control stack.

A description of the DEBUG function is provided in Part II.

4.4.2 Exiting the Debugger

To exit the debugger, use the QUIT debugger command. It causes the debugger to return control to the previous command level.

```
Debug 2> QUIT
Debug 1>
```

If you specify the QUIT command when the debugger command level is one (indicated by the prompt Debug 1>), the command causes the debugger to exit and returns you to the system's top level. For example:

```
Debug 1> QUIT
Lisp>
```

By default, the QUIT command displays a confirmation message before it exits if a continuable error causes the debugger to be invoked. For example:

```
Debug 1> QUIT
Do you really want to return to the previous command level?
```

If you respond to the message by typing YES, the debugger returns control to the previous command level.

```
Do you really want to return to the previous command level? YES
Lisp>
```

If you respond by typing NO, the debugger prompts you for another command.

```
Do you really want to return to the previous command level? NO
Debug 1>
```

You can prevent the debugger from displaying the confirmation message by specifying the QUIT command with a value other than NIL. For example:

```
Debug 1> QUIT T
Lisp>
```

A description of the QUIT command is provided in Section 4.4.3.2.

4.4.3 Using Debugger Commands

The debugger commands are words that describe the operation you want the debugger to perform. The debugger commands enable you to inspect and to modify the current control stack frame and to move to other stack frames. You must specify many of the debugger commands with one or more arguments. The command arguments modify command operations.

You can abbreviate debugger commands to as few characters as you like, as long as there is no ambiguity in the abbreviation.

Enter a debugger command by typing the command name or abbreviation and then pressing the RETURN key. For example:

```
Debug 1> BACKTRACE(RET)
```

If you press only the RETURN key, the debugger prompts you for another command.

Table 4-2 provides a summary of the debugger commands. Detailed descriptions of the commands are provided in Section 4.4.3.2.

Table 4-2
Debugger Commands

Command	Description
?	Displays help text about the debugger commands.
BACKTRACE	Displays a backtrace of the control stack.
BOTTOM	Moves the pointer to the first stack frame on the control stack.
CONTINUE	Enables you to correct a continuable error.
DOWN	Moves the pointer down the control stack.
ERROR	Redisplays the error message that was displayed when the debugger was invoked.

(Continued on next page)

Table 4-2 (Cont.)
Debugger Commands

Command	Description
EVALUATE	Evaluates a specified form.
GOTO	Moves the pointer to a specified stack frame.
HELP	Displays help text about the debugger commands.
QUIT	Exits to the previous command level.
REDO	Invokes the function in the current stack frame.
RETURN	Evaluates its arguments and causes the current stack frame to return the same values the evaluation returns.
SEARCH	Searches the control stack for a specified function.
SET	Sets the values of the components in the current stack frame.
SHOW	Displays information stored in the current stack frame.
STEP	Invokes the stepper for the function that is in the current stack frame.
TOP	Moves the pointer to the last stack frame in the control stack.
UP	Moves the pointer up the control stack.
WHERE	Redisplays the argument list and the function name in the current stack frame.

4.4.3.1 **Arguments** - Debugger command arguments modify the operations the debugger commands perform. You must specify some of the debugger commands with an argument. Some commands accept optional arguments. The arguments you can specify with the debugger commands are the following:

- Debugger command
- Symbol
- Form
- Integer
- Function name
- Modifier

NOTE

Only form arguments are evaluated.

The preceding arguments are self-explanatory with the exception of the integer and modifier arguments. Integer arguments represent control stack frame numbers. Each stack frame on the control stack has a frame number, which the debugger displays as part of the stack frame's output. The debugger reassigns these numbers each time it is invoked. You can specify a frame number in a debugger command to refer to a specific stack frame. If you refer to a frame number that is outside the current debugging session, an error is signaled. If you refer to the stack frame number of a frame that was established in another debugging session in a current nested session, the command in which you specify the frame number results in an erroneous or unpredictable result.

An argument that is a modifier is a word that changes the way a command operates. Table 4-3 provides a summary of the modifiers you can specify with debugger commands.

Table 4-3
Debugger Command Modifiers

Modifier	Command Modification
ALL	Operate on both significant and insignificant stack frames.
ARGUMENTS	Operate on the arguments specified with the function that is in the current stack frame.
CALL	Operate on the call to the current stack frame.
DOWN	Move the pointer down the control stack.
FUNCTION	Operate on the function object that is in the current stack frame.
HERE	Operate on the current stack frame.
NORMAL	Display the function name and the argument list that are in the control stack frames.
QUICK	Display the function name that is in the control stack frames.
TOP	Start a backtrace at the top of the control stack.
UP	Move the pointer up the control stack.
VERBOSE	Display the function name, argument list, local variable bindings, and special variable bindings that are in the control stack frames.

Enter an argument after the command it modifies and then press the RETURN key. For example:

```
Debug 1> DOWN ALL(RET)
```

Depending on which command you specify, an argument can be either required or optional. An argument whose value is an integer is usually optional; an argument whose value is a symbol or form is required. If you do not specify an argument that is required, the debugger prompts you for the argument. For example:

```
Debug 1> RETURN(RET)
First Value:
```

The debugger does not prompt for arguments if you specify them in the command line.

4.4.3.2 **Debugger Commands** - The VAX LISP debugger provides commands that you can use to move through and modify the system's control stack.

Help Commands

HELP
?

The HELP command displays help text about the debugger commands. You can specify this command with one argument. The argument must be the name of the debugger command about which you want help text. If you specify the HELP command without an argument, the debugger displays a list of the debugger commands.

You can abbreviate this command by using a question mark (?).

Evaluation Command

You can evaluate LISP expressions while you are in the debugger. If you want the LISP system to evalute a form, you can specify the form and then press the RETURN key. If you want the system to evaluate a symbol, you must use the EVALUATE command. You can also evaluate expressions by entering the break loop. For information on the break loop, see Section 4.3.

EVALUATE

The EVALUATE command explicitly evaluates a specified form. You must specify the command with an argument. The argument must be the form you want the LISP system to evaluate. The system evaluates the form in the lexical environment of the current stack frame.

Error-Handling Commands

The debugger handles errors that invoke the debugger. Each of the following debugger commands handles errors in a different way.

CONTINUE

The CONTINUE command causes the debugger to return NIL. This enables you to return from a continuable error or from a warning if the value of the *BREAK-ON-WARNINGS* variable is T. This command is not the same as the CONTINUE function.

QUIT

The QUIT command enables you to exit to the previous command level. If the current level of the debugger is one, the command causes the debugger to exit. You can specify this command with an optional argument. If a continuable error invokes the debugger and the argument is NIL, the debugger displays a confirmation message. If you respond to the message by typing YES, the command returns control to the previous command level. If the argument is not NIL, the debugger does not display a message. The default value for the optional argument is NIL.

REDO

The REDO command invokes the function in the current stack frame, causing the LISP system to reevaluate the function in that frame. This command is useful for correcting errors that are not continuable, such as unbound variables and undefined functions.

RETURN

The RETURN command evaluates its arguments and causes the debugger to force the current stack frame to return the same values the evaluation returns. You must specify the command with an argument. The argument must be a form. When the command is executed, the form is evaluated. When the evaluation is complete, the current stack frame returns the same values that the evaluated form returns.

STEP

The STEP command invokes the stepper for the function that is in the current stack frame. When the stepper is invoked, the LISP system reevaluates the function. This command is useful if you want to repeat an error to get information about the cause of the error.

Movement Commands

The movement commands move the debugger's pointer to another stack frame. The debugger displays the new stack frame's information.

BOTTOM

The BOTTOM command moves the pointer to the first significant stack frame on the control stack. You can specify this command with an optional argument. The argument must be the ALL modifier. If you specify it, the command moves the pointer to the first stack frame on the control stack whether it is significant or insignificant.

DOWN

The DOWN command moves the pointer down the significant stack frames on the control stack. You can specify this command with optional arguments. One of the optional arguments is the ALL modifier. If you specify it, the command moves the pointer down the significant and insignificant stack frames on the control stack.

You can also specify an optional integer argument. This argument indicates the number of stack frames down which the command is to move the pointer.

GOTO

The GOTO command moves the pointer to a specified stack frame. You must specify this command with an integer argument. The integer specifies the number of the stack frame to where you want the command to move the pointer.

SEARCH

The SEARCH command searches the control stack for a specified function name. You must specify this command with two arguments. One of the arguments must be either the UP or the DOWN modifier. The modifier specifies the direction of the command's search. The second argument must be the name of the function for which the command is to search.

You can also specify an optional integer argument. This argument must follow the function name argument in the command specification. The integer you specify indicates the number of occurrences of the specified function name that you want the command to skip.

TOP

The TOP command moves the pointer to the last significant stack frame on the control stack. You can specify this command with an optional argument. The argument must be the ALL modifier. If you specify it, the command moves the pointer to the last stack frame on the control stack whether it is significant or insignificant.

UP

The UP command moves the pointer up the significant stack frames on the control stack. You can specify this command with optional arguments. One of the optional arguments is the ALL modifier. If you specify it, the command moves the pointer up the significant and insignificant stack frames on the control stack.

You can also specify an optional integer argument. It indicates the number of stack frames up which the command is to move the pointer.

WHERE

The WHERE command redisplays the function name and argument list in the current stack frame.

Inspection and Modification Commands

You can inspect and change the information in a function call before the LISP system evaluates the call. To do this, use the inspection and modification commands.

ERROR

The ERROR command redisplays the error message that was displayed for the error that invoked the debugger.

SET

The SET command sets the values of the components in the current stack frame. You must specify this command with three arguments. One of the arguments must be a modifier. The modifier can be either ARGUMENTS or FUNCTION. The modifier determines what the command sets. The following list describes what is set when you specify each modifier:

- ARGUMENTS -- The value of an argument in the current stack frame.
- FUNCTION -- The function object in the current stack frame.

If you specify the ARGUMENTS modifier, the second argument must be the symbol that names the argument to be set, and the third argument must be a form that evaluates to the new value. If you specify the FUNCTION modifier, the second argument must be a form that evaluates to a function or the name of a function. The new function must take the same number of arguments the old function takes.

SHOW

The SHOW command displays information stored in the current stack frame. You must specify this command with an argument. The argument can be the ARGUMENTS, CALL, FUNCTION, or HERE modifier. The modifier determines what the command is to display. The following list describes what the command displays when you specify each modifier:

- ARGUMENTS -- A list of the arguments in the current stack frame.
- CALL -- The function call that created the current stack frame. The command displays the function call such that its output is easy to read. The arguments in the call are represented by their values.
- FUNCTION -- The function in the current stack frame. The function can be either interpreted, or compiled with the COMPILE function. The function cannot be displayed if it is a system function or is compiled with the COMPILE-FILE function or the DCL LISP/COMPILE command.
- HERE -- A description of the current stack frame.

Backtrace Command

BACKTRACE

The BACKTRACE command displays the argument list of each stack frame in the control stack starting from the top of the stack. You can specify the command with optional arguments. The arguments must be modifiers, which specify the style and extent of the backtrace.

The modifiers you can specify are ALL, NORMAL, QUICK, HERE, TOP, or VERBOSE. By default, the command uses the NORMAL and the TOP modifiers. The following list describes the style or extent the BACKTRACE command uses when you specify each modifier:

- ALL -- Displays significant and insignificant stack frames.
- NORMAL -- Displays the function name and argument list that are in each stack frame.
- QUICK -- Displays the function name in each stack frame.
- HERE -- Starts the backtrace at the current stack frame.
- TOP -- Starts the backtrace at the top of the control stack.
- VERBOSE -- Displays the function name, argument list, and local variable bindings in each stack frame.

4.4.4 Sample Debugging Sessions

1.

```
Lisp> (DEFUN FIRST-ELEMENT (X) (CAR X))
FIRST-ELEMENT
Lisp> (FIRST-ELEMENT 3)

Fatal error in function CAR (signaled with ERROR).
Argument must be a list: 3

Control Stack Debugger
Frame #11: (CAR 3)
Debug 1> DOWN
Frame #8: (BLOCK FIRST-ELEMENT (CAR X))
Debug 1> DOWN
Frame #5: (FIRST-ELEMENT 3)
Debug 1> SHOW HERE
It is a cons
Format: FIRST-ELEMENT x
-- Arguments --
X : 3
Debug 1> SET
Type of SET operation: ARGUMENT
Argument Name: X
New Value: '(1 2 3)
Debug 1> WHERE
Frame #5: (FIRST-ELEMENT (1 2 3))
Debug 1> REDO
1
Lisp>
```

The argument in a stack frame is changed from an integer to a list and the function is reevaluated with the correct argument.

2.

```
Lisp> (DEFUN PLUS-Y (X) (+ X Y))
PLUS-Y
Lisp> (PLUS-Y 4)

Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Symbol has no value: Y

Control Stack Debugger
Frame #8: (BLOCK PLUS-Y (+ X Y))
Debug 1> DOWN
Frame #5: (PLUS-Y 4)
Debug 1> UP
Frame #8: (BLOCK PLUS-Y (+ X Y))
Debug 1> (SETF Y 1)
1
Debug 1> WHERE
Frame #8: (BLOCK PLUS-Y (+ X Y))
Debug 1> EVALUATE
Evaluate: Y
1
Debug 1> DOWN
Frame #5: (PLUS-Y 4)
Debug 1> REDO
5
Lisp>
```

The value of the variable Y is set with the SETF macro and the body of the function PLUS-Y is reevaluated.

3.

```
Lisp> (DEFUN ONE-PLUS (X) (1+ X))
ONE-PLUS
Lisp> (ONE-PLUS '(1 2 3 4))

Fatal error in function 1+ (signaled with ERROR).
Argument must be a number: (1 2 3 4)

Control Stack Debugger
Frame #11: (1+ (1 2 3 4))
Debug 1> SET FUNCTION
Function: 'CAR
Debug 1> WHERE
Frame #11: (CAR (1 2 3 4))
Debug 1> DOWN
Frame #8: (BLOCK ONE-PLUS (1+ X))
Debug 1> UP
Frame #11: (CAR (1 2 3 4))
Debug 1> REDO
1
Lisp> (PPRINT-DEFINITION 'ONE-PLUS)
(DEFUN ONE-PLUS (X) (1+ X))
Lisp>
```

Shows that changing the contents of a stack frame does not change the contents of other stack frames or the function that was evaluated originally.

4.5 STEPPER

The stepper is a facility you can use to step interactively through the evaluation of a form. You can control the stepper with stepper commands as it displays and evaluates each subform of a specified form.

The stepper has a pointer that points to the current stack frame on the system's control stack. The current stack frame is the last frame for which the stepper displayed information.

The stepper prints its command interaction to the stream bound to the *DEBUG-IO* variable; it prints its output to the stream bound to the *TRACE-OUTPUT* variable.

4.5.1 Invoking the Stepper

You can invoke the stepper by calling the STEP macro with a form as an argument. The following example invokes the stepper with a call to a function named FACTORIAL:

```
Lisp> (STEP (FACTORIAL 3))
```

When the stepper is invoked, it displays a line of text that includes the first subform of the specified form and the stepper prompt. The output is displayed at the left margin of your terminal in the following format:

```
: #9: (FACTORIAL 3)
Step n>
```

The letter n in the prompt represents an integer, which indicates the number of the nested command level you are in. The value of n increases by one each time the level of the stepper increases. For example, the stepper displays the following prompt when you are in the third level of the stepper:

```
Step 3>
```

After the stepper is invoked, you can use the stepper commands to control the operations the stepper performs and the way the stepper displays output.

A description of the STEP macro is provided in COMMON LISP: The Language.

4.5.2 Exiting the Stepper

Usually, when you use the stepper, you press the RETURN key until the stepper steps through the entire specified form. If you want to exit from the stepper before it steps through a form, specify the QUIT stepper command. This command causes the stepper to return control to the previous command level that was active when the stepper was invoked.

```
Step 2> QUIT
Lisp>
```

By default, the QUIT command displays a confirmation message before it causes the stepper to exit. For example:

```
Step 2> QUIT
Do you really want to exit the stepper?
```

If you respond to the message by typing YES, the stepper exits and returns control to the command level that was active when it was invoked.

```
Do you really want to exit the stepper? YES
Lisp>
```

If you respond by typing NO, the stepper prompts you for another command.

```
Do you really want to exit the stepper? NO
Step 2>
```

You can prevent the stepper from displaying the confirmation message by specifying the QUIT command with a value other than NIL. For example:

```
Step 2> QUIT T
Lisp>
```

A description of the QUIT command is provided in Section 4.5.4.2.

4.5.3 Stepper Output

Once you invoke the stepper with a specified form, the stepper displays two types of information as the LISP system evaluates the form. The two types of information are the following:

- A description of each subform of the specified form
- A description of the return value from each subform

If the subform being evaluated is a symbol, the stepper displays the descriptions in a line of text that includes the following information:

- The nested level of the symbol
- The control stack frame number that indicates where the symbol and its return value are stored
- The symbol
- The return value

The stepper indicates the nested level of a symbol with an indentation. When the number of nested levels increases, the indentation increases. After making the appropriate indentation, the stepper displays the control stack frame number, the symbol, and the return value in the following format:

```
#n:  symbol => return-value
```

If the subform being evaluated is not a symbol, the stepper displays a line of text for each description. The description of a subform consists of the following information:

- The nested level of the subform
- The control stack frame number that indicates where the subform is stored
- The subform

The stepper indicates the nested level of a subform with an indentation. When the number of nested levels increases, the indentation increases. After making the appropriate indentation, the stepper displays the control stack frame number and the subform in the following format:

```
#n: (subform)
```

The description of a return value includes the following information:

- The nested level of the return value
- The control stack frame number that indicates where the return value is stored
- The return value

The stepper also indicates the nested level of each return value with an indentation. The indentation matches the indentation of the corresponding call. After making the appropriate indentation, the stepper displays the control stack frame number and the return value in the following format:

```
#n => return-value
```

Suppose you define a function named FACTORIAL.

```
Lisp> (DEFUN FACTORIAL (N)
        (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
FACTORIAL
```

The following example illustrates the format of the output the stepper displays when you invoke it with the form (FACTORIAL 3):

```
Lisp> (STEP (FACTORIAL 3))
: #9: (FACTORIAL 3)
Step 1> STEP
: : #15: (BLOCK FACTORIAL (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
Step 2> STEP
: : : #22: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step 3> STEP
: : : : #28: (<= N 1)
Step 4> STEP
: : : : : #33: N => 3
: : : : #28 => NIL
: : : : #27: (* N (FACTORIAL (- N 1)))
Step 4> STEP
: : : : : #32: N => 3
: : : : : #31: (FACTORIAL (- N 1))
Step 5> STEP
: : : : : : #36: (- N 1)
Step 6> STEP
: : : : : : : #41: N => 3
: : : : : : #36 => 2
: : : : : : #37: (BLOCK FACTORIAL (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
Step 6> OVER
: : : : : : #37 => 2
: : : : : #31 => 2
: : : : #27 => 6
: : : #22 => 6
: : #15 => 6
: #9 => 6
6
```

Note that the FACTORIAL function is a recursive function, and in the preceding example, there are three levels of recursion. The stepper indicates the nested level of each subform with an indentation, indicated with a colon followed by a space (:). The stepper indicates the number of the stack frame in which a call is stored with an integer. The integer is preceded with a number sign and followed by a colon (#n:).

The nested level of each return value matches the indentation of the corresponding subform. The stepper indicates the number of the control stack frame the LISP system pushes the value onto with an integer that matches the stack frame number of the corresponding subform. The integer is preceded by a number sign and followed by an arrow (#n =>) that points to the return value.

4.5.4 Using Stepper Commands

The stepper commands are words that describe the operation you want the stepper to perform. You must specify some commands with arguments. Arguments modify a command; they provide the stepper with additional information on how to execute the command.

You can abbreviate stepper commands to as few characters as you like, as long as there is no ambiguity in the abbreviation.

Each time a command is executed, the stepper displays a return value if the subform returns a value, displays the next subform, and prompts you for another command. Enter a stepper command by typing the command name or abbreviation and then pressing the RETURN key. For example:

```
Step 2> STEP(RET)
: : : #22: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step 3>
```

If you press only the RETURN key, the LISP system evaluates the subform the stepper displays. If the evaluation returns a value, the stepper displays the value and the next subform and then prompts you for another command.

```
Step 2>(RET)
: : : #22: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step 3>
```

Table 4-4 provides a summary of the stepper commands. Descriptions of the stepper commands are provided in Section 4.5.4.2.

Table 4-4
Stepper Commands

Command	Description
?	Displays help text about the stepper commands.
BACKTRACE	Displays a backtrace of a form's evaluation.
DEBUG	Invokes the debugger.
EVALUATE	Evaluates a specified form with the stepper disabled.
FINISH	Finishes the evaluation of the form that was specified in the call to the STEP macro with the stepper disabled.
HELP	Displays help text about the stepper commands.
OVER	Evaluates the subform in the current stack frame with the stepper disabled.
SHOW	Displays the subform in the current stack frame.
QUIT	Exits the stepper.
RETURN	Forces the current stack frame to return a value.
STEP	Evaluates the subform in the current stack frame with the stepper enabled.
UP	Evaluates subforms with the stepper disabled until the stepper gets back to a subform that contains the subform in the current stack frame.

4.5.4.1 **Arguments** - Stepper command arguments modify the operations the stepper commands perform. You must specify some stepper commands with an argument. Some commands accept optional arguments. The arguments you can specify with the stepper commands are the following:

- Integer
- Form
- Stepper command

NOTE

Only form arguments are evaluated.

Enter an argument after the command it modifies and press the RETURN key. For example:

```
Step 3> EVALUATE (<= N 1)(RET)
```

Depending on the command, an argument is either required or optional. If an argument is required and you omit it, the stepper prompts you for the argument. For example:

```
Step 3> EVALUATE(RET)
Evaluate: (<= N 1)
```

The stepper does not prompt for arguments if you specify them in the command line.

4.5.4.2 **Stepper Commands** - The stepper provides several commands that enable you to control how it steps through a forms evaluation.

Help Commands

HELP
?

The HELP command displays help text about the stepper commands. You can specify this command with one argument. The argument must be the name of the stepper command about which you want help text. If you specify the HELP command without an argument, the stepper displays a list of the stepper commands.

You can abbreviate this command by using a question mark (?).

Evaluation Command

You can evaluate expressions while you are in the stepper. If you want the LISP system to evaluate a form, you can specify the form and then press the RETURN key. If you want the system to evaluate a symbol, you must use the EVALUATE command.

EVALUATE

The EVALUATE command causes the LISP system to explicitly evaluate a specified form. You must specify the command with an argument. The argument must be the form you want the system to evaluate. The system evaluates the form in the lexical environment of the form currently being stepped.

Debugger Command

DEBUG

The DEBUG command invokes the debugger at the control stack frame that stores the call to the current form. When the debugger returns control to the stepper, the stepper prompts you for a command.

Display Command

SHOW

The SHOW command displays the subform that is in the current stack frame such that its representation is easy to read.

Exiting Command

QUIT

The QUIT command causes the stepper to exit and return control to the command level that was active when the stepper was invoked. You can specify this command with an optional argument. If you specify NIL, the stepper displays a confirmation message before it causes the stepper to exit. If you respond to the message by typing YES, the stepper exits. If you specify a value other than NIL, the stepper does not display a message. The default value for the optional argument is NIL.

Backtrace Command

BACKTRACE

The BACKTRACE command lists the subforms of the form being stepped through. You can specify the commmand with an optional integer argument. If you specify the argument, its value determines the number of subforms that are to be listed. The stepper works its way back the specified number of subforms and then lists the subforms in the order in which they were invoked. If you do not specify the argument, the stepper lists all the subforms the LISP system is evaluating.

Commands that Continue Evaluation of the Form Being Stepped

Several stepper commands continue the evaluation of the form that is being stepped, each command continuing the evaluation in a different way.

FINISH The FINISH command evaluates the form that you specified in the call to the STEP macro. You can specify the command with an optional argument. The argument must be a form. When the stepper executes the command, the LISP system evaluates the form. If the evaluation returns a value other than NIL, the stepper steps through the evaluation of the form until it reaches the end of the evaluation. If the evaluation returns NIL, the LISP system disables the stepper and then evaluates the form you specified in the call to the STEP macro. The default value for the optional argument is NIL.

OVER The OVER command causes the LISP system to evaluate the subform in the current stack frame with the stepper disabled.

RETURN The RETURN command causes the LISP system to evaluate its argument and causes the stepper to force the current stack frame to return the values returned by the evaluation. This command must be specified with an argument that must be a form. When you execute the command, the LISP system evaluates the form. When the evaluation is complete, the current stack frame returns the values returned by the evaluated form.

STEP The STEP command causes the LISP system to evaluate the subform in the current stack frame with the stepper enabled. This command is equivalent to pressing the RETURN key.

UP The UP command causes the LISP system to evaluate subforms with the stepper disabled until control returns to the subform that contains the subform in the current stack frame. You can specify the command with an optional integer argument (n). If you specify the argument, the system evalutates subforms with the stepper disabled until control returns to the subform that contains the subform in the current stack frame n levels deep. The default value of the argument is one.

4.5.5 Sample Stepper Sessions

1.

```
Lisp> (DEFUN FIRST-ELEMENT (X) (CAR X))
FIRST-ELEMENT
Lisp> (SETF MY-LIST '(FIRST SECOND THIRD))
(FIRST SECOND THIRD)
Lisp> (STEP (FIRST-ELEMENT MY-LIST))
: #9: (FIRST-ELEMENT MY-LIST)
Step 1> STEP
: : #14: MY-LIST => (FIRST SECOND THIRD)
: : #15: (BLOCK FIRST-ELEMENT (CAR X))
Step 2> STEP
: : : #22: (CAR X)
Step 3> EVALUATE (CAR X)
FIRST
Step 3> FINISH
FIRST
Lisp>
```

2.
```
Lisp> (DEFUN PLUS-Y (X) (+ X Y))
PLUS-Y
Lisp> (SETF Y 5)
5
Lisp> (STEP (PLUS-Y 10))
: #9: (PLUS-Y 10)
Step 1> STEP
: : #15: (BLOCK PLUS-Y (+ X Y))
Step 2> EVALUATE
Evaluate: (+ X Y)
15
Step 2> STEP
: : : #22: (+ X Y)
Step 3> BACKTRACE
(PLUS-Y 10)
: (BLOCK PLUS-Y (+ X Y))
: : (+ X Y)
Step 3> SHOW
(+ X Y)
Step 3> OVER
: : : #22 => 15
: : #15 => 15
: #9 => 15
15
Lisp
```

3.
```
Lisp> (DEFUN ADDITION (X) (+ X Y))
ADDITION
Lisp> (SETF Y 5)
5
Lisp> (STEP (ADDITION 4))
: #9: (ADDITION 4)
Step 1> STEP
: : #15: (BLOCK ADDITION (+ X Y))
Step 2> STEP
: : : #22: (+ X Y)
Step 3> BACKTRACE
(ADDITION 4)
: (BLOCK ADDITION (+ X Y))
: : (+ X Y)
Step 3> EVALUATE
Evaluate: (+ X Y)
9
Step 3> STEP
: : : : #27: X => 4
: : : : #26: Y => 5
: : : #22 => 9
: : #15 => 9
: #9 => 9
9
Lisp>
```

4.6 TRACER

The VAX LISP tracer is a macro you can use to inspect a program's evaluation. The tracer informs you when a function or macro is called during a program's evaluation by printing information about each call and return value to the stream bound to the *TRACE-OUTPUT* variable. To use the tracer, you must enable it for each function and macro you want traced.

NOTE

You cannot trace special forms.

4.6.1 Enabling the Tracer

You can enable the tracer for one or more functions and/or macros by specifying the function and macro names as arguments in a call to the TRACE macro. For example:

```
Lisp> (TRACE FACTORIAL ADDITION COUNTER)
(FACTORIAL ADDITION COUNTER)
```

The TRACE macro returns a list of the functions and macros that are to be traced.

If you try to trace a function or macro that is already being traced, a warning message is displayed. To avoid this error, call the TRACE macro without an argument to produce a list of the functions and macros for which tracing is enabled. For example:

```
Lisp> (TRACE)
(FACTORIAL ADDITION COUNTER)
```

A description of the TRACE macro is provided in Section 8.7.

4.6.2 Disabling the Tracer

To disable the tracer for a function or macro, specify the name of the function or macro in a call to the UNTRACE macro. It returns a list of the functions and macros for which tracing has just been disabled. For example:

```
Lisp> (UNTRACE FACTORIAL ADDITION COUNTER)
(FACTORIAL ADDITION COUNTER)
```

You can disable tracing for all the functions for which tracing is enabled by calling the UNTRACE macro without an argument. If you try to disable tracing for a function that is not being traced, a warning message is displayed.

The UNTRACE macro is described in COMMON LISP: The Language.

4.6.3 Tracer Output

Once you enable the tracer for a function or macro, the tracer displays two types of information each time that function or macro is called during a program's evaluation. The two types of information are the following:

- A description of each call to the specified function or macro
- A description of each return value from the specified function or macro

The description of a call to a function or macro consists of a line of text that includes the following information:

- The nested level of the call
- The control stack frame number that indicates where the call is stored
- The name and arguments of the function associated with the function or macro that is called

The tracer indicates the nested level of a call with an indentation. When the number of nested levels increases, the indentation increases. After making the appropriate indentation, the tracer displays the control stack frame number, the function name, and the arguments in the following format:

```
#n: (function-name arguments)
```

The tracer also displays a line of text for the return value of each evaluation. The line of text the tracer displays for each value includes the following information:

- The nested level of the return value
- The control stack frame number that indicates where the return value is stored
- The return value

The tracer indicates the nested level of each return value with an indentation. The indentation matches the indentation of the corresponding call. After making the indentation, the tracer displays the control stack frame number and the return value in the following format:

```
#n => return-value
```

Suppose you define a function named FACTORIAL.

```
Lisp> (DEFUN FACTORIAL (N)
        (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
FACTORIAL
```

The following example illustrates the format of the output the tracer displays when the function FACTORIAL is called with the argument 3:

```
Lisp> (FACTORIAL 3)
#11: (FACTORIAL 3)
. #27: (FACTORIAL 2)
. . #43: (FACTORIAL 1)
. . #43 => 1
. #27 => 2
#11 => 6
6
```

Note that the FACTORIAL function is a recursive function, and in the case of the preceding example, there are three levels of recursion. The tracer indicates the nested level of each call with an indentation. Each level of indentation is indicated with a period followed by a space (.). The tracer indicates the number of the stack frame in which a call is stored with an integer. The integer is preceded with a number sign and followed by a colon (#n:).

The nested level of each return value matches the indentation of the corresponding call. The tracer indicates the number of the control stack frame the LISP system pushes the value onto with an integer. This integer matches the stack frame number of the corresponding call. It is preceded with a number sign and followed by an arrow (#n =>) that points to the return value.

4.6.4 Tracer Options

You can modify the output of the tracer by specifying options in the call to the TRACE macro. Each option consists of a keyword-value pair. The following call to the TRACE macro shows the format in which to specify keyword-value pairs:

```
(TRACE (name keyword-1 value-1
             keyword-2 value-2
             ...))
```

You can also specify options for a list of functions and/or macros. The following call to the TRACE macro shows the format in which to specify the same options for a list of functions and macros:

```
(TRACE ((name-1 name-2 ...) keyword-1 value-1
                            keyword-2 value-2
                            ...))
```

NOTE

> Forms the system evaluates just before or just after a call to a function or macro for which tracing is enabled are evaluated in a null lexical environment. For information on lexical environments, see COMMON LISP: The Language.

A list of the keywords you can use to specify options follows:

- ```
  :DEBUG-IF      ---
  :PRE-DEBUG-IF    |-- Invoke the debugger
  :POST-DEBUG-IF --
  ```
- ```
  :PRINT      ---
  :PRE-PRINT    |-- Add information to tracer output
  :POST-PRINT --
  ```
- :STEP-IF -- Invokes the stepper
- :SUPPRESS-IF -- Removes information from tracer output
- :DURING -- Determines when a function or macro is traced

4.6.4.1 **Invoking the Debugger** - You can cause the tracer to invoke the debugger by specifying the :DEBUG-IF, :PRE-DEBUG-IF, or :POST-DEBUG-IF keyword. These keywords must be specified with a form. The LISP system evaluates the form before, after, or before and after each call to the function or macro for which the keyword modifies the tracer. If the form returns a value other than NIL, the tracer invokes the debugger after each evaluation.

4.6.4.2 **Adding Information to Tracer Output** - You can add information to tracer output by specifying the :PRINT, :PRE-PRINT, or :POST-PRINT keyword. You must specify these keywords with a list of forms. The LISP system evaluates the list of forms and the tracer displays the return values before, after, or before and after each call to the function or macro the keyword modifies the tracer for. The tracer displays the values one per line and indents them to match other tracer output. If the forms to be evaluated cause an error, the debugger is invoked.

4.6.4.3 **Invoking the Stepper** - You can cause the tracer to invoke the stepper by specifying the :STEP-IF keyword. You must specify this keyword with a form. The LISP system evaluates the form before each call to the function or macro for which the keyword modifies the tracer. If the form returns a value other than NIL, the tracer invokes the stepper.

4.6.4.4 **Removing Information from Tracer Output** - You can remove information from tracer output by specifying the :SUPPRESS-IF keyword. You must specify this keyword with a form. The LISP system evaluates the form before each call to the function or macro for which the keyword modifies the tracer. If the form returns a value other than NIL, the tracer does not display the arguments and the return value of the function or macro being traced.

4.6.4.5 **Defining When a Function or Macro Is Traced** - You can define when a function or macro, for which tracing is enabled, is to be traced by specifying the :DURING keyword. You must specify this keyword with a function or macro name or a list of function and/or macro names. The functions and macros for which the tracer is enabled are traced only when they are called (directly or indirectly) from within one of the functions or macros whose names are specified with the keyword.

4.7 THE EDITOR

The VAX LISP Editor is a powerful, extensible editor that enables you to create and edit LISP programs. Once you have located an error and you know which function in your program is causing the error, you can use the Editor to correct the error. Use the ED function to invoke the Editor. For a complete description of the ED function, the VAX LISP Editor, and instructions on how to use the Editor, see the <u>VAX LISP Editor Manual</u>.

CHAPTER 5

THE PRETTY PRINTER

A pretty printer is a facility that formats the printed representation of LISP objects. The pretty printer inserts indentations, spaces, and line breaks into its output to increase readability and help clarify the meaning of the object that is printed. A pretty printer is particularly useful for printing large and complex lists, arrays, and structures.

The VAX LISP pretty printer* consists of two routines: a dispatch routine and an output routine. The dispatch routine performs a series of checks on several control and formatting variables whose values influence the format of an object's printed representation. The pretty printer's output routine prints the output.

You can use the VAX LISP pretty printer in three ways.

- You can use it with the default values of the control and formatting variables. By default, the pretty printer produces output that can be read back into the LISP system.
- You can control its output by changing the values of the control variables.
- You can extend it by defining formatting functions.

This chapter explains the three ways of using the pretty printer.

5.1 USING THE PRETTY PRINTER

To use the pretty printer, do one of the following:

- Call a function that invokes the pretty printer -- FORMAT-USING-PPRINT-TEMPLATE, PPRINT, PPRINT-DEFINITION, or PPRINT-PLIST.
- Set the *PRINT-PRETTY* variable to T and call the PRIN1, PRIN1-TO-STRING, PRINC, PRINC-TO-STRING, PRINT, WRITE, or WRITE-TO-STRING function.
- Specify the :PRETTY keyword in a call to the WRITE or the WRITE-TO-STRING function.

---

* The VAX LISP pretty printer is based on the pretty-printer program that is described in the paper User Format Control in a Lisp Prettyprinter, ACM TOPLAS V5 #4, pp. 513-531, October 1983. The paper and the pretty-printer program were written by Richard C. Waters, Ph.D., of the MIT Artificial Intelligence Laboratory.

- Specify the ~N or ~:N directive in the control string argument of a call to the FORMAT function.

5.1.1 Pretty-Printing Functions

You can invoke the pretty printer by calling one of the functions listed in Table 5-1.

Table 5-1
Pretty-Printing Functions

Function	Description
FORMAT-USING-PPRINT-TEMPLATE	Processes its arguments and uses the results to pretty-print output to a specified stream. If the value of the stream argument is NIL, the function creates a string that contains the output. If the value of the stream argument is T, the function pretty-prints output to the stream that is the value of the *STANDARD-OUTPUT* variable.
PPRINT	Pretty-prints its object argument to a stream. The default stream is the value of the *STANDARD-OUTPUT* variable.
PPRINT-DEFINITION	Pretty-prints the function definition of its symbol argument to a stream. The default stream is the value of the *STANDARD-OUTPUT* variable.
PPRINT-PLIST	Pretty-prints the property list of its symbol argument to a stream. The default stream is the value of the *STANDARD-OUTPUT* variable.

Detailed descriptions of the FORMAT-USING-PPRINT-TEMPLATE, PPRINT-DEFINITION, and PPRINT-PLIST functions are provided in Part II. A description of the PPRINT function is provided in COMMON LISP: The Language.

5.1.2 *PRINT-PRETTY* Variable

If you set the *PRINT-PRETTY* variable to T, the pretty printer is enabled. The following example shows the output that is printed when the variable is set to NIL:

```
Lisp> (WRITE-TO-STRING #'COUNT-EVERYTHING)
"(LAMBDA (LIST) (BLOCK COUNT-EVERYTHING (COND ((NULL LIST) Ø) ((A
TOM LIST) 1) (T (+ (COUNT-EVERYTHING (CAR LIST)) (COUNT-EVERYTHIN
G (CDR LIST)))))))"
```

- Editor -- An extensible editor that enables you to edit programs and data structures.

5.1 CONTROL VARIABLES

VAX LISP provides two variables that control the output of the debugger, the stepper, and the tracer facilities: *DEBUG-PRINT-LENGTH* and *DEBUG-PRINT-LEVEL*. These variables are analogous to the COMMON LISP variables *PRINT-LENGTH* and *PRINT-LEVEL* but are used only in the debugger.

DEBUG-PRINT-LENGTH — Controls the number of displayed elements at each level of a nested data object. The variable's value must either be an integer or NIL. The default value is NIL (no limit).

DEBUG-PRINT-LEVEL — Controls the number of displayed levels of a nested data object. The variable's value must either be an integer or NIL. The default value is NIL (no limit).

5.2 CONTROL STACK

The control stack is the part of LISP memory that stores calls to functions, macros, and special forms. The stack consists of stack frames. Each time you call a function, macro, or special form, the VAX LISP system does the following:

1. Opens a stack frame.
2. Pushes the name of the function associated with the function, macro, or special form that was called onto the stack frame.
3. Pushes the function's arguments onto the stack frame.
4. Closes the stack frame when all the function's arguments are on the stack frame.
5. Evaluates the function.

Each control stack frame has a frame number, which is displayed as part of the stack frame's output. Stack frame numbers are displayed in the output of the debugger, the stepper, and the tracer.

There is always one active stack frame, and it can either be significant or insignificant. Significant stack frames are those that invoked documented and user-created functions. Insignificant stack frames are those that invoked undocumented functions.

Debugger commands show only significant stack frames unless you specify the ALL modifier with a debugger command (see Section 5.5.3.1). Significant stack frames store one of the following calls:

- A call to a function named by a symbol that is in the current package
- A call to a function that is accessible in the current package and is explicitly or implicitly called by another function that is in the current package

See *COMMON LISP: The Language* for information on packages.

Many stack frames in the control stack store internal, undocumented functions. These stack frames are insignificant to most users; therefore, by default, the debugger does not display their representation. However, if you are using the debugger and you want to examine these stack frames, you can specify the ALL modifier with debugger commands.

5.3 ACTIVE STACK FRAME

The active stack frame is a stack frame that stores a call to a function the LISP system is evaluating. The system can evaluate a function call in the active stack frame because the frame contains all the function's argument values. Only one stack frame is active at a time and an active stack frame can exist anywhere on the control stack.

The active stack frame can have a previous active stack frame and/or it can have a next active stack frame. The previous active stack frame represents the caller of the function in the current active stack frame.

5.4 BREAK LOOP

The break loop is a read-eval-print loop that you can invoke to debug a program. You can invoke the break loop while a program is being evaluated. If you do, the evaluation is interrupted and you are placed in the loop.

Table 5-2 (Cont.)
Pretty-Printer Control Variables

Variable	Default Value	Operation
PPRINT-LEFT-MARGIN	NIL	Sets the left margin for the pretty printer. NIL means zero.
PPRINT-MAJOR-WIDTH	20	Shifts logical units of an object to the left when the remaining width available for printing is less than its value (see Section 5.2.3.1).
PPRINT-MISER-WIDTH	40	Prints in miser mode when the remaining width available for printing is less than its value (see Section 5.2.3.2).
PPRINT-RIGHT-MARGIN	NIL	Sets the right margin for the pretty printer. NIL means 72.
PPRINT-START-LINE	NIL	Sets the line number of an object at which the pretty printer is to start printing. NIL means zero.
PRINT-ARRAY	T	Determines whether the contents of arrays are to be printed.
PRINT-CIRCLE	NIL	Determines whether an object is to be checked for circularity (see Section 5.2.1.3).
PRINT-LENGTH	NIL	Controls the number of elements that can be printed for each level of a nested object. NIL means no limit.
PRINT-LEVEL	NIL	Controls the number of levels that can be printed for a nested object. NIL means no limit.

Note the following:

- Line and column numbers are zero based.
- Printing starts in the column indicated by the *PPRINT-LEFT-MARGIN* variable and on the line indicated by the *PPRINT-START-LINE* variable.
- Printing ends in the column before the column indicated by the *PPRINT-RIGHT-MARGIN* and on the line before the line indicated by the *PPRINT-END-LINE* variable.
- The number of columns the pretty printer prints is the difference between the values of the *PPRINT-RIGHT-MARGIN* and *PPRINT-LEFT-MARGIN* variables.

- The number of lines the pretty printer prints is the difference between the values of the *PPRINT-END-LINE* and *PPRINT-START-LINE* variables.

The *PRINT-ARRAY*, *PRINT-CIRCLE*, *PRINT-LENGTH*, and *PRINT-LEVEL* variables are described in COMMON LISP: The Language. The rest of the variables listed in the preceding table are described in Part II.

5.2.1 Controlling How Much Is Printed

By default, the pretty printer prints the entire object that is specified in a call to a pretty-printing function. You can control how much of an object the pretty printer prints by setting the values of the *PPRINT-END-LINE*, *PPRINT-START-LINE*, *PRINT-ARRAY*, *PRINT-CIRCLE*, *PRINT-LENGTH*, and *PRINT-LEVEL* variables.

5.2.1.1 **Pretty-Printing Sections of an Object** - The values of the *PPRINT-START-LINE* and the *PPRINT-END-LINE* variables determine at which lines the pretty printer is to start and stop printing. By default, the values of these variables are NIL. The pretty printer interprets NIL to be zero for the *PPRINT-START-LINE* variable and the end of the object for the *PPRINT-END-LINE* variable.

You can instruct the pretty printer to print a section of an object by setting the values of the *PPRINT-START-LINE* and *PPRINT-END-LINE* variables to integers. The integers you specify determine at which lines the pretty printer is to start and stop printing. If you set the values to integers, the pretty printer skips the number of lines indicated by the value of the *PPRINT-START-LINE* variable and then starts printing. The pretty printer prints up to the line indicated by the value of the *PPRINT-END-LINE* variable, prints an ellipsis (...) at the end of the last line, and stops printing. The ellipsis indicates that the output is truncated. For example, suppose you define the following function:

```
Lisp> (DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS
                                   MARRIED?)
        (UNLESS (SYMBOLP NAME)
                (ERROR "~S must be a symbol." NAME))
        (SETF (GET NAME 'AGE) AGE
              (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
              (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED?)
        NAME)
RECORD-MY-STATISTICS
```

If the value of the *PPRINT-START-LINE* variable is four and the value of the *PPRINT-END-LINE* variable is six, a call to the PPRINT-DEFINITION function produces the following output:

```
Lisp> (PPRINT-DEFINITION 'RECORD-MY-STATISTICS)
  (SETF (GET NAME 'AGE) AGE
        (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS ...
```

Use the CONTINUE function again to complete the function's evaluation.

```
Break> (CONTINUE)
100
```

Changes that you make to global variables and global definitions while you are in the break loop remain in effect after you exit the loop and your program continues. For example, if you are in the break loop and you find that the value of the variable named *FIRST* has an incorrect value, you can change the variable's value. The change remains in effect after you exit the break loop and continue your program's evaluation.

NOTE

> The forms you type while you are in the break loop are evaluated in a null lexical environment, as though they are evaluated at top level. Therefore, you cannot examine the lexical variables of a program that you interrupt with the break loop. To examine those lexical variables, invoke the debugger (see Section 5.5). For information on lexical environments, see *COMMON LISP: The Language*.

5.4.4 Break Loop Variables

The break loop uses a copy of the top-level-loop variables (plus (+), hyphen (-), asterisk (*), slash (/), and so on) the same way the top-level loop uses them (see *COMMON LISP: The Language*). These variables preserve the input expressions you specify and the output values the VAX LISP system returns while you are in the break loop.

5.5 DEBUGGER

The VAX LISP debugger is a control stack debugger. You can use it interactively to inspect and modify the LISP system's control stack frames. The debugger has a pointer that points to the current stack frame. The current stack frame is the last frame for which the debugger displayed information. The debugger provides several commands that:

- Display help
- Evaluate a form or reevaluate the function call a stack frame stores

- Handle errors
- Change which stack frame is considered current
- Inspect or modify the function call in a stack frame
- Display a summary of the control stack

The debugger reads its input from and prints its output to the streams bound to the *DEBUG-IO* and the *TRACE-OUTPUT* variables.

NOTE

The stack frames the debugger displays are no longer active.

Before you use the debugger, you should be familiar with the VAX LISP control stack. The control stack is described in Section 5.2.

5.5.1 Invoking the Debugger

The VAX LISP system invokes the debugger when errors occur. You can invoke the debugger by calling the VAX LISP DEBUG function. For example:

```
Lisp> (DEBUG)
```

When the debugger is invoked, a message that identifies the debugger, a message that identifies the current stack frame preceded by "Apply" or "Eval", and the command prompt are displayed at the left margin of your terminal in the following format:

```
Control Stack Debugger
Apply #5: (DEBUG)
Debug n>
```

The letter n in the prompt represents an integer, which indicates the number of the nested command level you are in. The value of n increases by one each time the command level increases. For example, the top-level read-eval-print loop is level 0. If an error is invoked from the top-level loop, the debugger displays the prompt Debug 1>. If you make a mistake again causing an error while within the debugger, that error causes the debugger to display the prompt Debug 2>.

After the debugger is invoked, you can use the debugger commands to inspect and modify the contents of the system's control stack.

A description of the DEBUG function is provided in Part II.

5.5.2 Exiting the Debugger

To exit the debugger, use the QUIT debugger command. It causes the debugger to return control to the previous command level.

```
Debug 2> QUIT
Debug 1>
```

If you specify the QUIT command when the debugger command level is 1 (indicated by the prompt Debug 1>), the command causes the debugger to exit and returns you to the system's top level. For example:

```
Debug 1> QUIT
Lisp>
```

By default, the QUIT command displays a confirmation message before the debugger exits if a continuable error causes the debugger to be invoked. For example:

```
Debug 1> QUIT
Do you really want to return to the previous command level?
```

If you type YES, the debugger returns control to the previous command level.

```
Do you really want to return to the previous command level? YES
Lisp>
```

If you type NO, the debugger prompts you for another command.

```
Do you really want to return to the previous command level? NO
Debug 1>
```

You can prevent the debugger from displaying the confirmation message by specifying the QUIT command with a value other than NIL. For example:

```
Debug 1> QUIT T
Lisp>
```

A description of the QUIT command is provided in Section 5.5.3.2.

5.5.3 Using Debugger Commands

The debugger commands let you inspect and modify the current control stack frame and move to other stack frames. You must specify many of

the debugger commands with one or more arguments that qualify command operations. These arguments are listed in Section 5.5.3.1.

You can abbreviate debugger commands to as few characters as you like, as long as no ambiguity is in the abbreviation.

Enter a debugger command by typing the command name or abbreviation and then pressing the RETURN key. For example:

```
Debug 1> BACKTRACE<RET>
```

If you press only the RETURN key, the debugger prompts you for another command.

Table 5-2 provides a summary of the debugger commands. Detailed descriptions of the commands are provided in Section 5.5.3.2.

Table 5-2: Debugger Commands

Command	Description
BACKTRACE	Displays a backtrace of the control stack.
BOTTOM	Moves the current frame pointer to the first stack frame on the control stack.
CONTINUE	Continues execution by returning from the continuable error that invoked the debugger.
DOWN	Moves the current frame pointer down the control stack.
ERROR	Redisplays the error message that was displayed when the debugger was invoked.
EVALUATE	Evaluates a specified form.
GOTO	Moves the pointer to a specified stack frame.
HELP (or) ?	Displays help text about the debugger commands.
QUIT	Exits to the previous command level.
REDO	Reinvokes the function in the current stack frame.
RETURN	Evaluates its arguments and causes the current stack frame to return the same values the evaluation returns.
SEARCH	Searches the control stack for a specified function.

PPRINT-MISER-WIDTH, and *PPRINT-MAJOR-WIDTH* variables are 72, 60, and 20 respectively:

```
Lisp> (PPRINT-DEFINITION 'FACTORS-OF)
(DEFUN FACTORS-OF (INTEGER)
  (IF (OR (ZEROP INTEGER) (= 1 (ABS INTEGER)))
      (LIST INTEGER)
      (DO
       ((RESULT-LIST NIL)
        (TRY-THIS-INTEGER 2)
        (REST-TO-BE-FACTORED (ABS INTEGER)))
       ((= REST-TO-BE-FACTORED 1)
        (IF
         (MINUSP INTEGER)
         (CONS -1 (NREVERSE RESULT-LIST))
         (NREVERSE RESULT-LIST)))
        (LET ((NEW-REMAINDER
               (/ REST-TO-BE-FACTORED TRY-THIS-INTEGER)))
         (COND
          ((INTEGERP NEW-REMAINDER)
           (SETF REST-TO-BE-FACTORED NEW-REMAINDER)
           (PUSH TRY-THIS-INTEGER RESULT-LIST))
          (T (INCF TRY-THIS-INTEGER)))))))
```

In the second call to the IF special form, the pretty printer uses less indentation than in the first call to the form.

When the pretty printer shifts a major logical unit to the left, it also shifts the level of indentation to the left. Whether the pretty printer uses miser mode when it shifts a major logical unit to the left depends on the actual level of indentation, not on the level of indentation it would have used if it did not shift the unit.

Suppose the values of the *PPRINT-MISER-WIDTH* and *PPRINT-MAJOR-WIDTH* variables are 60. When the pretty printer produces the output for the FACTORS-OF function, it shifts a major logical unit to the left and it uses miser mode.

```
Lisp> (PPRINT-DEFINITION 'FACTORS-OF)
(DEFUN FACTORS-OF (INTEGER)
  (IF (OR (ZEROP INTEGER) (= 1 (ABS INTEGER)))
      (LIST INTEGER)
;---  |
     (DO
      ((RESULT-LIST NIL)
       (TRY-THIS-INTEGER 2)
       (REST-TO-BE-FACTORED (ABS INTEGER)))
      ((= REST-TO-BE-FACTORED 1)
       (IF
        (MINUSP INTEGER)
        (CONS -1 (NREVERSE RESULT-LIST))
        (NREVERSE RESULT-LIST)))
;---  |
     (LET ((NEW-REMAINDER
             (/ REST-TO-BE-FACTORED TRY-THIS-INTEGER)))
      (COND
       ((INTEGERP NEW-REMAINDER)
        (SETF REST-TO-BE-FACTORED NEW-REMAINDER)
        (PUSH TRY-THIS-INTEGER RESULT-LIST))
       (T (INCF TRY-THIS-INTEGER))))
;---  |
      )
;---  |
      ))
```

5.3 EXTENDING THE PRETTY PRINTER

The pretty printer consists of two routines that run concurrently: a dispatch routine and an output routine. The dispatch routine is the part of the pretty printer that determines which formatting function the pretty printer is to use to format an object's printed representation. Because the two routines run concurrently, formatting functions do not print output. They put formatting information on a queue for the output routine, which prints the object's representation.

When the pretty printer is invoked, the dispatch routine checks the data type of the object to be printed. The data type determines which formatting function the pretty printer calls to format the object's output. The routine checks the object's data type in the following order:

- Number, bit vector, character, string, or symbol
- List
- Structure
- Array
- Other types

Numbers, bit vectors, characters, strings, and symbols are objects that can usually be printed on one line. Therefore, the pretty printer prints these objects the same way the printer prints them. Lists, structures, and arrays are more complicated objects. When the printer prints these objects, the output is difficult to read. The pretty printer formats the printed output of these objects such that output can be read more easily.

The pretty printer calls default formatting functions to format objects that are standard LISP data types. You can extend the pretty printer by defining your own formatting functions and adding the functions to the dispatch-routine algorithm. You can add formatting functions to the algorithm in three ways.

- Use the PPRINT-FORMATTER function to associate a formatting function with a symbol that names a function. After you make the association, the pretty printer uses the formatting function to format the representation of calls to the function the symbol names.
- Replace the default formatting functions for lists. The system uses three default formatting functions for lists. To change one of these functions, you must bind a formatting function that you define to one of the following variables:
  - *PPRINT-LAMBDA-APPLICATION*
  - *PPRINT-DATA-LIST*
  - *PPRINT-FUNCTION-CALL*

  When the value of these variables is NIL, the pretty printer uses the system's default formatting functions.

- Push formatting functions that format arrays and other objects onto the list that is bound to one of the following formatting variables:

  - *PPRINT-ARRAY-FORMATTERS*

  - *PPRINT-SPECIAL-FORMATTERS*

  By default, the value of these variables is NIL. When the value is NIL, the dispatch routine disregards the variables and continues to search for the correct formatting function.

Before you define a formatting function and extend the pretty printer, you must understand the dispatch-routine algorithm. You must know where in the algorithm you want the routine to check whether your formatting function applies to the object being printed.

5.3.1 Dispatch-Routine Algorithm

This section describes the algorithm the pretty printer's dispatch routine uses to decide which formatting function to call to format an object. Figure 5-1 illustrates the steps taken by the algorithm.

The dispatch routine calls all formatting functions with one argument, the object being pretty-printed.

I. The dispatch routine calls the formatting functions in the list bound to the *PPRINT-SPECIAL-FORMATTERS* variable, starting with the first function. Calls to the EXPAND-PPRINT-TEMPLATE macro produce output regardless of the value the function returns.

   A. If a function returns a value other than NIL, the dispatch routine stops.

   B. If a function returns NIL, the routine does one of the following:

      1. If the function is not the last function in the list, the routine evaluates the next function in the list.

      2. If the function is the last function, the routine goes to Step II.

II. The routine checks whether the object is a number, bit vector, character, string, or symbol.

   A. If it is not one of these data types, the routine goes to Step III.

   B. If it is one of these data types, the dispatch routine stops and the pretty printer prints the object in a format that is similar to the output produced by the PRIN1 or the PRINC function.

III. The routine checks whether the object is a list.

   A. If the object is not a list, the routine goes to Step IV.

   B. If the object is a list, the routine checks whether the first element of the list is a symbol that is associated with a formatting function.

1. If the first element is such a symbol, the routine calls the formatting function the symbol is associated with and stops.

2. If the first element is not such a symbol, the routine calls one of the three default formatting functions for lists. These formatting functions are bound to variables. The routine checks whether the list represents a list of data, a call to a function, or an application of a lambda expression and uses the appropriate function. The variables the default formatting functions are bound to are the following:

   - *PPRINT-DATA-LIST*
   - *PPRINT-FUNCTION-CALL*
   - *PPRINT-LAMBDA-APPLICATION*

IV. The routine checks whether the object is a structure.

A. If the object is not a structure, the routine goes to Step V.

B. If the object is a structure, the routine checks whether a print function was specified in the structure's definition.

1. If a print function was specified in the definition, the dispatch routine calls that function to format the structure and stops.

2. If a print function is not specified in the definition, the routine calls the system's default formatting function for structures to format the structure and stops.

V. The routine checks whether the object is an array.

A. If the object is not an array, the routine goes to Step VI.

B. If the object is an array, the routine does the following:

1. Checks the value of the *PRINT-ARRAY* variable.

   a. If the value is not NIL, the routine goes to Step 2.

   b. If the value is NIL, the dispatch routine stops and the pretty printer prints the array in a format that is similar to the output produced by the PRIN1 or the PRINC function.

2. The dispatch routine evaluates the formatting functions in the list bound to the *PPRINT-ARRAY-FORMATTERS* variable, starting with the first function. Calls to the EXPAND-PPRINT-TEMPLATE macro produce output regardless of the value the function returns.

a. If a function returns a value other than NIL, the dispatch routine stops and the pretty printer uses the value to format the object.

b. If a function returns NIL, the routine does one of the following:

   i. If the function is not the last function in the list, the routine evaluates the next function in the list.

   ii. If the function is the last function, the routine goes to Step VI.

VI. The pretty printer prints the object in a format that is similar to the output produced by the PRIN1 or the PRINC function and the formatting routine stops.

Figure 5-1 illustrates the dispatch routine's algorithm.

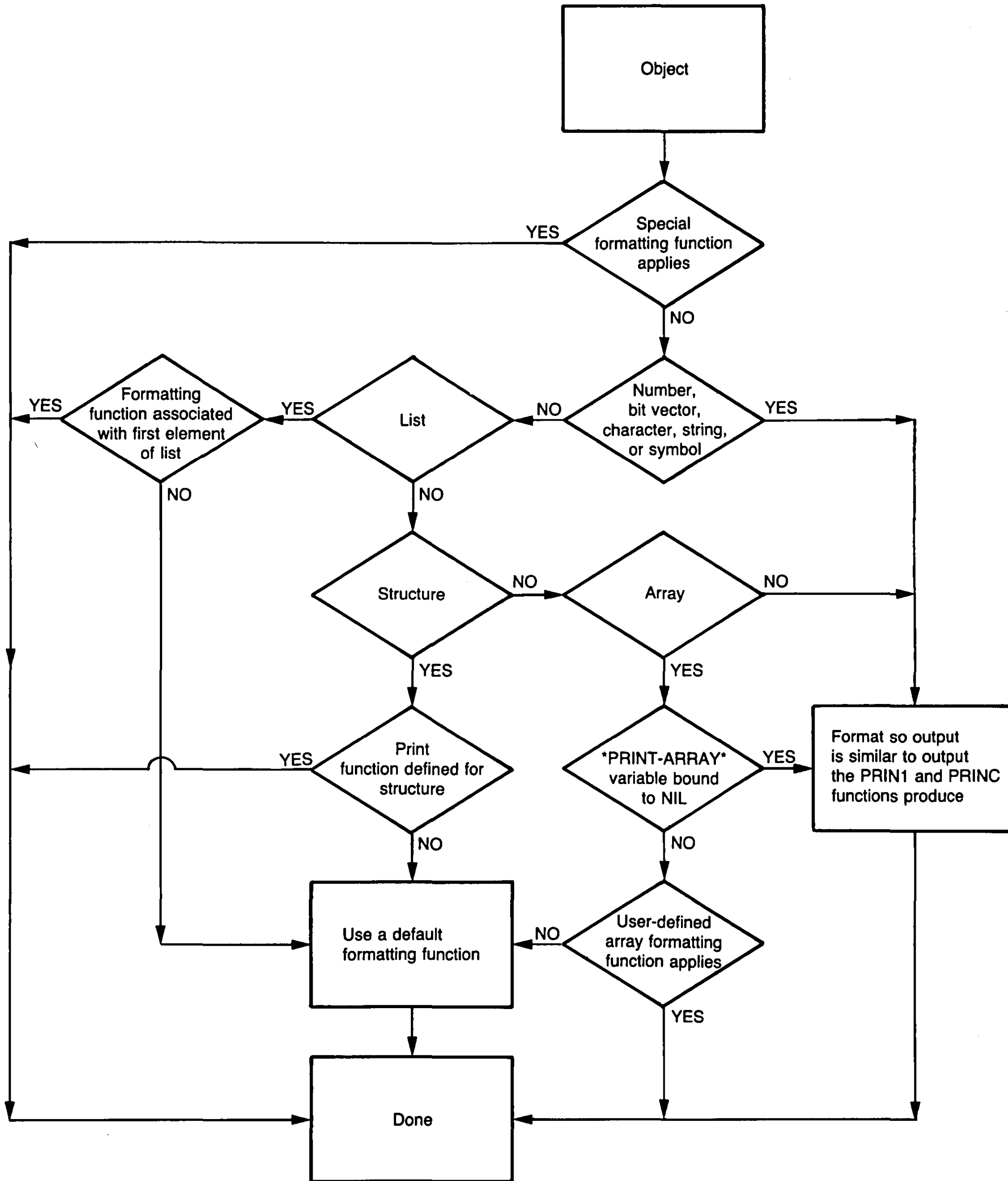


Figure 5-1 Dispatch-Routine Algorithm

The following list describes what the command displays when you specify each modifier:

- ARGUMENTS -- A list of the arguments in the current stack frame.
- CALL -- The function call that created the current stack frame. The command displays the function call so that its output is easy to read. The arguments in the call are represented by their values.
- FUNCTION -- The function in the current stack frame. The function can be either interpreted or compiled with the COMPILE function. The function cannot be displayed if it is a system function or if it is loaded from a compiled file.
- HERE -- A description of the current stack frame.

Backtrace Command

BACKTRACE

The BACKTRACE command displays the argument list of each stack frame in the control stack, starting from the top of the stack. You can specify the command with modifiers to specify the style and extent of the backtrace.

The modifiers you can specify are ALL, NORMAL, QUICK, HERE, TOP, or VERBOSE. By default, the command uses the NORMAL and the TOP modifiers. The following list describes the style or extent the BACKTRACE command uses when you specify each modifier:

- ALL -- Displays significant and insignificant stack frames.
- NORMAL -- Displays the function name and argument list in each stack frame.
- QUICK -- Displays the function name in each stack frame.
- HERE -- Starts the backtrace at the current stack frame.
- TOP -- Starts the backtrace at the top of the control stack.
- VERBOSE -- Displays the function name, argument list, and local variable bindings in each stack frame.

5.5.4 Using the DEBUG-CALL Function

The DEBUG-CALL function returns a list representing the call at the current debug stack frame. This function is a debugging tool and takes no arguments. The list returned by DEBUG-CALL can be used to access the values passed to the function in the current stack frame. If used outside the debugger, DEBUG-CALL returns NIL. The following example shows how to use the function:

```
Lisp> (DEFVAR ADJUSTABLE-STRING
               (MAKE-ARRAY 10 :ELEMENT-TYPE 'STRING-CHAR
                              :INITIAL-ELEMENT #\SPACE
                              :ADJUSTABLE T))
ADJUSTABLE-STRING
Lisp> (SCHAR ADJUSTABLE-STRING 3)

Fatal error in function SCHAR (signaled with ERROR).
Argument must be a simple-string: "          "

Control Stack Debugger
Apply #4: (SCHAR "          " 3)
Debug 1> (TYPE-OF (SECOND (DEBUG-CALL)))

(STRING 10)
Debug 1> RET #\SPACE
#\SPACE
```

In this case, the function in the current stack frame is SCHAR. The call to (DEBUG-CALL) returns the list (SCHAR " " 3). The form (SECOND (DEBUG-CALL)) returns the first argument to SCHAR in the current stack frame. Calling TYPE-OF with this LISP object determines that the first argument to SCHAR is of type (STRING 10) and not a simple string.

5.5.5 Sample Debugging Sessions

1.
```
Lisp> (DEFUN FIRST-ELEMENT (X) (CAR X))
FIRST-ELEMENT
Lisp> (FIRST-ELEMENT 3)

Fatal error in function CAR (signaled with ERROR).
Argument must be a list: 3

Control Stack Debugger
Apply #8: (CAR 3)
Debug 1> DOWN
Eval  #7: (CAR X)
Debug 1> DOWN
Eval  #6: (BLOCK FIRST-ELEMENT (CAR X))
Debug 1> DOWN
```

```
Apply #4: (FIRST-ELEMENT 3)
Debug 1> SHOW HERE
It is a cons
Format: FIRST-ELEMENT x
-- Arguments --
X : 3
Debug 1> SET
Type of SET operation: ARGUMENT
Argument Name: X
New Value: '(1 2 3)
Debug 1> WHERE
Apply #4: (FIRST-ELEMENT (1 2 3))
Debug 1> REDO
1
Lisp>
```

The argument in a stack frame is changed from an integer to a list, and the function is reevaluated with the correct argument.

```
Lisp> (DEFUN PLUS-Y (X) (+ X Y))
PLUS-Y
Lisp> (PLUS-Y 4)

Fatal error in function SYSTEM::INTERPRET (signaled with
ERROR).
Symbol has no value: Y

Control Stack Debugger
Eval  #8:  Y
Debug 1>  DOWN
Eval  #7: (+ X Y)
Debug 1>  DOWN
Eval  #6: (BLOCK PLUS-Y (+ X Y))
Debug 1> (SETF Y 1)
1
Debug 1> WHERE
Eval  #6: (BLOCK PLUS-Y (+ X Y))
Debug 1> EVALUATE
Evaluate: Y
1
Debug 1> DOWN
Apply #4: (PLUS-Y 4)
Debug 1> REDO
5
Lisp>
```

The value of the variable Y is set with the SETF macro, and the body of the function PLUS-Y is reevaluated.

3.
```
Lisp>(DEFUN ONE-PLUS (X) (1+ X))
ONE-PLUS
Lisp>(ONE-PLUS '(1 2 3 4))

Fatal error in function 1+ (signaled with ERROR).
Argument must be a number: (1 2 3 4)

Control Stack Debugger
Apply #8: (1+ (1 2 3 4))
Debug 1>SET FUNCTION
Function:'CAR
Debug 1>WHERE
Apply #8: (CAR (1 2 3 4))
Debug 1>DOWN
Eval #7: (1+ X)
Debug 1>UP
Apply #8: (CAR (1 2 3 4))
Debug 1>REDO
1
Lisp>(PPRINT-DEFINITION 'ONE-PLUS)
(DEFUN ONE-PLUS (X) (1+ X))
Lisp>
```

This example shows that changing the contents of a stack frame does not change the contents of other stack frames or the function that was originally evaluated.

5.6 STEPPER

The stepper is a facility you can use to step interactively through the evaluation of a form. You can control the stepper with stepper commands as it displays and evaluates each subform of a specified form.

The stepper has a pointer that points to the current stack frame on the system's control stack. The current stack frame is the last frame for which the stepper displayed information.

The stepper prints its command interaction to the stream bound to the *DEBUG-IO* variable; it prints its output to the stream bound to the *TRACE-OUTPUT* variable.

5.6.1 Invoking the Stepper

You can invoke the stepper by calling the STEP macro with a form as an argument. The following example invokes the stepper with a call to a function named FACTORIAL:

```
Lisp> (STEP (FACTORIAL 3))
```

When the stepper is invoked, it displays a line of text that includes the first subform of the specified form and the stepper prompt. The output is displayed at the left margin of your terminal in the following format:

```
#9: (FACTORIAL 3)
Step>
```

5.3.4 Templates

The EXPAND-PPRINT-TEMPLATE macro and the FORMAT-USING-PPRINT-TEMPLATE function must be called with a template argument whose value is a string. A template is a string of directives that are used to produce pretty-printer formatting code. Directives are single-character commands the macro and the function convert to pretty-printer formatting code. You can use directives to instruct the pretty printer to do the following:

- Use the specified text as literal text.
- Call the dispatch routine to format template arguments according to their data types.
- Produce spacing and line breaks.
- Call the dispatch routine to format nested structures.
- Call a formatting function.

Some of the directives must be followed by a parameter while other directives can be followed by a parameter. The four types of parameters are the following:

- Integer parameter (n). An integer parameter can be positive or negative. If you omit this parameter, a default value is used. An example of a directive specified with an integer parameter is "~8", which means add eight spaces.
- Function-name parameter (f). A function-name parameter must be a symbol that has a formatting function definition. You must terminate the symbol with a whitespace character, such as #\SPACE or the end of the template. If a directive requires this parameter, you cannot omit it. This parameter does not have a default value. An example of a directive specified with a function-name parameter is "&My-Formatting-Function", which means call the function MY-FORMATTING-FUNCTION with no arguments.
- Subtemplate parameter. Directives you can specify with a subtemplate parameter delimit the subtemplate directly or indirectly with quotes. An example of a directive specified with a subtemplate parameter is " [*.*]", which means the square bracket directive delimits the subtemplate *.* directly. Another example is "$\"*~*\"", which means the dollar sign directive ($) delimits the subtemplate *~* indirectly.
- Number-sign parameter (#). You can specify the number-sign parameter in place of integer and function-name parameters. This parameter indicates that the next argument in the template's argument list is to be used as a parameter instead of a literal value. For example, (EXPAND-PPRINT-TEMPLATE "~#" 8) has the same result as (EXPAND-PPRINT-TEMPLATE "~8"). Another example is (EXPAND-PPRINT-TEMPLATE "$EXAMPLE" X), which has the same result as (EXPAND-PPRINT-TEMPLATE "$#" X #'EXAMPLE).

The EXPAND-PPRINT-TEMPLATE macro and the FORMAT-USING-PPRINT-TEMPLATE function can also be specified with a list of arguments that are used by the specified template. The arguments in the argument list can be any LISP expression, but the simplest case is when an argument is a subobject of the object being pretty-printed. Some directives simply modify pretty-printer output without referring to an argument while other directives refer to the arguments in the template's argument list. Directives that refer to arguments base their output on the argument.

NOTE

Even though the arguments in the template's argument list can be any type of LISP expression, the rest of this chapter refers to the arguments as subobjects.

Table 5-3 lists the pretty-printer directives with brief descriptions. Detailed descriptions are provided in Part II in the descriptions of the EXPAND-PPRINT-TEMPLATE macro and the FORMAT-USING-PPRINT-TEMPLATE function. Sections 5.3.4.1 through 5.3.4.5 explain how to use the directives.

Table 5-3
Pretty-Printer Directives

Directive	Description
' '	Apostrophes enclose literal text.
*	An asterisk calls the dispatch routine to format the corresponding subobject by data type.
P	P calls the dispatch routine to format the corresponding subobject such that its output is similar to the output the PRIN1 function produces.
C	C calls the dispatch routine to format the corresponding subobject such that its output is similar to the output the PRINC function produces.
S	S calls the dispatch routine to format the corresponding subobject such that its output is similar to the output the PRINC function produces. The subobject is not counted when the pretty printer computes the current level and length of indentation.
I	I instructs the pretty printer to ignore the corresponding subobject.
~	Tilde inserts spaces. You can specify this directive with an optional integer parameter.
T	T inserts spaces; it is similar to a tab. You can specify this directive with an optional integer parameter.

(Continued on next page)

Table 5-3 (Cont.)
Pretty-Printer Directives

Directive	Description
+	Plus sign changes the pretty printer's current indentation level. You can specify this directive with an optional integer parameter.
!	Exclamation point inserts a line break.
N	N inserts a line break if not specified within the brace or parentheses directive. If specified within one of these directives, N inserts a line break if the pretty printer cannot print the subobject the brace or parentheses directive refers to on one line.
B	B inserts a line break if the pretty printer cannot print the next subobject on the current line.
M	M inserts a line break if the pretty printer cannot print the entire object on one line or if the remaining width available for printing is less than the value of the *PPRINT-MISER-WIDTH* variable.
-	Hyphen, when specified with an integer parameter, is equivalent to ~nN.
,	Comma, when specified with an integer parameter, is equivalent to ~nB.
;	Semicolon, when specified with an integer parameter, is equivalent to ~1TnB.
_	Underscore, when specified with an integer parameter, is equivalent to ~nM.
{ }	Braces enclose a subtemplate that formats a logical unit of a subobject.
[ ]	Square brackets enclose a subtemplate that formats the elements of a list. It refers to one subobject in the template's argument list.
( )	Parentheses, when specified with an integer and subtemplate parameters, are an abbreviation for {n '(' [subtemplate] ')'}.
.	Period is used within the square bracket or parenthesis directive. It causes the next directive that refers to a subobject to use a list that contains the remaining elements of the subobject.
< >	Angle brackets are used within the square bracket or parenthesis directive. They cause the pretty printer to repeat the enclosed subtemplate until all the elements of the subobject are used.
&	Ampersand causes the dispatch routine to call the specified formatting function with no arguments.

(Continued on next page)

Table 5-3 (Cont.)
Pretty-Printer Directives

Directive	Description
%	Percent sign causes the dispatch routine to call the specified formatting function to format the corresponding subobject in the template's argument list.
$	Dollar sign checks the data type of the corresponding subobject in the template's argument list. If the subobject is a list, the dispatch routine uses the specified formatting function or subtemplate with the list as the argument. If it is not a list, the dispatch routine formats the subobject according to its data type.

If you use a pretty-printer template that contains an error, an error message is printed and the pretty printer stops. The pretty printer does not enter the debugger; it returns control to the current command level.

The case of the directives is not significant.

You can specify arbitrary amounts of whitespace in templates to improve readability. The whitespace characters, such as the #\NEWLINE and the #\SPACE characters, have no effect on pretty-printed output unless they are enclosed by the apostrophe directive.

You must call the EXPAND-PPRINT-TEMPLATE macro one or more times in a formatting function's definition. You can call the macro successively rather than call it once to produce the same output. Consider the following:

```
Lisp> (DEFUN MY-FORMATTER (OBJECT)
        (PROGN
          (EXPAND-PPRINT-TEMPLATE "'I like'")
          (EXPAND-PPRINT-TEMPLATE "~'VAX LISP.'")))
MY-FORMATTER
```

The preceding formatting function definition includes two calls to the EXPAND-PPRINT-TEMPLATE macro. A call to the function produces the same output as the call to the EXPAND-PPRINT-TEMPLATE macro in the following example:

```
Lisp> (DEFUN MY-FORMATTER (OBJECT)
        (EXPAND-PPRINT-TEMPLATE "'I like VAX LISP.'"))
MY-FORMATTER
```

The FORMAT-USING-PPRINT-TEMPLATE function is useful for pretty-printing an object directly or for seeing the output a template produces. A description of this function is provided in Part II.

5.3.4.1 **Literal Text** - If you want to specify literal text in a template you must enclose the text with the apostrophe directive ('). When you use this directive, the case of the characters is preserved. For example:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL "'I like VAX LISP.'")
"I like VAX LISP."
```

To include an apostrophe in literal text, quote the apostrophe with an apostrophe. For example:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL "' ''(CAR 3)'")
" '(CAR 3)"
```

5.3.4.2 **Formatting Template Arguments** - Several pretty-printer directives refer to subobjects in the template's argument list. Five of these directives are an asterisk (*), P, C, S, and I. The asterisk directive (*) signals a recursive call to the dispatch routine to format the subobject it refers to. The dispatch routine uses the subobject's data type to determine the format the pretty printer is to use to print the subobject's output. The following example includes the asterisk directive:

```
Lisp> (SETF ORDER '(FIRST SECOND THIRD))
(FIRST SECOND THIRD)
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                     "'(' * ' ' * ' ' * ')'"
                                     (NTH Ø ORDER)
                                     (NTH 1 ORDER)
                                     (NTH 2 ORDER))
"(FIRST SECOND THIRD)"
```

The asterisk directive causes the pretty printer's dispatch routine to be called recursively for each element of the list (FIRST SECOND THIRD).

You can also format a subobject in the template's argument list with the P, C, or S directive. If you specify the P directive, the subobject the directive refers to is formatted such that its output is similar to the output the PRIN1 function produces. If you specify the C or the S directive, the subobject the directive refers to is formatted such that its output is similar to the output the PRINC function produces. The difference between the C and the S directives is that a subobject that the C directive refers to is counted as part of the value of the *PRINT-LENGTH* variable and a subobject that the S directive refers to is not counted.

The following example shows the output the pretty printer produces when you use these three directives:

```
Lisp> (SETF ORDER '("FIRST" "SECOND" "TH\"I\"RD"))
("FIRST" "SECOND" "TH\"I\"RD")
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                     "'(' P ' ' C ' ' S ')'"
                                     (NTH Ø ORDER)
                                     (NTH 1 ORDER)
                                     (NTH 2 ORDER))
"(\"FIRST\" SECOND TH\"I\"RD)"
```

The pretty printer prints the first element of the template's argument list with escape characters because the P directive caused the *PRINT-ESCAPE* variable to be bound to T. The pretty printer prints the second and third arguments without escape characters because the C and S directives caused the *PRINT-ESCAPE* variable to be bound to NIL. The third argument is pretty-printed with embedded double quotes because they are part of the argument's literal text.

The I directive also refers to arguments in the template's argument list. This directive instructs the pretty printer to ignore the argument it refers to. The argument is evaluated, but the pretty printer produces no output.

5.3.4.3 **Spaces and Line Breaks** - You can use several directives to control the spacing the pretty printer uses to format output. These directives can be categorized as follows:

- Directives that add or subtract spaces
- Directives that add line breaks
- Directives that add or subtract spaces or add line breaks, depending on specific conditions

The most direct way to add spaces to pretty-printer output is to specify the tilde directive (~). You can specify it with an integer parameter, which defaults to one. The integer can be positive or negative. If you specify a positive integer, the pretty printer adds the number of spaces equal to the integer. If you specify a negative integer, the pretty printer deletes the number of spaces indicated by the integer. The pretty printer can delete only spaces that it has a record of. The following example includes the tilde directive.

```
Lisp> (SETF ORDER '(FIRST SECOND THIRD))
(FIRST SECOND THIRD)
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'(' * ~ * ~ * ')'"
                                    (NTH Ø ORDER)
                                    (NTH 2 ORDER))
"(FIRST SECOND THIRD)"
```

The plus sign directive (+) alters the current default level of indentation. You can specify this directive with an integer parameter, which defaults to one. The integer can be positive or negative. If you specify a positive integer, the pretty printer increases the indentation by the number of spaces indicated by the integer. If you specify a negative integer, the pretty printer decreases the indentation by the number of spaces indicated by the integer. If you specify one, the indentation the pretty printer prints after line breaks is increased by one space.

If you want the pretty printer to produce output that has a tabular format, specify the T directive. You can specify this directive with an integer parameter. The pretty printer uses the parameter to determine the width of the columns. It inserts spaces in the output. The spaces make the difference between the current level of indentation and the cursor position (after the pretty printer inserts the spaces) divisible by the parameter value. If you do not specify an integer parameter, the pretty printer chooses a reasonable default value. However, the default value might not be large enough, especially if you are pretty-printing a large data structure.

Consider the template "'FIRST' T4 'SECOND'". This template instructs the pretty printer to print the first character of SECOND Ix4 spaces after the first character of FIRST, where I is the smallest positive integer such that the two words do not overlap. In this case, I is two and there are three spaces between the words FIRST and SECOND in output.

You might want to explicitly insert a space in the output to prevent the output from running together. Suppose you specify the template "'FIRST' T5 'SECOND'". The output this template produces looks like the following:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'FIRST' T5 'SECOND'"
"FIRSTSECOND"
```

Note how the words run together because FIRST is five characters long. To insert spaces between the words you can specify the tilde directive in the template. The template "'FIRST'~ T5 'SECOND'", produces the following output:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'FIRST' ~ T5 'SECOND'")
"FIRST      SECOND"
```

When the pretty printer inserts a line break into its output, it calls the FRESH-LINE function and indents the correct number of spaces. The FRESH-LINE function outputs a #\NEWLINE character if the stream the pretty printer is printing to is not at the beginning of a line. If you specify consecutive directives that cause a line break, the pretty printer outputs only one #\NEWLINE character.

You can cause the pretty printer to insert a line break by specifying the exclamation point (!) directive. The following example illustrates the use of the exclamation point directive:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'(' * ! * ! * ')'"
                                    (NTH Ø ORDER)
                                    (NTH 1 ORDER)
                                    (NTH 2 ORDER))
"(FIRST
SECOND
THIRD)"
```

In the preceding example, the pretty printer did not add indentation after it inserted line breaks. The three lines of text are printed at the left margin.

The following example shows how you can use the exclamation point and the tilde directives to format output with line breaks and spacing:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'(' * ! ~2 * ! * ')'"
                                    (NTH Ø ORDER)
                                    (NTH 1 ORDER)
                                    (NTH 2 ORDER))
"(FIRST
  SECOND
THIRD)"
```

In this example, the tilde directive caused the pretty printer to insert two spaces before it printed the second subobject. The spaces are inserted only before the second subobject and they are not inserted before the third subobject.

The next example illustrates the use of the exclamation point and the plus sign directives to control the indentation of pretty-printer output:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'(' * ! +2 * ! * ')'"
                                    (NTH 0 ORDER)
                                    (NTH 1 ORDER)
                                    (NTH 2 ORDER))
"(FIRST
SECOND
  THIRD)"
```

The change in the level of indentation caused by the plus sign directive did not appear until the next line break was signaled. To change the level of indentation before the pretty printer prints the second subobject, you must specify the plus sign directive before the first exclamation point directive in the template. For example:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL
                                    "'(' * +2 ! * ! * ')'"
                                    (NTH 0 ORDER)
                                    (NTH 1 ORDER)
                                    (NTH 2 ORDER))
"(FIRST
  SECOND
  THIRD)"
```

You can instruct the pretty printer to insert a line break if a specific condition exists by including the N, B, or M directive in a template.

The N directive instructs the pretty printer to insert a line break if one of the following conditions exists:

- The directive is not specified within the brace or parentheses directive (see Section 5.3.4.4).
- If the directive is specified within the brace or parentheses directive and the subobject the brace or parentheses directive refers to does not fit on one line.
- If the directive is specified within the brace or parentheses directive and another directive in the subtemplate has already instructed the pretty printer to insert a line break.

Consider the following example:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL "'A' N 'B' N 'C'")
"A
B
C"
```

The pretty printer inserts a line break each time it encounters the N directive. In the following example, the N directive is specified within the brace directive:

```
Lisp> (FORMAT-USING-PPRINT-TEMPLATE NIL "{'A' N 'B' N 'C'}")
"ABC"
```

the EVALUATE command. See *COMMON LISP: The Language* for a description of dynamic and lexical environment variables.

Some COMMON LISP functions (for example, EVALHOOK, APPLYHOOK, and MACROEXPAND) take an optional environment argument. The value bound to the *STEP-ENVIRONMENT* variable can be passed as an environment to these functions to allow evaluaton of forms in the context of the stepped form.

5.6.5.3 **Example Use of Stepper Variables** - The following example illustrates the use of the *STEP-FORM* and *STEP-ENVIRONMENT* special variables.

```
Lisp> (SETF X "Top level value of X")
"Top level value of X"
Lisp> (DEFUN FIBONACCI (X)
        (IF (< X 3) 1
            (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))))
FIBONACCI
Lisp> (STEP (FIBONACCI 5))
#4: (FIBONACCI 5)
Step> STEP
: #10: (BLOCK FIBONACCI (IF (< X 3) 1
                           (+ (FIBONACCI (- X 1))
                              (FIBONACCI (- X 2)))))
Step> STEP
: : #14: (IF (< X 3) 1 (+ (FIBONACCI (- X 1))
                          (FIBONACCI (- X 2))))
Step> STEP
: : : #18: (< X 3)
Step> STEP
: : : : #22: X => 5
: : : #18 => NIL
: : : #17: (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))
Step> STEP
: : : : #21: (FIBONACCI (- X 1))
Step> STEP
: : : : : #25: (- X 1)
Step> STEP
: : : : : : #29: X => 5
: : : : : #25 => 4
: : : : : #27: (BLOCK FIBONACCI (IF (< X 3) 1
                                   (+ (FIBONACCI (- X 1))
                                      (FIBONACCI (- X 2)))))
Step> STEP
: : : : : : #31: (IF (< X 3) 1
                     (+ (FIBONACCI (- X 1))
                        (FIBONACCI (- X 2))))
Step> STEP
: : : : : : : #35: (< X 3)
```

```
Step> STEP
: : : : : : : : #39: X => 4
: : : : : : : #35 => NIL
: : : : : : : #34: (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))
Step> STEP
: : : : : : : : #38: (FIBONACCI (- X 1))
Step> EVAL *STEP-FORM*
(FIBONACCI (- X 1))
Step> STEP
: : : : : : : : : #42: (- X 1)
Step> STEP
: : : : : : : : : : #46: X => 4
: : : : : : : : : #42 => 3
: : : : : : : : : #44: (BLOCK FIBONACCI
                            (IF (< X 3) 1
                              (+ (FIBONACCI (- X 1))
                                 (FIBONACCI (- X 2))))))
Step> EVAL *STEP-FORM*
(BLOCK FIBONACCI
  (IF (< X 3) 1 (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))))
Step> STEP
: : : : : : : : : : #48: (IF (< X 3) 1
                              (+ (FIBONACCI (- X 1))
                                 (FIBONACCI (- X 2))))
Step> STEP
: : : : : : : : : : : #52: (< X 3)
Step> STEP
: : : : : : : : : : : : #56: X => 3
: : : : : : : : : : : #52 => NIL
: : : : : : : : : : : #51: (+ (FIBONACCI (- X 1))
                               (FIBONACCI (- X 2)))
Step> STEP
: : : : : : : : : : : : #55: (FIBONACCI (- X 1))
Step> EVAL X
3
Step> (EVAL 'X)
"Top level value of X"
Step> EVAL *STEP-FORM*
(FIBONACCI (- X 1))
Step> (EVALHOOK 'X NIL NIL NIL)
"Top level value of X"
Step> (EVALHOOK 'X NIL NIL *STEP-ENVIRONMENT*)
3
Step> (EVALHOOK (CADR *STEP-FORM*) NIL NIL *STEP-ENVIRONMENT*)
2
Step> STEP
: : : : : : : : : : : : : #59: (- X 1)
Step> STEP
: : : : : : : : : : : : : : #63: X => 3
: : : : : : : : : : : : : #59 => 2
: : : : : : : : : : : : : #61: (FIBONACCI
                                 (IF (< X 3) 1
```

```
                                        (+ (FIBONACCI (- X 1))
                                           (FIBONACCI (- X 2)))))
    Step> FINISH
    5
```

This example shows that the *STEP-FORM* special variable is bound to the form being evaluated while stepping. The example also shows that the *STEP-ENVIRONMENT* special variable is bound to the lexical environment in which the currently stepped form is being evaluated.

The call to EVALHOOK evaluates the form (- X 1) in the lexical environment of the stepper, that is, with the local binding of X. A call to EVALHOOK with a null environment specified shows that X's value in the null lexical environment differs from that in the stepper. The EVAL command uses the *STEP-ENVIRONMENT* environment; the EVAL function uses the null lexical environment.

5.6.6 Sample Stepper Sessions

1.
```
Lisp> (DEFUN FIRST-ELEMENT (X) (CAR X))
FIRST-ELEMENT
Lisp> (SETF MY-LIST '(FIRST SECOND THIRD))
(FIRST SECOND THIRD)
Lisp> (STEP (FIRST-ELEMENT MY-LIST))
#10: (FIRST-ELEMENT MY-LIST)
Step> STEP
: #15: MY-LIST => (FIRST SECOND THIRD)
: #17: (BLOCK FIRST-ELEMENT (CAR X))
Step> STEP
: : #22: (CAR X)
Step> EVALUATE (CAR X)
FIRST
Step> FINISH
FIRST
Lisp>
```

2.
```
Lisp> (DEFUN PLUS-Y (X) (+ X Y))
PLUS-Y
Lisp> (SETF Y 5)
5
Lisp> (STEP (PLUS-Y 10))
#10: (PLUS-Y 10)
Step> STEP
: #17: (BLOCK PLUS-Y (+ X Y))
Step> EVALUATE
Evaluate: (+ X Y)
15
Step> STEP
: : #22: (+ X Y)
```

```
Step> BACKTRACE
(+ X Y)
(BLOCK PLUS-Y (+ X Y))
(PLUS-Y 10)
Step> SHOW
(+ X Y)
Step> OVER
: : #22 => 15
: #17 => 15
#10 => 15
15
Lisp>
```

3.
```
Lisp> (DEFUN ADDITION (X) (+ X Y))
ADDITION
Lisp> (SETF Y 5)
5
Lisp> (STEP (ADDITION 4))
#10: (ADDITION 4)
Step> STEP
: #17: (BLOCK ADDITION (+ X Y))
Step>
: : #22: (+ X Y)
Step> BACKTRACE
(+ X Y)
(BLOCK ADDITION (+ X Y))
(ADDITION 4)
Step> EVALUATE
Evaluate: (+ X Y)
9
Step> STEP
: : : #27: X => 4
: : : #27: Y => 5
: : #22 => 9
: #17 => 9
#10 => 9
9
Lisp>
```

5.7 TRACER

The VAX LISP tracer is a macro you can use to inspect a program's evaluation. The tracer informs you when a function or macro is called during a program's evaluation by printing information about each call and return value to the stream bound to the *TRACE-OUTPUT* variable. To use the tracer, you must enable it for each function and macro you want traced.

Suppose you define a function named FACTORIAL.

```
Lisp> (DEFUN FACTORIAL (N)
        (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
FACTORIAL
```

The following example illustrates the format of the output the tracer displays when the function FACTORIAL is called with the argument 3:

```
Lisp> (FACTORIAL 3)
#11: (FACTORIAL 3)
. #27: (FACTORIAL 2)
. . #43: (FACTORIAL 1)
. . #43 => 1
. #27 => 2
#11 => 6
6
```

The FACTORIAL function is a recursive one and, in the case of the preceding example, has three levels of recursion. The tracer indicates the nested level of each call with indentation. Each level of indentation is indicated with a period followed by a space (.). The tracer indicates the number of the stack frame in which a call is stored with an integer. The integer is preceded with a number sign and followed by a colon (#n:).

The nested level of each return value matches the indentation of the corresponding call. The tracer indicates the number of the control stack frame onto which the LISP system pushes the value with an integer. This integer matches the stack frame number of the corresponding call and is preceded with a number sign and followed by an arrow (#n =>) that points to the return value.

5.7.4 Tracer Options

You can modify the output of the tracer by specifying options in the call to the TRACE macro. Each option consists of a keyword-value pair. The format in which to specify keyword-value pairs for the TRACE macro is:

(TRACE (*function-name keyword-1 value-1*
        keyword-2 value-2
        ...))

You can also specify options for a list of functions and/or macros. The TRACE macro format in which to specify the same options for a list of functions and macros is:

(TRACE ((*name-1 name-2* ...) *keyword-1 value-1*
        keyword-2 value-2
        ...))

NOTE

Forms the system evaluates just before or just after a call to a function or macro for which tracing is enabled are evaluated in a null lexical environment. For information on lexical environments, see *COMMON LISP: The Language*.

The keywords you can use to specify options are:

- ```
  :DEBUG-IF      ---
  :PRE-DEBUG-IF    |-- Invoke the debugger
  :POST-DEBUG-IF --
  ```
- ```
  :PRINT      ---
  :PRE-PRINT    |-- Add information to tracer output
  :POST-PRINT --
  ```
- :STEP-IF -- Invokes the stepper
- :SUPPRESS-IF -- Removes information from tracer output
- :DURING -- Determines when a function or macro is traced

5.7.4.1 **Invoking the Debugger** - You can cause the tracer to invoke the debugger by specifying the :DEBUG-IF, :PRE-DEBUG-IF, or :POST-DEBUG-IF keyword. These keywords must be specified with a form. The LISP system evaluates the form before, after, or before and after each call to the function or macro being traced. If the form returns a value other than NIL, the tracer invokes the debugger after each evaluation.

5.7.4.2 **Adding Information to Tracer Output** - You can add information to tracer output by specifying the :PRINT, :PRE-PRINT, or :POST-PRINT keyword. You must specify these keywords with a list of forms. The LISP system evaluates each form in the list and the tracer displays their return values before, after, or before and after each call to the function or macro being traced. The tracer displays the values one per line and indents them to match other tracer output. If the forms to be evaluated cause an error, the debugger is invoked.

5.7.4.3 **Invoking the Stepper** - You can cause the tracer to invoke the stepper by specifying the :STEP-IF keyword. You must specify this keyword with a form. The LISP system evaluates the form before each call to the function or macro being traced. If the form returns a value other than NIL, the tracer invokes the stepper.

5.7.4.4 **Removing Information from Tracer Output** - You can remove information from tracer output by specifying the :SUPPRESS-IF keyword. You must specify this keyword with a form. The LISP system evaluates the form before each call to the function or macro being traced. If the form returns a value other than NIL, the tracer does not display the arguments and the return value of the function or macro being traced.

5.7.4.5 **Defining When a Function or Macro Is Traced** - You can define when a function or macro, for which tracing is enabled, is to be traced by specifying the :DURING keyword. You must specify this keyword with a function or macro name or a list of function and/or macro names. The functions and macros for which the tracer is enabled are traced only when they are called (directly or indirectly) from within one of the functions or macros whose names are specified with the keyword.

5.7.5 Tracer Variables

You can use two special variables with the TRACE macro. These are helpful debugging tools: *TRACE-CALL* and *TRACE-VALUES*. With these variables and the preceding tracer options, you can control when to debug or step depending on the arguments to a function or the return values from a function.

5.7.5.1 ***TRACE-CALL*** - The *TRACE-CALL* variable is bound to the function or macro call being traced. The following example shows how to use the variable:

```
Lisp> (DEFUN FIBONACCI (X)
        (IF (< X 3) 1
            (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))))
FIBONACCI

Lisp> (TRACE (FIBONACCI
               :PRE-DEBUG-IF (< (SECOND *TRACE-CALL*) 2)
               :SUPPRESS-IF T))
(FIBONACCI)
Lisp> (FIBONACCI 5)
Control Stack Debugger
Apply #30: (DEBUG)
Debug 1> DOWN
Eval  #27: (FIBONACCI (- X 2))
Debug 1> DOWN
Eval  #26: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
```

```
Debug 1> DOWN
Eval #25: (IF (< X 3) 1
              (+ (FIBONACCI (- X 1))
                 (FIBONACCI (- X 2))))
Debug 1> DOWN
Eval #24: (BLOCK FIBONACCI
            (IF (< X 3) 1
                (+ (FIBONACCI (- X 1))
                   (FIBONACCI (- X 2)))))
Debug 1> DOWN
Apply #22: (FIBONACCI 3)
Debug 1> (CADR (DEBUG-CALL))
3
Debug 1> CONTINUE
Control Stack Debugger
Apply #22: (DEBUG)
Debug 1> CONTINUE
5
```

- In this example, FIBONACCI is first defined.
- Then the TRACE macro is called for FIBONACCI. TRACE is specified to invoke the debugger if the first argument to FIBONACCI (the function call being traced) is less than 2. Since the PRE-DEBUG-IF option is specified, the debugger is invoked before the call to FIBONACCI. As the :SUPPRESS-IF option has a value of T, calls to FIBONACCI do not cause any trace output.
- The DOWN command moves the pointer down the control stack.
- The DEBUG-CALL function returns a list representing the current debug frame function call. In this case, the CADR of the list is 3. This accesses the first argument to the function in the current stack frame.
- Finally the CONTINUE command continues the evaluation of FIBONACCI.

5.7.5.2 ***TRACE-VALUES*** - The *TRACE-VALUES* variable is bound to the list of values returned by a traced function. Consequently, the variable can be used only with the :POST- options to the TRACE macro. Before being bound to the return values, the variable returns NIL. The following example shows how to use the variable:

```
Lisp> (TRACE (FIBONACCI
               :POST-DEBUG-IF (> (FIRST *TRACE-VALUES*) 2)))
(FIBONACCI)
Lisp> (FIBONACCI 5)
#5: (FIBONACCI 5)
```

```
. #13: (FIBONACCI 4)
. . #21: (FIBONACCI 3)
. . . #29: (FIBONACCI 2)
. . . #29=> 1
. . . #29: (FIBONACCI 1)
. . . #29=> 1
. . #21=> 2
. . #21: (FIBONACCI 2)
. . #21=> 1
Control Stack Debugger
Apply #14: (DEBUG)
Debug 1> BACKTRACE
-- Backtrace start --
Apply #14: (DEBUG)
Eval  #11: (FIBONACCI (- X 1)
Eval  #10: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
Eval  #9: (IF (< X 3) 1
              (+ (FIBONACCI (- X 1))
                 (FIBONACCI (- X 2))))
Eval  #8: (BLOCK FIBONACCI
             (IF (< X 3) 1
                 (+ (FIBONACCI (- X 1))
                    (FIBONACCI (- X 2)))))
Apply #6: (FIBONACCI 5)
Eval  #3: (FIBONACCI 5)
Apply #1: (EVAL (FIBONACCI 5))
-- Backtrace end --
Apply #14: (DEBUG)
Debug 1> CONTINUE
. #13=> 3
. #13: (FIBONACCI 3)
. . #21: (FIBONACCI 2)
. . #21=> 1
. . #21: (FIBONACCI 1)
. . #21=> 1
. #13=> 2
Control Stack Debugger
Apply #6: (DEBUG)
Debug 1> CONTINUE
#5=> 5
5
```

TRACE is called for FIBONACCI (the same function as in the previous example) to start the debugger if the value returned exceeds 2. The value returned exceeds 2 twice -- once when it returns 3 and at the end when it returns 5.

5.8 THE EDITOR

The VAX LISP Editor is a powerful, extensible editor that enables you to create and edit LISP programs. Once you have located an error and you know which function in your program is causing the error, you can use the Editor to correct the error. Use the ED function to invoke the Editor. For a complete description of the ED function, the VAX LISP Editor, and instructions on how to use the Editor, see the *VAX LISP Editor Manual*.

Show the pretty-printed output of two 2-dimensional arrays after the formatting function TWO-BY-TWO-ARRAYS is added to the dispatch-routine algorithm.

```
Lisp> (POP PPRINT-ARRAY-FORMATTERS)
TWO-BY-TWO-ARRAYS
```

Pops the formatting function TWO-BY-TWO-ARRAYS off the list that is bound to the *PPRINT-ARRAY-FORMATTERS* variable.

11.

```
Lisp> (DEFUN PPRINT-FURTHER-INDENTED (OBJECT LEFT-MARGIN
                                       &OPTIONAL
                                       (STREAM
                                        *STANDARD-OUTPUT*))
        (UNLESS (AND (INTEGERP LEFT-MARGIN)
                     (>= LEFT-MARGIN Ø))
                (ERROR
                  "The left-margin ~S must be a positive integer."
                       LEFT-MARGIN))
        (LET ((*PPRINT-LEFT-MARGIN* LEFT-MARGIN))
          (PPRINT OBJECT)))
PPRINT-FURTHER-INDENTED
```

Defines a formatting function that causes the pretty printer to print an object with the left margin specified.

CHAPTER 6

CALLING EXTERNAL ROUTINES

VAX LISP provides a facility that enables you to call external routines from within a VAX LISP program. Using this facility, VAX LISP programs can call the following:

- Routines written in languages that adhere to the VAX Calling Standard
- Run-time library (RTL) routines
- VMS and RMS system services

To call an external routine, the routine must follow the VAX Procedure Calling Standard. If you call a routine written in another language, the routine also must be linked into a position-independent shareable image.

The call-out facility cannot call external routines that require an extensive, non-VMS-oriented software environment. Routines written in APL and interpreted BASIC are examples of such routines. You can use VMS subprocess and mailbox facilities to communicate with such routines. VAX LISP provides functions for subprocess operations (see Part II).

Programs written in other VAX languages cannot call VAX LISP routines.

The _VAX Architecture Handbook_ and the _VAX/VMS Run-Time Library Reference Manual_ contain detailed information about calling external routines and passing parameters. You should be familiar with these subjects before you use the VAX LISP call-out facility.

A routine that can be called is termed a "procedure" in the manuals mentioned above. This chapter, however, uses the expression "external routine" to maintain consistency with the VAX LISP language terminology.

Before a LISP program can call external routines, you must create, compile, and debug the routines, and then you must perform the following steps:

1. Link the external routines into a VMS shareable image.
2. Define the external routines in LISP.
3. Call the external routines.

Figure 6-1 illustrates the steps you must perform to call out to an external routine.

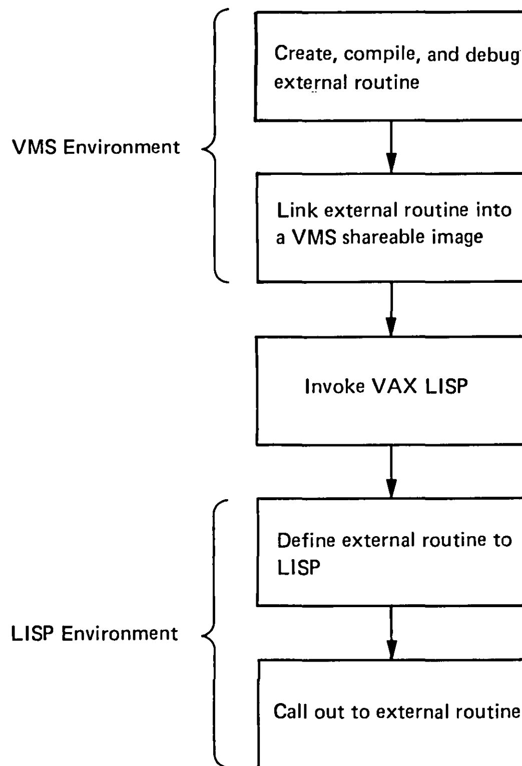


Figure 6-1 Calling External Routines

This chapter describes the standard VAX calling conventions, explains the three steps in the preceding list, provides a list of examples that show you how to use the call-out facility, and supplies information about the following topics:

- Data type conversions
- Calling VMS and RMS system services
- Errors during external-routine execution
- Suspending a LISP system that contains calls to external routines

6.1 STANDARD VAX CALLING CONVENTIONS

The VAX Procedure Calling Standard defines a uniform method for language routines to call one another -- see the VAX Architecture Handbook and the VAX/VMS Run-Time Library Reference Manual. This standard prescribes how external routines receive and return control, how parameters are passed, and how function values are returned. By means of the standard call conventions, most languages used with the VAX/VMS operating system can call library routines and routines written in other VAX native-mode languages. Interpreted and compiled VAX LISP programs cannot conform to the standard because of the nature of the LISP language. For this reason, VAX LISP provides a facility that enables you to call routines written in other VAX languages, which do conform to the standard. The next four sections provide a brief summary of the VAX Procedure Calling Standard.

6.1.1 Transfer of Control

VAX LISP calls external routines with a CALLG instruction. These routines return control to programs that call them with a RET VAX instruction.

6.1.2 Parameter Lists

Arguments are passed to an external routine in a parameter list. The LISP system constructs a parameter list each time a LISP program calls an external routine. The VAX Procedure Calling Standard defines a parameter list as a sequence of longword (4-byte) entries. The first byte of the first entry in the list is a parameter count, which indicates the number of parameters that follow in the list.

The succeeding longwords contain a data value, a pointer to a data value, or a pointer to a descriptor of the data value, depending on the specified passing mechanism.

6.1.3 Parameter-Passing Mechanisms

The VAX Procedure Calling Standard defines three mechanisms by which parameters are passed to external routines:

- By immediate value -- the parameter is the value
- By reference -- the parameter is the address of the value
- By descriptor -- the parameter is the address of a descriptor of the value

By default, the VAX LISP system uses the reference and the descriptor mechanisms to pass parameters. The system uses the descriptor mechanism to pass parameters of the string type; it uses the reference mechanism to pass all other LISP object types. Section 6.3.2.2 provides information on how to specify a parameter's passing mechanism.

6.1.4 Function Return Values

An external routine can be a subroutine or a function. A subroutine is invoked only to produce side effects on the parameters passed to it, and it returns no value as a result of its execution. A function, on the other hand, returns a value after execution and might perform side effects. The function value is returned in one of three ways.

- If the data type is scalar and requires 32 bits or less of storage, the value is returned in register R0.
- If the data type is scalar and requires from 33 to 64 bits of storage, the low-order bits of the value are returned in register R0, and the high-order bits of the value are returned in register R1.
- If the data type requires more than 64 bits of storage or if it is not a scalar type, the call-out facility allocates the required storage and passes the address of that storage as a new parameter added to the beginning of the argument list.

In VAX LISP, you can specify up to 254 parameters for subroutines or functions. Functions can return values that are bits, integers, or floating-point numbers. The return methods the LISP system uses for values of each data type are summarized in Table 6-1.

Table 6-1
Function Return Methods

Data Type	Return Method
:BIT :BYTE :UNSIGNED-BYTE :WORD :UNSIGNED-WORD :LONGWORD :UNSIGNED-LONGWORD :F-FLOATING	General register R0
:D-FLOATING :G-FLOATING	R0: Low-order result R1: High-order result
:H-FLOATING	New parameter

6.2 LINKING A SHAREABLE IMAGE

Before a LISP program can call external routines, you must link the required object modules into one or more position-independent VMS shareable images. The following example links the object modules TEST.OBJ and FUN.OBJ into a shareable image called MYIMAGE.EXE. The UNIVERSAL linker option is used to list the entry points that are available to the call-out facility.

```
$ LINK/SHAREABLE=MYIMAGE TEST,FUN,SYS$INPUT:/OPTIONS
UNIVERSAL=ENTRY_1,ENTRY_2,ENTRY_3
```

The number of individual shareable images that can be mapped into VAX LISP depends on VMS shareable image restrictions and the available address space.

If you specify a base address as an option in a command that invokes the VMS linker or if the linker issues a warning message that informs you that the shareable image is based, you cannot call external routines in that image.

You can call external routines in shareable images that contain writeable sections. Routines that are written in VAX-11 FORTRAN, which use COMMON blocks, are examples of routines that produce such code. A shareable image contains a writeable section if the external routine contains a program section (PSECT) that has the write (WRT) and the share (SHR) attributes. To determine whether a program section in a shareable image has these attributes, examine the image's map file.

Before you can call an external routine in a shareable image that contains writeable sections, install the shareable image with the VMS INSTALL utility.

The procedure for linking shareable images is explained in the _VAX/VMS Linker Reference Manual_.

6.3 DEFINING AN EXTERNAL ROUTINE

Programs written in VAX LISP cannot call external routines the same way as programs written in other VMS languages that adhere to the VAX Procedure Calling Standard. When a program calls an external routine the program must identify information about the routine. Other VMS languages identify the information by compiling code into object modules that are linked by the VMS linker. Since VAX LISP does not create object modules that can be linked, it must identify information about an external routine another way.

After you link an external routine into a shareable image, enter the VAX LISP environment and define the routine using the VAX LISP DEFINE-EXTERNAL-ROUTINE macro. It provides the VAX LISP system with the information it needs to create a parameter list and to locate and call the external routine. A description of the DEFINE-EXTERNAL-ROUTINE macro is provided in Part II.

The formal definition of an external routine consists of two components:

- External-routine name and options
- Formal-parameter descriptions

Sections 6.3.1 and 6.3.2 describe these components.

6.3.1 External-Routine Name and Options

When you define an external routine, you must specify a name for it. In addition, you can specify options that provide the LISP system with information about how to handle the external routine.

6.3.1.1 **External-Routine Name** - An external-routine name is a symbol that the call-out facility uses to access information about a routine.

You can specify an external-routine name with options. If you specify options, specify the name and options as a list whose first element is the name; if you do not specify options, specify the name as a symbol.

The symbol that names the external routine can be different than the actual name of the external routine. If the symbol is different, specify the external-routine name with the entry-point option (see Section 6.3.1.2).

6.3.1.2 **External-Routine Options** - You can assign specific characteristics to an external routine by specifying options in the routine's definition. Each option consists of a keyword-value pair. Specify a keyword-value pair as follows:

```
keyword value
```

Specify external-routine options in a list whose first element is the name of the routine the options characterize. The format in which to specify the name and options follows:

```
(name keyword-1 value-1 keyword-2 value-2 ...)
```

A list of the keywords that you can use to specify options for external routines and the characteristics the options define follows:

- :CHECK-STATUS-RETURN -- Status return checking
- :ENTRY-POINT -- Entry point
- :IMAGE-NAME -- Image name
- :RESULT -- Result data type
- :TYPE-CHECK -- Type checking

The option values are not evaluated.

NOTE

You must specify the image name option unless you are calling a system service.

Status Return Checking: The VAX Procedure Calling Standard specifies a method for returning the completion status of a function (see Section 6.1.4). The :CHECK-STATUS-RETURN keyword specifies whether the call-out facility is to examine the contents of register RØ upon return from the external routine. You can specify the keyword with either T or NIL. If you specify T, the content of the register is examined, and the routine's return value is interpreted as a VMS status code or a user status code. If the severity of the return value is warning, error, or severe-error, a LISP continuable error is signaled (see Section 6.8). If you specify NIL (the default value), all status codes are ignored.

Entry Point: The entry point of an external routine is specified with the :ENTRY-POINT keyword. You must specify this keyword with the string that represents the name of the entry point that is to be called if the name is different than the name you specify for the external routine. The default string is the print name of the external-routine name you specify in the call to the DEFINE-EXTERNAL-ROUTINE macro.

Image Name: An external routine's image name is specified with the :IMAGE-NAME keyword. You must include this keyword in a routine's definition unless you are calling a system service. Specify the :IMAGE-NAME keyword with the string that represents the VMS file name of the external routine's shareable image. The file specification is merged with SYS$SHARE:.EXE.

Result Data Types: The :RESULT keyword specifies the LISP data type the external routine is to return to the LISP system. You can specify this keyword with the LISP data type or with a list that contains both the LISP and the VAX data types. Valid VAX LISP data type values are INTEGER, FIXNUM, BIGNUM, SHORT-FLOAT, FLOAT, DOUBLE-FLOAT, LONG-FLOAT, BIT, BIT-VECTOR, NIL and NULL. The VAX data types you can specify are: :BIT, :BYTE, :UNSIGNED-BYTE, :WORD, :UNSIGNED-WORD, :LONGWORD, :UNSIGNED-LONGWORD, :F-FLOATING, :D-FLOATING, :G-FLOATING, and :H-FLOATING. The default value is the NIL LISP data type.

If the specified external routine is a subroutine invoked for side effects, using the LISP type NIL forces the call-out facility to return no value when the external routine returns control to the LISP program. If you specify the NULL LISP type, NIL is returned when the external routine returns control to the program.

Type Checking: The actual parameter data types that are passed to an external routine and the defined formal parameter data types can be checked for compatibility. The :TYPE-CHECK keyword controls whether the LISP system generates type-checking code. You can specify the keyword with either T or NIL. If you specify T, the LISP system generates code that checks the type of actual LISP objects when you call the CALL-OUT macro. If the types of the routine's formal and the actual parameters are incompatible, an error is signaled. If you specify NIL (the default value), the system does not generate type-checking code.

6.3.2 Formal-Parameter Descriptions

External-routine definitions can include a formal-parameter list. A formal-parameter list consists of formal-parameter descriptions; each description includes the parameter name and the options that define the parameter's characteristics. The order of the formal parameters in the formal-parameter list specifies the order of the actual parameters the external routine requires.

6.3.2.1 **Formal-Parameter Name** - A formal-parameter name is a symbol that names a formal parameter. The symbol must be either unique within the routine's definition or NIL.

You can specify a formal-parameter name with options. If you specify options, specify the name and options as a list whose first element is the name; if you do not specify options, specify the name as a symbol.

6.3.2.2 **Formal-Parameter Options** - You can define characteristics for a formal parameter by specifying options in the parameter's description. Each option consists of a keyword-value pair. A keyword-value pair must be specified as follows:

```
keyword value
```

You must specify options in a list whose first element is the name of the parameter they characterize. The format in which to specify the name and options follows:

```
(name keyword-1 value-1 keyword-2 value-2 ...)
```

A list of the keywords that you can use to specify options in formal-parameter descriptions and the characteristics the options define follows:

- :ACCESS -- Access method
- :LISP-TYPE -- LISP data type
- :MECHANISM -- Passing mechanism
- :VAX-TYPE -- VAX data type

The option values are not evaluated.

Access Method: The access method of a parameter is defined with the :ACCESS keyword. Valid option values are :IN and :IN-OUT. The values inform the LISP system of the type of access an external routine requires for a parameter. The default value is :IN.

If you specify the :IN value in a formal parameter description, the actual parameter passed to the external routine has input access. A parameter that has input access is assumed to be read-only. An external routine cannot modify its value. The actual parameter is evaluated at run time to produce an actual parameter value.

The actual parameter has input-output access if you specify the :IN-OUT value in a formal parameter description. A parameter that has input-output access has to be a valid SETF form. The form you specify is evaluated to produce a value. The value is passed to the external routine, where it is modified. The effect of modifying the value in the external routine is similar to applying the SETF macro to the argument form and specifying the new value.

NOTE

If the call-out facility passes an actual parameter value, which is not an integer, to an external routine with input-output access, the external routine modifies the actual LISP object. The macro does not create a new object. For example, if the value of LISP variables A and B are the same floating-point number, passing A as an input-output parameter might modify the value of B also.

LISP Data Type: The :LISP-TYPE keyword specifies the LISP type of an actual parameter. You can specify this keyword with the types: numbers, simple strings, simple bit vectors, simple floating-point arrays, or alien structures. The INTEGER type is the default. See COMMON LISP: The Language for a list of valid LISP types.

If the values you specify for the LISP data type and the VAX data type are not compatible, an error is signaled.

Passing Mechanism: The mechanism by which an actual parameter is to be passed to an external routine is specified with the :MECHANISM keyword. The values you can specify with the :MECHANISM keyword are :IMMED, :REF, and :DESCR. These values correspond to the three defined mechanisms that are described in Section 6.1.3: immediate value, reference, and descriptor.

- The immediate value mechanism passes a copy of the actual parameter in the argument list. You can use this mechanism only for parameters that have input access and that are numeric data types that require no more than 32 bits of storage space.
- The reference mechanism passes the address of the actual parameter to the external routine. This mechanism is the default for parameters that are of any LISP data type except string.
- The descriptor mechanism passes the address of a descriptor for the actual parameter to the external routine. The descriptor is a data structure that contains the address of the parameter, as well as its data type and size. Parameters that are of the LISP string type default to this mechanism.

You cannot specify specific VMS descriptor classes in external routine definitions. The DEFINE-EXTERNAL-ROUTINE macro assigns an appropriate descriptor class to a routine when the LISP system evaluates it. The values the macro assigns are DSC$K_CLASS_S or DSC$K_CLASS_A. To pass an actual parameter using a user-specified descriptor, define the descriptor and the parameter to be alien structures and pass the alien-structure descriptor with the reference mechanism. For information on defining alien structures, see Chapter 7.

VAX Data Type: Specify the VAX data type of an actual parameter, which is to be passed to an external routine, with the :VAX-TYPE keyword. Data types you can specify as values are :BIT, :BYTE, :UNSIGNED-BYTE, :WORD, :UNSIGNED-WORD, :LONGWORD, :UNSIGNED-LONGWORD, :F-FLOATING, :D-FLOATING, :G-FLOATING, :H-FLOATING, and :TEXT. The default VAX type is the type that most nearly corresponds to a LISP object type. For example, :LONGWORD corresponds to FIXNUM and BIGNUM, :F-FLOATING corresponds to SINGLE-FLOAT, and :TEXT corresponds to SIMPLE-STRING.

NOTE

> If the values you specify for the LISP data type and the VAX data type are not compatible, an error is signaled. Therefore, if the default VAX data type does not match the required type, you must specify the :VAX-TYPE keyword.

6.4 CALLING AN EXTERNAL ROUTINE

After you define an external routine, you can call it by specifying the routine's name in a call to the VAX LISP CALL-OUT macro. The CALL-OUT macro produces code that performs the following operations:

1. Activates the external routine's shareable image if the image is not already activated
2. Creates an actual parameter list using the actual parameter arguments; it creates the list according to the VAX Procedure Calling Standard
3. Transfers control to the external routine

NOTE

You cannot use the VAX-11 Symbolic Debugger when you are in the LISP environment and you cannot use the VAX LISP debugger inside your external routine. Therefore, you must debug external routines before you call them from LISP.

When you call the CALL-OUT macro, the value you specify for the first argument must be the name of a predefined external routine. In addition to specifying the name, you can specify actual parameters.

6.4.1 Predefined Name

The name that you specify in a call to the CALL-OUT macro must be a symbol that names an external routine, which was previously defined with the DEFINE-EXTERNAL-ROUTINE macro.

6.4.2 Actual Parameters

You can specify actual parameters in a call to the CALL-OUT macro. They correspond, by position, to the formal parameters that are defined for the external routine being called. The parameters are evaluated before control is transferred to the external routine.

You can omit a parameter by putting an explicit NIL in the position of the corresponding formal parameter. You cannot use expressions that evaluate to NIL. For example:

```
Lisp> (CALL-OUT SYS$RENAME OLD NIL NIL NEW)
1
```

The positions in the actual parameter list that correspond to NIL contain zeros to coincide with the VAX Procedure Calling Standard for omitted arguments. If you supply fewer actual parameters than are specified in the external routine's formal definition, the argument count in the parameter list contains only the number of actual parameters. If you supply more arguments than are specified in the routine's formal definition, the LISP system signals an error.

6.4.3 Internal Data Structures

When you interpret or compile an external routine definition, the LISP system creates internal data structures. When the VAX LISP compiler or interpreter expands the CALL-OUT macro, the macro uses these internal data structures to define its correct expansion. In addition, the code that results from the CALL-OUT macro expansion uses some of the internal data structures.

If you use the CALL-OUT macro to call an external routine whose definition is in the same source file, the internal data structures are present when the LISP system interprets or compiles the file. However, if you use the CALL-OUT macro to call an external routine that is defined in a separate file, the data structures are not present. Because the CALL-OUT macro expansion code uses these internal data structures, you must load the file that contains the external routine's definition (compiled or source) into the LISP system before you perform the following operations:

- Compile code that contains a use of the CALL-OUT macro
- Execute code that contains a call to an external routine
- Execute compiled code that contains a call to an external routine

One way that you can load a file, which contains an external routine's definition, into the compiler is to load the file as an initialization file using the /INITIALIZE DCL command qualifier (see Section 2.8.4). For more information on loading files, see Section 2.6.

6.5 EXAMPLES OF USING THE CALL-OUT FACILITY

This section provides examples of using the VAX LISP call-out facility.

1.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
          (MTH$DACOSD :IMAGE-NAME "VMSRTL"
                      :RESULT DOUBLE-FLOAT)
          (X :LISP-TYPE DOUBLE-FLOAT
             :VAX-TYPE :D-FLOATING))
MTH$DACOSD
```

Defines an RTL routine, called MTH$DACOSD, which returns the double precision arc cosine of a number in degrees. The routine takes one read-only argument, which is a D format floating-point number and returns the result as a D format floating-point number.

```
Lisp> (CALL-OUT MTH$DACOSD 0.33333d0)
2.417999466331081d16
```

Calls the RTL routine MTH$DACOSD, and returns the routine's value.

2. ```
   Lisp> (DEFINE-EXTERNAL-ROUTINE
                (ERASE-PAGE :IMAGE-NAME "SCRSHR"
                            :ENTRY-POINT "LIB$ERASE_PAGE"
                            :CHECK-STATUS-RETURN T)
                (LINE :LISP-TYPE INTEGER
                      :VAX-TYPE :WORD)
                (COL  :LISP-TYPE INTEGER
                      :VAX-TYPE :WORD))
   ERASE-PAGE
   ```

   Defines an RTL screen management routine, called ERASE-PAGE, which erases the terminal screen. Note that the image name for the screen package is SCRSHR and not VMSRTL. The routine does not return the RTL status, but the call-out facility checks the status internally. A VAX data type is specified for each argument because the default type -- :LONGWORD -- is not the type required by the RTL routine.

   ```
   Lisp> (CALL-OUT ERASE-PAGE 1 1)
   ```

   Calls the RTL routine LIB$ERASE_PAGE to erase the terminal screen starting at the upper left corner.

3. ```
   Lisp> (DEFINE-EXTERNAL-ROUTINE
                (PUT-SCREEN :IMAGE-NAME "SCRSHR"
                            :ENTRY-POINT "LIB$PUT_SCREEN"
                            :RESULT INTEGER)
                (CHARS :LISP-TYPE STRING)
                (LINE  :LISP-TYPE INTEGER
                       :VAX-TYPE :WORD)
                (COL   :LISP-TYPE INTEGER
                       :VAX-TYPE :WORD)
                (FLAG  :LISP-TYPE INTEGER
                       :VAX-TYPE :WORD))
   PUT-SCREEN
   ```

   Defines an RTL screen management routine, called PUT-SCREEN, which writes to the terminal screen.

   ```
   Lisp> (CALL-OUT PUT-SCREEN "This is a test.")
   lhis is a cest.
   ```

   Calls the RTL routine PUT-SCREEN, which writes the string "This is a test" to the current screen position and then returns the status code (1).

   ```
   Lisp> (CALL-OUT PUT-SCREEN "Another test." 5 10)
   1  Another test.
   ```

   Calls the RTL routine PUT-SCREEN, which writes the string "Another test" starting on line five and in column ten and then returns the status code (1).

   ```
   Lisp> (CALL-OUT PUT-SCREEN "A third test." NIL NIL 2)
   1 third test.
   ```

   Calls the RTL routine PUT-SCREEN, which writes the string "A third test" to the current screen position with the reverse video attribute and then returns the status code (1).

4.
```
Lisp> (DEFINE-EXTERNAL-ROUTINE
          (GET-SCREEN :IMAGE-NAME "SCRSHR"
                      :ENTRY-POINT "LIB$GET_SCREEN"
                      :RESULT INTEGER)
          (INPUT    :LISP-TYPE STRING
                    :ACCESS :IN-OUT)
          (PROMPT   :LISP-TYPE STRING)
          (OUT-LEN  :LISP-TYPE INTEGER
                    :VAX-TYPE :UNSIGNED-WORD
                    :ACCESS :IN-OUT))
GET-SCREEN
```

Defines an RTL screen management routine, called GET-SCREEN, which gets input from the terminal screen. The definition includes two input-output access parameters. You must use the SETF macro to assign values to these parameters before you call the RTL routine.

```
Lisp> (SETF TEXT-LENGTH 0)
0
Lisp> (SETF IN-TEXT (MAKE-STRING 80 :INITIAL-ELEMENT
                                              #\SPACE))
"
  "
Lisp> (CALL-OUT GET-SCREEN IN-TEXT "Enter your data: "
                               TEXT-LENGTH)
Enter your data: This is input from the terminal.<RET>
1
```

Calls the RTL routine GET-SCREEN, which modifies the values of the variables IN-TEXT and TEXT-LENGTH for the text that is input. The routine modifies only the number of characters specified by the new value of the TEXT-LENGTH variable. If the number of characters in the specified text is greater than the value of the variable, the routine does not modify the remaining characters. The following call to the SUBSEQ function returns the actual input from the terminal.

```
Lisp> (SUBSEQ IN-TEXT 0 TEXT-LENGTH)
"This is input from the terminal."
```

5. This example shows you how to call out to an external routine that is written in another language. The following program is written in FORTRAN:

```
    FUNCTION NUMBERS(X,Y)
    IMPLICIT INTEGER*4(A-Z)

    NUMBERS=Y*(X + Y ** X)/X
    RETURN
    END
```

A routine written in FORTRAN, called NUMBERS, which manipulates two integers and returns an integer.

```
$ FORTRAN NUMBERS
```

Compiles the FORTRAN program NUMBERS.

```
$ LINK/SHAREABLE=DBA2:[SMITH]EXAMPLE NUMBERS,SYS$INPUT:/OPTION
UNIVERSAL=NUMBERS<CTRL/Z>
$
```

Links the FORTRAN routine NUMBERS into a shareable image. The name NUMBERS is specified as an entry point that is globally available.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
          (NUMBERS :IMAGE-NAME "DBA2:[SMITH]EXAMPLE"
                   :RESULT INTEGER)
             X Y)
NUMBERS
```

Defines an external routine, called NUMBERS, which manipulates two integers and returns an integer. The image name specification includes a directory specification because the routine does not reside in SYS$SHARE. The arguments do not have options because, by default, they are assumed to be longword integers that are passed by reference.

```
Lisp> (CALL-OUT NUMBERS 5 7)
23536
```

Calls the external routine NUMBERS, which returns the function value.

6. This example illustrates a more complex use of the call-out facility. Assume that an external routine named COMPLEX exists outside of the LISP system.

```
Lisp> (DEFINE-ALIEN-STRUCTURE COMPLEX-NUMBER
          (REAL    :G-FLOATING 0 8)
          (COMPLEX :G-FLOATING 8 16))
COMPLEX-NUMBER
```

Defines a complex number (double-precision). The alien-structure facility is used to define complex numbers because the data type cannot be represented directly in VAX LISP. See Chapter 7 for a description of the alien-structure facility.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
          (CMPLX_EXP :IMAGE-NAME "DBA2:[SMITH]COMPLEX"
          (OUTPUT :LISP-TYPE ALIEN-STRUCTURE
                  :ACCESS :IN-OUT)
          (INPUT  :LISP-TYPE ALIEN-STRUCTURE))
CMPLX_EXP
```

Defines the external routine, called CMPLX_EXP, which uses complex numbers.

```
Lisp> (SETQ C1 (MAKE-COMPLEX-NUMBER :REAL 5.0d1
                                    :COMPLEX 6.123456d-4))
#<Alien Structure COMPLEX-NUMBER #x5010324>
Lisp> (SETQ C2 (MAKE-COMPLEX-NUMBER :REAL 0.0d0
                                    :COMPLEX 0.0d0))
#<Alien Structure COMPLEX-NUMBER #x5010348>
```

These expressions create two complex numbers, C1 and C2.

```
Lisp> (CALL-OUT CMPLX_EXP C2 C1)
```

Calls the external routine CMPLX_EXP. The routine changes the values in the alien structure C2. A value is not returned from the function call because the default result type (NIL) was used.

6.6 DATA TYPE CONVERSIONS

The internal representation of some LISP objects differs from the standard VAX format for the corresponding data types. If the data types do not have the same internal format, the call-out facility converts the LISP type to a VAX data type before it passes the parameter to the external routine. Likewise, after the LISP system evaluates the external routine, the call-out facility might have to convert the resulting VAX data to a LISP object before it can return the data to the LISP system.

6.6.1 Converting LISP Objects to VAX Data Types

The :LISP-TYPE and :VAX-TYPE keyword specifications in an external routine definition determine whether an actual parameter must be converted. If conversion is necessary, the conversion operation is independent of the mechanism that passes the parameters.

The call-out facility does not convert VAX LISP floating-point numbers, strings, and floating-point arrays before it passes them to an external routine. Therefore, the actual parameter type must correspond exactly to the specified VAX formal type. For example, if the VAX type is :H-FLOATING, the LISP object must be of type LONG-FLOAT.

The call-out facility does not convert array indices before it passes them to an external routine; the right-most index must vary faster than the left-most index. In FORTRAN, the left-most array index varies faster. Therefore, if you call a FORTRAN external routine, you must ensure correct index order.

The only string type the call-out facility can pass to an external routine is the LISP simple string. The facility does not support strings that contain such things as fill pointers and displacements.

NOTE

Occurrences of the #\NEWLINE character might not be interpreted correctly by an external routine.

LISP integer, bit, and bit vector types are converted to VAX integers. Table 6-2 lists the conversion operations.

Table 6-2
LISP Object to VAX Integer Conversions

LISP Object Type	Conversion Operation
FIXNUM BIGNUM	The call-out facility converts fixnums and bignums to the appropriate VAX integer type.
BIT	The call-out facility converts a bit to a VAX byte, word, or longword by placing the bit in the low order bit of the VAX data object and padding the object on the left with zeros.

(Continued on next page)

Table 6-2 (Cont.)
LISP Object to VAX Integer Conversions

LISP Object Type	Conversion Operation
BIT-VECTOR	The call-out facility converts a bit vector of length 32 or less to a VAX byte, word, or longword by placing the bits right-justified in the VAX object and padding the object on the left with zeros.

6.6.2 Converting VAX Data Types to LISP Objects

The call-out facility might have to convert the VAX data resulting from the execution of an external routine to a LISP object before it can return the data to the LISP system. VAX floating-point numbers, strings, and arrays require no conversion; the facility returns them as the corresponding LISP object types.

VAX integer types do require conversion. The call-out facility converts them to fixnum, bignum, bit, or bit vector LISP object types. Table 6-3 lists the valid conversion operations.

Table 6-3
VAX Integer to LISP Object Conversions

LISP Object Type	Conversion Operation
FIXNUM BIGNUM	The call-out facility converts a VAX byte, word, or longword to a fixnum if the value fits in a fixnum field; otherwise, it converts a VAX byte to a bignum.
BIT	The call-out facility converts the least significant bit of a VAX integer to a LISP bit object.
BIT-VECTOR	The call-out facility converts a VAX integer to a LISP bit vector of length n (where n<=32). The facility uses the low order n bits of the VAX integer for the conversion. If the bit vector is larger than the VAX integer, the facility fills the high-order bits of the LISP bit vector with zeros.

6.7 CALLING SYSTEM SERVICES

The call-out facility provides a mechanism for LISP programs to call standard VMS and RMS system services. Sections 6.7.1 and 6.7.2 provide the information you need to define and call system services. Section 6.7.3 lists the system services that are supported by the VAX LISP call-out facility. Section 6.7.4 provides examples of calling system services.

6.7.1 Defining System Services

Defining VMS and RMS system services is similar to defining other external routines with a few restrictions. You must be familiar with the explanation of defining an external routine, which is provided in Section 6.3, to understand the following list of restrictions:

- You must omit the image name parameter from the DEFINE-EXTERNAL-ROUTINE macro specification. Omission of this parameter causes the macro to assume that the function being defined is a system service. If you use the name of a system service but supply an image name (not NIL), the LISP system assumes that you want an entry point in an ordinary shareable image of that name rather than the VMS system service given as the external routine name.

- The external-routine name or the entry-point name in the DEFINE-EXTERNAL-ROUTINE macro specification must be one of the system service names listed in Table 6-4 or Table 6-5.

- The data types of the arguments in the argument list in the DEFINE-EXTERNAL-ROUTINE macro specification must correspond to the system service's data types. The definitions for VMS system service arguments are provided in the _VAX/VMS System Services Reference Manual_, and the definitions for RMS system service areguments are provided in the _VAX/VMS RMS Reference Manual_.

- The order and the correct number of system service arguments in the DEFINE-EXTERNAL-ROUTINE macro specification must correspond exactly to the order and number specified by the service's definition (see the _VAX/VMS System Services Reference Manual_ or the _VAX/VMS RMS Reference Manual_).

6.7.2 Calling Out to System Services

Calling VMS and RMS system services is similar to calling other external routines with a few restrictions. You must be familiar with the explanation of calling out to an external routine, which is provided in Section 6.4, to understand the following list of restrictions:

- You must always call system services with a complete argument list, even if you omit the last several arguments. Put NIL in place of the omitted arguments -- including omitted trailing arguments.

- You must omit parameters that correspond to an ASTADR parameter in the system service, although you might have to define the field to account for the correct number of arguments. If you put a value other than NIL into the field, unpredictable or erroneous behavior results.

- If you call an asynchronous routine, such as SYS$QIO, you must place input-output access data in statically allocated alien structures (see Chapter 7). You must do this even if the next call on the CALL-OUT macro has the SYS$WAITFR system service or one of its variants as its argument. An example of such input-output access data is the I/O status block (IOSB) parameter of the SYS$QIO system service.

6.7.3 System Services

Tables 6-4 and 6-5 contain alphabetized lists of the VMS and RMS services supported by the call-out facility. For more information on VMS services, see the VAX/VMS System Services Reference Manual. For more information on RMS services, see the VAX/VMS RMS Reference Manual.

Table 6-4
VMS Services Supported by the Call-Out Facility

System Service	Function
SYS$ADJWSL	Adjusts the current limit of a process's working set size by a specified number of pages.
SYS$ALLOC	Allocates a device for exclusive use by a process and its subprocesses.
SYS$ASCTIM	Converts an absolute or a delta time from 64-bit system time format to an ASCII string.
SYS$ASSIGN	Assigns a process an I/O channel so that input/output operations can be performed on a device, or establishes a logical link with a remote node on a network.
SYS$BINTIM	Converts an ASCII string to an absolute or a delta time value in the 64-bit system time format.
SYS$BRDCST	Broadcasts a message to one or more terminals.
SYS$CANCEL	Cancels all pending I/O requests on a specific channel.
SYS$CANWAK	Cancels all scheduled wake-up requests for a process from the timer queue, including those made by the caller or by other processes.
SYS$CREMBX	Creates a virtual mailbox device named MBAn: and assigns an I/O channel to it.
SYS$CREPRC	Enables a process to create another process.
SYS$DALLOC	Deallocates a previously allocated device.
SYS$DASSGN	Deassigns an I/O channel acquired for input/output operations with the SYS$ASSIGN system service.
SYS$DELLOG	Deletes a logical name and its equivalence name from the process, group, or system logical name table.
SYS$DELMBX	Marks a permanent mailbox for deletion.
SYS$DELPRC	Enables a process to delete itself or another process.

(Continued on next page)

Table 6-4 (Cont.)
VMS Services Supported by the Call-Out Facility

System Service	Function
SYS$DEQ	Unlocks resources that the calling process previously locked using the SYS$ENQ system service.
SYS$ENQ	Enables you to queue requests to access a resource or to convert the current lock request mode to another lock request mode.
SYS$ENQW	Combines the SYS$ENQ and SYS$WAITFR system services.
SYS$FAO	Converts binary values into ASCII characters and returns the converted characters in an output string.
SYS$FAOL	Converts binary values into ASCII characters and returns the converted characters in an output string.
SYS$FORCEX	Causes the SYS$EXIT system service call to be issued for a specified process.
SYS$GETCHN	Returns information about a device to which an I/O channel has been assigned.
SYS$GETDVI	Returns information about an I/O device.
SYS$GETJPI	Returns accounting, status, and identification information about a specified process.
SYS$GETSYI	Returns status and identification information about the system.
SYS$GETTIM	Returns the current system time in 64-bit format.
SYS$HIBER	Enables a process to make itself inactive but to remain known to the system so it can be interrupted, for example, to receive ASTs.
SYS$NUMTIM	Converts an absolute or a delta time from a 64-bit system time format to binary integer date and time values.
SYS$QIO	Initiates an input or output operation by queuing a request to a channel associated with a specific device.
SYS$QIOW	Combines the SYS$QIO and SYS$WAITFR system services.
SYS$RESUME	Causes a process previously suspended by the SYS$SUSPEND system service to resume execution, or cancels the effect of a subsequent suspend request.

(Continued on next page)

Table 6-4 (Cont.)
VMS Services Supported by the Call-Out Facility

System Service	Function
SYS$SCHDWK	Schedules the wakeup of a process that has placed itself in a state of hibernation with the SYS$HIBER system service.
SYS$SETIME	Changes or recalibrates the current system time.
SYS$SETIMR	Enables a process to schedule the setting of an event flag and/or the queuing of an AST at some future time.
SYS$SETPRI	Changes a process's base priority.
SYS$SETPRN	Enables a process to establish or to change its own process name.
SYS$SETPRV	Enables a process to enable or disable specified user privileges.
SYS$SETRWM	Enables a process to indicate what action a system service should take when it lacks a system resource required for its execution.
SYS$SNDACC	Controls accounting log activity and enables a process to write an arbitrary data message into the accounting log file.
SYS$SNDERR	Writes an arbitrary message to the system error log file.
SYS$SNDOPR	Enables a process to send a message to one or more terminals designated as operator's terminals and to optionally receive a reply.
SYS$SNDSMB	Used by the operating system to queue user's print files to a system printer or to queue command procedure files for detached job execution.
SYS$SUSPND	Enables a process to suspend itself or another process.
SYS$WAITFR	Tests a specific event flag and returns immediately if the flag is set.
SYS$WAKE	Activates a process that has placed itself in a state of hibernation with the SYS$HIBER system service.
SYS$WFLAND	Enables a process to specify a mask of event flags for which it wishes to wait.
SYS$WFLOR	Tests the event flags specified by a mask within a specified cluster and returns immediately if any of them is set.

Table 6-5
RMS Services Supported by the Call-Out Facility

System Service	Function
SYS$CLOSE	Terminates file processing and closes the file.
SYS$CONNECT	Establishes a record stream by associating and connecting a record access block (RAB) with a file access block (FAB).
SYS$CREATE	Constructs a new file according to the attributes you specify in the file access block (FAB).
SYS$DELETE	Removes an existing record from a relative or an indexed file.
SYS$DISCONNECT	Terminates a record stream by breaking the connection between a record access block (RAB) and a file access block (FAB).
SYS$DISPLAY	Retrieves file attribute information about a file and places the information in fields in the file access block (FAB) and in the extended attributes blocks (XAB) chained to the FAB.
SYS$ENTER	Inserts a file name into a directory.
SYS$ERASE	Deletes a VAX RMS disk file and removes the file's directory entry as specified in the path to the file.
SYS$EXIT	Exits VAX LISP.
SYS$EXTEND	Increases the amount of space allocated to a VAX RMS disk file.
SYS$FIND	Locates a specified record in a file and returns its record's file address in the record's file address (RFA) field of the record access block (RAB).
SYS$FLUSH	Writes out all modified I/O buffers and file attributes associated with the file.
SYS$FREE	Unlocks all records that were previously locked for the record stream.
SYS$GET	Enables a record to be retrieved from a file.
SYS$NXTVOL	Enables you to proceed to the next volume in the set before the end of the current volume is reached on input, or before the end of the tape is reached on output.
SYS$OPEN	Makes an existing file available for processing by your program.
SYS$PARSE	Analyzes the file specification string and fills in various name block (NAM) fields.

(Continued on next page)

Table 6-5 (Cont.)
RMS Services Supported by the Call-Out Facility

System Service	Function
SYS$PUT	Inserts a record into a file.
SYS$READ	Retrieves a specified number of bytes from a file and transfers them to memory.
SYS$RELEASE	Unlocks the record pointed to by the contents of the record's file address (RFA) field of the record access block (RAB).
SYS$REMOVE	Deletes a file name from a directory.
SYS$RENAME	Renames a file.
SYS$REWIND	Sets the context of a stream to the first record in the file.
SYS$RMSRUNDWN	Closes all files opened by VAX RMS for the image or process and halts I/O activity.
SYS$SEARCH	Scans a directory file and fills in various name block (NAM) fields.
SYS$SETDDIR	Enables you to read and/or change the default directory string for the process.
SYS$SETDFPROT	Enables you to read and/or write the default file protection for the process.
SYS$SPACE	Enables you to position a file forward or backward a specified number of blocks.
SYS$TRUNCATE	Removes records from the end of a sequential file.
SYS$UPDATE	Enables you to modify the contents of an existing record in a file residing on a disk device.
SYS$WAIT	Suspends execution until an asynchronous record operation completes.
SYS$WRITE	Transfers a user-specified number of bytes to a VAX RMS file of any file organization.

6.7.4 Examples of Calling System Services

This section provides examples of how to call system services.

1.
```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SYS$DALLOC :RESULT INTEGER)
                (DEVNAM :LISP-TYPE STRING)
                (ACMODE :LISP-TYPE INTEGER
                        :MECHANISM :IMMED))
SYS$DALLOC
```

   Defines the VMS system service SYS$DALLOC.

```
Lisp> (CALL-OUT SYS$DALLOC "TTH7:" NIL)
2312
```

Calls the VMS system service SYS$DALLOC. NIL is specified to account for the omitted parameter; this ensures that the correct number of arguments are specified.

2. Suppose that the LISP variables OLD and NEW are bound to statically allocated alien structures (see Chapter 7), which are the file attribute blocks to be used in a rename operation.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SYS$RENAME :RESULT INTEGER)
                (OLD-FAB :LISP-TYPE ALIEN-STRUCTURE)
                 NIL     ;Error and success routines
                 NIL     ;must be omitted from the call
                (NEW-FAB :LISP-TYPE ALIEN-STRUCTURE))
SYS$RENAME
```

Defines the RMS system service SYS$RENAME.

```
Lisp> (CALL-OUT SYS$RENAME OLD NIL NIL NEW)
1
```

Calls the RMS system service SYS$RENAME. NIL is specified to account for the omitted parameters. This ensures that the correct number of arguments are specified. NIL is specified for the error and status routines because ASTADR parameters must be omitted.

6.8 ERRORS DURING EXTERNAL-ROUTINE EXECUTION

Errors that occur during the activation or the execution of an external routine are trapped by the VAX LISP error handler. The types of errors that might occur during these operations include VMS errors that occur while you are accessing a shareable image and error conditions that the external routine signals (by way of the VMS error-signaling mechanism). You cannot correct these errors.

NOTE

The VAX LISP error handler regards signaled conditions as fatal errors (including conditions that have a success status).

Status codes returned by an external routine, however, do not always represent uncorrectable errors. The operation that the call-out facility performs when a routine returns a status code is determined by the value that is specified with the :CHECK-STATUS-RETURN keyword in the routine's definition. If the value is T, the facility examines the contents of register R0 and interprets the routine's return value as a VMS status code or a user status code. If the severity of the return value is warning, error, or severe-error, the LISP system signals a continuable error. If the :CHECK-STATUS-RETURN keyword is specified with NIL, all status codes are ignored.

You can include your own error handler in an external routine to intercept signaled error conditions. Because the VAX LISP error handler was defined prior to your defining your own handler, the call-out facility passes to the VAX LISP error handler only the error conditions that your error handler does not accept.

6.9 SUSPENDING A LISP SYSTEM THAT CONTAINS CALLS TO EXTERNAL ROUTINES

You can suspend an executing LISP system that contains external routine definitions or calls to external routines. When you suspend such a system, you must be aware of certain restrictions to ensure correct operation of the resumed system. They exist because mapped images or memory acquired from outside the LISP environment (with LIB$GET_VM) are unmapped when the LISP system exits, and they cannot be automatically remapped during a resume operation that follows a suspend operation. Defined external routines are automatically remapped the next time the external routine is called. If you are not aware of the restrictions, other side effects might create undesirable results. Undesirable results can occur from the following:

- Acquiring memory with LIB$GET_VM
- Data initialization
- Open files

6.9.1 Acquiring Memory with LIB$GET_VM

Memory acquired with the VMS LIB$GET_VM function within an external routine is deleted when you exit the LISP system and is not remapped by a resume operation. This prevents you from storing data in acquired memory between calls across a suspend/resume cycle on the routine. Many RTL routines, for example, use this function, and you cannot resume the routines.

6.9.2 Data Initialization

When an external routine contains code that sets flags for an initialization and takes branches based on those flags, the flags are reset when the routine's image is remapped. As a result, the first time you call the routine after a resume operation, the routine executes as if it were executing for the first time, causing a problem if you want to retain the saved data during a suspend/resume cycle.

If you want to retain data across a suspend/resume cycle, do not produce code that depends on a first-time flag. Use one of the following methods:

- Retain data as individual LISP objects, which can be parameters the call-out facility can pass to external routines.
- Store data in alien structures.

Undesired side effects do not occur if external routines you need to use are defined in a series with the DEFINE-EXTERNAL-ROUTINE macro and the resulting system is suspended prior to a call to an external routine. The VAX LISP system retains the information the external routine definition provides.

6.9.3 Open Files

When you exit the LISP system, open files are closed. A resume operation does not reopen files that were opened by external routines in the suspended system.

CHAPTER 7

DEFINING ALIEN STRUCTURES

Alien structures are structured records that are used to exchange data between LISP programs and external routines that refer to VAX data structures, which cannot be accessed with LISP code. Typical alien structures are byte-aligned collections of integers, floating-point numbers, strings, and bit vectors.

VAX LISP provides a facility that enables you to define, create, and access alien structures. This facility is used primarily with the VAX LISP call-out facility; it is used to create parameters for external routines that have arguments or control blocks that are too complicated for the call-out facility to convert (see Section 6.6).

Before you can use an alien structure, you must define the structure with the VAX LISP DEFINE-ALIEN-STRUCTURE macro. This macro is similar to the DEFSTRUCT macro described in _COMMON LISP: The Language_.

An alien-structure definition consists of the following components:

- Alien-structure name and options
- Field descriptions

An example of a definition follows:

```
(DEFINE-ALIEN-STRUCTURE SPACE
  (AREA-1 :SIGNED-INTEGER Ø 4)
  (AREA-2 :SIGNED-INTEGER 4 8))
```

The preceding definition defines an alien structure named SPACE. The structure is defined to be an object that consists of two fields, AREA-1 and AREA-2, which are stored internally as VAX 32-bit integers. The numbers in the definition specify the structure's field lengths in bytes. When the LISP system evaluates the definition, the DEFINE-ALIEN-STRUCTURE macro does the following:

- Defines an access function for each field and names the functions SPACE-AREA-1 and SPACE-AREA-2. They are 1-argument functions, which return the LISP integers that correspond to the VAX integers stored in fields AREA-1 and AREA-2. The access functions are acceptable place indicators in a call to the SETF macro.
- Defines the symbol SPACE to be the name of a data type.
- Defines a predicate function named SPACE-P. The predicate function is a 1-argument function, which returns T if its argument is of type SPACE.

- Defines a constructor function named MAKE-SPACE. The constructor function creates structures of type SPACE.
- Defines a copier function named COPY-SPACE. The copier function is a 1-argument function, which returns a copy of its argument if the argument is of type SPACE.

This chapter describes the alien-structure definition components, provides examples of how to define alien structures, and lists the functions and macros you can use with the alien-structure facility.

See Part II for a description of the DEFINE-ALIEN-STRUCTURE macro.

7.1 ALIEN-STRUCTURE NAME AND OPTIONS

When you define an alien structure, you must specify a name for the structure. In addition, you can specify options that provide the DEFINE-ALIEN-STRUCTURE macro with information about how to name the functions it creates.

7.1.1 Alien-Structure Name

An alien-structure name is a symbol that names a new data type. You can specify an alien structure name with options. If you specify options, specify the name and options as a list whose first element is the name; if you do not specify options, specify the name as a symbol.

7.1.2 Alien-Structure Options

You can assign specific characteristics to an alien structure by specifying options in the structure's definition. Each option consists of a keyword-value pair. A keyword-value pair must be specified as a list as follows:

```
(keyword value)
```

You must specify options in a list whose first element is the name of the alien structure they characterize. The format in which to specify the name and options follows:

```
(name (keyword-1 value-1) (keyword-2 value-2) ...)
```

A list of the keywords that you can use to specify options for alien structures and the characteristics the options define follows:

- :CONC-NAME -- Access function names
- :CONSTRUCTOR -- Constructor function name
- :COPIER -- Copier function name
- :PREDICATE -- Predicate function name
- :PRINT-FUNCTION -- Print function

7.1.2.1 **Access Function** - Access functions are 1- or 2-argument functions you can use to access fields that are defined for an alien structure. Each field is assigned an access function. By default, the DEFINE-ALIEN-STRUCTURE macro produces names for the access functions by prefixing each field name with the name of the alien structure and a hyphen (-). For example, the macro produces access functions named SPACE-AREA-1 and SPACE-AREA-2 when the LISP system evaluates the following definition:

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

If you specify the :CONC-NAME keyword in a structure's definition, the function names are the field names prefixed with the name you specify with the keyword. When the LISP system evaluates the following definition, the DEFINE-ALIEN-STRUCTURE macro produces access functions named GALAXY-AREA-1 and GALAXY-AREA-2:

```
Lisp> (DEFINE-ALIEN-STRUCTURE (SPACE (:CONC-NAME GALAXY-))
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

If you specify NIL with the :CONC-NAME keyword, the function names are the same as the field names, AREA-1 and AREA-2.

7.1.2.2 **Constructor Function** - You can create new structures from an alien-structure definition by using a constructor function. By default, the DEFINE-ALIEN-STRUCTURE macro names a constructor function by prefixing the alien-structure's name with the string MAKE and a hyphen. For example, the macro names the constructor function MAKE-SPACE when the LISP system evaluates the following definition:

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

If you specify a symbol as the value of the :CONSTRUCTOR keyword, the DEFINE-ALIEN-STRUCTURE macro uses the symbol to name the constructor function. When the LISP system evaluates the following definition, the macro names the constructor function CREATE:

```
Lisp> (DEFINE-ALIEN-STRUCTURE (SPACE (:CONSTRUCTOR CREATE))
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

If you specify NIL with the :CONSTRUCTOR keyword, the DEFINE-ALIEN-STRUCTURE macro does not define a constructor function and you cannot create alien structures of that type.

Constructor functions accept optional data initialization keywords. The DEFINE-ALIEN-STRUCTURE macro creates a data initialization keyword for each field you specify in an alien structure definition. The value of an initialization keyword is the value you assign to the field. When the LISP system evaluates the following definition, the

DEFINE-ALIEN-STRUCTURE macro produces two data initialization keywords named :AREA-1 and :AREA-2:

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

The :AREA-1 keyword assigns a value to the field AREA-1 and the :AREA-2 keyword assigns a value to the field AREA-2. For example:

```
Lisp> (MAKE-SPACE :AREA-1 5 :AREA-2 1Ø)
#<Alien Structure SPACE #x5Ø36E8>
```

Constructor functions produced by the DEFINE-ALIEN-STRUCTURE macro accept two keywords in addition to the data initialization keywords that are constructed from the alien structure's field names. Table 7-1 describes the two keywords and their corresponding values.

Table 7-1
Constructor Function Keywords

Keyword	Value
:ALIEN-DATA-LENGTH integer	The number of bytes of memory to be allocated for the alien structure's data vector. This keyword allows efficient use of storage when you are using alien structures as data buffers for variable size records. The default is large enough to store the defined alien structure.
:ALLOCATION value	The type of allocation to be used for the alien structure. Valid values are :DYNAMIC and :STATIC. :DYNAMIC is the default. If :STATIC is specified, the alien structure is allocated in static space and its virtual address is not changed during a garbage collection (see Section 8.3.2).

7.1.2.3 **Copier Function** - A copier function is a 1-argument function you can use to create a copy of an existing alien structure. By default, the DEFINE-ALIEN-STRUCTURE macro names the copier function by prefixing the alien-structure's name with the string COPY and a hyphen. For example, the macro produces a copier function named COPY-SPACE when the LISP system evaluates the following definition:

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

You can specify the :COPY keyword with a symbol value to modify the default name for the copier function. The DEFINE-ALIEN-STRUCTURE macro uses the symbol you specify to name the function. When the LISP system evaluates the following definition, the macro produces a copier function named REPRODUCE:

```
Lisp> (DEFINE-ALIEN-STRUCTURE (SPACE (:COPIER REPRODUCE))
          (AREA-1 :UNSIGNED-INTEGER Ø 4)
          (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

If you specify NIL with the :COPIER keyword, the DEFINE-ALIEN-STRUCTURE macro does not define a copier function.

7.1.2.4 **Predicate Function** - A predicate function is a 1-argument function that determines whether its argument is an occurrence of the defined alien structure. The DEFINE-ALIEN-STRUCTURE macro creates the predicate function name by attaching the alien-structure's name to the characters -P. The macro names the predicate function SPACE-P when the LISP system evaluates the following definition:

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
          (AREA-1 :UNSIGNED-INTEGER Ø 4)
          (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

You can specify the :PREDICATE keyword with a symbol value in an alien structure's definition. When you specify this option, the predicate function name is the symbol value. For example, the following definition produces the predicate function CHECK:

```
Lisp> (DEFINE-ALIEN-STRUCTURE (SPACE (:PREDICATE CHECK))
          (AREA-1 :UNSIGNED-INTEGER Ø 4)
          (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
```

If you specify NIL with the :PREDICATE keyword, the DEFINE-ALIEN-STRUCTURE macro does not define a predicate function.

7.1.2.5 **Print Function** - You can use the :PRINT-FUNCTION keyword to specify the function that is to print an alien structure. A print function has three arguments:

- Name -- the name of the alien structure to be printed
- Stream -- the stream to print to
- Integer -- the print depth

7.2 ALIEN-STRUCTURE FIELD DESCRIPTIONS

Alien structures are composed of fields, each of which has a description that specifies the following:

- Field name

- Field type
- Start and end positions in the structure's data area
- Options that define the field's characteristics

When you specify a field description, you must specify the description as a list whose first element is the name of the field. Specify the field arguments in the following format:

```
(name type start-position end-position options)
```

The following list is an example of a field description:

```
(FIELD-1 :STRING 0 9 :OCCURS 10 :OFFSET 15)
```

7.2.1 Field Name

An alien-structure field name is a symbol that names a field. Functions that access and set the values of alien-structure fields refer to field names to retrieve field description data.

7.2.2 Field Type

An alien-structure field type specifies how a field is to be interpreted by the LISP system. The data in a field is stored as VAX data and is converted to a LISP object when the data is accessed.

Each field associates a VAX data type with the LISP data object the system creates when a function accesses the field. An alien-structure's field type defines the conversion methods the LISP system is to use to transform a field's LISP object to a VAX data type and its VAX data type to a LISP object.

Although the alien-structure facility provides predefined data types, you can define alien-field types with the VAX LISP DEFINE-ALIEN-FIELD-TYPE macro.

7.2.2.1 **Predefined Types** - The VAX LISP alien-structure facility defines a number of types for alien-structure fields. Table 7-2 lists the predefined types and their internal storage representations.

Table 7-2
Predefined Alien-Structure Field Types

Type	Internal Storage Representation
:STRING	VAX character string
:VARYING-STRING	VAX character string; the first 16-bit word of the data vector contains a count of the number of characters in the string (this is the body of the VMS varying string type)

(Continued on next page)

Table 7-2 (Cont.)
Predefined Alien-Structure Field Types

Type	Internal Storage Representation
:SIGNED-INTEGER	Signed two's complement integer
:UNSIGNED-INTEGER	Unsigned integer
:BIT-VECTOR	Unsigned integer
:F-FLOATING	F_floating data
:G-FLOATING	G_floating data
:D-FLOATING	D_floating data
:H-FLOATING	H_floating data
:POINTER	See below
:SELECTION	See below

Descriptions of the :POINTER and the :SELECTION alien-field types follow:

:POINTER Specification -- (:POINTER [name] [:DISPLACED value]): If you specify the :POINTER alien-field type, the field will contain a VAX pointer, which points to the start of the data area of a statically allocated alien structure. If you specify the name argument, the update function checks that the new value of the field points to the name of the specified alien structure. The :DISPLACED keyword causes the stored VAX pointer to point to the start of the alien-structure data area plus the number of bytes specified for the value. You can omit the parentheses if you do not specify the field name and the :DISPLACED keyword. The following field description includes the type :POINTER:

```
(:AREA-1 (:POINTER SPACE) Ø 4)
```

:SELECTION Specification -- (:SELECTION sØ s1 s2 ...): If you specify the :SELECTION alien-field type, the DEFINE-ALIEN-STRUCTURE macro evaluates each element in the list (sn). When the field is accessed, it is interpreted as an unsigned integer, and the corresponding sn value is returned. The SETF form receives one of the values and stores the corresponding integer in the field. The following field description includes the type :SELECTION:

```
(:AREA-1 (:SELECTION 'JUPITER 'MARS 'VENUS) Ø 4)
```

7.2.2.2 **Defining Types** - In addition to the predefined alien-field types, you can define your own field types with the DEFINE-ALIEN-FIELD-TYPE macro. It is described in Part II.

7.2.3 Field Position

You establish the position of a field in an alien structure's data area by specifying the start and end arguments in a field description specification. These arguments are rational numbers that determine the start and end positions of the field.

7.2.3.1 Start Position

The first field in an alien structure's data area starts in position zero. Each field is measured in units of 8-bit bytes. The value can be a ratio; you can, therefore, specify fields within arbitrary bit boundaries. For example, a field with a start value of one-half starts on the fourth bit of the data area. Because the units are 8-bit bytes, a start value of one-third causes an error when you call the DEFINE-ALIEN-STRUCTURE macro.

The LISP system evaluates the start position when it expands the DEFINE-ALIEN-STRUCTURE macro.

7.2.3.2 End Position

The last position a field occupies is the position that precedes the field's end position value. For example, if a field's start position is zero and its end position is four, the field occupies positions Ø to 3.

The end position is measured in 8-bit bytes and the value can be a ratio. The LISP system evaluates the end position when it expands the DEFINE-ALIEN-STRUCTURE macro.

7.2.3.3 Gaps

A gap is memory space that you can allocate as part of an alien structure. Defined functions cannot access gaps. Even though gaps can exist between fields, the LISP system does not generate forms that access and set fields that include gaps; that is, LISP-level code does not process gaps.

7.2.3.4 Overlapping Fields

Alien-structure fields can overlap. This enables you to access data from more than one field at a time. If you change the data in a field that overlaps other fields, the other overlapping fields are also changed.

Overlapping fields are useful when you want data to be interpreted in more than one way. The following definition defines an alien structure that contains fields that overlap.

```
Lisp> (DEFINE-ALIEN-STRUCTURE MASK
        (NUMBER :UNSIGNED-INTEGER Ø 4)
        (BIT-Ø  :UNSIGNED-INTEGER Ø 1/8)
        (BIT-1  :UNSIGNED-INTEGER 1/8 2/8)
        (BIT-2  :UNSIGNED-INTEGER 2/8 3/8)
        (BIT-3  :UNSIGNED-INTEGER 3/8 4/8)
        (BIT-4  :UNSIGNED-INTEGER 4/8 5/8))
MASK
```

After you define the alien structure, you must create a structure by calling the alien structure's constructor function. The following example shows how to use the SETF macro to set the value of the symbol NEWMASK to a new alien-structure of type MASK.

```
Lisp> (SETF NEWMASK (MAKE-MASK))
#<Alien Structure MASK #x50C600>
```

Two ways to set the two and the four bits in NEWMASK and to clear all other bits follow:

```
Lisp> (SETF (MASK-NUMBER NEWMASK) (+ 4 16))
20

Lisp> (SETF (MASK-NUMBER NEWMASK) 0
            (MASK-BIT-2 NEWMASK) 1
            (MASK-BIT-4 NEWMASK) 1)
1
```

7.2.4 **Field Options**

You can define characteristics for the fields specified for an alien structure by specifying field options in the structure's definition. Each option consists of a keyword-value pair. You must specify a keyword-value pair as follows:

```
keyword value
```

Specify options in a list whose first element is the name of the field the options characterize. The format of such a list follows:

```
(name keyword-1 value-1 keyword-2 value-2 ...)
```

A list of the keywords you can use to specify field options and the characteristics the options define follows:

- :DEFAULT -- Initial value
- :READ-ONLY -- Whether a field can be set
- :OCCURS -- Number of times a field repeats
- :OFFSET -- Offset

7.2.4.1 **Initial Value** - You can specify the initial value of a field in a call to the alien-structure's constructor function. Each field description you specify in a call to the DEFINE-ALIEN-STRUCTURE macro causes the constructor function to accept a keyword-value pair that consists of the field name prefixed with a colon (:) and the field's initial value. If you specify different values for overlapping fields, the field values that result are undefined.

To specify an initial value for a field, specify the value with the :DEFAULT keyword in the alien-structure's definition. The LISP system evaluates the initial value when you use the alien structure's constructor function. If you do not specify a value for the field in the call to the constructor function, the system uses the initial value you specified in the alien structure's definition.

7.2.4.2 **Accessing and Setting a Field** - The :READ-ONLY keyword enables you to specify whether a field can be accessed or set. The value you specify with the keyword defines the direction characteristic of the field; the value can be either T or NIL. If you specify T, the DEFINE-ALIEN-STRUCTURE macro generates access functions that are not acceptable place indicators in a call to the SETF macro. If you specify NIL, the macro generates access functions that are acceptable place indicators in a call to the SETF macro.

7.2.4.3 **Repeating Fields** - A field can be repeated within an alien structure. The integer you specify with the :OCCURS keyword determines the number of times the field is repeated.

The argument a field's access function takes depends on whether you specify the :OCCURS keyword in the field's description. If you do not specify the :OCCURS keyword, the access function takes the field name as its argument. If you specify this keyword, the access function takes the field name and an index for arguments. The index is an integer that indicates the occurrence of the field. The first occurrence of the field has an index of zero. Consider the following definition:

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER 0 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8 :OCCURS 4))
SPACE
```

When the LISP system evaluates this definition, the access functions AREA-1 and AREA-2 have the following formats:

```
(SPACE-AREA-1 field)
(SPACE-AREA-2 field index)
```

The LISP system evaluates the value you specify with the :OCCURS keyword when it expands the DEFINE-ALIEN-STRUCTURE macro.

You can change the values of the alien structure's fields with the SETF macro. For example, you can change the value of the field AREA-1 to five by applying the SETF macro to the access function SPACE-AREA-1.

```
Lisp> (SETF PLACE (MAKE-SPACE))
#<Alien Structure SPACE x50C618>
Lisp> (SETF (SPACE-AREA-1 PLACE) 5)
5
```

7.2.4.4 **Offset** - A field offset is the distance in 8-bit bytes from the start of one occurrence of a field to the start of the next occurrence of the field. Specifying an offset enables you to access data files that consist of repeated substructures. You define an offset by specifying a rational number with the :OFFSET keyword. If you specify a value that is greater than the field length, the DEFINE-ALIEN-STRUCTURE macro produces gaps in the alien structure. You can fill them by defining one or more other fields with the :OCCURS and the :OFFSET keywords.

The LISP system evaluates the value you specify with the :OFFSET keyword when it expands the DEFINE-ALIEN-STRUCTURE macro.

7.3 EXAMPLES OF DEFINING ALIEN STRUCTURES

This section provides examples of how to define an alien structure.

1. ```
   Lisp> (DEFINE-ALIEN-STRUCTURE MY-ALIEN (FIELD-1 :STRING Ø 9))
   MY-ALIEN
   ```

   Defines an alien structure, named MY-ALIEN, which contains one field, named FIELD-1. The structure is a string that begins on the first byte and is ten characters long.

2. Below is an example of a Pascal record structure definition.

   ```
   TYPE
    FAMILY_REC = RECORD
    {A record structure definition.}
     SURNAME : PACKED ARRAY[1..2Ø] OF CHAR;
     FATHER : RECORD
      NAME : PACKED ARRAY[1..2Ø] OF CHAR;
      AGE : INTEGER;
      END;
     MOTHER : RECORD
      NAME : PACKED ARRAY[1..2Ø] OF CHAR;
      AGE : INTEGER;
      END;
     NUM_CHILDREN : INTEGER;
     FOR I = Ø TO NUM_CHILDREN DO
      BEGIN
       CHILDREN :  RECORD
       NAME : PACKED ARRAY[1..2Ø] OF CHAR;
       AGE : INTEGER;
       SEX : (FEMALE MALE);
       END;
      END;
    END;
   ```

   An equivalent LISP record structure definition looks like the following:

   ```
   Lisp> (DEFINE-ALIEN-STRUCTURE FAMILY-REC
         "A record structure definition."
           (SURNAME :STRING Ø 2Ø)
           (FATHER-NAME  :STRING 2Ø 4Ø)
           (FATHER-AGE   :UNSIGNED-INTEGER 4Ø 44)
           (MOTHER-NAME  :STRING 44 64)
           (MOTHER-AGE   :UNSIGNED-INTEGER 64 68)
           (NUM-CHILDREN :UNSIGNED-INTEGER 68 72 :DEFAULT 2)
           (CHILD-NAME   :STRING 72 92 :OCCURS 2Ø :OFFSET 25)
           (CHILD-AGE    :UNSIGNED-INTEGER 92 96 :OCCURS 2Ø
                                                 :OFFSET 25)
           (CHILD-SEX (:SELECTION 'FEMALE 'MALE) 96 97
                                                 :OCCURS 2Ø
                                                 :OFFSET 25))
   FAMILY-REC
   ```

   Defines the record FAMILY-REC. The definition contains the :DEFAULT, :OCCURS, and :OFFSET keywords.
   ```

7.4 ALIEN-STRUCTURE FUNCTIONS AND MACROS

In addition to the DEFINE-ALIEN-STRUCTURE macro, VAX LISP provides the following alien-structure functions and macros:

- ALIEN-STRUCTURE-LENGTH function -- Returns the length of an alien structure
- ALIEN-FIELD function -- Accesses a field of a specified type from an alien structure
- DEFINE-ALIEN-FIELD-TYPE macro -- Defines alien-structure field types

Descriptions of the functions and macro are provided in Part II.

CHAPTER 8

VAX LISP I/O EXTENSIONS

VAX LISP provides a number of extensions to the COMMON LISP I/O system. These extensions fall into the following three categories:

- A facility for defining new stream types. VAX LISP lets you define new types of character streams. Section 8.1 describes this facility.

- Information about streams. VAX LISP data types and functions provide more information about streams than is possible using only COMMON LISP facilities. Section 8.2 describes these data types and functions.

- New I/O functions. VAX LISP provides a number of I/O functions in addition to those defined in COMMON LISP. Section 8.3 describes these functions.

8.1 DEFINING NEW TYPES OF STREAMS

COMMON LISP provides several types of streams; for example, synonym streams, broadcast streams, and echo streams. VAX LISP lets you define new types of streams that have characteristics different from those defined in COMMON LISP. You might want to define a new type of stream when you need input or output operations to have side effects unavailable with COMMON LISP streams.

To define a new type of stream, do the following:

- Design the stream. You need to decide how instances of the stream should respond to each valid I/O function.

- Define a means of creating instances of the stream. See Section 8.1.2.

- Define the action of each I/O function when acting on streams of that type. See Section 8.1.3.

8.1.1 Overview of VAX LISP I/O

In the VAX LISP I/O system, every instance of a stream includes a function called the stream dispatch function. Whenever an I/O function is called with a stream as its argument, the stream dispatch function for that stream executes. The stream dispatch function is passed, as its first argument, an I/O request specifier whose value indicates the I/O function that was called. The stream dispatch function must, for every valid I/O request specifier, take the appropriate action for that I/O function operating on that type of stream.

In VAX LISP, streams are implemented as structures. Every stream type structure definition includes the STREAM structure definition provided in VAX LISP. In addition, stream type definitions may contain slots specific to that type of stream. For example, the structure that implements a synonym stream contains a slot for the *symbol* to whose value the synonym stream is equated.

8.1.2 Defining Stream Structures

To define a new stream type, use the DEFSTRUCT macro to create a structure definition that includes the STREAM structure definition. Define additional slots as needed to satisfy the requirements of the stream type you have designed.

Your structure definition must set the following slots in the STREAM structure:

- The DOES-INPUT-P and DOES-OUTPUT-P slots, whose values indicate whether input operations and output operations, respectively, are valid on the new stream type.

- The DISPATCH-FUNCTION slot, whose value is a function you write that performs each of the input and/or output operations that can result from function calls on the stream. Section 8.1.3 describes stream dispatch functions. The value of the DISPATCH-FUNCTION slot may also be a symbol with a function definition.

The following example defines a new stream type called SAMPLE-STREAM:

```
(DEFSTRUCT (SAMPLE-STREAM
               (:CONSTRUCTOR MAKE-SAMPLE-STREAM
                              (INPUT-STREAM OUTPUT-STREAM))
               (:COPIER NIL)
               (:INCLUDE STREAM (DOES-INPUT-P T)
                                (DOES-OUTPUT-P T)
                                (DISPATCH-FUNCTION
                                   #'SAMPLE-STREAM-DISPATCH)))
  (INPUT-STREAM NIL :TYPE STREAM :READ-ONLY T)
  (OUTPUT-STREAM NIL :TYPE STREAM :READ-ONLY T))
```

This definition results in the following:

- A definition for the structure type SAMPLE-STREAM, instances of which contain the slots INPUT-STREAM and OUTPUT-STREAM in addition to those slots inherited from the STREAM structure definition
- A new type, SAMPLE-STREAM
- A new predicate, SAMPLE-STREAM-P
- A by-position constructor function whose format is:

  MAKE-SAMPLE-STREAM *input-stream output-stream*

- Accessor functions SAMPLE-STREAM-INPUT-STREAM and SAMPLE-STREAM-OUTPUT-STREAM

8.1.3 Stream Dispatch Functions

When an I/O function is called on a stream, the dispatch function for that stream type executes. Each stream type's dispatch function must perform the operations for each I/O function that can be called on streams of that type.

When it executes, the stream dispatch function receives at least two arguments:

1. An I/O request specifier (a keyword that corresponds to the I/O function that was called on the stream).
2. The stream on which the function was called.

The stream dispatch function receives additional arguments that correspond to the additional arguments with which the I/O function was called. A stream dispatch function must be able to receive any number of arguments without error, although it need not process all arguments it receives.

For example, if MY-SAMPLE-STREAM is an instance of SAMPLE-STREAM, the following call:

```
(READ-CHAR MY-SAMPLE-STREAM NIL 'EOF-ENCOUNTERED NIL)
```

results in a call to SAMPLE-STREAM-DISPATCH with five arguments. The first argument is the I/O request specifier, :READ-CHAR in this case. The second through fifth arguments are MY-SAMPLE-STREAM, NIL, 'EOF-ENCOUNTERED, and NIL.

Table 8-1 lists the I/O request specifiers that stream dispatch functions must handle. Not all I/O functions have a corresponding specifier, because some I/O functions (such as WRITE-LINE and TERPRI) are defined and implemented in terms of lower-level functions.

Table 8-1: I/O Request Specifiers

Must Be Handled by All Streams

:CLOSE	:ELEMENT-TYPE

Must Be Handled by All Input Streams

:CLEAR-INPUT	:LISTEN2
:NREAD-LINE	:READ-CHAR
:READ-LINE	:UNREAD-CHAR

Must Be Handled by All Output Streams

:CLEAR-OUTPUT	:FINISH-OUTPUT
:FORCE-OUTPUT	:FRESH-LINE
:IMMEDIATE-OUTPUT-P	:LINE-POSITION
:RIGHT-MARGIN	:WRITE-CHAR
:WRITE-STRING	

Note: See Section 8.3 for a description of the functions IMMEDIATE-OUTPUT-P, LISTEN2, LINE-POSITION, NREAD-LINE, and RIGHT-MARGIN. All other functions are described in *COMMON LISP: The Language*.

The :ABORT flag argument to CLOSE is passed as the first argument with the :CLOSE request.

The stream dispatch function SAMPLE-STREAM-DISPATCH might be written as follows. Note the use of &REST to ensure that SAMPLE-STREAM-DISPATCH can be called with an indefinite number of arguments without error.

```
(DEFUN SAMPLE-STREAM-DISPATCH
       (REQUEST STREAM &OPTIONAL ARG1 ARG2 ARG3 &REST ARG4)
  (DECLARE (IGNORE ARG4))
  (CASE (THE KEYWORD REQUEST)
    (:READ-CHAR
        (LET ((CHAR (READ-CHAR
                         (SAMPLE-STREAM-INPUT-STREAM STREAM)
                         ARG1 ARG2 ARG3)))
          (UNLESS (EQ CHAR ARG2)
            (WRITE-CHAR CHAR
                         (SAMPLE-STREAM-OUTPUT-STREAM STREAM)))
          CHAR))
    (:WRITE-CHAR (WRITE-CHAR ARG1
                              (SAMPLE-STREAM-OUTPUT-STREAM STREAM)))
    (:UNREAD-CHAR (UNREAD-CHAR ARG1
                                (SAMPLE-STREAM-INPUT-STREAM STREAM)))
    .
    .
    .
    (T (ERROR "~A does not recognize the ~A request"
             STREAM REQUEST))))
```

SAMPLE-STREAM-DISPATCH provides special handling when READ-CHAR is called on a stream of type SAMPLE-STREAM. For other I/O functions, SAMPLE-STREAM-DISPATCH simply calls the same function on either the input stream or the output stream. SAMPLE-STREAM-DISPATCH signals an error if it is called with an unrecognized I/O request specifier.

8.2 GETTING INFORMATION ABOUT STREAMS

VAX LISP provides access to more detailed information about streams than is called for in *COMMON LISP: The Language*. VAX LISP provides a separate data type for each stream type, a predicate for each stream type, and functions to retrieve elements that were used to construct streams.

Table 8-2 lists the stream data types and predicates. The STREAMP predicate is satisfied by objects of any of the stream data types.

Table 8-2: Stream Data Types and Predicates

Data Type	**Predicate Function**
BROADCAST-STREAM	BROADCAST-STREAM-P *object*
CONCATENATED-STREAM	CONCATENATED-STREAM-P *object*
DRIBBLE-STREAM	DRIBBLE-STREAM-P *object*
ECHO-STREAM	ECHO-STREAM-P *object*
FILE-STREAM	FILE-STREAM-P *object*
STRING-STREAM	STRING-STREAM-P *object*
SYNONYM-STREAM	SYNONYM-STREAM-P *object*
TERMINAL-STREAM	TERMINAL-STREAM-P *object*
TWO-WAY-STREAM	TWO-WAY-STREAM-P *object*

Table 8-3 lists functions that retrieve information from streams. You cannot use SETF with these functions.

Table 8-3: Stream Informational Functions

Function	**Return Value**
BROADCAST-STREAM-STREAMS *broadcast-stream*	List of streams
CONCATENATED-STREAM-STREAMS *concatenated-stream*	List of streams
ECHO-STREAM-INPUT-STREAM *echo-stream*	Stream
ECHO-STREAM-OUTPUT-STREAM *echo-stream*	Stream
SYNONYM-STREAM-SYMBOL *synonym-stream*	Symbol
TWO-WAY-STREAM-INPUT-STREAM *two-way-stream*	Stream
TWO-WAY-STREAM-OUTPUT-STREAM *two-way-stream*	Stream

8.3 NEW I/O FUNCTIONS

VAX LISP provides several I/O functions in addition to those defined in *COMMON LISP: The Language*. Most of these functions are variations on existing COMMON LISP functions. This section describes the functions in alphabetical order. The optional arguments *input-stream* and *output-stream* have the defaults specified in *COMMON LISP: The Language*:

- *input-stream* defaults to *STANDARD-INPUT*. If a value of T is supplied, the value of *TERMINAL-IO* is used.
- *output-stream* defaults to *STANDARD-OUTPUT*. If a value of T is supplied, the value of *TERMINAL-IO* is used.

IMMEDIATE-OUTPUT-P Function

Returns T if an output stream does not buffer its output and NIL otherwise. The I/O system uses this function to improve output performance by buffering output when the stream itself does not perform buffering.

Format

IMMEDIATE-OUTPUT-P &OPTIONAL *output-stream*

Argument

output-stream

An output stream.

Return Value

T if *output-stream* does not buffer output and NIL otherwise.

LINE-POSITION Function

Returns the number of characters that have been output on the current line if that number can be determined and NIL otherwise.

Format

LINE-POSITION &OPTIONAL *output-stream*

Argument

output-stream

An output stream.

Return Value

A fixnum or NIL.

LISTEN2 Function

Returns two values. The first is identical to the value returned by the COMMON LISP LISTEN function; the second is T if end-of-file was encountered on the input stream, and NIL otherwise. You can use this function wherever you would normally use LISTEN.

Format

LISTEN2 &OPTIONAL *input-stream*

Argument

input-stream

An input stream.

Return Value

Two values:

1. T if a character is immediately available from *input-stream* and NIL otherwise.

2. T if end-of-file was encountered on *input-stream* and NIL otherwise.

NREAD-LINE Function

NREAD-LINE, a destructive version of the COMMON LISP READ-LINE function, places the characters that were read into the string supplied as its first argument. NREAD-LINE returns the number of characters read, a flag indicating whether end-of-file was encountered, and a string containing the line if the line could not fit into the string supplied.

Format

NREAD-LINE *string*
&OPTIONAL *input-stream eof-error-p eof-value-p recursive-p*

Arguments

string

A character string. NREAD-LINE updates *string* with the line that was read. If *string* has a fill pointer, the fill pointer is adjusted so that *string* appears to contain exactly what was read from the stream. If *string* is adjustable and the size of the line exceeds the size of *string*, then *string* is extended.

Since NREAD-LINE does not return *string*, you must maintain a pointer to *string*.

input-stream eof-error-p eof-value-p recursive-p

These arguments correspond to the arguments to READ-LINE documented in *COMMON LISP: The Language*.

Return Value

Three values:

1. A fixnum indicating the number of characters that were in the line.

2. T if the line was terminated by end-of-file and NIL otherwise.

3. NIL if the line fit into *string*: otherwise a string containing the line.

OPEN-STREAM-P Function

Returns T if a stream is open, and NIL otherwise.

Format

OPEN-STREAM-P *stream*

Argument

stream

A stream.

Return Value

T or NIL.

RIGHT-MARGIN Function

Returns the default right margin used by the pretty printer when printing to the stream. The current margin used by the pretty printer is controlled by the variable *PRINT-RIGHT-MARGIN*.

Format

RIGHT-MARGIN &OPTIONAL *output-stream*

Argument

output-stream

An output stream.

Return Value

A non-negative fixnum indicating the default right margin for *output-stream*.

If you use the PATHNAME function to create a pathname called THIS-PATHNAME, whose host field value is the current node, the NAMESTRING function does not include the host in the namestring it returns. The following call to the SETF macro sets THIS-PATHNAME to the pathname that is created with the PATHNAME function:

```
Lisp> (SETF THIS-PATHNAME
            (PATHNAME "MIAMI::DBA1:[SMITH]LOGIN.COM;4"))
#S(PATHNAME :HOST "MIAMI" :DEVICE "DAB1" :DIRECTORY "SMITH" :NAME
 "LOGIN" :TYPE "COM" :VERSION 4)
```

When the NAMESTRING function is called with THIS-PATHNAME as its argument, the namestring that is returned does not include the pathname's host.

```
Lisp> (NAMESTRING THIS-PATHNAME)
"DBA1:[SMITH]LOGIN.COM;4"
```

Suppose you use the PATHNAME function to create a pathname called THAT-PATHNAME whose host field value is BOSTON. The following call to the SETF macro sets THAT-PATHNAME to the pathname that is created with the PATHNAME function:

```
Lisp> (SETF THAT-PATHNAME
            (PATHNAME "BOSTON::DBA1:[SMITH]LOGIN.COM;4"))
#S(PATHNAME :HOST "BOSTON" :DEVICE "DBA1" :DIRECTORY "SMITH"
:NAME "LOGIN" :TYPE "COM" :VERSION 4)
```

Because the current node is MIAMI and the host field value of THAT-PATHNAME is BOSTON, the NAMESTRING function returns a namestring that includes all the pathname field values.

```
Lisp> (NAMESTRING THAT-PATHNAME)
"BOSTON::DBA1:[SMITH]LOGIN.COM;4"
```

If you want to invoke DECnet-VAX and you want to specify the current host, specify the host with an access control string or specify zero as the host. For example:

```
Lisp> (SETF THAT-PATHNAME
            (PATHNAME "0::THATDEVICE:[SMITH]LOGIN.COM"))
#S(PATHNAME :HOST "0" :DEVICE "THATDEVICE" :DIRECTORY "SMITH" :NA
ME "LOGIN" :TYPE "COM" :VERSION NIL)
Lisp> (NAMESTRING THAT-PATHNAME)
"0::THATDEVICE:[SMITH]LOGIN.COM"
```

It was noted in Table 8-3 that in VAX LISP the host field of a pathname can include an access control string. If the NAMESTRING function is called with a pathname argument whose host field includes an access control string, the namestring that is returned includes the host, even if the value in the pathname's host field is the same as the current node.

Assume that the current host is MIAMI. The following SETF expression sets THIS-PATHNAME to the pathname that is created with the PATHNAME function:

```
Lisp> (SETF THIS-PATHNAME
            (PATHNAME
              "MIAMI\"SMITH MYPASSWORD\"::THISDEVICE:[SMITH]FILE"))
#S(PATHNAME :HOST "MIAMI:\SMITH password\"" :DEVICE "THISDEVICE"
:DIRECTORY "SMITH" :NAME "FILE" :TYPE NIL VERSION: NIL)
```

The host field of the pathname that is created contains the host MIAMI and the access control string SMITH MYPASSWORD. The NAMESTRING function, when called with THIS-PATHNAME as its argument, returns a namestring that includes all the pathname field values.

```
Lisp> (NAMESTRING THIS-PATHNAME)
"MIAMI\"SMITH password\"::THISDEVICE:[SMITH]FILE"
```

8.3 THE GARBAGE COLLECTOR

When VAX LISP is executing, LISP objects are created dynamically. Some of the objects that are created are always used and referred to, while others are referred to for only a short time. When a LISP object can no longer be referred to, the space that the object occupies can be reclaimed by the VAX LISP system. This process of reclaiming space is called garbage collection.

The VAX LISP garbage collector is a stop-and-copy garbage collector. The LISP system includes a dynamic memory pool, which is divided into two equal-sized spaces: dynamic-Ø space and dynamic-1 space. At a given time, LISP objects are allocated in either dynamic-Ø or dynamic-1 space. When the memory in the current space is exhausted, LISP processing is temporarily suspended, and the LISP data objects that can still be referenced are copied to the other space. The objects that cannot be referenced are not copied.

You can ignore garbage collections of dynamic memory space when you are writing LISP programs. Garbage collections occur automatically when the current dynamic space is exhausted, and LISP processing continues when a garbage collection is complete.

Sections 8.3.1 through 8.3.6 provide information about the VAX LISP garbage collector.

8.3.1 Frequency of Garbage Collection

The frequency of garbage collection is proportional to the amount of dynamic memory space that is available in the VAX LISP system. You can set the amount of dynamic memory space that is to be available by specifying the DCL /MEMORY command qualifier (see Section 2.8.9) when you invoke the LISP system. Garbage collection occurs less often if you use this qualifier to increase the size of the dynamic memory space.

The degree to which the frequency of garbage collection and the size of dynamic memory affects run-time efficiency depends on the program that is being executed. If a program creates more permanent objects than objects that can be referred to for a short period of time, the garbage collector has to perform more copy operations. As a result, the program slows down. The fewer the copy operations the garbage collector has to perform, the faster the garbage collection is finished.

8.3.2 Static Space

LISP objects that are created in static space are not collected by the garbage collector. These objects do not move and they are not deleted, even if they can no longer be referred to. You can create objects in static space by using the MAKE-ARRAY function with the :ALLOCATION keyword (see Section 8.7) or by using the constructor functions that are defined by the DEFINE-ALIEN-STRUCTURE macro for alien structures (see Section 7.1.2.2).

8.3.3 LISP Processing

LISP processing is suspended during a garbage collection. The VMS operating system queues asynchronous functions, such as those defined by the VAX LISP BIND-KEYBOARD-FUNCTION function, for delivery after garbage collection is finished. Asynchronous functions are discussed in Section 8.5.

8.3.4 Messages

When a garbage collection occurs, a message is displayed when the operation begins and when it is finished. You can suppress these messages by changing the value of the VAX LISP *GC-VERBOSE* variable to NIL. When the value is NIL, messages are not displayed.

You can also specify the contents of the messages by changing the values of the VAX LISP *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables. The *GC-VERBOSE*, *PRE-GC-MESSAGE*, and *POST-GC-MESSAGE* variables are described in Part II.

NOTE

> If you suppress or change the garbage collection messages and a garbage collection is initiated due to a control stack overflow, it is difficult to determine whether your program is in a recursive loop. Therefore, you should not suppress or change the messages before you debug your program.

8.3.5 Available Space

Garbage collection generally occurs when a LISP object is being created. If a garbage collection occurs and not enough dynamic memory space is available to allocate the object, an error is signaled. When this situation exists, you can suspend the LISP image and resume it later with more dynamic-memory space. For information about how to suspend and resume a LISP image, see Section 2.9.

8.3.6 Garbage Collection Failure

The garbage collection process may fail to complete. If, for example, a garbage collection is initiated because of control stack overflow, the size of the control stack must increase and the amount of dynamic memory space must decrease. If the reduced dynamic memory space cannot contain all the LISP objects that can be referred to, the LISP image is terminated and control returns to the DCL level. This condition is usually caused by a user programming error, such as a function that is recursive and nonterminating.

8.4 INPUT AND OUTPUT

VAX LISP I/O is implemented with two sets of low-level functions. One set of functions handles terminal I/O by way of direct QIOs to the terminal driver. The other set of functions handles all other I/O (particularly to disk files) by way of calls to VAX Record Management Services (RMS). See the VAX/VMS RMS Reference Manual for information about VAX RMS.

The VAX LISP implementation dependencies for I/O have to do with the following topics:

- Newline character
- Terminal input
- End-of-file operations
- Record length
- File organization
- Functions

The implementation-dependent information about these topics is provided in Sections 8.4.1 through 8.4.6.

8.4.1 Newline Character

COMMON LISP defines the #\NEWLINE character as a character that is returned from the READ-CHAR function as an end-of-line indicator. In VAX LISP, the character code for the #\NEWLINE character has an integer value of 255.

In VAX LISP, the WRITE-CHAR and WRITE-STRING functions interpret the #\NEWLINE character as follows:

- When the WRITE-CHAR function is called with the #\NEWLINE character as its argument value, the function starts writing a new line. This call is equivalent to a call to the TERPRI function (see COMMON LISP: The Language).

- When the WRITE-STRING function is called with an argument string that contains the #\NEWLINE character, the function divides the string into two lines. The following example shows the output that is dispayed by the WRITE-STRING function when the #\NEWLINE character is not used:

```
Lisp> (WRITE-STRING (CONCATENATE 'STRING
                                 "NEW"
                                 "LINE"))
NEWLINE
"NEWLINE"
```

  Both of the strings NEW and LINE are displayed on the same line. A call to the WRITE-STRING function, which includes a string argument that contains the #\NEWLINE character, looks like the following:

```
Lisp> (WRITE-STRING (CONCATENATE 'STRING
                                 "NEW"
                                 (STRING #\NEWLINE)
                                 "LINE"))
NEW
LINE
"NEW
LINE"
```

  This call to the WRITE-STRING function displays the strings NEW and LINE on separate lines.

The #\NEWLINE character is the only character that causes a new line to be written. VAX LISP writes carriage returns and linefeeds without special interpretation.

8.4.2 Terminal Input

In VAX LISP, terminals perform input operations in line mode. Input is returned by the READ-CHAR function only after you press the RETURN or ESCAPE key or type CTRL/Z.

The READ-CHAR function returns ASCII characters as data unless one of the following conditions exists:

- A character is used by the VMS terminal driver for terminal control.
- A character is defined to invoke an asynchronous function.

See the VAX/VMS I/O User's Guide (Volume 1) for information on terminal control characters, and see Section 8.5 for information about asynchronous functions.

You can change the mode in which your terminal performs input operations by invoking the VAX LISP SET-TERMINAL-MODES function with the :PASSALL keyword (see Part II). For example:

```
Lisp> (SET-TERMINAL-MODES :PASS-ALL T)
T
```

If the value of the :PASS-ALL keyword is T, the SET-TERMINAL-MODES function puts your terminal in passall mode. When a terminal is in passall mode, control characters processed by the VMS system and characters defined to invoke asynchronous functions are not recognized by the LISP system. In addition, the READ-CHAR function performs input operations differently than it does when the terminal is in line mode. In line mode, the READ-CHAR function does not return a character until you press the RETURN key; in passall mode, it returns a character as soon as the character is typed. See <u>COMMON LISP: The Language</u> for a description of the READ-CHAR function.

To put your terminal back into line mode, invoke the SET-TERMINAL-MODES function with the :PASS-ALL keyword set to NIL.

```
Lisp> (SET-TERMINAL-MODES :PASS-ALL NIL)
T
```

8.4.3 End-of-File Operations

In VAX LISP, read operations from a file do not indicate the end of the file until the operation after the last character in the file is performed.

Read operations from a terminal do not indicate the end of a file in VAX LISP.

In VAX LISP, you can close a stream that is connected to your terminal if the stream is not related to the stream bound to the *TERMINAL-IO* variable. If you attempt to close the stream bound to the *TERMINAL-IO* variable, no action is performed.

8.4.4 Record Length

VAX LISP uses RMS to process file I/O. Therefore, the maximum record length in VAX LISP must conform to the maximum record length in RMS. A maximum of 32,767 characters can be written to a disk file, and a maximum of 9,995 characters can be written to a magnetic tape. If you exceed these record-length limits, an error is signaled and nothing is written to the file.

The WRITE-CHAR function causes an immediate operation when it is called with a terminal stream. As a result, there is no limit on the number of calls you can make to the WRITE-CHAR function before you invoke the TERPRI function if you are writing to a terminal.

Your user-buffered I/O byte limit quota determines the maximum string length you can write to your terminal. You can find out what the quota is by invoking the VAX LISP GET-PROCESS-INFORMATION function with the :BIO-BYTE-QUOTA keyword (see Part II). For example:

```
Lisp> (GET-PROCESS-INFORMATION "SMITH" :BIO-BYTE-QUOTA)
(:BIO-BYTE-QUOTA 30000)
```

NOTE

You can prevent your buffered I/O byte limit quota from overflowing by including calls to the TERPRI function or by specifying the #\NEWLINE character in your output.

8.4.5 File Organization

VAX LISP reads RMS files sequentially. Character files created by VAX LISP have sequential organization, variable-length records, and the implied carriage-return attribute. Files created for binary output (for example, the WRITE-BYTE function) have sequential organization, variable-length records, and no carriage-control attributes.

8.4.6 Functions

Four COMMON LISP functions used for I/O have VAX LISP dependencies and need further explanation. The implementation information for the following functions is provided in the next four sections:

- FILE-LENGTH
- FILE-POSITION
- OPEN
- WRITE-CHAR

8.4.6.1 **FILE-LENGTH Function** - The length of a file is measured in units of the OPEN function's :ELEMENT-TYPE keyword. In VAX LISP, files cannot be measured in these units for all the supported element types. Therefore, the FILE-LENGTH function returns NIL.

You can determine the total number of 8-bit bytes that can occupy a file by invoking the GET-FILE-INFORMATION function with the :END-OF-FILE-BLOCK and :FIRST-FREE-BYTE keywords, and then performing the following steps:

1. Multiply the value returned for the :END-OF-FILE-BLOCK keyword minus one by 512
2. Add the value you get in Step 1 to the value returned for the :FIRST-FREE-BYTE keyword

For more information on the GET-FILE-INFORMATION function, see Part II.

8.4.6.2 **FILE-POSITION Function** - The FILE-POSITION function returns or sets the current position within a random-access file. VAX LISP does not support random-access files; therefore, the function returns NIL.

8.4.6.3 **OPEN Function** - Before you can access a file, you must open it with the OPEN function or the WITH-OPEN-FILE macro. The OPEN function can be specified with keywords that determine the type of stream that is to be created and how errors are to be handled. The keywords you can specify are the following:

- :DIRECTION
- :ELEMENT-TYPE
- :IF-EXISTS
- :IF-DOES-NOT-EXIST

VAX LISP has restrictions on the values you can specify for the preceding keywords. The rest of this section explains the restrictions.

You can specify the :IO value for the :DIRECTION keyword only if the specified stream is connected to a terminal or mailbox. When you specify the :IO value, the target device must exist before the OPEN function is called. Therefore, if you specify this value for the :DIRECTION keyword, you cannot specify the :IF-EXISTS keyword, and you can specify the :IF-DOES-NOT-EXIST keyword only with the :ERROR value.

The :IF-EXISTS keyword accepts :RENAME and :RENAME-AND-DELETE as values, but the operation they perform is the same as the operation performed by the :NEW-VERSION value. The :RENAME, :RENAME-AND-DELETE, and :NEW-VERSION values create a new file with the same file name but increase the version number by one.

VAX LISP supports all the values for the :ELEMENT-TYPE keyword except CHARACTER. VAX LISP allows you to open binary streams, but the maximum byte size for a stream is 512 8-bit bytes.

8.4.6.4 **WRITE-CHAR Function** - The WRITE-CHAR function disregards the bit and font attributes of characters.

8.5 ASYNCHRONOUS FUNCTIONS

An asynchronous function is a function that is invoked when a specific event occurs. If an asynchronous function is defined for an event, the VAX LISP system interrupts the current LISP process and invokes the asynchronous function when the event occurs. When the asynchronous function exits, the VAX LISP system resumes the process at the point where it was interrupted.

VAX LISP provides a function you can use to define asynchronous functions: BIND-KEYBOARD-FUNCTION. It binds an ASCII control character to a function. Once a control character is bound to a function, you can cause the VAX LISP system to interrupt the current evaluation and call the function asynchronously by typing the control character.

Asynchronous functions are not always called as soon as the defined event occurs (typing of a control key). If a low-level LISP function, such as CDR or CONS, is being evaluated or a garbage collection is being performed, asynchronous functions are placed in a queue until they can be evaluated. Delays in asynchronous function evaluation are generally not perceptible. An example of when you might perceive a delay is when the system performs a garbage collection.

If you suspend the LISP system when asynchronous functions are defined, the functions that are defined by the BIND-KEYBOARD-FUNCTION function are still defined when the system is resumed. The key/function bindings are not lost.

In addition to the BIND-KEYBOARD-FUNCTION function are the VAX LISP functions GET-KEYBOARD-FUNCTION and UNBIND-KEYBOARD-FUNCTION. The GET-KEYBOARD-FUNCTION function returns information about a function that is bound to a control character, and the UNBIND-KEYBOARD-FUNCTION function removes the binding of a function from a control character.

Descriptions of the BIND-KEYBOARD-FUNCTION, GET-KEYBOARD-FUNCTION, and UNBIND-KEYBOARD-FUNCTION functions are provided in Part II.

8.6 THE COMPILER

8.6.1 Compiler Restrictions

The VAX LISP compiler translates interpreted function definitions into function objects that contain VAX instructions. The COMPILE function causes these objects to be bound as the definitions of the symbols that name them. The COMPILE-FILE function puts the objects into an output file. Because of the way these two functions handle such objects, a restriction exists for the use of each of the functions.

8.6.1.1 **COMPILE Function** - The compiler cannot compile pieces of code unless they are function definitions. Therefore, you cannot use the COMPILE function to compile a function unless you create the function in a null lexical environment (not top level). An example of a LISP expression that cannot be evaluated follows:

```
Lisp> (LET ((COUNTER Ø))
          (COMPILE NIL #'(LAMBDA () (INCF COUNTER))))
```

The COMPILE function cannot compile the function object in the preceding example because it depends on the lexical environment in which it was created. In the following example, the COMPILE function is called with a lambda expression rather than a function object:

```
Lisp> (LET ((COUNTER Ø))
          (COMPILE NIL '(LAMBDA () (INCF COUNTER))))
```

The call to the COMPILE function in the preceding example compiles the lambda expression. The value that is returned is a compiled object that increments the dynamic value of COUNTER. The compiled object does not increment the local value of COUNTER, which encloses the call to the COMPILE function.

8.6.1.2 **COMPILE-FILE Function** - The COMPILE-FILE function encloses each top-level form of the file it is compiling with an anonymous function definition. Therefore, the function cannot put a compiled function object that is recognized as data into an output file. Consider the following form:

```
Lisp> (SETQ F '#.(COMPILE NIL '(LAMBDA (C) (PRINT C))))
#<Compiled Function #:G1149 #x504C4C>
```

When the COMPILE-FILE function reads the preceding form from a file that is being compiled, an anonymous function is created. This function becomes part of the third element of the list whose first element is the SETQ special form. The preceding call to the SETQ special form can be compiled but it cannot be put into the output file.

8.6.2 Compiler Optimizations

VAX LISP allows you to control two qualities of compiled code: the speed of the generated code and whether run-time safety checking is to be performed. The default values for these qualities is one. You can set the values globally and locally. To set the values globally in VAX LISP, you can either use the DCL LISP command with the /COMPILE and /OPTIMIZE qualifiers (see Sections 2.8.2 and 2.8.10) or specify the OPTIMIZE declaration in a call to the PROCLAIM function (see COMMON LISP: The Language). Both of these methods of setting the quality values produce the same results. For example, if you are at the DCL level of operation and you want to set the global values of the speed quality to three and the safety quality to two, use the following DCL command specification:

```
$ LISP/COMPILE/OPTIMIZE=(SPEED:3,SAFETY:2) MYPROG.LSP
```

If you are in LISP and you want to set the global values of the speed and safety qualities, specify the PROCLAIM function as the first form in the file. For example, to set the values of the qualities to the same values that were set in the preceding example, specify the following call to the PROCLAIM function as the first form in the file MYPROG.LSP:

```
(PROCLAIM '(OPTIMIZE (SPEED 3) (SAFETY 2)))
```

You can also set the quality values locally. To do this, you must use the OPTIMIZE declaration within the form for which you want the values to be set. Local optimization quality values override global quality values.

If you are more concerned about the safety of your code than the speed at which it is evaluated, the value of the safety quality must be greater than one, or the value of the speed quality must be less than two. When this relationship exists between the two quality values, the compiler generates safe code. Safe code is code that checks arguments to ensure that the arguments are of the proper data type. Examples of safe code are the following:

- Code that uses generic arithmetic
- Code that checks if the arguments of calls to functions that require list arguments are lists

- Code that checks whether indices used to access arrays are bound

If you are more interested in producing code that is evaluated fast than in producing safe code, the value of the speed quality must be greater than or equal to two, and the value of the safety quality must be less than or equal to one. When this relationship exists between the two quality values, the compiler considers type declarations and generates type-specific code. Type-specific code executes faster than safe code. If you want the compiler to generate type-specific code, you must specify declarations in your code in addition to setting the values of the speed and the safety qualities to the correct values.

Consider the following code, and suppose the value of the safety quality is one and the speed quality is two:

```
(DEFUN LOOP-OVER-A-SUBLIST (INPUT-LIST)
  (DO ((I (GET-INITIAL-VALUE) (1+ I))
       (L INPUT-LIST (CDR L)))
      ((OR (>= I (THE FIXNUM *FINAL-VALUE*))
           (ENDP L))
       L)
    (DECLARE (FIXNUM I)
             (LIST L))
    (DO-SOME-WORK L I)))
```

Since the value of the safety quality is less than two and the value of the speed quality is greater than one, the compiler regards the type declarations. In this example, the types FIXNUM and LIST are declared with the the following form:

```
(DECLARE (FIXNUM I)
         (LIST L))
```

When the example code is compiled, the compiler uses the type declarations and translates the 1+, CDR, ENDP, and >= functions in the code as follows:

- The 1+ function becomes one VAX instruction.
- The CDR function becomes one VAX instruction.
- The ENDP function is transformed into the NULL function.
- The >= function becomes two VAX instructions: a longword comparison and a branch.

It is critical that the value of the *FINAL-VALUE* variable and the return value of the GET-INITIAL-VALUE function are fixnums. Also, the INPUT-LIST argument specified for the LOOP-OVER-A-SUBLIST function must be a true list (not an atom or a dotted list).

If a declaration is violated, the error that results is not signaled. For example, if you call the LOOP-OVER-A-SUBLIST function with the symbol LOOP, an error results because the argument is not a list, but the error is not signaled. Errors such as this can cause damage to the LISP environment, which cannot be repaired. By default, the values of the speed and safety qualities are set such that error checking and signaling code are generated for all operations; such values prevent you from damaging the LISP environment.

If the INPUT-LIST argument in the preceding example is not guaranteed to always be a list, you can add an explicit type check before the DO loop. The following form is an example of an explicit type check:

```
(UNLESS (LISTP INPUT-LIST)
    ;but doesn't check for a dotted-list
  (ERROR "Cannot loop through this object: ~S." INPUT-LIST))
```

The check performed by the LISTP function is evaluated at run time even though the compiler might heed the FIXNUM and LIST declarations.

8.7 FUNCTIONS AND MACROS

Several functions and macros described in COMMON LISP: The Language have implementation dependencies. This section provides information for such functions and macros. Table 8-4 lists the names of the functions and macros and provides a brief explanation of the type of information that is implementation dependent. The rest of the section provides detailed descriptions presented alphabetically by name. Each description consists of the function's or macro's use, implementation-dependent information, format, applicable arguments, return value, and examples of use.

See COMMON LISP: The Language for complete descriptions of the functions and macros described in this section.

Table 8-4
Summary of Implementation-Dependent Functions and Macros

Name	Function or Macro	Implementation-Dependent Information
APROPOS	Function	Optional argument and DO-SYMBOLS macro
APROPOS-LIST	Function	Optional argument and DO-SYMBOLS macro
BREAK	Function	Facility invoked
COMPILE-FILE	Function	Keywords and return value
DESCRIBE	Function	Displayed output
DIRECTORY	Function	Merge the argument
DRIBBLE	Function	Terminal I/O while in the Editor is not saved; cannot nest calls
ED	Function	Arguments
GET-INTERNAL-RUN-TIME	Function	Meaning of return value
LOAD	Function	Finds latest file
LONG-SITE-NAME	Function	Logical name and return value
MACHINE-INSTANCE	Function	Logical name and return value

(Continued on next page)

Table 8-4 (Cont.)
Summary of Implementation-Dependent Functions and Macros

Name	Function or Macro	Implementation-Dependent Information
MACHINE-VERSION	Function	Return value
MAKE-ARRAY	Function	:ALLOCATION keyword
REQUIRE	Function	Modules
ROOM	Function	Displayed output
SHORT-SITE-NAME	Function	Logical name and return value
TIME	Macro	Displayed output
TRACE	Macro	Keywords
WARN	Function	Facility invoked

APROPOS

Searches through packages for symbols whose print names contain a specified string. The function is not sensitive to the case of characters. The string can be either the print name or a substring of the symbol's print name.

The APROPOS function displays a message that shows the string that is being searched for and the name of the package that is being searched. When the function finds a symbol whose print name contains the string, the function displays the symbol's name. If the symbol has a value, the function displays the phrase "has a value" after the symbol as follows:

```
*MY-SYMBOL*, has a value
```

If the symbol has a function definition, the function displays the phrase "has a definition" after the symbol as follows:

```
MY-FUNCTION, has a definition
```

In VAX LISP, the APROPOS function uses the DO-SYMBOLS macro rather than the DO-ALL-SYMBOLS macro. As a result, the function displays only symbols that are accessible from the current package. For information on packages, see COMMON LISP: The Language.

Format

```
APROPOS string &OPTIONAL package
```

Arguments

string

The string to be searched for in the symbols' print names. If you specify a symbol for this argument, the symbol's print name is used.

package

An optional argument. If you specify the argument, the symbols in the specified package are searched. If you specify T, all packages are searched. If you do not specify the argument, the symbols that are accessible in the current package are searched.

Return Value

No value.

Example

```
Lisp> (APROPOS "*PRINT")

Symbols in package USER containing the string "*PRINT":
  *PRINT-LEVEL*, has a value
  *PRINT-GENSYM*, has a value
  *PRINT-RADIX*, has a value
  *PRINT-PRETTY*, has a value
  *PRINT-CASE*, has a value
  *PRINT-CIRCLE*, has a value
  *PRINT-BASE*, has a value
  *PRINT-ESCAPE*, has a value
  *PRINT-LENGTH*, has a value
  *PRINT-ARRAY*, has a value
  *PRINT-SLOT-NAMES-AS-KEYWORDS*, has a value
```

Searches the package USER for the string *PRINT and displays a list of the symbols that contain the specified string.

APROPOS-LIST

Searches through packages for symbols whose print names contain a specified string. The function is not sensitive to the case of characters. The string can be either the print name or a substring of the symbol's print name.

When the function completes its search, it returns a list of the symbols whose print names contain the string.

In VAX LISP, the APROPOS-LIST function uses the DO-SYMBOLS macro rather than the DO-ALL-SYMBOLS macro. As a result, the function includes only symbols that are accessible from the current package in the list it returns. For information on packages, see COMMON LISP: The Language.

Format

```
APROPOS-LIST string &OPTIONAL package
```

Arguments

string

The string to be searched for in the symbols' print names. If you specify a symbol for this argument, the symbol's print name is used.

package

An optional argument. If you specify the argument, the symbols in the specified package are searched. If you specify T, all packages are searched. If you do not specify the argument, the symbols that are accessible in the current package are searched.

Return Value

A list of the symbols whose print names contain the specified string.

Example

```
Lisp> (APROPOS-LIST "ARRAY")
(ARRAY-TOTAL-SIZE ARRAY-DIMENSION ARRAY-DIMENSIONS *PPRINT-ARRAY-
FORMATTERS* SIMPLE-ARRAY ARRAY-DIMENSION-LIMIT ARRAY-ELEMENT-TYPE
 ARRAYP *PRINT-ARRAY* ARRAY-RANK ARRAY-RANK-LIMIT MAKE-ARRAY ARRA
Y-TOTAL-SIZE-LIMIT ARRAY-ROW-MAJOR-INDEX ADJUST-ARRAY ARRAY ARRAY
-IN-BOUNDS-P ADJUSTABLE-ARRAY-P ARRAY-HAS-FILL-POINTER-P)
```

Searches the symbols that are accessible in the current package for the string ARRAY and returns a list of the symbols that contain the specified string.

BREAK

Invokes a break loop. A break loop is a nested read-eval-print loop. For more information about break loops, see Section 4.3.

Format

```
BREAK &OPTIONAL format-string &REST args
```

Arguments

format-string

An optional argument. The string of characters that is passed to the FORMAT function to create the break-loop message.

args

An optional argument. The arguments that are passed to the FORMAT function as arguments for the format string.

Return Value

When the CONTINUE function is called to exit the break loop, the BREAK function returns NIL.

Example

```
(WHEN (EMERGENCY-SITUATION-P STATUS)
    (BREAK "Emergency situation ~D encountered." STATUS))
```

Calls the BREAK function if the value of the EMERGENCY-SITUATION-P function is not NIL. The break message contains the status code.

COMPILE-FILE

Compiles a specified LISP source file and writes the compiled code as a binary fast-loading file (type FAS).

Format

COMPILE-FILE input-pathname &KEY {keyword value}*

Arguments

input-pathname

A pathname, namestring, symbol, or stream. The compiler uses the value of the *DEFAULT-PATHNAME-DEFAULTS* variable to fill in file specification components that are not specified.

keyword value

Optional keyword-value pairs, which specify the options for the compilation. All the keywords are VAX LISP extensions except :OUTPUT-FILE. Table 8-5 lists the keywords and the values that can be specified with them.

Table 8-5
COMPILE-FILE Options

Keyword-Value Pair	Description
:LISTING value	Specifies whether the compiler is to produce a listing file. The value can be T, NIL, or a pathname, namestring, symbol, or stream. If you specify T, the compiler produces a listing file. The listing file is assigned the same name as the source file with the file type LIS, and is placed in the directory in which the source file is. If you specify NIL, no listing is produced. The default value is NIL. If you specify a pathname, namestring, symbol, or stream, the compiler uses the value as the specification of the listing file. The compiler uses the LIS file type and the value of the *DEFAULT-PATHNAME-DEFAULTS* variable to fill the components of the file specification that are not specified.

(Continued on next page)

Table 8-5 (Cont.)
COMPILE-FILE Options

Keyword-Value Pair	Description
:MACHINE-CODE value	Specifies whether the compiler is to include the machine code it produces for each function and macro it compiles in the listing file. The value can be either T or NIL. If you specify T, the listing file contains the machine code. If you specify NIL, the listing file does not contain machine code. The default value is NIL.
:OPTIMIZE value	Specifies the optimization qualities the compiler is to use during compilation. The value must be a list of sublists. Each sublist must contain a symbol and a value, which specify the optimization qualities and corresponding values that the compiler is to use during compilation. For example: ((SPACE 2) (SAFETY 1)) The default value for each quality is one. For a detailed discussion about compiler optimizations, see Section 8.6.2.
:OUTPUT-FILE value	Specifies whether the compiler is to produce a fast-loading file. The value can be T, NIL, or a pathname, namestring, symbol, or stream. If you specify T, the compiler produces a fast-loading file. The output file is assigned the same name as the source file with the file type FAS and is placed in the directory the source file is in. The default value is T. If you specify NIL, no fast-loading file is produced.

(Continued on next page)

Table 8-5 (Cont.)
COMPILE-FILE Options

Keyword-Value Pair	Description
	If you specify a pathname, namestring, symbol, or stream, the compiler uses the value as the specification of the output file. The compiler uses the FAS file type and the value of the *DEFAULT-PATHNAME-DEFAULTS* variable to fill the components of the file specification that are not specified.
:VERBOSE value	Specifies whether the compiler is to display the name of functions and macros it compiles. The value can be either T or NIL. If you specify T, the compiler displays the name of each function and macro. If a listing file exists, the compiler also includes the names in the listing file. If you specify NIL, the names are not displayed or included in the listing file. The default value is the value of the *COMPILE-VERBOSE* variable (see Part II).
:WARNINGS value	Specifies whether the compiler is to display warning messages. The value can be either T or NIL. If you specify T, the compiler displays warning messages. If a listing file exists, the compiler also includes the messages in the listing file. If you specify NIL, warning messages are not displayed or included in the listing file. The default value is the value of the *COMPILE-WARNINGS* variable (see Part II).

Return Value

If the compiler generated an output file, a namestring is returned. Otherwise, NIL is returned.

Examples

1. Lisp> (COMPILE-FILE "FACTORIAL" :VERBOSE T)

   ```
   Starting compilation of file DBA1:[SMITH]FACTORIAL.LSP;1

   FACTORIAL compiled.

   Finished compilation of file DBA1:[SMITH]FACTORIAL.LSP;1
   0 Errors, 0 Warnings
   "DBA1:[SMITH]FACTORIAL.FAS;1"
   ```

   Compiles the file FACTORIAL.LSP, which is in the current directory. A fast-loading file named FACTORIAL.FAS is produced. The compilation is logged to the terminal because the :VERBOSE keyword is specified with the value T.

2. Lisp> (COMPILE-FILE "FACTORIAL" :OUTPUT-FILE NIL

   ```
                                 :LISTING T
                                 :WARNINGS NIL
                                 :VERBOSE NIL)
   NIL
   ```

   Compiles the file FACTORIAL.LSP, which is in the current directory. A fast-loading file is not produced, because the :OUTPUT-FILE keyword is specified with the value NIL. A listing file named FACTORIAL.LIS is produced. Warning messages are suppressed because the :WARNINGS keyword is specified with the value NIL.

DESCRIBE

Displays the information about a specified object. If the specified object has a documentation string, this function displays it in addition to the other information it displays. The type of information the function displays depends on the type of the object. For example, if a symbol is specified, the function displays the symbol's value, definition, properties, and other types of information. If a floating-point number is specified, the number's internal representation is displayed in a way that is useful for tracking such things as roundoff errors.

Format

```
DESCRIBE object
```

Argument

object

The object about which information is to be displayed.

Return Value

No value.

Examples

1. Lisp> (DESCRIBE 'C)

```
It is the symbol C
Package:  USER
Value:    unbound
Function: undefined
```

   Displays information about the symbol C.

2. Lisp> (DESCRIBE 'FACTORIAL)

```
It is the symbol FACTORIAL
Package:  USER
Value:    unbound
Function: a compiled-function
    FACTORIAL n
```

   Displays information about the symbol FACTORIAL.

3. Lisp> (DESCRIBE PI)

```
It is the long-float 3.14159265358979323846264338327950
Sign:         +
Exponent:     2 (radix 2)
Significand:  0.78539816339744830961566084581988L0
```

   Displays information about the object PI.

4. Lisp> (DESCRIBE '#(1 2 3 4 5))

```
It is a simple-vector
Dimensions:    (5)
Element type: t
Adjustable:    no
Fill Pointer: no
Displaced:     no
```

   Displays information about the simple-vector #(1 2 3 4 5).

DIRECTORY

Converts its argument to a pathname and returns a list of the pathnames for the files matching the specification. The DIRECTORY function is similar to the DCL DIRECTORY command.

Format

```
DIRECTORY pathname
```

Argument

pathname

The pathname, namestring, stream, or symbol for which the list of file system pathnames is to be returned. In VAX LISP, this argument is merged with the following default file specification:

```
host::device:[directory]*.*;*
```

The host, device, and directory values are supplied by the *DEFAULT-PATHNAME-DEFAULTS* variable.

Specifying just a directory is equivalent to specifying a directory with wild cards (*) in the name, type, and version fields of the argument. For example, the following two expressions are equivalent:

```
(DIRECTORY "[MYDIRECTORY]") <=>

     (DIRECTORY "[MYDIRECTORY]*.*;*")
```

Both of these expressions return a list of pathnames that represent the files in the directory MYDIRECTORY.

Specifying just a directory with a specified version field is equivalent to specifying a directory and version with wild cards (*) in the name and type fields of the argument. For example, the following two expressions are equivalent:

```
(DIRECTORY "[MYDIRECTORY];0") <=>

     (DIRECTORY "[MYDIRECTORY]*.*;")
```

Both of these expressions return a list of the pathnames that represent the newest versions of the files in the directory MYDIRECTORY.

The following equivalent expressions return the list of pathnames for files in your default directory:

```
(DIRECTORY "") <=>

     (DIRECTORY (DEFAULT-DIRECTORY))
```

Return Value

A list of pathnames if the specified pathname is matched and NIL if the pathname is not matched.

Example

```
Lisp> (DEFUN MY-DIRECTORY (&OPTIONAL (FILENAME ""))
        (LET ((PATHNAME (PATHNAME FILENAME))
              (DIRECTORY (DIRECTORY FILENAME)))
          (COND ((NULL DIRECTORY)
                 (FORMAT T
                         "~%No files match ~A.~%"
                         (NAMESTRING FILENAME)))
                (T (FORMAT T
                        "~%The following ~:[files are~;file is ~]
                        in the directory ~A:[~A]:"
                           (EQUAL (LENGTH DIRECTORY) 1)
                           (PATHNAME-DEVICE
                            (NTH Ø DIRECTORY))
                           (PATHNAME-DIRECTORY
                            (NTH Ø DIRECTORY)))
                   (DOLIST (DIRECTORY)
                     (FORMAT T "~&~T~A" (FILE-NAMESTRING X)))
                   (TERPRI)))
          (VALUES)))
MY-DIRECTORY
Lisp> (MY-DIRECTORY)
The following files are in the directory DBA1:[SMITH.TESTS]:
  TEST5.DRB;1
  TEST1.LSP;7
  TEST1.LSP;6
  TEST1.LSP;5
  EXAMPLE.TXT;2
  TEST3.LSP;15
  TEST6.LSP;1
Lisp> (MY-DIRECTORY ".LSP;")
The following files are in the directory DBA1:[SMITH.TESTS]:
  TEST1.LSP;7
  TEST3.LSP;15
  TEST6.LSP;1
```

- The call to the DEFUN macro defines a function that formats the output of the DIRECTORY function, making the output more readable. The function is defined such that it accepts an optional argument and does not return a value.

- The first call to the function MY-DIRECTORY shows how the function formats the directory output when an argument is not specified.

- The second call to the function MY-DIRECTORY includes an argument; the output includes only the latest versions of file names of type LSP.

DRIBBLE

Sends the input and output of an interactive LISP session to a specified pathname, enabling you to save a record of what you do during the session in the form of a file.

When you want to stop the DRIBBLE function from sending input and output to the pathname, close the file by calling the function without an argument.

In VAX LISP, there are two restrictions on the use of the DRIBBLE function.

- When you are in the Editor, terminal I/O is not recorded in a dribble file.
- You cannot nest calls to the DRIBBLE function.

Format

```
DRIBBLE &OPTIONAL pathname
```

Argument

pathname

The pathname to which the input and output of the LISP session is to be sent.

Return Value

If a pathname was specified with the function, no value is returned. If the function was sending input and output to a pathname and the function was called again, without a pathname, to stop it, T is returned. If the function was called without a pathname and it had not been called previously with a pathname, NIL is returned.

Examples

1.
```
Lisp> (DRIBBLE 'NEWFNCTN.LSP)
Dribbling to DBA1:[SMITH]NEWFNCTN.LSP;1
NIL
Lisp>
```

Creates a dribble file named NEWFNCTN.LSP. The LISP system sends input and output to the file until you call the DRIBBLE function again (without an argument) or exit LISP.

2.
```
Lisp> (DRIBBLE)
T
```

Closes the dribble file that was previously opened.

ED

Invokes the VAX LISP Editor. This function can be specified with an optional argument whose value can be a namestring, pathname, or symbol. In VAX LISP, the argument's value can also be a list. In addition, you can specify a :TYPE argument whose value can be the :FUNCTION or :VALUE keyword.

Format

```
ED &OPTIONAL x &KEY :TYPE keyword
```

Arguments

x

The namestring, pathname, symbol, or list that is to be edited. If you specify a list, the list must be a generalized variable that can be specified in a call to the SETF macro. The list is evaluated and it returns a value you can edit. When you write the buffer containing the value, the Editor replaces the value of the generalized variable with the new value.

If you specify a symbol, you can also specify the keyword argument. The value of the keyword informs the Editor whether you want to edit the symbol's function or macro definition or its value.

keyword

You can specify this argument if the x argument is a symbol. The value is a keyword that affects the interpretation of the x argument's value. You can specify one of the following keywords:

:FUNCTION	The Editor is invoked to edit the function or macro definition associated with the specified symbol.
:VALUE	The Editor is invoked to edit the specified symbol's value.

The default value for the :TYPE keyword is the :FUNCTION keyword.

Return Value

No value.

Examples

1. Lisp> (ED "[SMITH.LISP]NEWPROG.LSP")

   Invokes the Editor to edit the file NEWPROG.LSP in the directory SMITH.LISP.

2. Lisp> (ED 'FACTORIAL)

   Invokes the Editor to edit a function named FACTORIAL.

3. Lisp> (ED '*PPRINT-SPECIAL-FORMATTERS* :TYPE :VALUE)

   Invokes the Editor to edit the value of the symbol *PPRINT-SPECIAL-FORMATTERS*.

4.
```
Lisp> (DEFSTRUCT ROOM
        DOORS
        WINDOWS
        OUTLETS
        COLOR)
HOUSE
Lisp> (SETQ ROOM2 (MAKE-ROOM :DOORS 1
                             :WINDOWS 3
                             :OUTLETS 4
                             :COLOR 'BLUE))
#S(ROOM :DOORS 1 :WINDOWS 3 :OUTLETS 4 :COLOR BLUE)
Lisp> (ED '(ROOM-COLOR ROOM2))
```

- The call to the DEFSTRUCT macro defines a structure named ROOM.
- The call to the SETQ special form creates an instance of the structure ROOM.
- The call to the ED function invokes the Editor to edit the COLOR slot of the structure bound to ROOM2.

GET-INTERNAL-RUN-TIME

Returns an integer that represents the elapsed CPU time used for the current process. The function value is measured in terms of the INTERNAL-TIME-UNITS-PER-SECOND constant. A description of the INTERNAL-TIME-UNITS-PER-SECOND constant is provided in COMMON LISP: The Language.

Format

GET-INTERNAL-RUN-TIME

Return Value

The elapsed CPU time used for the current process.

Example

```
Lisp> (DEFMACRO MY-TIME (FORM)
        `(LET* ((START-REAL-TIME (GET-INTERNAL-REAL-TIME))
                (START-RUN-TIME (GET-INTERNAL-RUN-TIME))
                (VALUE ,FORM)
                (END-RUN-TIME (GET-INTERNAL-RUN-TIME))
                (END-REAL-TIME (GET-INTERNAL-REAL-TIME)))
           (FORMAT *TRACE-OUTPUT*
                   "~&Run Time: ~,2F sec., ~
                    Real Time: ~,2F sec.~%"
                   (/ (- END-RUN-TIME START-RUN-TIME)
                      INTERNAL-TIME-UNITS-PER-SECOND)
                   (/ (- END-REAL-TIME START-REAL-TIME)
                      INTERNAL-TIME-UNITS-PER-SECOND))
          VALUE))
MY-TIME
```

Defines a macro that displays timing information about the evaluation of a specified form.

LOAD

Reads and evaluates the contents of a file into the LISP environment.

In VAX LISP, if the specified file name does not specify an explicit file type, the LOAD function locates the source file (type LSP) or fast-loading file (type FAS) with the latest file write date and loads it. This ensures that the latest version of the file is loaded, whether or not the file is compiled.

Format

LOAD filename &KEY {keyword value}*

Arguments

filename

The name of the file to be loaded.

keyword value

Optional keyword-value pairs, which specify the options for the load operation. Table 8-6 lists the keywords and the corresponding values you can specify.

Table 8-6
LOAD Options

Keyword-Value Pair	Description
:IF-DOES-NOT-EXIST value	Specifies whether the LOAD function signals an error if the file does not exist. The value can be either T or NIL. If you specify T, the function signals an error if the file does not exist. If you specify NIL, the function returns NIL if the file does not exist. The default value is T.
:PRINT value	Specifies whether the value of each file that is loaded is printed to the stream bound to the *STANDARD-OUTPUT* variable. The value can be either T or NIL. If you specify T, the value of each file is printed to the stream. If you specify NIL, no action is taken. The default value is NIL.

(Continued on next page)

Table 8-6 (Cont.)
LOAD Options

Keyword-Value Pair	Description
:VERBOSE value	Specifies whether the LOAD function is to print a message in the form of a comment to the stream bound to the *STANDARD-OUTPUT* variable. The value can be either T or NIL. If you specify T, the function prints a message. The message includes information, such as the name of the file that is being loaded. If you specify NIL, the function uses the value of *LOAD-VERBOSE* variable. The default is NIL.

Return Value

A value other than NIL if the load operation is successful.

Example

```
Lisp> (COMPILE-FILE "FACTORIAL")

Starting compilation of file DBA1:[SMITH]FACTORIAL.LSP;1

FACTORIAL compiled.

Finished compilation of file DBA1:[SMITH]FACTORIAL.LSP;1
Ø Errors, Ø Warnings
"DBA1:[SMITH]FACTORIAL.FAS;1"
Lisp> (LOAD "FACTORIAL")
; Loading contents of file DBA1:[SMITH]FACTORIAL.FAS;1
;   FACTORIAL
; Finished loading DBA1:[SMITH]FACTORIAL.FAS;1
T
```

- The call to the COMPILE-FILE function produces a fast-loading file named FACTORIAL.FAS.
- The call to the LOAD function locates the fast-loading file FACTORIAL.FAS and loads the file into the LISP environment.

LONG-SITE-NAME

Translates the logical name LISP$LONG_SITE_NAME. If the first character of the resulting string is an at sign (@), the remainder of the string is assumed to be a file specification. The file is read and its content is returned as a string that represents the physical location of the computer hardware on which the VAX LISP system is running. If the first character of the translation is not an at sign, the translation itself is returned as the long-site name.

Format

```
LONG-SITE-NAME
```

Return Value

The contents of a file or the translation of the logical name LISP$LONG_SITE_NAME is returned as a string that represents the physical location of the computer hardware on which the VAX LISP system is running. If a long-site name is not defined, NIL is returned.

Example

```
Lisp> (LONG-SITE-NAME)
"Smith's Computer Company
Artificial Intelligence Group
22 Plum Road
Canterbury, Ohio 47190"
```

Returns a detailed description of the physical location of the computer hardware on which a VAX LISP system is running.

MACHINE-INSTANCE

Translates the logical name LISP$MACHINE_INSTANCE.

Format

```
MACHINE-INSTANCE
```

Return Value

The translation of the logical name LISP$MACHINE_INSTANCE is returned as a string. If the logical name is not defined and DECnet-VAX is running, the node name is returned. If the logical name is not defined and DECnet-VAX is not running, NIL is returned.

Example

```
Lisp> (MACHINE-INSTANCE)
"MIAMI"
```

The name of the computer hardware being used.

MACHINE-VERSION

Returns the content of the system identification (SID) register as a string that represents the version of computer hardware on which the VAX LISP system is running. The contents of the SID are determined by the type of CPU -- for example, 780, 750, or 730. For more information about CPU types, see the VAX Architecture Handbook.

Format

```
MACHINE-VERSION
```

Return Value

The contents of the SID register are returned as a string.

Example

```
Lisp> (MACHINE-VERSION)
"SID Register: #x01383550"
```

The version of the VAX computer hardware being used.

MAKE-ARRAY

Creates arrays. When this function is used with the VAX LISP :ALLOCATION keyword and :STATIC value, it creates a statically allocated array.

During system usage, the garbage collector changes the memory addresses of most LISP objects. You can prevent the garbage collector from changing addresses by allocating objects in static space. Arrays, vectors, and strings can be statically allocated if you use the :ALLOCATION keyword and :STATIC value in a call on the MAKE-ARRAY function. Once an object is statically allocated, its virtual address does not change.

NOTE

> A statically allocated object maintains its memory address even if a SUSPEND/RESUME operation is performed.

Calling the MAKE-ARRAY function with the :ALLOCATION :STATIC keyword-value pair is useful if you are creating a large array. Preventing the garbage collector from moving the array, causes the garbage collector to go faster.

The MAKE-ARRAY function has a number of keywords that can be used in conjunction with the :ALLOCATION keyword. Use the appropriate keywords to construct static vectors and strings. See <u>COMMON LISP: The Language</u> for information on the MAKE-ARRAY keywords.

Format

    MAKE-ARRAY dimensions :ALLOCATION keyword

Arguments

dimensions

> A list of positive integers that are to be the dimensions of the array.

keyword

> Whether the LISP object is to be statically allocated. You can specify one of the following keywords:

:DYNAMIC	The LISP object is not to be statically allocated. This is the default.
:STATIC	The LISP object is to be statically allocated.

Return Value

> The statically allocated object.

Example

```
Lisp> (DEFPARAMETER BIT-BUFFER
        (MAKE-ARRAY '(1000 1000) :ELEMENT-TYPE 'BIT
                                 :ALLOCATION :STATIC))
BIT-BUFFER
```

Creates a large array of bits named BIT-BUFFER, which is not intended to be removed from the system.

REQUIRE

Searches LISP memory for a specified module. If the module is not loaded, the function loads the files that you specify for the module. If the module is loaded, its files are not reloaded.

When you call the REQUIRE function in VAX LISP, the function checks whether you explicitly specified pathnames that name the files it is to load. If you specify pathnames, the function loads the files the pathnames represent. If you do not specify pathnames, the function searches for the module's files in the following order:

1. The function searches the current directory for a source file or a fast-loading file with the specified module name. If the function finds such a file, it loads the file into LISP memory. This search forces the function to locate a module you have created before it locates a module of the same name that is present in one of the public places (see following steps).

2. If the logical name LISP$MODULES is defined, the function searches the directory this logical name refers to for a source file or a fast-loading file with the specified module name. This search enables the VAX LISP sites to maintain a central directory of modules.

3. The function searches the directory that the logical name LISP$SYSTEM refers to for a source file or a fast-loading file with the specified module name. This search enables you to locate modules that are provided with the VAX LISP system.

4. If the function does not find a file with the specified module name, an error is signaled.

When you load a module, the pathname that refers to the directory that contains the module is bound to the *MODULE-DIRECTORY* variable. A description of the *MODULE-DIRECTORY* variable is provided in Part II.

Format

    REQUIRE module-name &OPTIONAL pathname

Arguments

module-name

> A string or a symbol that names the module whose files are to be loaded.

pathname

> A pathname or a list of pathnames that represent the files to be loaded into LISP memory. The files are loaded in the same order the pathnames are listed, from left to right.

Return Value

> Undefined

Example

```
Lisp> *MODULES*
("CALCULUS" "NEWTONIAN-MECHANICS")
Lisp> (REQUIRE 'RELATIVE)
T
Lisp> *MODULES*
("RELATIVE" "CALCULUS" "NEWTONIAN-MECHANICS")
```

- The first call to the *MODULES* variable shows that the modules CALCULUS and NEWTONIAN-MECHANICS are loaded.
- The call to the REQUIRE function checks whether the module RELATIVE is loaded. The previous call to the *MODULES* variable indicated that the module was not loaded, therefore, the function loaded the module RELATIVE.
- The second call to the *MODULES* variable shows that the module RELATIVE was loaded.

ROOM

Displays information about LISP memory. Information is displayed for the following memory spaces:

- Read-only space
- Static space
- Dynamic space
- Control stack
- Binding stack

The following information is provided for each type of space:

- Total number of memory pages that can be used
- Current number of memory pages being used
- Percentage of free memory pages available for use

The information for each storage type is displayed on one line in the following format:

```
Read-Only Storage   Total Size: 4864, Current Allocation: 4209, Free: 13%
```

All counts are in pages.

Format

ROOM &OPTIONAL value

Argument

value

Optional argument whose value can be either T or NIL. If you specify NIL, the function displays the same information that it displays when the argument is not specified. If you specify T, the function displays additional information for the read-only, static, and dynamic storage spaces. The additional information consists of a breakdown of the storage space being used by each VAX LISP data type. The information is displayed in the following tabular format:

```
Read-Only Storage   Total Size: 4864, Current Allocation: 4209, Free: 13%
    (reserved)      0  Functions:    189  Arrays:         0  B-Vectors:    42
    Strings:      480  U-Vectors:   3174  Bignums:        1  (reserved)     0
    (reserved)      0  Sngl Flos:      1  Dbl Flos:       1  Long Flos:     1
    Ratios:         0  Complexes:      0  Symbols:        0  Conses:      320
    (reserved)      0  S Flo Vecs:     0  D Flo Vecs:     0  L Flo Vecs:    0
```

Table 8-7 lists the headings and VAX LISP data types the ROOM function displays for each type of storage space.

Return Value

No value.

Table 8-7
Data Type Headings

Heading	Data Type
Functions	Compiled function descriptors
Arrays	Nonsimple array descriptors
B-Vectors	Boxed vectors -- simple vectors of LISP objects
Strings	Character strings
U-Vectors	Unboxed vectors -- simple vectors that contain compiled code, alien structures, or integers of type (mod n)
Bignums	Bignums
Sngl Flos	Single-format floating-point numbers
Dbl Flos	Double-format floating-point numbers
Long Flos	Long-format floating-point numbers
Ratios	Ratios
Complexes	Complex numbers
Symbols	Symbols
Conses	Conses
S Flo Vecs	Simple single-format floating-point vectors
D Flo Vecs	Simple double-format floating-point vectors
L Flo Vecs	Simple long-format floating-point vectors

Examples

1. Lisp> (ROOM)

```
Read-Only Storage   Total Size: 4864, Current Allocation: 4209, Free: 13%
Static Storage      Total Size: 2560, Current Allocation: 1971, Free: 23%
Dynamic-0 Storage   Total Size: 4962, Current Allocation: 1748, Free: 65%
Control Stack       Total Size: 254, Current Allocation: 1, Free: 100%
Binding Stack       Total Size: 94, Current Allocation: 1, Free: 100%
```

Displays a list of the current memory storage information.

2. Lisp> (ROOM T)

```
Read-Only Storage   Total Size: 4864, Current Allocation: 4209, Free: 13%
    (reserved)     0  Functions:    189  Arrays:        0  B-Vectors:    42
    Strings:     480  U-Vectors:   3174  Bignums:       1  (reserved)     0
    (reserved)     0  Sngl Flos:      1  Dbl Flos:      1  Long Flos:     1
    Ratios:        0  Complexes:      0  Symbols:       0  Conses:      320
    (reserved)     0  S Flo Vecs:     0  D Flo Vecs:    0  L Flo Vecs:    0

Static Storage      Total Size: 2560, Current Allocation: 1971, Free: 23%
    (reserved)     0  Functions:    253  Arrays:        1  B-Vectors:     2
    Strings:     452  U-Vectors:    501  Bignums:       0  (reserved)     0
    (reserved)     0  Sngl Flos:      2  Dbl Flos:      2  Long Flos:     0
    Ratios:        0  Complexes:      0  Symbols:     310  Conses:      448
    (reserved)     0  S Flo Vecs:     0  D Flo Vecs:    0  L Flo Vecs:    0
```

```
Dynamic-Ø Storage   Total Size: 4962, Current Allocation: 1549, Free: 69%
    (reserved)     Ø  Functions:      1  Arrays:         1  B-Vectors:    296
    Strings:     239  U-Vectors:      6  Bignums:        4  (reserved)      Ø
    (reserved)     Ø  Sngl Flos:      1  Dbl Flos:       1  Long Flos:      1
    Ratios:        1  Complexes:      Ø  Symbols:       2Ø  Conses:       978
    (reserved)     Ø  S Flo Vecs:     Ø  D Flo Vecs:     Ø  L Flo Vecs:     Ø

Control Stack       Total Size: 254, Current Allocation: 1, Free: 1ØØ%
Binding Stack       Total Size:  94, Current Allocation: 1, Free: 1ØØ%
```

Displays a detailed list of the current memory storage information.

SHORT-SITE-NAME

Translates the logical name LISP$SHORT_SITE_NAME.

Format

```
SHORT-SITE-NAME
```

Return Value

The translation of the logical name LISP$SHORT_SITE_NAME is returned as a string. If the logical name is not defined, NIL is returned.

Example

```
Lisp> (SHORT-SITE-NAME)
"Artificial Intelligence Group"
```

Returns a short description of the physical location of the computer hardware on which a VAX LISP system is running.

TIME

Evaluates a form, displays the form's CPU time and real time, and returns the values the form returns.

The time information is displayed in the following format:

```
CPU Time: Ø.Ø3 sec., Real Time: Ø.23 sec.
```

If garbage collections occur during the evaluation of a call to the TIME macro, the macro displays another line of time information. This line includes information about the CPU time and real time used by the garbage collector.

Format

```
TIME form
```

Argument

form

The form that is to be evaluated.

Return Value

The form's return values are returned.

Example

```
Lisp> (TIME (TEST))
CPU Time: Ø.Ø3 sec., Real Time: Ø.23 sec.
6
```

Displays the amount of time used in compiling the form (TEST) and then returns the value the form returned (6).

TRACE

Enables tracing for one or more functions and macros.

VAX LISP allows you to specify a number of options that suppress the TRACE macro's displayed output or that cause additional information to be displayed. The options are specified as keyword-value pairs. The keyword-word value pairs you can specify are listed in Table 8-8.

NOTE

The arguments specified in a call to the TRACE macro are not evaluated.

Format

```
TRACE &REST trace-description
```

Argument

trace-description

One or more optional arguments. If an argument is not specified, the TRACE macro returns a list of the functions and macros that are currently being traced. Trace-description arguments can be specified in three formats:

- One or more function and/or macro names can be specified.

```
name-1 name-2 ...
```

- The name of each function or macro can be specified with keyword-value pairs. The keyword-value pairs specify the operations the TRACE macro is to perform when it traces the specified function or macro. The name and the keyword-value pairs must be specified as a list whose first element is the name.

```
(name keyword-1 value-1
      keyword-2 value-2 ...)
```

- A list of function and/or macro names can be specified with keyword-value pairs. The keyword-value pairs specify the operations the TRACE macro is to perform when it traces each function and/or macro in the list. The list of names and the keyword-value pairs must be specified as a list whose first element is the list of names.

```
((name-1 name-2 ...) keyword-1 value-1
                     keyword-2 value-2 ...)
```

Table 8-8 lists the keywords and values that can be specified. The forms that are referred to in the value descriptions are evaluated in the null lexical environment (not at top level).

Table 8-8
TRACE Options

Keyword-Value Pair	Description
:DEBUG-IF form	Specifies a form that is to be evaluated before and after each call to the specified function or macro. If the form returns a value other than NIL, the VAX LISP debugger is invoked before and after the function or macro is called.
:PRE-DEBUG-IF form	Specifies a form that is to be evaluated before each call to the specified function or macro. If the form returns a value other than NIL, the VAX LISP debugger is invoked before the specified function or macro is called.
:POST-DEBUG-IF form	Specifies a form that is to be evaluated after each call to the specified function or macro. If the form returns a value other than NIL, the VAX LISP debugger is invoked after the specified function or macro is called.
:PRINT form-list	Specifies a list of forms that are to be evaluated and whose values are to be displayed before and after each call to the specified function or macro. The values are displayed one per line and are indented to match other output displayed by the TRACE macro. If the TRACE macro cannot evaluate the argument, the debugger is invoked (see Section 4.4).
:PRE-PRINT form-list	Specifies a list of forms that are to be evaluated and whose values are to be displayed before each call to the specified function or macro. The values are displayed one per line and are indented to match other output displayed by the TRACE macro. If the TRACE macro cannot evaluate the argument, the debugger is invoked (see Section 4.4).

(Continued on next page)

Table 8-8 (Cont.)
TRACE Options

Keyword-Value Pair	Description
:POST-PRINT form-list	Specifies a list of forms that are to be evaluated and whose values are to be displayed after each call to the specified function or macro. The values are displayed one per line and are indented to match other output displayed by the TRACE macro. If the TRACE macro cannot evaluate the argument, the debugger is invoked (see Section 4.4).
:STEP-IF form	Specifies a form that is to be evaluated before each call to the specified function or macro. If the form returns a value other than NIL, the stepper is invoked and the function or macro is stepped through. See Section 4.5 for information on the stepper.
:SUPPRESS-IF form	Specifies a form that is to be evaluated before each call to the specified function or macro. If the form returns a value other than NIL, the TRACE macro does not display the arguments and the return value of the specified function or macro.
:DURING name	Specifies a function or macro name or a list of function and macro names. The specified function or macro is traced only when it is called (directly or indirectly) from within one of the specified functions or macros.

Return Value

A list of the functions currently being traced.

Examples

1. Lisp> (TRACE FACTORIAL COUNT1 COUNT2)
   (FACTORIAL COUNT1 COUNT2)

   Enables the tracer for the functions FACTORIAL, COUNT1, and COUNT2.

2. Lisp> (TRACE)
   (FACTORIAL COUNT1 COUNT2)

   Returns a list of the functions for which the tracer is enabled.

WARN

Invokes the VAX LISP error handler. The error handler displays an error message and checks the value of the *BREAK-ON-WARNINGS* variable. If the value is NIL, the error handler causes the WARN function to return NIL; if the value is not NIL, the error handler checks the value of the *ERROR-ACTION* variable. The value of the *ERROR-ACTION* variable can be either the :EXIT or the :DEBUG keyword. If the value is :EXIT, the error handler causes the LISP system to exit; if the value is :DEBUG, the handler invokes the VAX LISP debugger.

For more information on warnings, see Section 3.2.3.

Format

WARN format-string &REST args

Arguments

format-string

The string of characters that is passed to the FORMAT function to create a warning message.

args

The arguments that are passed to the FORMAT function as arguments for the format string.

Return Value

NIL

Examples

```
Lisp> (DEFUN LOG-ERROR-STATUS (VMS-STATUS)
        (DECLARE (SPECIAL *ERROR-LOG*))
        (LET ((MESSAGE (GET-VMS-MESSAGE VMS-STATUS #*1111)))
        (IF MESSAGE
            (WRITE-LINE MESSAGE *ERROR-LOG*)
            (WARN
             "There is no message for VMS status #X~8,'0X."
                  VMS-STATUS)))
LOG-ERROR-STATUS
```

Defines a function that is an error logging facility. The function logs the VMS status that is returned from a call-out to a system service or an RTL routine. If the call-out facility returns an error status that has no corresponding message text, a warning message is displayed, and no log entry is produced.

ALIEN-FIELD

Accesses the value of a field of a specified type from an alien structure. The function ignores the alien structure's predefined fields.

You can modify alien structures if you use the ALIEN-FIELD function with the SETF macro. This function is most useful if you use it when you are debugging a program that uses alien structures.

For more information about alien structures, see Chapter 7.

Format

ALIEN-FIELD alien-structure type start end

Arguments

alien-structure

The name of the alien structure from which a field value is to be accessed.

type

The type of the field from which a value is to be accessed. This argument can be either a symbol that names an alien-structure field type or a list of which the first element defines the field type.

start

A rational number that specifies the start position (in bytes) of a field in the alien-structure's data area. There is no default.

end

A rational number that specifies the end position (in bytes) of a field in the alien-structure's data area. There is no default.

Return Value

The value of a field of the specified alien structure.

Example

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER Ø 4)
        (AREA-2 :UNSIGNED-INTEGER 4 8))
SPACE
Lisp> (ALIEN-FIELD (MAKE-SPACE) :UNSIGNED-INTEGER Ø 1)
Ø
```

- The call to the DEFINE-ALIEN-STRUCTURE macro defines an alien structure named SPACE.
- The call to the ALIEN-FIELD function returns the value of a byte-sized field in a structure of type SPACE.

ALIEN-STRUCTURE-LENGTH

Returns the length of an alien structure in bytes.

Format

ALIEN-STRUCTURE-LENGTH alien-structure

Argument

alien-structure

The name of the alien structure whose length is to be returned.

Return Value

The length of the alien structure.

Example

```
Lisp> (DEFINE-ALIEN-STRUCTURE MY-STRUCTURE (A :STRING Ø 9))
MY-STRUCTURE
Lisp> (ALIEN-STRUCTURE-LENGTH (MAKE-MY-STRUCTURE))
9
```

- The call to the DEFINE-ALIEN-STRUCTURE macro defines an alien structure named MY-STRUCTURE.
- The call to the ALIEN-STRUCTURE-LENGTH function returns the length of the alien structure.

ATTACH

Connects your terminal to a process and puts the current LISP process into a VMS hibernation state, a state in which a process is inactive but can become active at a later time. You can use this function to switch terminal control from one process to another.

The ATTACH function is similar to the DCL ATTACH command. For information about the ATTACH command, see the VAX/VMS Command Language User's Guide.

Format

```
ATTACH process
```

Argument

process

The name or identification of the process (PID) to which your terminal is to be connected. To specify the process name, use a string or a symbol; to specify the PID, use an integer.

Return Value

Undefined

Examples

1.
```
Lisp> (SPAWN)
$ ATTACH SMITH
Lisp> (ATTACH "SMITH_1")
%DCL-S-RETURNED, control returned to process SMITH_1
$
```

- The call to the SPAWN function creates a subprocess named SMITH_1.
- The DCL ATTACH command attaches your terminal back to the process SMITH.
- The call to the VAX LISP ATTACH function returns control to the process SMITH_1.

2.
```
Lisp> (DEFUN ATTACH-MAIN NIL
        (ATTACH (SECOND (GET-PROCESS-INFORMATION
                         NIL
                         :OWNER-PID))))
ATTACH-MAIN
```

Defines a function that attaches back to the main process if the LISP system is running as a subprocess.

BIND-KEYBOARD-FUNCTION

Binds an ASCII keyboard control character (characters of codes Ø to 32) to a function. When a control character is bound to a function, you can execute the function by typing the control character on your terminal keyboard. When you type the control character, the LISP system is interrupted at its current point, and the function the control character is bound to is called asynchronously. The LISP system then evaluates the function and returns control to where the interruption occurred.

You can delete the binding of a function and a control character by using the UNBIND-KEYBOARD-FUNCTION function. You can use the GET-KEYBOARD-FUNCTION function to get information about a function that is bound to a control character.

NOTE

> When you bind a control character to a function, the stream bound to the *TERMINAL-IO* variable must be connected to your terminal.

See Section 8.5 for an explanation about calling functions asynchronously.

Format

```
BIND-KEYBOARD-FUNCTION control-character function
     &KEY :ARGUMENTS list
```

Arguments

control-character

The ASCII control character to be bound to the function. You can bind a function to any control character except CTRL/Q or CTRL/S.

function

The function to which the control character is to be bound.

list

The list of arguments to be passed to the specified function when it is called. The arguments in the list are evaluated when the BIND-KEYBOARD-FUNCTION function is called.

Return Value

T

Examples

1. ```
   Lisp> (BIND-KEYBOARD-FUNCTION #\^B #'BREAK)
   T
   Lisp> CTRL/B
   Break 1>
   ```

   Binds CTRL/B to the BREAK function. You can then invoke a break loop by typing CTRL/B.
   ```

2.
```
Lisp> (BIND-KEYBOARD-FUNCTION #\^E #'ED)
T
Lisp> CTRL/E
 .
 .  (now in the Editor)
 .
```

Binds CTRL/E to the ED function. You can then invoke the Editor by typing CTRL/E.

3.
```
Lisp> (BIND-KEYBOARD-FUNCTION #\^G
                              #`THROW-TO-COMMAND-LEVEL
                              :ARGUMENTS
                              `(:CURRENT))
T
Lisp> (SPAWN CTRL/G
Lisp>
```

Binds CTRL/G to the THROW-TO-COMMAND-LEVEL function with the argument :CURRENT. By typing CTRL/G, you can abort a function call or a command in the debugger.

CALL-OUT

Calls a defined external routine. If you specify an external routine that has not been defined with the DEFINE-EXTERNAL-ROUTINE macro, the LISP system signals an error.

For information about how to use the VAX LISP call-out facility, see Chapter 6.

Format

```
CALL-OUT external-routine &OPTIONAL {actual-parameter}*
```

Arguments

external-routine

The name of a defined external routine.

actual-parameter

An actual parameter to be passed to the external routine. The parameter corresponds by position to a formal parameter defined for the routine. The LISP system evaluates the parameter expression before the external routine is called. You can omit a parameter by putting an explicit NIL in the parameter's position (you cannot use an expression that evaluates to NIL). The corresponding position in the parameter list will contain a zero to coincide with the VAX Procedure Calling Standard. If you specify fewer actual parameters than were specified in the formal definition, the argument count in the parameter list will contain only the number of actual arguments. The LISP system signals an error if you supply more arguments than were specified in the formal definition.

Return Value

The value returned by the external routine, NIL, or no value. The value is dependent upon the value you specify with the DEFINE-EXTERNAL-ROUTINE macro's :RESULT keyword.

Example

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
        (ERASE-PAGE :IMAGE-NAME "SCRSHR"
                    :ENTRY-POINT "LIB$ERASE_PAGE"
                    :CHECK-STATUS-RETURN T)
        (LINE   :LISP-TYPE INTEGER
                :VAX-TYPE :WORD)
        (COLUMN :LISP-TYPE INTEGER
                :VAX-TYPE :WORD))
ERASE-PAGE
Lisp> (CALL-OUT ERASE-PAGE 1 1)
```

- The call to the DEFINE-EXTERNAL-ROUTINE macro defines an RTL screen management routine, named ERASE-PAGE, which erases the terminal screen (see the description of the DEFINE-EXTERNAL-ROUTINE macro).
- The call to the CALL-OUT macro calls the RTL routine ERASE-PAGE, which causes the line and the column arguments to be treated as omitted arguments in the parameter list. Since the arguments are omitted, the RTL routine uses the default arguments and erases the terminal screen.

CHAR-NAME-TABLE

Displays a formatted list of the VAX LISP character names.

Format

CHAR-NAME-TABLE

Return Value

No value.

Example

```
Lisp> (CHAR-NAME-TABLE)

Hex Code  Preferred Name  Other Names

   00     NULL            NUL
   01     ^A              SOH
   02     ^B              STX
   03     ^C              ETX
   04     ^D              EOT
   05     ^E              ENQ
   06     ^F              ACK
   07     BELL            ^G  BEL
   08     BACKSPACE       ^H  BS
   09     TAB             ^I  HT
   0A     LINEFEED        ^J  LF
   0B     ^K              VT
   0C     PAGE            ^L  FORMFEED  FF
   0D     RETURN          ^M  CR
   0E     ^N              SO
   0F     ^O              SI
   10     ^P              DLE
   11     ^Q              XON  DC1
   12     ^R              DC2
   13     ^S              XOFF  DC3
   14     ^T              DC4
   15     ^U              NAK
   16     ^V              SYN
   17     ^W              ETB
   18     ^X              CAN
   19     ^Y              EM
   1A     ^Z              SUB
   1B     ESCAPE          ESC  ALTMODE
   1C     FS
   1D     GS
   1E     RS
   1F     US
   20     SPACE           SP
   7F     RUBOUT          DELETE  DEL
   84     IND
   85     NEL
   86     SSA
   87     ESA
   88     HTS
   89     HTJ
   8A     VTS
   8B     PLD
   8C     PLU
   8D     RI
   8E     SS2
   8F     SS3
   90     DCS
   91     PU1
   92     PU2
   93     STS
   94     CCH
   95     MW
   96     SPA
   97     EPA
   9B     CSI
   9C     ST
   9D     OSC
   9E     PM
   9F     APC
   FF     NEWLINE
```

COMPILEDP

A predicate that checks whether an object is a symbol that has a compiled function definition.

Format

COMPILEDP name

Argument

name

The symbol whose function call is to be checked.

Return Value

The interpreted function definition if the symbol has an interpreted function definition that was compiled with the COMPILE function. Returns T if the symbol has a compiled definition that was not compiled with the COMPILE function. Returns NIL if the symbol does not have a compiled function definition.

Example

```
Lisp> (DEFUN ADD2 (X) (+ X 2))
ADD2
Lisp> (COMPILEDP 'ADD2)
NIL
Lisp> (COMPILE 'ADD2)
ADD2 compiled.
ADD2
Lisp> (COMPILEDP 'ADD2)
(LAMBDA (X) (BLOCK ADD2 (+ X 2)))
```

- The call to the DEFUN macro defines a function named ADD2.
- The first call to the COMPILEDP function returns NIL because the function ADD2 has not been compiled.
- The call to the COMPILE function compiles the function ADD2.
- The second call to the COMPILEDP function returns the interpreted function definition because the function ADD2 was compiled previously.

COMPILE-VERBOSE

Controls the amount of information that the compiler displays.

The COMPILE-FILE function binds the value of the :VERBOSE keyword to the *COMPILE-VERBOSE* variable. If the :VERBOSE keyword is not specified, the function rebinds *COMPILE-VERBOSE* variable to its value. If the value is not NIL, the compiler displays the name of each function as it is compiled; if the value is NIL, the compiler does not display the function names. The default value is T.

Example

```
Lisp> (COMPILE-FILE 'MATH)
Starting compilation of file DBA1:[SMITH]MATH.LSP;1

FACTORIAL compiled.
FIBONACCI compiled.

Finished compilation of file DBA1:[SMITH]MATH.LSP;1
0 Errors, 0 Warnings
"DBA1:[SMITH]MATH.LSP;1"
Lisp> (SETF *COMPILE-VERBOSE* NIL)
NIL
Lisp> (COMPILE-FILE 'MATH)
"DBA1:[SMITH]MATH.LSP;1"
```

- The first call to the COMPILE-FILE function shows the output the compiler displays during the compilation of a file when the *COMPILE-VERBOSE* variable is set to T.
- The call to the SETF macro sets the value of the variable to NIL.
- The second call to the COMPILE-FILE function compiles the file without displaying output because the variable's value is NIL.

COMPILE-WARNINGS

Controls whether the compiler displays warning messages during a compilation.

The COMPILE-FILE function binds the value of the :WARNINGS keyword to the *COMPILE-WARNINGS* variable. If the :WARNINGS keyword is not specified, the function rebinds the *COMPILE-WARNINGS* variable to its value. If the value is not NIL, the compiler displays warning messages; if the value is NIL, the compiler does not display warning messages. The default value is T.

NOTE

The compiler always displays fatal and continuable error messages.

Example

```
Lisp> (COMPILE-FILE 'MATH)
Starting compilation of file DBA1:[SMITH]MATH.LSP;2

Warning in FACTORIAL
  N bound but not referenced.
FACTORIAL compiled.
Warning in FIBONACCI
  N bound but not referenced.
FIBONACCI compiled.

Finished compilation of file DBA1:[SMITH]MATH.LSP;2
0 Errors, 2 Warnings
"DBA1:[SMITH]MATH.LSP;2"
Lisp> (SETF *COMPILE-WARNINGS* NIL)
NIL
Lisp> (COMPILE-FILE 'MATH)
Starting compilation of file DBA1:[SMITH]MATH.LSP;2

FACTORIAL compiled.
FIBONACCI compiled.

Finished compilation of file DBA1:[SMITH]MATH.LSP;2
0 Errors, 2 Warnings
"DBA1:[SMITH]MATH.LSP;2"
```

- The first call to the COMPILE-FILE function shows the output the compiler displays during the compilation of a file when the *COMPILE-WARNINGS* variable is set to T.
- The call to the SETF macro sets the value of the variable to NIL.
- The second call to the COMPILE-FILE function compiles the file without displaying warning messages in the output because the variable's value is NIL.

CONTINUE

Enables you to exit the break loop. When you call this function, it causes the BREAK function to return NIL and the evaluation of your program to continue from the point where the break loop was entered.

Format

```
CONTINUE
```

Return Value

```
NIL
```

Example

```
Lisp> (BREAK)
  .
  .
  .
Break 1> (CONTINUE)
NIL
```

- The call to the BREAK function invokes the break loop.
- The call to the CONTINUE function exits the break loop and returns you to the top-level loop.

DEBUG

Invokes the VAX LISP debugger.

For information about how to use the VAX LISP debugger, see Section 4.4.

Format

DEBUG

Return Value

Returns NIL. You can cause the debugger to return other values (see Section 4.4.3).

Example

```
Lisp> (DEBUG)
Control Stack Debugger
Frame #5: (DEBUG)
Debug 1>
```

Invokes the VAX LISP debugger. When you invoke the debugger, it displays an identifying message, stack frame information, and the debugger prompt.

DEBUG-PRINT-LENGTH

Controls the output that the debugger, stepper, and tracer facilities display. This variable controls the number of objects these facilities can display at each level of a nested data object. The variable's value can be either a positive integer or NIL. If the value is a positive integer, the integer indicates the number of objects at each level of a nested object to be displayed. If the value is NIL, there is no limit on the number of objects that can be displayed. The default value is NIL.

The value of this variable might cause the printer to truncate output. An ellipsis (...) indicates truncation.

This variable is similar to the *PRINT-LENGTH* variable described in COMMON LISP: The Language.

Example

```
Lisp> (SETF ALPHABET '(A B C D E F G H I J K))
(A B C D E F G H I J K)
Lisp> (SETF *DEBUG-PRINT-LENGTH* 5)
5
Lisp> (+ 2 ALPHABET)

Fatal error in function + (signaled with ERROR).
Argument must be a number: (A B C D E F G H I J K)

Control Stack Debugger
Frame #5: (+ 2 (A B C D E ...))
Debug 1> (SETF *DEBUG-PRINT-LENGTH* 3)
3
Debug 1> WHERE
Frame #5: (+ 2 (A B C ...))
```

- The call to the SETF macro sets the symbol ALPHABET to a list of single-letter symbols.
- The evaluation of the *DEBUG-PRINT-LENGTH* variable shows the value of the variable is five.
- The call to the plus sign (+) function causes the LISP system to invoke the debugger. Note that the debugger only displays five elements of the list that is bound to the symbol ALPHABET the first time it displays stack frame numbered five.
- The call to the SETF macro within the debugger sets the value of the *DEBUG-PRINT-LENGTH* variable to three.
- The debugger displays three elements of the list after you change the value of the variable.

DEBUG-PRINT-LEVEL

Controls the output that the debugger, stepper, and tracer facilities display. This variable controls the number of levels of a nested object these facilities can display. The variable's value can be either a positive integer or NIL. If the value is a positive integer, the integer indicates the number of levels of a nested object to be displayed. If the value is NIL, there is no limit on the number of levels that can be displayed. The default value is NIL.

The value of this variable might cause the printer to truncate output. A number sign (#) indicates truncation.

This variable is similar to the *PRINT-LEVEL* variable described in COMMON LISP: The Language.

Example

```
Lisp> (SETF ALPHABET '(A (B (C (D (E))))))
(A (B (C (D (E)))))
Lisp> (SETF *DEBUG-PRINT-LEVEL* 3)
3
Lisp> (+ 2 ALPHABET)

Fatal error in function + (signaled with ERROR).
Argument must be a number: (A (B (C (D (E)))))

Control Stack Debugger
Frame #5: (+ 2 (A (B #)))
Debug 1> (SETF *DEBUG-PRINT-LEVEL* NIL)
NIL
Debug 1> WHERE
Frame #5: (+ 2 (A (B (C (D (E))))))
```

- The call to the SETF macro sets the symbol ALPHABET to a nested list.
- The evaluation of the *DEBUG-PRINT-LEVEL* variable shows the value of the variable is three.
- The call to the plus sign (+) function causes the LISP system to invoke the debugger. Note that the debugger only displays three levels of the nested list that is bound to the symbol ALPHABET the first time it displays stack frame numbered five.
- The call to the SETF macro within the debugger sets the value of the *DEBUG-PRINT-LEVEL* variable to the empty list.
- The debugger displays all the levels of the nested list after you change the value of the variable.

DEFAULT-DIRECTORY

Returns a pathname with the host, device, and directory fields filled with the values of the current default directory.

The DEFAULT-DIRECTORY function is similar to the DCL SHOW DEFAULT command. For information about the SHOW DEFAULT command, see the VAX/VMS Command Language User's Guide.

You can change the default directory by using the SETF macro. Setting your default directory with this macro also resets the value of the *DEFAULT-PATHNAME-DEFAULTS* variable. Performing this operation is similar to using the DCL SET DEFAULT command. See Section 8.2 and COMMON LISP: The Language for information about pathnames and the *DEFAULT-PATHNAME-DEFAULTS* variable.

Format

DEFAULT-DIRECTORY

Return Value

The pathname that refers to the default directory.

Examples

1.
```
Lisp> (DEFAULT-DIRECTORY)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH" :
NAME NIL :TYPE NIL :VERSION NIL)
Lisp> (SETF (DEFAULT-DIRECTORY) "[.TESTS]")
"[.TESTS]"
Lisp> (DEFAULT-DIRECTORY)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH.TE
STS" :NAME NIL :TYPE NIL :VERSION NIL)
```

- The first call to the DEFAULT-DIRECTORY function returns the pathname that points to the default directory.
- The call to the SETF macro changes the value of the default directory to SMITH.TESTS.
- The second call to the DEFAULT-DIRECTORY function verifies the directory change.

2.
```
Lisp> (DEFAULT-DIRECTORY)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH.TE
STS" :NAME NIL :TYPE NIL :VERSION NIL)
Lisp> *DEFAULT-PATHNAME-DEFAULTS*
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH.TE
STS" :NAME NIL :TYPE NIL :VERSION NIL)
Lisp> (NAMESTRING (DEFAULT-DIRECTORY))
"DBA1:[SMITH.TESTS]"
Lisp> (SETF (DEFAULT-DIRECTORY) "[-]")
"[-]"
Lisp> (NAMESTRING (DEFAULT-DIRECTORY))
"DBA1:[SMITH]"
Lisp> (NAMESTRING *DEFAULT-PATHNAME-DEFAULTS*)
"DBA1:[SMITH]"
```

- The first call to the DEFAULT-DIRECTORY function returns the pathname that points to the default directory.

DEFAULT-DIRECTORY Function

- The call to the *DEFAULT-PATHNAME-DEFAULTS* variable shows that its value is the same as the value returned by the DEFAULT-DIRECTORY function.
- The call to the NAMESTRING function returns the pathname as a namestring.
- The call to the SETF macro changes the value of the default directory to DBA1:[SMITH].
- The last two calls to the NAMESTRING function show that the return values of the DEFAULT-DIRECTORY function and the *DEFAULT-PATHNAME-DEFAULTS* variable are still the same.

DEFINE-ALIEN-FIELD-TYPE

Defines alien-structure field types.

For information about alien structures, see Chapter 7.

Format

```
DEFINE-ALIEN-FIELD-TYPE name internal-type primitive-type
     access-function setf-function
```

Arguments

name

The name of the alien-field type being defined.

internal-type

A LISP data type indicating the type of internal LISP object to which the field is to be mapped.

primitive-type

Either one of the predefined alien-field types or a type that was previously defined with the DEFINE-ALIEN-FIELD-TYPE macro. A LISP object defined by this argument is extracted from the alien structure's data when the field is accessed. The object is then passed to the specified access function. Predefined alien-field types are listed in Table 7-2.

access-function

The access function to which the LISP object defined by the primitive-type argument is passed. The function returns an object that is of the type defined by the internal-type argument.

setf-function

The set function with which the LISP object is to be passed. The function returns an object whose type is the type of the default SETF form as defined by the primitive-type argument. When the object is returned, it is packed into the alien structure's field data.

Return Value

The name of the alien-field type.

NOTE

Functions that access and set field values can take more than one argument; additional arguments are optional. When the type argument in the DEFINE-ALIEN-STRUCTURE macro's field description is a list, the first element of the list is the field type and the remaining elements are expressions the LISP system evaluates when it evaluates the access function. The resulting values are passed as additional arguments to the functions that access or set the field.

Examples

1.
```
Lisp> (DEFINE-ALIEN-FIELD-TYPE INTEGER-STRING-8
                                'INTEGER
                                :STRING
                                #'(LAMBDA
                                   (X)
                                   (PARSE-INTEGER X))
                                #'(LAMBDA
                                   (X)
                                   (FORMAT NIL "~8" X)))
INTEGER-STRING-8
Lisp> (DEFINE-ALIEN-STRUCTURE TWO-ASCII-INTEGERS
        (INT-1 INTEGER-STRING-8 Ø 8)
        (INT-2 INTEGER-STRING-8 8 16))
TWO-ASCII-INTEGERS
```

- The call to the DEFINE-ALIEN-FIELD-TYPE macro defines a field type named INTEGER-STRING-8. The field type INTEGER-STRING-8 causes an alien structure to convert strings to integers.
- The call to the DEFINE-ALIEN-STRUCTURE macro defines an alien structure named TWO-ASCII-INTEGERS that has two fields, each of type INTEGER-STRING-8.

2.
```
Lisp> (DEFINE-ALIEN-FIELD-TYPE SELECTION
                                T
                                :UNSIGNED-INTEGER
                                #'(LAMBDA
                                   (N &REST S-LIST)
                                   (NTH N S-LIST))
                                #'(LAMBDA
                                   (X &REST S-LIST)
                                   (POSITION X S-LIST)))
SELECTION
```

Defines an alien-field type named SELECTION. This type causes an alien structure to evaluate an unsigned integer either when the LISP system evaluates a field of this type or when the SETF macro is applied to a field to produce LISP objects.

DEFINE-ALIEN-STRUCTURE

Defines alien structures. An alien structure is a VAX-formatted memory structure.

The syntax of the DEFINE-ALIEN-STRUCTURE macro is similar to the DEFSTRUCT macro described in COMMON LISP: The Language.

For an explanation of how to define an alien structure, see Chapter 7.

Format

```
DEFINE-ALIEN-STRUCTURE name-and-options [doc-string]
     {field-description}*
```

Arguments

name-and-options

The name and the options of a new data type. The name argument must be a symbol. The options define the characteristics of the alien structure. Specify the options with keyword-value pairs. Specify a keyword-value pair as a list in the following format:

```
(keyword value)
```

If you do not specify options, you can specify the name-and-options argument as a symbol.

```
name
```

If you specify options, specify the name-and-options argument as a list whose first element is the name.

```
(name {(keyword value)}*)
```

Table 1 lists the keyword-value pairs that you can specify.

Table 1
DEFINE-ALIEN-STRUCTURE Options

Keyword-Value Pair	Description
:CONC-NAME name	Names the access functions. The value can be either a symbol or NIL. If you specify a symbol, the symbol becomes a prefix in the access function names. If you include a hyphen (-) in the symbol, specify it as part of the prefix. If you specify NIL, the access function names are the same as the field names. By default, the prefix is the alien-structure name followed by a hyphen.

(Continued on next page)

Table 1 (Cont.)
DEFINE-ALIEN-STRUCTURE Options

Keyword-Value Pair	Description
:CONSTRUCTOR name	Names the constructor function. The value can be either a symbol or NIL. If you specify a symbol, the symbol becomes the name of the constructor function. If you specify NIL, the macro does not define a constructor function. If you do not specify this keyword, the constructor function's name is the prefix MAKE- attached to the alien-structure name.
:COPIER name	Names the copier function. The value can be either a symbol or NIL. If you specify a symbol, the symbol becomes the name of the copier function. If you specify NIL, the macro does not create a copier function. If you do not specify this keyword, the copier function's name is the prefix COPY- attached to the alien-structure name.
:PREDICATE name	Names the predicate function. The value can be either a symbol or NIL. If you specify a symbol, the symbol becomes the name of the predicate function. If you specify NIL, the macro does not define a predicate function. If you do not specify this keyword, the macro names the predicate function by attaching the structure name to the characters -P.
:PRINT-FUNCTION function-name	Specifies the print function for the alien structure. The value must be a function. If you do not specify this keyword, the LISP system displays the alien structure in the following format: #<Alien Structure name number> In the preceding format, name is the name of the alien structure and number is a unique identification number, which distinguishes alien structures that have the same name.

DEBUG Function

Invokes the VAX LISP debugger.

For information about how to use the VAX LISP debugger, see Chapter 5.

Format

```
DEBUG
```

Return Value

Returns NIL. You can cause the debugger to return other values (see Chapter 5).

Example

```
Lisp> (DEBUG)
Control Stack Debugger
Apply #5: (DEBUG)
Debug 1>
```

Invokes the VAX LISP debugger. When you invoke the debugger, it displays an identifying message, stack frame information, and the debugger prompt.

DEBUG-CALL Function

Returns a list representing the current debug frame function call. This function is a debugging tool and takes no arguments. The list returned by the DEBUG-CALL function can be used to access the values passed to the function in the current stack frame.

Format

DEBUG-CALL

Return Value

A list representing the current debug frame function call. NIL is returned if this function is called outside the debugger.

Example

```
Lisp> (DEFVAR ADJUSTABLE-STRING
              (MAKE-ARRAY 10 :ELEMENT-TYPE 'STRING-CHAR
                             :INITIAL-ELEMENT #\SPACE
                             :ADJUSTABLE T))
ADJUSTABLE-STRING
Lisp> (SCHAR ADJUSTABLE-STRING 3)

Fatal error in function SCHAR (signaled with ERROR).
Argument must be a simple-string: "          "

Control Stack Debugger
Apply #4: (SCHAR "          " 3)
Debug 1> (TYPE-OF (SECOND (DEBUG-CALL)))

(STRING 10)
Debug 1> RET #\SPACE
#\SPACE
```

In this case, the function in the current stack frame is SCHAR. The call to (DEBUG-CALL) returns the list (SCHAR " " 3). The form (SECOND (DEBUG-CALL)) returns the first argument to SCHAR in the current stack frame. Calling TYPE-OF with this LISP object determines that the first argument to SCHAR is of type (STRING 10) and not a simple string. See the TRACE macro description for another example of the use of the DEBUG-CALL function.

DEBUG-PRINT-LENGTH Variable

Controls the output that the debugger, stepper, and tracer facilities display. This variable controls the number of objects these facilities can display at each level of a nested data object. The variable's value can be either a positive integer or NIL. If the value is a positive integer, the integer indicates the number of objects at each level of a nested object to be displayed. If the value is NIL, no limit is on the number of objects that can be displayed. The default value is NIL.

The value of this variable might cause the printer to truncate output. An ellipsis (...) indicates truncation.

This variable is similar to the *PRINT-LENGTH* variable described in *COMMON LISP: The Language*.

Example

```
Lisp> (SETF ALPHABET '(A B C D E F G H I J K))
(A B C D E F G H I J K)
Lisp> (SETF *DEBUG-PRINT-LENGTH* 5)
5
Lisp> (+ 2 ALPHABET)

Fatal error in function + (signaled with ERROR).
Argument must be a number: (A B C D E F G H I J K)

Control Stack Debugger
Apply #5: (+ 2 (A B C D E ...))
Debug 1> (SETF *DEBUG-PRINT-LENGTH* 3)
3
Debug 1> WHERE
Apply #5: (+ 2 (A B C ...))
```

- The call to the SETF macro sets the symbol ALPHABET to a list of single-letter symbols.
- The value of the *DEBUG-PRINT-LENGTH* variable is set to 5.
- The illegal call to the plus sign (+) function causes the LISP system to invoke the debugger. The debugger displays only five elements of the list that is the value of the symbol ALPHABET the first time it displays the stack frame numbered 5.
- The call to the SETF macro within the debugger sets the value of the *DEBUG-PRINT-LENGTH* variable to 3.
- The debugger displays three elements of the list, after you change the value of the variable.

DEBUG-PRINT-LEVEL Variable

Controls the output that the debugger, stepper, and tracer facilities display. This variable controls the number of levels of a nested object these facilities can display. The variable's value can be either a positive integer or NIL. If the value is a positive integer, the integer indicates the number of levels of a nested object to be displayed. If the value is NIL, no limit is on the number of levels that can be displayed. The default value is NIL.

The value of this variable might cause the printer to truncate output. A number sign (#) indicates truncation.

This variable is similar to the *PRINT-LEVEL* variable described in *COMMON LISP: The Language*.

Example

```
Lisp> (SETF ALPHABET '(A (B (C (D (E))))))
(A (B (C (D (E)))))
Lisp> (SETF *DEBUG-PRINT-LEVEL* 3)
3
Lisp> (+ 2 ALPHABET)

Fatal error in function + (signaled with ERROR).
Argument must be a number: (A (B (C (D (E)))))

Control Stack Debugger
Apply #5: (+ 2 (A (B #)))
Debug 1> (SETF *DEBUG-PRINT-LEVEL* NIL)
NIL
Debug 1> WHERE
Apply #5: (+ 2 (A (B (C (D (E))))))
```

- The call to the SETF macro sets the symbol ALPHABET to a nested list.
- The value of the *DEBUG-PRINT-LEVEL* variable is set to 3.
- The illegal call to the plus sign (+) function causes the LISP system to invoke the debugger. The debugger displays only three levels of the nested list (that is the value of the symbol ALPHABET) the first time it displays the stack frame numbered 5.
- The call to the SETF macro within the debugger sets the value of the *DEBUG-PRINT-LEVEL* variable to NIL.
- The debugger displays all the levels of the nested list, after you change the value of the variable.

Table 3 (Cont.)
DEFINE-EXTERNAL-ROUTINE Options

Keyword-Value Pair	Description
	specify NIL, the call-out facility does not check the severity of the return value. NIL is the default value.
:ENTRY-POINT string	Names the external routine's entry point. The value must be a string. The macro converts the name to uppercase characters. The default value is the print name of the external-routine name.
:IMAGE-NAME pathname	Specifies the shareable image that was created for the external routine. The file specification is merged with the file SYS$SHARE:.EXE.
:RESULT type	Specifies the type of LISP object the external routine is to return. The value can be a LISP type, a type-spec-list, or NIL. A type-spec-list has the following format: :RESULT (:LISP-TYPE LISP-type :VAX-TYPE VAX-type) NIL specifies that the routine returns no value. The default value is NIL.
:TYPE-CHECK value	Specifies whether the call-out facility is to check the types of the actual parameters passed to the external routine and the LISP types specified in the formal parameter specification for compatibility. The value can be either T or NIL. If you specify T, the facility checks the types for compatibility; if you specify NIL, the facility does not check the parameter types. The default value is NIL.

doc-string

The documentation string that is to be attached to the symbol that names the external routine. The documentation string is of type EXTERNAL-ROUTINE. See COMMON LISP: The Language for information on the DOCUMENTATION function.

formal-parameter-description

A parameter description that is to be passed to the external routine. Specify the descriptions in the following format:

(name options)

The name argument must be a unique symbol within the definition or NIL. This argument is the formal name of a parameter. The symbol is used by functions that access and set external-routine parameters.

The options define the characteristics of a formal parameter. Specify the options with keyword-value pairs.

keyword value

If you do not specify options, you can specify the formal-parameter-description argument as a symbol.

name

If you specify options, specify the argument as a list whose first element is the name.

(name {keyword value}*)

The option values are not evaluated.

Table 4 lists the keyword-value pairs you can specify.

Table 4
DEFINE-EXTERNAL-ROUTINE Formal Parameter Options

Keyword-Value Pair	Description
:ACCESS value	Specifies the type of access the external routine needs for the actual parameter. The value can be either :IN or :IN-OUT. The default value is :IN. If you specify :IN, the parameter has input access. If you specify :IN-OUT, the parameter has input-output access.
:LISP-TYPE type	Specifies the LISP type of the parameter value the call-out facility is to pass to the external routine. The values you can specify are the types: number, simple string, simple bit vector, simple floating-point array, or alien structure. The default value is INTEGER.

(Continued on next page)

Table 4 (Cont.)
DEFINE-EXTERNAL-ROUTINE Formal Parameter Options

Keyword-Value Pair	Description
:MECHANISM value	Specifies the parameter-passing mechanism the external routine is to expect for the actual parameter. The values you can specify are :IMMED, :REF, and :DESCR. The default value is :DESCR for LISP string-type parameters and is :REF for other LISP data types.
:VAX-TYPE type	Specifies the VAX data type of the parameter value the external routine is to return. The values you can specify are :BIT, :BIT-STRING, :BYTE, :UNSIGNED-BYTE, :WORD, :UNSIGNED-WORD, :LONGWORD, :UNSIGNED-LONGWORD, :F-FLOATING, :D-FLOATING, :G-FLOATING, :H-FLOATING, :TEXT, and :ALIEN-STRUCTURE. The default value is :LONGWORD.

Return Value

The symbol that names the external routine.

Example

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
        (ERASE-PAGE :IMAGE-NAME "SCRSHR"
                    :ENTRY-POINT "LIB$ERASE_PAGE"
                    :CHECK-STATUS-RETURN T)
        (LINE   :LISP-TYPE INTEGER
                :VAX-TYPE :WORD)
        (COLUMN :LISP-TYPE INTEGER
                :VAX-TYPE :WORD))
ERASE-PAGE
```

Defines an RTL screen management routine, called ERASE-PAGE, which erases the terminal screen. The image name for the screen package is SCRSHR and not VMSRTL. The RTL status is not to be returned from the function, but the status is to be checked internally by the call-out facility. A VAX data type is specified for each argument because the default type -- :LONGWORD -- is not the type required by the RTL routine.

More examples of how to define external routines are provided in Section 6.5. The examples in Section 6.5 also show you how to call out to defined external routines.

ERROR-ACTION

Determines the action the VAX LISP error handler is to take when an error occurs. The value of this variable can be the :EXIT or the :DEBUG keyword. If the value is :EXIT, the error handler causes the LISP system to exit; if the value is :DEBUG, the handler invokes the VAX LISP debugger. The default value is :DEBUG for interactive LISP sessions; the default value is :EXIT otherwise.

Example

```
Lisp> (CAR 'A)

Fatal error in function CAR (signaled with ERROR).
Argument must be a list: A.

Control Stack Debugger
Frame #5: (CAR A)
Debug 1> QUIT
Lisp> (SETF *ERROR-ACTION* :EXIT)
:EXIT
Lisp> (CAR 'A)

Fatal error in function CAR (signaled with ERROR).
Argument must be a list: A.

$
```

- When the first error occurs the LISP system invokes the VAX LISP debugger because the value of the *ERROR-ACTION* variable is :DEBUG (the default).
- The call to the SETF macro sets the value of the variable to :EXIT.
- The second time the error occurs the LISP system exits and control returns to the VMS command level.

EXIT

Invokes the VMS Exit system service, causing the LISP system to exit and to return control to the VMS command level.

You can pass the status of the LISP system to the VMS command level when you exit the LISP system by specifying an optional argument. When the LISP system exits, the argument's value is passed to the VMS command level.

Format

EXIT &OPTIONAL status

Argument

status

A fixnum or a keyword that indicates the status of the LISP system that is to be returned to the VMS command level when the LISP system exits. The keywords you can specify and the types of status they return are the following:

:ERROR	Error status
:SUCCESS	Success status
:WARNING	Warning status

Return Value

No value.

Examples

1.
```
Lisp> (EXIT)
$
```

Exits the LISP system.

2.
```
Lisp> (EXIT :ERROR)

$ SHOW SYMBOL $STATUS
  $STATUS = "%X112D8012"
```

Exits the LISP system. When control returns to the VMS command level, the VAX LISP exit status contains an error status.

EXPAND-PPRINT-TEMPLATE

Translates a template and an object that is to be pretty-printed into pretty-printer code. A call to this macro has the side effect of creating queue entries that cause the pretty printer to produce output.

NOTE

You cannot call this macro at top level.

You must include calls to this macro in user-defined extensions to the pretty printer. For an explanation on to how extend the pretty printer, see Section 5.3.

Format

```
EXPAND-PPRINT-TEMPLATE template &REST {subobject}*
```

Arguments

template

A string of directives. You can specify an alphabetic directive as either a lower- or uppercase character. Table 5 lists the directives that you can use to define templates. In addition to listing the directives, the table provides the following information for each directive:

- The type of parameter that can follow the directive: literal text, integer, function name, or subtemplate. When you specify a parameter, do not include spaces between the directive and the parameter. If you specify a function name parameter, the name must be followed by a whitespace character or the end of the string. You can use a number sign (#) parameter instead of an integer or function name parameter to refer to a subobject in the template's argument list (the second argument).
- Whether the directive operates on a subobject in the template's argument list.
- A brief description of the directive.

This argument is the same as the template argument for the FORMAT-USING-PPRINT-TEMPLATE function.

NOTE

When the pretty printer inserts a line break, it prints the #\NEWLINE character, if necessary, and then prints the current indentation. Consecutive uses of a line-break directive result in the pretty printer printing one #\NEWLINE character.

subobject

A subobject that a directive in the template argument refers to. The list of subobjects is the template's argument list. This argument is the same as the subobject argument for the FORMAT-USING-PPRINT-TEMPLATE function.

Return Value

A value other than NIL. This macro is usually called for side effects.

Table 5
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
' '	Literal text	No	Apostrophes. The pretty printer prints the literal text that a pair of apostrophes enclose. To include an apostrophe in literal text, quote the apostrophe with an apostrophe.
*	None	Yes	Asterisk. An asterisk signals a recursive call to the pretty-printer dispatch routine. The routine passes the corresponding subobject to the pretty printer to be formatted. The format the pretty printer uses for the subobject is dependent upon the subobject's data type and not upon succeeding directives.
P	None	Yes	PRIN1. The pretty printer formats the corresponding subobject such that its output is similar to the output the PRIN1 function produces.
C	None	Yes	PRINC. The pretty printer formats the corresponding subobject such that its output is similar to the output the PRINC function produces.
S	None	Yes	Special. The pretty printer formats the corresponding subobject such that its output is similar to the output the PRINC function produces. The value of the *PRINT-LEVEL* variable is ignored when the subobject is printed.

(Continued on next page)

Table 5 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
I	None	Yes	Ignore. The pretty printer evaluates the corresponding subobject but produces no output.
~	Integer	No	Tilde. The pretty printer inserts a space. The integer parameter is optional and it can have a positive or negative value. If you specify a positive integer, it indicates the number of spaces to be inserted. If you specify a negative integer, it indicates the number of spaces to be deleted. The pretty printer deletes only spaces. The default value is one.
T	Integer	No	Tab. The pretty printer inserts spaces to make the output have a tabular appearance. The integer parameter (n) is optional. If you specify the parameter, the pretty printer starts printing at the current indentation level plus x spaces, where n evenly divides x. The default value depends on the entries in the template.
+	Integer	No	Plus sign. The pretty printer changes the current indentation level. The integer parameter is optional and can have a positive or negative value. If you specify the parameter, the pretty printer changes the indentation level the specified number of spaces. The default value is one.
!	None	No	Exclamation point. The pretty printer inserts a line break.

(Continued on next page)

Table 5 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
N	None	No	Conditional line break. The pretty printer inserts a line break if the directive is not specified within the brace or parentheses directive. If it is specified within one of these directives, the pretty printer inserts a line break if it cannot print the subobject the brace or parentheses directive refers to on one line.
B	None	No	Conditional line break. The pretty printer inserts a line break if the next subobject cannot be printed on the current line.
M	None	No	Conditional line break. The pretty printer inserts a line break if the remainder of the object cannot be printed on one line or if the remaining width available for printing is less than the value of the *PPRINT-MISER-WIDTH* variable.
-	Integer	No	Hyphen. The pretty printer inserts spaces. If the entire object cannot be printed on one line, the pretty printer inserts a line break. The integer parameter is optional; -n is an abbreviation for ~nN.
,	Integer	No	Comma. The pretty printer inserts spaces. If the next subobject cannot be printed on the current line, the pretty printer inserts a line break. The integer parameter is optional; ,n is an abbreviation for ~nB.
;	Integer	No	Semicolon. The pretty printer inserts one space and if necessary, inserts additional spaces to make the output have a tabular appearance. If the next subobject cannot be printed on the current line, the pretty printer inserts a line break. The integer parameter is optional; ;n is an abbreviation for ~1TnB.

(Continued on next page)

Table 5 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
_	Integer	No	Underscore. The pretty printer inserts spaces. If the remainder of the object cannot be printed on one line or if the remaining width available for printing is less than the value of the *PPRINT-MISER-WIDTH* variable, the pretty printer inserts a line break. The integer parameter is optional; _n is an abbreviation for ~nM.
{ }	Integer Subtemplate	No	Braces. This directive encloses a subtemplate that formats a logical unit of the object being pretty-printed. The directive is useful for causing the indentation level to remain at an appropriate level by default. The integer parameter is optional. If you specify the integer parameter after the open brace, the pretty printer increments the indentation level by the specified number of spaces. The default value of the integer parameter is the position of the cursor after printing the object's first subobject.
[ ]	Subtemplate	Yes	Square brackets. This directive encloses a subtemplate. The subtemplate refers to one subobject in the template's argument list, which must be a list. The subtemplate formats the elements of the list. The pretty printer stops formatting the list after the last element of the list or the last directive in the subtemplate is used. It stops even if the list has more elements or the subtemplate contains more directives.

(Continued on next page)

Table 5 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
( )	Integer Subtemplate	Yes	Parentheses. This directive formats one subobject in the template's argument list. The subobject must be a list. The integer parameter is optional. The default value of the integer parameter is the position of the cursor after printing the first element of the list. The directive (n subtemplate) is an abbreviation for the following: {n '('[subtemplate]')'}
.	None	None	Period. This directive can be used only inside the square brackets or parentheses directive. The period directive causes the next directive that operates on a subobject to use a list that contains the remaining elements of the subobject.
< >	Subtemplate	No	Angle brackets. This directive can be used only inside the square brackets or parentheses directive. The pretty printer uses the suptemplate parameter repeatedly until the last element of the list is used.
&	Function name	No	Ampersand. The pretty printer calls the specified formatting function with no arguments.
%	Function name	Yes	Percent sign. The pretty printer calls the specified formatting function with one argument, the subobject to be formatted.
$	Function name or \"Subtemplate\"	Yes	Dollar sign. If the subobject is a list, the pretty printer calls the specified function or subtemplate with one argument, the subobject to be formatted. If the subobject is not a list, it is passed to the formatting routine, which formats it according to its data type. If you specify a subtemplate, the subtemplate must be enclosed in double quotes (\"subtemplate\").

Example

```
Lisp> (DEFUN SETQ-FORMATTER (OBJECT)
        (EXPAND-PPRINT-TEMPLATE "(* <*_*!>)" OBJECT))
SETQ-FORMATTER
```

Defines a formatting function for lists whose first element is the symbol SETQ. The parentheses directive causes the pretty printer to print parentheses around the list. An indentation increment is not specified, so the pretty printer will use the default indentation to line up the subobjects one under the other. The angle bracket directive specifies that the objects in the list element it refers to are to be formatted in pairs. The exclamation point directive will force a line break after the pretty printer prints each pair so it prints each pair on a separate line -- even when the entire SETQ expression fits on one line.

More examples of defining formatting functions are provided in Section 5.3.5.

DIRECTORY Function

Converts its argument to a pathname and returns a list of the pathnames for the files matching the specification. The DIRECTORY function is similar to the DCL DIRECTORY command.

Format

DIRECTORY *pathname*

Argument

pathname

The pathname, namestring, stream, or symbol for which the list of file system pathnames is to be returned. In VAX LISP/VMS, this argument is merged with the following default file specification:

host::*device*:[*directory*]*.*;*

The *host*, *device*, and *directory* values are supplied by the *DEFAULT-PATHNAME-DEFAULTS* variable.

Specifying just a directory is equivalent to specifying a directory with wild cards (*) in the name, type, and version fields of the argument. For example, the following two expressions are equivalent:

```
(DIRECTORY "[MYDIRECTORY]")

(DIRECTORY "[MYDIRECTORY]*.*;*")
```

Both expressions return a list of pathnames that represent the files in the directory MYDIRECTORY.

Specifying just a directory with a specified version field is equivalent to specifying a directory and version with wild cards (*) in the name and type fields of the argument. For example, the following two expressions are equivalent:

```
(DIRECTORY "[MYDIRECTORY];0")

(DIRECTORY "[MYDIRECTORY]*.*;")
```

Both expressions return a list of the pathnames that represent the newest versions of the files in the directory MYDIRECTORY.

The following equivalent expressions return the list of pathnames for files in your default directory:

DIRECTORY Function (cont.)

```
(DIRECTORY "")

(DIRECTORY (DEFAULT-DIRECTORY))
```

Return Value

A list of pathnames, if the specified pathname is matched, and NIL, if the pathname is not matched.

Example

```
Lisp> (DEFUN MY-DIRECTORY (&OPTIONAL (FILENAME ""))
        (LET ((PATHNAME (PATHNAME FILENAME))
              (DIRECTORY (DIRECTORY FILENAME)))
          (COND ((NULL DIRECTORY)
                 (FORMAT T
                         "~%No files match ~A.~%"
                         (NAMESTRING FILENAME)))
                (T (FORMAT T
                        "~%The following ~:[files are~;file is ~]
                        in the directory ~A:[~A]:"
                           (EQUAL (LENGTH DIRECTORY) 1)
                           (PATHNAME-DEVICE
                            (NTH 0 DIRECTORY))
                           (PATHNAME-DIRECTORY
                            (NTH 0 DIRECTORY)))
                   (DOLIST (X DIRECTORY)
                     (FORMAT T "~&~2T~A" (FILE-NAMESTRING X)))
                   (TERPRI)))
          (VALUES)))
MY-DIRECTORY
Lisp> (MY-DIRECTORY)
The following files are in the directory DBA1:[SMITH.TESTS]:
  TEST5.DRB;1
  TEST1.LSP;7
  TEST1.LSP;6
  TEST1.LSP;5
  EXAMPLE.TXT;2
  TEST3.LSP;15
  TEST6.LSP;1
Lisp> (MY-DIRECTORY ".LSP;")
The following files are in the directory DBA1:[SMITH.TESTS]:
  TEST1.LSP;7
  TEST3.LSP;15
  TEST6.LSP;1
```

- The call to the DEFUN macro defines a function that formats the output of the DIRECTORY function, making the output more readable. The function is defined such that it accepts an optional argument and does not return a value.

Table 6 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
~	Integer	No	Tilde. The pretty printer inserts a space. The integer parameter is optional and it can have a positive or negative value. If you specify a positive integer, it indicates the number of spaces to be inserted. If you specify a negative integer, it indicates the number of spaces to be deleted. The pretty printer deletes only spaces. The default value is one.
T	Integer	No	Tab. The pretty printer inserts spaces to make the output have a tabular appearance. The integer parameter (n) is optional. If you specify the parameter, the pretty printer starts printing at the current indentation level plus x spaces, where n evenly divides x. The default value depends on the entries in the template.
+	Integer	No	Plus sign. The pretty printer changes the current indentation level. The integer parameter is optional and can have a positive or negative value. If you specify the parameter, the pretty printer changes the indentation level the specified number of spaces. The default value is one.
!	None	No	Exclamation point. The pretty printer inserts a line break.
N	None	No	Conditional line break. The pretty printer inserts a line break if the directive is not specified within the brace or parentheses directive. If it is specified within one of these directives, the pretty printer inserts a line break if it cannot print the subobject the brace or parentheses directive refers to on one line.

(Continued on next page)

Table 6 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
B	None	No	Conditional line break. The pretty printer inserts a line break if the next subobject cannot be printed on the current line.
M	None	No	Conditional line break. The pretty printer inserts a line break if the remainder of the object cannot be printed on one line or if the remaining width available for printing is less than the value of the *PPRINT-MISER-WIDTH* variable.
-	Integer	No	Hyphen. The pretty printer inserts spaces. If the object cannot be printed on one line, the pretty printer inserts a line break. The integer parameter is optional; -n is an abbreviation for ~nN.
,	Integer	No	Comma. The pretty printer inserts spaces. If the next subobject cannot be printed on the current line, the pretty printer inserts a line break. The integer parameter is optional; ,n is an abbreviation for ~nB.
;	Integer	No	Semicolon. The pretty printer inserts one space and if necessary, inserts additional spaces to make the output have a tabular appearance. If the next subobject cannot be printed on the current line, the pretty printer inserts a line break. The integer parameter is optional; ;n is an abbreviation for ~1TnB.
_	Integer	No	Underscore. The pretty printer inserts spaces. If the remainder of the object cannot be printed on one line or if the remaining width available for printing is less than the value of the *PPRINT-MISER-WIDTH* variable, the pretty printer inserts a line break. The integer parameter is optional; _n is an abbreviation for ~nM.

(Continued on next page)

Table 6 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
{ }	Integer Subtemplate	No	Braces. This directive encloses a subtemplate that formats a logical unit of the object being pretty-printed. The directive is useful for causing the indentation level to remain at an appropriate level by default. The integer parameter is optional. If you specify the integer parameter after the open brace, the pretty printer increments the indentation level by the specified number of spaces. The default value of the integer parameter is the position of the cursor after printing the object's first subobject.
[ ]	Subtemplate	Yes	Square brackets. This directive encloses a subtemplate. The subtemplate refers to one subobject in the template's argument list, which must be a list. The subtemplate formats the elements of the list. The pretty printer stops formatting the list after the last element of the list or the last directive in the subtemplate is used. It stops even if the list has more elements or the subtemplate contains more directives.
( )	Integer Subtemplate	Yes	Parentheses. This directive formats one subobject in the template's argument list. The subobject must be a list. The integer parameter is optional. The default value of the integer parameter is the position of the cursor after printing the first element of the list. The directive (n subtemplate) is an abbreviation for the following: {n '('[subtemplate]')'}

(Continued on next page)

Table 6 (Cont.)
Pretty-Printer Directives

Directive	Parameter	Subobject	Description
.	None	None	Period. This directive can be used only inside the square brackets or parentheses directive. The period directive causes the next directive that operates on a subobject to use a list that contains the remaining elements of the subobject.
< >	Subtemplate	No	Angle brackets. This directive can be used only inside the square brackets or parentheses directive. The pretty printer uses the suptemplate parameter repeatedly until the last element of the list is used.
&	Function name	No	Ampersand. The pretty printer calls the specified formatting function with no arguments.
%	Function name	Yes	Percent sign. The pretty printer calls the specified formatting function with one argument, the subobject to be formatted.
$	Function name or \"Subtemplate\"	Yes	Dollar sign. If the subobject is a list, the pretty printer calls the specified function or subtemplate with one argument, the subobject to be formatted. If the subobject is not a list, it is passed to the formatting routine, which formats it according to its data type. If you specify a subtemplate, the subtemplate must be enclosed in double quotes (\"subtemplate\").

Example

```
Lisp> (SETF WEATHER '(SUN CLOUDS RAIN SNOW))
(SUN CLOUDS RAIN SNOW)
Lisp> (FORMAT-USING-PPRINT-TEMPLATE T "(1 * ! * ! * ! *)" WEATHER)
(SUN
 CLOUDS
 RAIN
 SNOW)
NIL
```

- The call to the SETF macro sets the value of the symbol WEATHER to a list.
- The call to the FORMAT-USING-PPRINT-TEMPLATE function causes the pretty printer to use the template "(1 * ! * ! * ! *)" to format the list.

ENLARGE-BINDING-STACK Function

Enlarges the VAX LISP binding stack by the specified number of pages. Use this function if the default size of the binding stack is too small to accommodate a large or complex program.

If the binding stack overflows in the course of program execution, a continuable error is signaled. Continuing from this error enlarges the binding stack and allows program execution to continue.

Enlarging the binding stack -- either by use of ENLARGE-BINDING-STACK or by continuing from a binding stack overflow -- causes a garbage collection.

Format

ENLARGE-BINDING-STACK *number-of-pages*

Argument

number-of-pages

The number of 512-byte pages by which to enlarge the binding stack.

Return Value

Undefined.

ENLARGE-CONTROL-STACK Function

Enlarges the VAX LISP control stack by the specified number of pages. Use this function if the default size of the control stack is too small to accommodate a large or complex program.

If the control stack overflows in the course of program execution, a continuable error is signaled. Continuing from this error enlarges the control stack and allows program execution to continue.

Enlarging the control stack -- either by use of ENLARGE-CONTROL-STACK or by continuing from a control stack overflow -- causes a garbage collection.

Format

ENLARGE-CONTROL-STACK *number-of-pages*

Argument

number-of-pages

The number of 512-byte pages by which to enlarge the control stack.

Return Value

Undefined.

GC

Invokes the garbage collector. The LISP system initiates garbage collection during normal system use whenever necessary. You cannot disable this process. However, the GC function enables you to initiate garbage collection during system interaction.

NOTE

> The LISP system does not use the GC function to initiate garbage collections. Therefore, redefining the GC function does not prevent garbage collection.

You might want to use the GC function to invoke the garbage collector just before a time-critical part of a LISP program. Using the GC function this way reduces the possibility of the LISP system initiating a garbage collection when a critical part of the program is executing.

See Section 8.3 for a description of the garbage collector.

Format

```
GC
```

Return Value

T when garbage collection is completed.

Example

```
Lisp> (GC)
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
```

Invokes the garbage collector. Whether the messages are printed when a garbage collection occurs depends on the value of the *GC-VERBOSE* variable.

GC-VERBOSE

A variable whose value is used as a flag to determine whether the LISP system is to display messages when a garbage collection occurs. If the flag is NIL, the system displays messages. If the flag is not NIL, the system displays a message before and after a garbage collection occurs. The default value is T.

The messages the LISP system displays are controlled by the VAX LISP *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables.

For more information on garbage collector messages, see Section 8.3.4.

Example

```
Lisp> *GC-VERBOSE*
T
Lisp> (GC)
; Stating garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
Lisp> (SETF *GC-VERBOSE* NIL)
NIL
Lisp> (GC)
T
```

- The first evaluation of the *GC-VERBOSE* variable returns the default value T, which indicates that the LISP system will display a message before and after a garbage collection occurs (depending on the values of the *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables).

- The call to the GC function shows the default messages the system displays when a garbage collection occurs and the variable's value is T.

- The call to the SETF macro sets the value of the variable to NIL.

- The second call to the GC function shows that the system does not display messages when the variable's value is NIL.

GET-DEVICE-INFORMATION

Returns information about a device. The keywords you specify with the function determine the type of information the function returns.

This function is similar to the $GETDVI VMS system service. For more information on the $GETDVI system service, see the VAX/VMS System Services Reference Manual and the VAX/VMS I/O User's Guide (Volume 1).

Format

GET-DEVICE-INFORMATION device &REST {keyword}*

Arguments

device

The string that names the device about which information is to be returned.

keyword

Optional keywords that specify types of information about the specified device. Do not specify values with the keywords.

Table 7 lists the keywords that you can specify and the values they return.

Table 7
GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
:ACP-PID	An integer that specifies the ACP process ID.
:ACP-TYPE	An integer that specifies the ACP type code.
:BUFFER-SIZE	An integer that specifies the buffer size.
:CLUSTER-SIZE	An integer that specifies the volume cluster size.
:CYLINDERS	An integer that specifies the number of cylinders on the device.
:DEVICE-CHARACTERISTICS	A vector of 32 bits that specifies the device characteristics. See the VAX/VMS I/O User's Guide for information about device characteristics.
:DEVICE-CLASS	An integer that specifies the device class.
:DEVICE-DEPENDENT-Ø	A bit vector that specifies device-dependent information.

(Continued on next page)

Table 7 (Cont.)
GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
:DEVICE-DEPENDENT-1	A bit vector that specifies device-dependent information.
:DEVICE-NAME	A string that specifies the device name.
:DEVICE-TYPE	An integer that specifies the device type.
:ERROR-COUNT	An integer that specifies the number of errors that have occurred on the device.
:FREE-BLOCKS	An integer that specifies .the number of free blocks on the device; otherwise, NIL.
:LOGICAL-VOLUME-NAME	A string that specifies the logical name associated with the volume on the device. This keyword is valid only for disks.
:MAX-BLOCKS	An integer that specifies the maximum number of logical blocks that can exist on the device.
:MAX-FILES	An integer that specifies the maximum number of files that can exist on the device.
:MOUNT-COUNT	An integer that specifies the number of times the device has been mounted.
:NEXT-DEVICE-NAME	A string that specifies the name of the next volume in the volume set.
:OPERATION-COUNT	An integer that specifies the number of operations that have been performed on the device.
:OWNER-UIC	An integer that specifies the UIC of the owner.
:PID	An integer that specifies the process ID of the owner.
:RECORD-SIZE	An integer that specifies the blocked record size.
:REFERENCE-COUNT	An integer that specifies the number of channels assigned to the device.

(Continued on next page)

Table 7 (Cont.)
GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
:ROOT-DEVICE-NAME	A string that specifies the name of the root volume in the volume set.
:SECTORS	An integer that specifies the number of sectors per track.
:SERIAL-NUMBER	An integer that specifies the serial number.
:TRACKS	An integer that specifies the number of tracks per cylinder.
:TRANSACTION-COUNT	An integer that specifies the number of files open on the device.
:UNIT	An integer that specifies the unit number.
:VOLUME-COUNT	An integer that specifies the number of volumes in the volume set.
:VOLUME-NAME	A string that specifies the name of the volume on the device.
:VOLUME-NUMBER	An integer that specifies the number of the volume on the device.
:VOLUME-PROTECTION	A vector of 32 bits that specifies the volume protection mask.

Return Value

The keywords and their values are returned as a list in the following format:

```
(:keyword-1 value-1 :keyword-2 value-2 ...)
```

The function preserves the order of the keyword-value pairs in the argument list.

If you do not specify keywords, the function returns a list of all the keyword-value pairs. If the device does not exist, the function returns NIL.

Example

```
Lisp> (GET-DEVICE-INFORMATION "DBA1"
                              :DEVICE-NAME
                              :ERROR-COUNT
                              :MOUNT-COUNT)
(:DEVICE-NAME "_DBA1:" :ERROR-COUNT Ø :MOUNT-COUNT 1)
```

Returns the device name, the error count, and the mount count for the device DBA1.

GET-FILE-INFORMATION

Returns information about a file. The keywords that you specify with the function determine the type of information the function returns. The keywords correspond to RMS file access block (FAB) and extended attribute block (XAB) fields. See the VAX/VMS RMS Reference Manual for information on FAB and XAB fields.

Format

GET-FILE-INFORMATION pathname &REST {keyword}*

Arguments

pathname

A pathname, namestring, symbol, or stream that represents the name of the file about which information is to be returned.

keyword

Optional keywords that return specific types of information about the specified file. Do not specify values with the keywords.

Table 8 lists the keywords that you can specify and the values they return.

Table 8
GET-FILE-INFORMATION Keywords

Keyword	Return Value
:ALLOCATION-QUANTITY	An integer that specifies the number of blocks that are allocated for the file.
:BACKUP-DATE	The last universal date and time the file was backed up. If the file has not been backed up, the function returns NIL.
:BLOCK-SIZE	An integer that specifies the block size.
:CREATION-DATE	The universal date and time the file was created.
:DEFAULT-EXTENSION	An integer that specifies the number of blocks that were added to the file's size when the file was extended.
:END-OF-FILE-BLOCK	An integer that specifies the block in which the file ends.
:EXPIRATION-DATE	The universal date and time the file expires. If an expiration date is not recorded, the function returns NIL.

(Continued on next page)

GC-VERBOSE Variable

A variable whose value is used as a flag to determine whether the LISP system is to display messages when a garbage collection occurs. If the flag is NIL, the system displays no messages. If the flag is not NIL, the system displays a message before and after a garbage collection occurs. The default value is T.

The messages the LISP system displays are controlled by the VAX LISP *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables.

For more information on garbage collector messages, see Chapter 7.

Example

```
Lisp> *GC-VERBOSE*
T
Lisp> (GC)
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
Lisp> (SETF *GC-VERBOSE* NIL)
NIL
Lisp> (GC)
T
```

- The first evaluation of the *GC-VERBOSE* variable returns the default value T, which indicates that the LISP system will display a message before and after a garbage collection occurs (depending on the values of the *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables).
- The call to the GC function shows the default messages the system displays when a garbage collection occurs and the variable's value is T.
- The call to the SETF macro sets the value of the variable to NIL.
- The second call to the GC function shows that the system does not display messages when the variable's value is NIL.

GENERALIZED-PRINT-FUNCTION-ENABLED-P Function

Used to globally enable a generalized print function or test whether a generalized print function is enabled. GENERALIZED-PRINT-FUNCTION-ENABLED-P is a predicate, and it can be used as a place form with SETF.

See Chapter 6 for more information about using generalized print functions.

Format

GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*

Argument

name

A symbol identifying the generalized print function to be enabled or tested.

Return Value

T or NIL.

Example

```
Lisp> (GENERALIZED-PRINT-FUNCTION-ENABLED-P 'PRINT-NIL-AS-LIST)
NIL
Lisp> (DEFINE-GENERALIZED-PRINT-FUNCTION PRINT-NIL-AS-LIST
                 (OBJECT STREAM)
                 (NULL OBJECT)
          (PRINC "( )" STREAM))
PRINT-NIL-AS-LIST
Lisp> (SETF (GENERALIZED PRINT-FUNCTION-ENABLED-P
                 'PRINT-NIL-AS-LIST)
        T)
T
Lisp> (PPRINT NIL)
( )
```

- The first use of the GENERALIZED-PRINT-FUNCTION-ENABLED-P function returns NIL, because no generalized print function named PRINT-NIL-AS-LIST has been defined.
- The call to DEFINE-GENERALIZED-PRINT-FUNCTION defines the generalized print function PRINT-NIL-AS-LIST.
- The call to SETF globally enables the generalized print function PRINT-NIL-AS-LIST.
- The PPRINT call prints ( ), because the generalized print function is enabled globally and pretty printing is enabled.

GET-GC-REAL-TIME

Enables you to inspect the elapsed time used by the garbage collector during program execution. This function is useful for tuning programs.

The function measures its value in terms of the INTERNAL-TIME-UNITS-PER-SECOND constant. This value is cumulative. It includes the elapsed time used for all the garbage collections that have occurred. A description of the INTERNAL-TIME-UNITS-PER-SECOND constant is provided in COMMON LISP: The Language.

When a suspended system is resumed, the elapsed time is set to zero.

For more information on the garbage collector, see Section 8.3.

Format

GET-GC-REAL-TIME

Return Value

The real time that has been used by the garbage collector.

Examples

1.
```
Lisp> (GET-GC-REAL-TIME)
348570000
Lisp> (GC)
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
Lisp> (GET-GC-REAL-TIME)
401210000
```

- The first call to the GET-GC-REAL-TIME function returns the real time used by the garbage collector.
- The call to the GC function invokes a garbage collection.
- The second call to the GET-GC-REAL-TIME function returns the updated real time that has been used by the garbage collector.

2.
```
Lisp> (DEFMACRO GC-ELAPSED-TIME (FORM)
         `(LET* ((START-GC (GET-GC-REAL-TIME)) (VALUE ,FORM)
                 (END-GC (GET-GC-REAL-TIME)))
            (FORMAT *TRACE-OUTPUT*
                 "~%GC elapsed time: ~D seconds~%"
                 (TRUNCATE
                   (- END-GC START-GC)
                   INTERNAL-TIME-UNITS-PER-SECOND))))
GC-ELAPSED-TIME
Lisp> (GC-ELAPSED-TIME (SUSPEND "MYFILE.TXT"))
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
; Starting garbage collection due to SUSPEND function.
; Starting garbage collection due to SUSPEND function.
GC elapsed time: 54 seconds
NIL
```

- The call to the DEFMACRO macro defines a macro named GC-ELAPSED-TIME, which evaluates a form and displays the amount of elapsed time that was used by the garbage collector during a form's evaluation.

- The call to the GC-ELAPSED-TIME function displays the amount of elapsed time the garbage collector used when the LISP system evaluated the form (SUSPEND "MYFILE.TXT").

GET-GC-RUN-TIME

Enables you to inspect the CPU time used by the garbage collector during program execution. This function is useful for tuning programs.

The function measures its value in terms of the CPU-TIME-UNITS-PER-SECOND constant. This value is cumulative. It includes the CPU time used for all the garbage collections that have occurred. A description of the CPU-TIME-UNITS-PER-SECOND constant is provided in COMMON LISP: The Language.

When a suspended system is resumed, the CPU time is set to zero.

For more information on the garbage collector, see Section 8.3.

Format

GET-GC-RUN-TIME

Return Value

The CPU time that has been used by the garbage collector.

Examples

1.
```
Lisp> (GET-GC-RUN-TIME)
6933
Lisp> (GC)
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
Lisp> (GET-GC-RUN-TIME)
8423
```

- The first call to the GET-GC-RUN-TIME function returns the CPU time used by the garbage collector.
- The call to the GC function invokes a garbage collection.
- The second call to the GET-GC-RUN-TIME function returns the updated CPU time that has been used by the garbage collector.

2.
```
Lisp> (DEFMACRO GC-CPU-TIME (FORM)
        `(LET* ((START-GC (GET-GC-RUN-TIME)) (VALUE ,FORM)
                (END-GC (GET-GC-RUN-TIME)))
           (FORMAT *TRACE-OUTPUT*
                   "~%GC CPU time: ~D seconds~%"
                   (TRUNCATE
                     (- END-GC START-GC)
                     INTERNAL-TIME-UNITS-PER-SECOND))))
GC-CPU-TIME
Lisp> (GC-CPU-TIME (SUSPEND "MYFILE.TXT"))
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
; Starting garbage collection due to SUSPEND function.
; Starting garbage collection due to SUSPEND function.
GC CPU time: 284 seconds
NIL
```

- The call to the DEFMACRO macro defines a macro named GC-CPU-TIME, which evaluates a form and displays the amount of CPU time that was used by the garbage collector during a form's evaluation.
- The call to the GC-CPU-TIME function displays the amount of CPU time the garbage collector used when the LISP system evaluated the form (SUSPEND "MYFILE.TXT").

GET-KEYBOARD-FUNCTION

Returns information about the function that is bound to a control character.

Format

GET-KEYBOARD-FUNCTION control-character

Argument

control-character

The control character to which a function is bound.

Return Value

The function that is bound to the control character and the function's argument list. If a function is not bound to the specified control character, the function returns NIL twice.

Examples

1.
```
Lisp> (BIND-KEYBOARD-FUNCTION #\^B #'BREAK)
T
Lisp> (GET-KEYBOARD-FUNCTION #\^B)
#<Compiled Function BREAK #x261510> ;
NIL
```

- The call to the BIND-KEYBOARD-FUNCTION function binds CTRL/B to the BREAK function.
- The call to the GET-KEYBOARD-FUNCTION function returns the function that is bound to CTRL/B and the function's argument list, which is NIL.

2.
```
Lisp> (GET-KEYBOARD-FUNCTION #\^S)
NIL ;
NIL
```

Returns NIL twice because a function is not bound to CTRL/S.

GET-PROCESS-INFORMATION

Returns information about a process. The keywords you specify with the function determine the type of information the function returns.

This function is similar to the $GETJPI VMS system service. For more information on the $GETJPI system service, see the VAX/VMS System Services Reference Manual and the VAX/VMS I/O User's Guide (Volume 1).

Format

GET-PROCESS-INFORMATION process &REST {keyword}*

Arguments

process

The name or the identification of the process (PID) about which information is to be returned. You can specify a string, an integer, or NIL. If you specify a string, the argument is the process name; if you specify an integer, the argument is the PID. If you specify NIL, the information the function returns corresponds to the current process.

keyword

Optional keywords that return specific types of information about the process. Do not specify values with the keywords.

Table 9 lists the keywords that you can specify and the values they return.

Table 9
GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:ACCOUNT	A string that specifies the account.
:ACTIVE-PAGE-TABLE-COUNT	An integer that specifies the active page table count.
:AST-ACTIVE	A vector of four bits that specifies the number of access modes that have active asynchronous system traps (ASTs) for the process.
:AST-COUNT	An integer that specifies the remaining AST quota.
:AST-ENABLED	A vector of four bits that specifies the number of access modes that have enabled ASTs for the process.
:AST-QUOTA	An integer that specifies the AST quota.

(Continued on next page)

Table 9 (Cont.)
GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:AUTHORIZED-PRIVILEGES	A vector of 64 bits that specifies the privileges the process is authorized to enable.
:BASE-PRIORITY	An integer that specifies the base priority.
:BATCH	Either T or NIL. The function returns T if the process is a batch job; otherwise, returns NIL.
:BIO-BYTE-COUNT	An integer that specifies the remaining buffered I/O byte count quota.
:BIO-BYTE-QUOTA	An integer that specifies the buffered I/O byte count quota.
:BIO-COUNT	An integer that specifies the remaining buffered I/O operation quota.
:BIO-OPERATIONS	An integer that specifies the number of buffered I/O operations the process has performed.
:BIO-QUOTA	An integer that specifies the buffered I/O operation quota.
:CPU-LIMIT	An integer that specifies the CPU time limit of the process in 10-millisecond units.
:CPU-TIME	An integer that specifies the accumulated CPU time of the process in 10-millisecond units.
:CURRENT-PRIORITY	An integer that specifies the current priority.
:CURRENT-PRIVILEGES	A vector of 64 bits that specifies the current privileges.
:DEFAULT-PAGE-FAULT-CLUSTER	An integer that specifies the default page fault cluster size.
:DEFAULT-PRIVILEGES	A vector of 64 bits that specifies the default privileges.
:DIO-COUNT	An integer that specifies the remaining direct I/O operation quota.

(Continued on next page)

Table 9 (Cont.)
GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:DIO-OPERATIONS	An integer that specifies the number of direct I/O operations the process has performed.
:DIO-QUOTA	An integer that specifies the direct I/O operation quota.
:ENQUEUE-COUNT	An integer that specifies the number of lock manager enqueues.
:ENQUEUE-QUOTA	An integer that specifies the lock manager enqueue quota.
:EVENT-FLAG-WAIT-MASK	A vector of 32 bits that specifies the event flag wait mask.
:FIRST-FREE-PØ-PAGE	An integer that specifies the first free page at the end of the program region.
:FIRST-FREE-P1-PAGE	An integer that specifies the first free page at the end of the control region.
:GLOBAL-PAGES	An integer that specifies the number of global pages in the working set.
:GROUP	An integer that specifies the group field of the UIC.
:IMAGE-NAME	A string that specifies the current image file name.
:IMAGE-PRIVILEGES	A vector of 64 bits that specifies the privileges with which the current image of the process was installed.
:JOB-SUBPROCESS-COUNT	An integer that specifies the number of subprocesses.
:LOCAL-EVENT-FLAGS	A vector of 32 bits that specifies the local event flags the process has in effect.
:LOGIN-TIME	An integer in internal time that specifies the time the process was created.
:MEMBER	An integer that specifies the member field of the UIC.
:MOUNTED-VOLUMES	An integer that specifies the number of mounted volumes.

(Continued on next page)

Table 9 (Cont.)
GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:OPEN-FILE-COUNT	An integer that specifies the remaining open file quota.
:OPEN-FILE-QUOTA	An integer that specifies the open file quota.
:OWNER-PID	An integer that specifies the process ID of the owner.
:PAGE-FAULTS	An integer that specifies the number of page faults.
:PAGE-FILE-COUNT	An integer that specifies the number of paging file pages being used by the process.
:PAGE-FILE-QUOTA	An integer that specifies the paging file quota.
:PAGES-AVAILABLE	An integer that specifies the number of virtual pages available for expansion.
:PID	An integer that specifies the process ID.
:PROCESS-NAME	A string that specifies the name of the process.
:SITE-SPECIFIC	A longword that specifies the contents of the site-specific longword.
:STATE	An integer that specifies the state.
:STATUS	A vector of 32 bits that specifies the status flags.
:SUBPROCESS-COUNT	An integer that specifies the number of subprocesses owned by the process.
:SUBPROCESS-QUOTA	An integer that specifies the subprocess quota.
:TERMINAL	A string that specifies the name of the terminal with which the process is interacting.
:TERMINATION-MAILBOX	An integer that specifies the termination mailbox unit number.
:TIMER-QUEUE-COUNT	An integer that specifies the remaining timer queue entry quota.

(Continued on next page)

Table 9 (Cont.)
GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:TIMER-QUEUE-QUOTA	An integer that specifies the timer queue entry quota.
:UIC	An integer that specifies the UIC.
:USERNAME	A string that specifies the user name.
:VIRTUAL-ADDRESS-PEAK	An integer that specifies the peak virtual address space size.
:WORKING-SET-AUTHORIZED-EXTENT	An integer that specifies the maximum authorized working set extent.
:WORKING-SET-AUTHORIZED-QUOTA	An integer that specifies the authorized working set quota.
:WORKING-SET-COUNT	An integer that specifies the number of process pages in the working set.
:WORKING-SET-DEFAULT	An integer that specifies the default working set size.
:WORKING-SET-EXTENT	An integer that specifies the current working set size extent.
:WORKING-SET-PEAK	An integer that specifies the peak working set size.
:WORKING-SET-QUOTA	An integer that specifies the current working set quota.
:WORKING-SET-SIZE	An integer that specifies the current working set size.

Return Value

The keywords and their values are returned as a list in the following format:

```
(:keyword-1 value-1 :keyword-2 value-2 ...)
```

The function preserves the order of the keyword-value pairs in the argument list.

If you do not specify keywords, the function returns a list of all the keyword-value pairs. If the specified process does not exist, the function returns NIL.

Examples

1. Lisp> (GET-PROCESS-INFORMATION "SMITH"
                                         :BATCH
                                         :CPU-TIME
                                         :BASE-PRIORITY
                                         :GLOBAL-PAGES)
   (:BATCH NIL :CPU-TIME 45884 :BASE-PRIORITY 4 :GLOBAL-PAGES 68
)

   Returns the value of the batch setting, the CPU time, the base priority, and the number of global pages used for the process SMITH.

2. Lisp> (DEFUN HOME NIL
            (LET ((PID
                   (SECOND (GET-PROCESS-INFORMATION
                            NIL
                            :OWNER-PID))))
              (IF (ZEROP PID) NIL (ATTACH PID))))
   HOME

   Defines a function that just returns NIL if the LISP system is running in the main process and attaches you to the parent process if the system is running in a subprocess.

GET-TERMINAL-MODES

Returns information about the terminal characteristics of the device associated with the *TERMINAL-IO* variable when you invoke the LISP system. If the specified stream is not connected to a terminal, the LISP system signals an error. The keywords you specify with the function determine the type of information the function returns.

This function is similar to the DCL SHOW TERMINAL command. For more information on the SHOW TERMINAL command, see the VAX/VMS Command Language User's Guide.

Format

GET-TERMINAL-MODES &REST {keyword}*

Argument

keyword

Optional keywords that return the terminal characteristics of the stream that is bound to the *TERMINAL-IO* variable. Do not specify values with the keywords.

Table 10 lists the keywords that you can specify and the values they return.

Table 10
GET-TERMINAL-MODES Keywords

Keyword	Return Value
:BROADCAST	Either T or NIL. The function returns T if your terminal can receive broadcast messages such as MAIL notifications and REPLY messages; otherwise, returns NIL.
:ECHO	Either T or NIL. The function returns T if the terminal displays the input character that it receives; otherwise, returns NIL. If the function returns NIL, the terminal displays only data output from the system or a user application program.

(Continued on next page)

Table 10 (Cont.)
GET-TERMINAL-MODES Keywords

Keyword	Return Value
:ESCAPE	Either T or NIL. The function returns T if ANSI standard escape sequences transmitted from the terminal are handled as a single multiple-character terminator; otherwise, returns NIL. The terminal driver checks the escape sequences for syntax before passing them to the program. For more information on escape sequences, see the VAX/VMS I/O User's Guide (Volume 1).
:HALF-DUPLEX	Either T or NIL. The function returns T if the terminal's operating mode is half-duplex and it returns NIL if the the operating mode is full-duplex. For a description of terminal operating modes, see the VAX/VMS I/O User's Guide (Volume 1).
:PASS-ALL	Either T or NIL. The function returns T if the system does not expand tab characters to blanks, fill carriage return or line feed characters, recognize control characters, and receive broadcast messages. The function returns NIL if the system passes all data to an application program as binary data.
:TYPE-AHEAD	Either T or NIL. The function returns T if the terminal accepts input that is typed when there is no outstanding read and it returns NIL if the terminal driver is dedicated and accepts input only when a program or the system issues a read.
:WRAP	Either T or NIL. The function returns T if the terminal generates a carriage return and a line feed when the end of a line is reached and it returns NIL otherwise. The end of the line is determined by the terminal-width setting.

Return Value

The keywords and their values are returned as a list in the following format:

```
(:keyword-1 value-1 :keyword-2 value-2 ...)
```

The function preserves the order of the keyword-value pairs in the argument list.

If you do not specify keywords, the function returns a list of the keyword-value pairs. The list is returned in a format such that it can be specified as an argument in a call to the SET-TERMINAL-MODES function.

Example

```
Lisp> (GET-TERMINAL-MODES)
(:BROADCAST T :ECHO T :ESCAPE NIL :HALF-DUPLEX NIL :PASS-ALL NIL
:TYPE-AHEAD T :WRAP T)
```

Returns a list of all the keyword-value pairs.

GET-VMS-MESSAGE

Returns the system message associated with a specified VMS status.

Format

GET-VMS-MESSAGE status &OPTIONAL flags

Arguments

status

A fixnum that specifies the VMS status code of the message that is to be returned. See the VAX/VMS System Messages and Recovery Procedures Manual for information on VMS message status codes.

flags

A bit vector of length four that specifies the content of the message. The default value is #*0000, which indicates that the process default message flags are to be used. The information that is included in the message when each of the four bits is set follows:

Bit	Information
0	Text
1	Message ID
2	Severity
3	Facility

Return Value

Returns the message that corresponds to the specified status code as a string. The function returns NIL if you specify a status code that does not exist.

Examples

1. Lisp> (GET-VMS-MESSAGE 32)
   "%SYSTEM-W-NOPRIV, no privilege for attempted operation"

   Returns the VMS message text for message 32 with all flags set.

2. Lisp (GET-VMS-MESSAGE 32 #*1001)
   "%SYSTEM, no privilege for attempted operation"

   Returns the VMS message text for message 32 with only the facility and text flags set.

HASH-TABLE-REHASH-SIZE

Returns the rehash size of a hash table. The rehash size indicates how much a hash table is to increase when it is full. The value is specified when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see COMMON LISP: The Language.

Format

HASH-TABLE-REHASH-SIZE hash-table

Argument

hash-table

The name of the hash table whose rehash size is to be returned.

Return Value

An integer greater than zero or a floating-point number greater than one. If an integer is returned, the value indicates the number of entries that are to be added to the table. If a floating-point number is returned, the value indicates the ratio of the new size to the old size.

Example

```
Lisp> (SETF *PRINT-ARRAY* NIL)
NIL
Lisp> (SETF TABLE-1 (MAKE-HASH-TABLE :TEST #'EQUAL
                                     :SIZE 200
                                     :REHASH-SIZE 1.5
                                     :REHASH-THRESHOLD .95))
#<Hash Table #x503BA8>
Lisp> (HASH-TABLE-REHASH-SIZE TABLE-1)
1.5
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets the hash table created by the call to the MAKE-HASH-TABLE function to TABLE-1.
- The call to the HASH-TABLE-REHASH-SIZE function returns the rehash size of the hash table TABLE-1.

HASH-TABLE-REHASH-THRESHOLD

Returns the rehash threshold for a hash table. The rehash threshold indicates how full a hash table can get before its size has to be increased. The value is specified when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see COMMON LISP: The Language.

Format

```
HASH-TABLE-REHASH-THRESHOLD hash-table
```

Argument

hash-table

The name of the hash table whose rehash threshold is to be returned.

Return Value

An integer greater than zero and less than hash table's rehash size or a floating-point number greater than zero and less than one.

Example

```
Lisp> (SETF *PRINT-ARRAY* NIL)
NIL
Lisp> (SETF TABLE-1 (MAKE-HASH-TABLE :TEST #'EQUAL
                                     :SIZE 200
                                     :REHASH-SIZE 1.5
                                     :REHASH-THRESHOLD .95))
#<Hash Table #x503BA8>
Lisp> (HASH-TABLE-REHASH-THRESHOLD TABLE-1)
0.95
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets the hash table created by the call to the MAKE-HASH-TABLE function to TABLE-1.
- The call to the HASH-TABLE-REHASH-THRESHOLD function returns the rehash threshold of the hash table TABLE-1.

HASH-TABLE-SIZE

Returns the initial size of a hash table. The value is specified when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see COMMON LISP: The Language.

Format

HASH-TABLE-SIZE hash-table

Argument

hash-table

The name of the hash table whose initial size is to be returned.

Return Value

An integer that indicates the initial size of the hash table.

Example

```
Lisp> (SETF *PRINT-ARRAY* NIL)
NIL
Lisp> (SETF TABLE-1 (MAKE-HASH-TABLE :TEST #'EQUAL
                                     :SIZE 200
                                     :REHASH-SIZE 1.5
                                     :REHASH-THRESHOLD .95))
#<Hash Table #x503BA8>
Lisp> (HASH-TABLE-SIZE TABLE-1)
1.5
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets the hash table created by the call to the MAKE-HASH-TABLE function to TABLE-1.
- The call to the HASH-TABLE-SIZE function returns the initial size of the hash table TABLE-1.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
:PAGE-FILE-COUNT	An integer that specifies the number of paging file pages remaining to the process.
:PAGE-FILE-QUOTA	An integer that specifies the paging file quota.
:PAGES-AVAILABLE	An integer that specifies the number of virtual pages available for expansion.
:PID	An integer that specifies the process ID.
:PID-OF-PARENT	An integer that specifies the PID of the parent process. This integer differs from :OWNER-PID in that :PID-OF-PARENT refers to the top-level process, while :OWNER-PID refers to the process immediately above the current process or subprocess.
:PROCESS-CREATION-FLAGS	A 32-bit bit-vector that specifies the flags used to create the process.
:PROCESS-INDEX	An integer that specifies the index number of the process at a given instant. (Process index numbers are reassigned to different processes over time.)
:PROCESS-NAME	A string that specifies the name of the process.
:SITE-SPECIFIC	A longword that specifies the contents of the site-specific longword.
:STATE	An integer that specifies the state.
:STATUS	A vector of 32 bits that specifies the status flags.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
:SUBPROCESS-COUNT	An integer that specifies the number of subprocesses owned by the process.
:SUBPROCESS-QUOTA	An integer that specifies the subprocess quota.
:TERMINAL	A string that specifies the name of the terminal with which the process is interacting.
:TERMINATION-MAILBOX	An integer that specifies the termination mailbox unit number.
:TIMER-QUEUE-COUNT	An integer that specifies the remaining timer queue entry quota.
:TIMER-QUEUE-QUOTA	An integer that specifies the timer queue entry quota.
:UAF-FLAGS	A 12-bit bit-vector that specifies the UAF flags of the user who owns the process.
:UIC	An integer that specifies the UIC.
:USERNAME	A string that specifies the user name.
:VIRTUAL-ADDRESS-PEAK	An integer that specifies the peak virtual address space size.
:WORKING-SET-AUTHORIZED-EXTENT	An integer that specifies the maximum authorized working set extent.
:WORKING-SET-AUTHORIZED-QUOTA	An integer that specifies the authorized working set quota.
:WORKING-SET-COUNT	An integer that specifies the number of process pages in the working set.

POST-GC-MESSAGE

Controls the message the LISP system displays after a garbage collection occurs. The value of this variable can be NIL, a string of message text, or the null string (""). If the value is NIL, the system displays a system message. If the value is a string, the system displays the string. If the variable's value is the null string (""), the system displays no output. The default value is NIL.

The system messages appear in the following form:

```
; Finished garbage collection due to GC function.
```

Example

```
Lisp> (GC)
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
Lisp> (SETF *POST-GC-MESSAGE* "")
""
Lisp> (GC)
; Starting garbage collection due to GC function.
T
Lisp> (SETF *POST-GC-MESSAGE* "GC -- finished")
"GC -- finished"
Lisp> (GC)
; Starting garbage collection due to GC function.
GC -- finished
T
```

- The first call to the GC function shows the garbage collection messages the LISP system displays by default.
- The first call to the SETF macro sets the value of the *POST-GC-MESSAGE* variable to the null string ("").
- The second call to the GC function shows that the system does not display a message when a garbage collection is finished when the variable's value is the null string.
- The second call to the SETF macro sets the value of the variable to the string "GC -- finished".
- The third call to the GC function shows that the system displays the new message when a garbage collection is finished if the variable's value is a string.

PPRINT-ARRAY-FORMATTERS

A variable whose value is a list of pretty-printer formatting functions that are tested for applicability to an array that is being printed. The default value is NIL.

You can add formatting functions to this variable's value by using the PUSH macro. The elements that you add to the list must be either formatting functions or symbols that have formatting-function definitions. The functions in the list must take one argument, the object to be pretty-printed, and must produce pretty-printer formatting code. You can assume that the object is an array.

A formatting function should return NIL if the pretty printer is to attempt additional formatting. A formatting function should end with either a call to the EXPAND-PPRINT-TEMPLATE macro or with a value other than NIL if the pretty printer is not to attempt additional formatting.

The functions that are stored as the value of the *PPRINT-ARRAY-FORMATTERS* variable are used as follows:

- If the object to be pretty-printed is an array other than a string or bit vector, the LISP system evaluates each formatting function in the list starting with the first function in the list.

  NOTE

  The pretty-printer algorithm checks for strings and bit vectors prior to checking the value of the *PPRINT-ARRAY-FORMATTERS* variable.

- If a function returns NIL, the system evaluates the next function. A call to the EXPAND-PPRINT-TEMPLATE macro results in translated pretty-printer code, which is used in the output.

- If a function returns a value other than NIL, the dispatch routine stops. Each call to the EXPAND-PPRINT-TEMPLATE macro produces translated pretty-printer code, which the pretty printer uses to format the object.

For more information about the use of the *PPRINT-ARRAY-FORMATTERS* variable, see Section 5.3.2.3.

Example

```
Lisp> (DEFUN TWO-BY-TWO-ARRAYS (OBJECT)
        (IF (AND (ARRAYP OBJECT)
                 (EQUAL '(2 2) (ARRAY-DIMENSIONS OBJECT)))
            (EXPAND-PPRINT-TEMPLATE "'A 2-dimensional array:
                                    ' +4 ! * ~ T3 * ! * ~ T3
                                    * !"
                                    (AREF OBJECT Ø Ø)
                                    (AREF OBJECT Ø 1)
                                    (AREF OBJECT 1 Ø)
                                    (AREF OBJECT 1 1))
        NIL))
TWO-BY-TWO-ARRAYS
Lisp> (PUSH 'TWO-BY-TWO-ARRAYS *PPRINT-ARRAY-FORMATTERS*)
(TWO-BY-TWO-ARRAYS)
```

```
Lisp> (PPRINT '#2A((1 2) (3 4)))
A 2-dimensional array:

    1  2
    3  4
```

- The call to the DEFUN macro defines a formatting function named TWO-BY-TWO-ARRAYS, which formats the output of 2-dimensional arrays.
- The call to the PUSH macro pushes the formatting function TWO-BY-TWO-ARRAYS onto the list that is bound to the *PPRINT-ARRAY-FORMATTERS* variable.
- The call to the PPRINT function shows the pretty-printed output of a 2-dimensional array after the formatting function TWO-BY-TWO-ARRAYS is added to the list bound to the variable *PPRINT-ARRAY-FORMATTERS* variable.

PPRINT-CHECK-INDENTATION

Checks the indentation of a list that is to be pretty-printed. You can include calls to this function in pretty-printer formatting function definitions. If the remaining width on the terminal screen available for pretty-printing is less than the value of the *PPRINT-MAJOR-WIDTH* variable, the pretty printer shifts the specified list to the left.

NOTE

You cannot call the PPRINT-CHECK-INDENTATION function at top level.

Format

PPRINT-CHECK-INDENTATION list formatting-function

Arguments

list

The list whose indentation is to be checked.

formatting-function

A 1-argument formatting function that is to be used to format the specified list's pretty-printed output. The argument is the specified list.

Return Value

The LISP code the pretty printer is to evaluate.

Example

```
Lisp> (DEFUN FACTORIAL-FORMATTER (NUMBER-LIST)
        (PPRINT-CHECK-INDENTATION NUMBER-LIST
                                  #'FACTORIAL-FORMAT))
FACTORIAL-FORMATTER
```

Defines formatting function named FACTORIAL-FORMATTER, which includes a call to the PPRINT-CHECK-INDENTATION function.

PPRINT-DATA-LIST

Specifies the default pretty-printer formatting function the pretty printer is to use to format a list of data. A list of data is a list that does not represent a function call or an application of a lambda expression. The value of this variable can be a function, a symbol that has a function definition, or NIL. The function must take one argument, the object to be pretty-printed, and must produce pretty-printer formatting code. You can assume that the object is a list.

If the value of the *PPRINT-DATA-LIST* variable is NIL, the pretty printer uses the system's default formatting function to format the list.

For more information about the use of the *PPRINT-DATA-LIST* variable, see Section 5.3.2.2.

Example

```
Lisp> (DEFUN MY-DATA-LIST (OBJECT)
        (EXPAND-PPRINT-TEMPLATE "(1 * <;6 *>)" OBJECT))
MY-DATA-LIST
Lisp> (SETF *PPRINT-DATA-LIST* 'MY-DATA-FORMAT)
MY-DATA-FORMAT
Lisp> (PPRINT '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30))
(1     2     3     4     5     6     7     8     9     10    11
 12    13    14    15    16    17    18    19    20    21    22
 23    24    25    26    27    28    29    30)
```

- The call to the DEFUN macro defines a formatting function named MY-DATA-LIST, which formats a list of data.
- The call to the SETF macro binds the formatting function MY-DATA-LIST to the variable *PPRINT-DATA-LIST*.
- The call to the PPRINT function pretty-prints the output of a data list in the format specified by the formatting function MY-DATA-LIST.

PPRINT-DEFINITION

Pretty-prints the function value of a symbol to a stream.

Format

PPRINT-DEFINITION symbol &OPTIONAL stream

Arguments

symbol

The symbol whose function value is to be pretty-printed.

stream

The stream to which the code is to be pretty-printed. The default stream is the stream bound to the *STANDARD-OUTPUT* variable.

Return Value

No value.

Examples

1.
```
Lisp> (DEFUN FACTORIAL (N)
"Returns the factorial of an integer."
(COND ((<= N 1) 1) (T (* N (FACTORIAL (- N 1))))))
FACTORIAL
Lisp> (PPRINT-DEFINITION 'FACTORIAL)
(DEFUN FACTORIAL (N)
  "Returns the factorial of an integer."
  (COND ((<= N 1) 1) (T (* N (FACTORIAL (- N 1))))))
```

- The call to the DEFUN macro defines a function called FACTORIAL, which returns the factorial of an integer.
- The call to the PPRINT-DEFINITION function pretty-prints the function value of the symbol FACTORIAL.

2.
```
Lisp> (DEFUN RECORD-MY-STATISTICS
(NAME AGE SIBLINGS MARRIED?)
(UNLESS (SYMBOLP NAME)
(ERROR "~S must be a symbol." NAME))
(SETF (GET NAME 'AGE) AGE
(GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
(GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED?) NAME)
RECORD-MY-STATISTICS
Lisp> (PPRINT-DEFINITION 'RECORD-MY-STATISTICS)
(DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
  (UNLESS (SYMBOLP NAME)
          (ERROR "~S must be a symbol." NAME))
  (SETF (GET NAME 'AGE) AGE
        (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
        (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED?)
  NAME)
```

- The call to the DEFUN macro defines a function called RECORD-MY-STATISTICS.
- The call to the PPRINT-DEFINITION function pretty-prints the function value of the symbol RECORD-MY-STATISTICS.

PPRINT-END-LINE

Specifies the line number of an object at which the pretty printer is to stop printing. Line numbers start at zero, so if you set the variable to n, the pretty printer stops printing after it prints n lines of output. If the value of this variable truncates printing, the pretty printer prints an ellipsis (...) at the end of the last line to indicate that the output was truncated, and immediately stops printing. The default value is NIL, which the pretty printer interprets to be the end of the object.

You can use the *PPRINT-END-LINE* variable with the *PPRINT-START-LINE* variable to instruct the pretty printer to print a specific section of a program. For example, if the value of the *PPRINT-START-LINE* variable is four and the value of the *PPRINT-END-LINE* variable is 10, the pretty printer prints lines 4 to 9 of an object.

For more information about the use of the *PPRINT-END-LINE* variable, see Section 5.2.1.1.

Example

```
Lisp> (DEFUN RECORD-MY-STATISTICS
(NAME AGE SIBLINGS MARRIED?)
(UNLESS (SYMBOLP NAME)
(ERROR "~S must be a symbol." NAME))
(SETF (GET NAME 'AGE) AGE
(GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
(GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED?) NAME)
RECORD-MY-STATISTICS
Lisp> (SETF *PPRINT-END-LINE* 3)
3
Lisp> (PPRINT-DEFINITION 'RECORD-MY-STATISTICS)
(DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
  (UNLESS (SYMBOLP NAME)
          (ERROR "~S must be a symbol." NAME)) ...
```

- The call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The call to the SETF macro sets the value of the *PPRINT-END-LINE* variable to three.
- The call to the PPRINT-DEFINITION function shows the effect the variable's value has on the output the pretty printer prints. The pretty printer prints three lines followed by an ellipsis, which indicates the output was truncated.

PPRINT-FORMATTER

Returns the pretty-printer formatting function that is associated with a symbol. The pretty printer uses the formatting function the PPRINT-FORMATTER function returns to pretty-print lists whose first element is the specified symbol. By default, several symbols have formatting functions associated with them. If a formatting function is not associated with the specified symbol, the PPRINT-FORMATTER function returns NIL.

You can specify the PPRINT-FORMATTER function in a call to the SETF macro.

For a complete description of how to use the PPRINT-FORMATTER to associate a formatting function with a symbol, see Section 5.3.2.1.

Format

    PPRINT-FORMATTER symbol

Argument

symbol

    The symbol whose formatting function is to returned.

Return Value

    The formatting function associated with the specified symbol.

Examples

1.
```
Lisp> (PPRINT-FORMATTER 'SETF)
SYSTEM-PPRINT::SETQ-FORMAT
```

   Checks whether a formatting function is defined for lists whose first element is the symbol SETF.

2.
```
Lisp> (DEFUN LET-FORMATTER (OBJECT)
        (EXPAND-PPRINT-TEMPLATE "(2 *  (1 <* !>) <- *>)"
                                OBJECT))
LET-FORMATTER
Lisp> (SETF (PPRINT-FORMATTER 'LET) 'LET-FORMATTER)
LET-FORMATTER
Lisp> (PPRINT '(LET ((Y 1) (Z 2)) (CONS Y Z)))
(LET ((Y 1)
      (Z 2))
  (CONS Y Z))
```

   - The call to the DEFUN macro defines a formatting function for lists whose first element is the symbol LET.
   - The call to the SETF macro associates the formatting function LET-FORMATTER with the symbol LET.
   - The call to the PPRINT function shows the output the pretty printer produces for lists whose first element is the symbol LET after the symbol is associated with the function LET-FORMATTER.

PPRINT-FUNCTION-CALL

Specifies the default pretty-printer formatting function the pretty printer is to use to format lists that represent a function call. A list represents a function call if its first element is a symbol that has a function defintion. The value of this variable can be a function, a symbol that has a function definition, or NIL. The function must take one argument, the object to be pretty-printed, and must produce pretty-printer formatting code. You can assume that the argument is a list whose first element is a symbol that has a function definition.

If the value of the *PPRINT-FUNCTION-CALL* variable is NIL, the pretty printer uses the system's default formatting function.

For more information about the use of the *PPRINT-FUNCTION-CALL* variable, see Section 5.3.2.2.

Example

```
Lisp> (DEFUN FUNCTION-FORMATTER (OBJECT)
        (EXPAND-PPRINT-TEMPLATE "(1';A call to the function:'
                                  ! * !
                                 ';with the following arguments'
                                  ! <* ;4>)"
                                  OBJECT))
FUNCTION-FORMATTER
Lisp> (LET ((*PPRINT-FUNCTION-CALL* 'FUNCTION-FORMATTER))
        (PPRINT '(+ 1 2 3 4 5)))
(;A call to the function:
 +
 ;with the following arguments
 1   2   3   4   5)
```

- The call to the DEFUN macro defines a formatting function that adds comments to the pretty-printed output of a function call. The comments identify the function name and the arguments that are passed to the function.

- The call to the LET special form binds the formatting function FUNCTION-FORMATTER to the formatting variable *PPRINT-FUNCTION-CALL*. The binding causes the dispatch routine to use the FUNCTION-FORMATTER function as the default function for lists whose first element is a symbol that names a function. Since the symbol + represents the name of a function, the pretty printer formats the list (+ 1 2 3 4 5) with the FUNCTION-FORMATTER function.

PPRINT-LAMBDA-APPLICATION

Specifies the pretty printer's default formatting function for lists that represent applications of lambda expressions. A list is assumed to represent the application of a lambda expression if the first element of the list is a list whose first element is the symbol LAMBDA. The following list represents an application of a lambda expression:

```
((LAMBDA (X Y) (+ X Y)) 3 4)
```

The value of this variable can be a function, a symbol that has a function definition, or NIL. The function must take one argument, the object to be pretty-printed, and must produce pretty-printer formatting code. You can assume that the argument represents a call to a lambda expression.

If the value of the *PPRINT-LAMBDA-APPLICATION* variable is NIL, the pretty printer uses the system's default formatting function.

For more information about the use of the *PPRINT-LAMBDA-APPLICATION* variable, see Section 5.3.2.2.

Example

```
Lisp> (PPRINT '((LAMBDA (X Y Z) (IF X Y Z)) 8 9 10))
((LAMBDA (X Y Z) (IF X Y Z)) 8 9 10)
Lisp> (DEFUN MY-LAMBDA-FORMAT (OBJECT)
        (EXPAND-PPRINT-TEMPLATE
          "{1 '(; This is the lambda:' ! * ! '
                ; The lambda is being applied to these
                  values:' !"
          (CAR OBJECT))
        (DO ((REST (REST OBJECT) (REST REST))
             (THIS (CAR REST) (CAR REST)))
            ((NULL REST) (EXPAND-PPRINT-TEMPLATE "')'}"))
          (EXPAND-PPRINT-TEMPLATE " * " THIS)
          (UNLESS (= 1 (LENGTH REST))
                  (EXPAND-PPRINT-TEMPLATE " , "))))
MY-LAMBDA-FORMAT
Lisp> (LET ((*PPRINT-LAMBDA-APPLICATION* 'MY-LAMBDA-FORMAT))
        (PPRINT '((LAMBDA (X Y Z) (IF X Y Z)) 8 9 10)))
(; This is the lambda:
 (LAMBDA (X Y Z) (IF X Y Z))
 ; The lambda is being applied to these values:
 8 9 10)
```

- The call to the PPRINT function shows the default pretty-printer output for lambda expressions.

- The call to the DEFUN macro defines a formatting function named MY-LAMBDA-FORMAT, which adds comments to pretty-printed output of lambda expressions. The comments in the definition identify the lambda expression and the arguments with which the lambda expression is called.

- The call to the LET special form binds the formatting function MY-LAMBDA-FORMAT to the formatting variable *PPRINT-LAMBDA-APPLICATION*. The binding causes the pretty-printer dispatch routine to use the MY-LAMBDA-FORMAT function as the default formatting function for lambda expressions. Since ((LAMBDA (X Y Z) (IF X Y Z)) 8 9 10) represents an application of a lambda expression, the dispatch routine calls the function MY-LAMBDA-FORMAT to format the output.

PPRINT-LEFT-MARGIN

Specifies the left margin for the pretty printer. Columns are numbered starting at zero so if you set this variable to n, the pretty printer leaves n empty columns to the left of the left margin.

If the value of the *PPRINT-LEFT-MARGIN* variable is NIL, the pretty printer uses the current stream position as the left margin. The default value is NIL.

Example

```
Lisp> (DEFUN RECORD-MY-STATISTICS
(NAME AGE SIBLINGS MARRIED?)
(UNLESS (SYMBOLP NAME)
(ERROR "~S must be a symbol." NAME))
(SETF (GET NAME 'AGE) AGE
(GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
(GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED) NAME)
RECORD-MY-STATISTICS
Lisp> (SETF *PPRINT-LEFT-MARGIN* 9)
9
Lisp> (PPRINT-DEFINITION 'RECORD-MY-STATISTICS)
         (DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
           (UNLESS (SYMBOLP NAME)
                   (ERROR "~S must be a symbol." NAME))
           (SETF (GET NAME 'AGE) AGE
                 (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
                 (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED)
           NAME)
```

- The call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The call to the PPRINT-DEFINITION function pretty-prints the function definition with the left margin set to zero.
- The call to the SETF macro sets the value of the *PPRINT-LEFT-MARGIN* variable to nine.
- The call to the PPRINT-DEFINITION function shows the effect the variable's value has on the output of the pretty printer. The pretty printer starts printing each line in column nine.

PPRINT-MAJOR-WIDTH

Controls when logical units of an object, such as DO, LET, and PROG forms, are shifted to the left. If the available line width is less than the variable's value, the pretty printer shifts units of an object to the left. When the pretty printer shifts text to the left, the amount of line width available for printing increases. The default value is 20.

The pretty printer can shift a logical unit of an object to the left only if you declare the logical unit to be a major logical unit. To declare a major logical unit, you must include the PPRINT-CHECK-INDENTATION function in a formatting-function definition.

When the pretty printer shifts a logical unit to the left, it surrounds the unit with comment lines that end with a vertical bar (|). The vertical bar indicates the indentation the pretty printer would have used if the structure was not shifted.

You can prevent the pretty printer from shifting text by setting the *PPRINT-MAJOR-WIDTH* variable to zero.

For more information about shifting logical units of an object to the left and the use of the *PPRINT-MAJOR-WIDTH* variable, see Section 5.2.3.2.

Example

```
Lisp> (DEFUN FACTORS-OF (INTEGER)
(IF (OR (ZEROP INTEGER) (= 1 (ABS INTEGER)))
(LIST INTEGER)
(DO ((RESULT-LIST NIL)
(TRY-THIS-INTEGER 2)
(REST-TO-BE-FACTORED 1)
(IF (MINUSP INTEGER)
(CONS -1 (NREVERSE RESULT-LIST))
(NREVERSE RESULT-LIST)))
(LET ((NEW-REMAINDER
(/ REST-TO-BE-FACTORED TRY-THIS-INTEGER)))
(COND ((INTEGERP NEW-REMAINDER)
(SETF REST-TO-BE-FACTORED NEW-REMAINDER)
(PUSH TRY-THIS-INTEGER RESULT-LIST))
(T (INCF TRY-THIS-INTEGER))))))
FACTORS-OF
Lisp> (SETF *PPRINT-MAJOR-WIDTH* 55)
55
Lisp> (PPRINT-DEFINITION 'FACTORS-OF)
(DEFUN FACTORS-OF (INTEGER)
  (IF (OR (ZEROP INTEGER) (= 1 (ABS INTEGER)))
      (LIST INTEGER)
;---   |
     (DO ((RESULT-LIST NIL)
          (TRY-THIS-INTEGER 2)
          (REST-TO-BE-FACTORED (ABS INTEGER)))
         ((= REST-TO-BE-FACTORED 1)
          (IF (MINUSP INTEGER)
              (CONS -1 (NREVERSE RESULT-LIST))
              (NREVERSE RESULT-LIST)))
```

```
;---    |
     (LET ((NEW-REMAINDER
             (/ REST-TO-BE-FACTORED TRY-THIS-INTEGER)))
       (COND ((INTEGERP NEW-REMAINDER)
               (SETF REST-TO-BE-FACTORED NEW-REMAINDER)
               (PUSH TRY-THIS-INTEGER RESULT-LIST))
              (T (INCF TRY-THIS-INTEGER))))
;---    |
       )
;---   |
      ))
```

- The call to the DEFUN macro defines a function named FACTORS-OF.
- The call to the SETF macro sets the value of the *PPRINT-MAJOR-WIDTH* variable to 55.
- The call to the PPRINT-DEFINITION function shows the pretty-printed output of the FACTORS-OF function definition with a major logical unit shifted to the left.

PPRINT-MISER-WIDTH

Controls miser mode printing. If the available line width between the current indentation and the end of the line is less than the value of this variable, the pretty printer enables miser mode. When the pretty printer is in miser mode, all indentations are one space and some spaces are replaced with line breaks. The default value is 40.

You can prevent the pretty printer from printing in miser mode by setting the *PPRINT-MISER-WIDTH* variable to zero.

For more information about miser mode and the use of the *PPRINT-MISER-WIDTH* variable, see Section 5.2.3.1.

Example

```
Lisp> (DEFUN FACTORS-OF (INTEGER)
(IF (OR (ZEROP INTEGER) (= 1 (ABS INTEGER)))
(LIST INTEGER)
(DO ((RESULT-LIST NIL)
(TRY-THIS-INTEGER 2)
(REST-TO-BE-FACTORED 1)
(IF (MINUSP INTEGER)
(CONS -1 (NREVERSE RESULT-LIST))
(NREVERSE RESULT-LIST)))
(LET ((NEW-REMAINDER
(/ REST-TO-BE-FACTORED TRY-THIS-INTEGER)))
(COND ((INTEGERP NEW-REMAINDER)
(SETF REST-TO-BE-FACTORED NEW-REMAINDER)
(PUSH TRY-THIS-INTEGER RESULT-LIST))
(T (INCF TRY-THIS-INTEGER))))))
FACTORS-OF
Lisp> (SETF *PPRINT-MISER-WIDTH* 57)
57
Lisp> (PPRINT-DEFINITION 'FACTORS-OF)
(DEFUN FACTORS-OF (INTEGER)
  (IF (OR (ZEROP INTEGER) (= 1 (ABS INTEGER)))
      (LIST INTEGER)
      (DO
       ((RESULT-LIST NIL)
        (TRY-THIS-INTEGER 2)
        (REST-TO-BE-FACTORED (ABS INTEGER)))
       ((= REST-TO-BE-FACTORED 1)
        (IF
         (MINUSP INTEGER)
         (CONS -1 (NREVERSE RESULT-LIST))
         (NREVERSE RESULT-LIST)))
       (LET ((NEW-REMAINDER
              (/ REST-TO-BE-FACTORED TRY-THIS-INTEGER)))
        (COND
         ((INTEGERP NEW-REMAINDER)
          (SETF REST-TO-BE-FACTORED NEW-REMAINDER)
          (PUSH TRY-THIS-INTEGER RESULT-LIST))
         (T (INCF TRY-THIS-INTEGER)))))))
```

- The call to the DEFUN macro defines a function named FACTORS-OF.
- The call to the SETF macro sets the value of the *PPRINT-MISER-WIDTH* variable to 57.
- The call to the PPRINT-DEFINITION function shows the pretty-printed output of the FACTORS-OF function definition with miser mode enabled.

PPRINT-PLIST

Pretty-prints the property list of a symbol to a stream. A property list is a list of symbol-value pairs; each symbol is associated with a value or an expression. The PPRINT-PLIST function prints the property list in a way that emphasizes the relationship between the symbols and their values. The function prints only the symbol-value pairs for which the symbol is accessible in the current package. For information on packages, see COMMON LISP: The Language.

NOTE

The form (PPRINT-PLIST 'ME) is not equivalent to the form (PPRINT (SYMBOL-PLIST 'ME)).

Format

PPRINT-PLIST symbol &OPTIONAL stream

Arguments

symbol

The symbol whose property list the pretty printer is to print.

stream

The stream to which the pretty printer is to print the code. The default stream is the stream bound to the *STANDARD-OUTPUT* variable.

Return Value

No value.

Examples

1.
```
Lisp> (SETF (GET 'CHILDREN 'SONS) '(DANNY GEOFFREY))
(DANNY GEOFFREY)
Lisp> (SETF (GET 'CHILDREN 'DAUGHTERS) 'SAMANTHA)
SAMANTHA
Lisp> (PPRINT-PLIST 'CHILDREN)
(DAUGHTERS SAMANTHA
 SONS (DANNY GEOFFREY))
```

- The calls to the SETF macro give the symbol CHILDREN the properties SONS and DAUGHTERS. The property list of the symbol CHILDREN has two properties: DAUGHTERS whose value is SAMANTHA and SONS whose value is the list (DANNY GEOFFREY).
- The call to the PPRINT-PLIST function pretty-prints the property list of the symbol CHILDREN. The pretty-printed output emphasizes the relationship between each property and its value.

2.

```
Lisp> (DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
        (UNLESS (SYMBOLP NAME)
                (ERROR "~S must be a symbol." NAME))
        (SETF (GET NAME 'AGE) AGE
              (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
              (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED)
       NAME)
RECORD-MY-STATISTICS
Lisp> (DEFUN SHOW-MY-STATISTICS (NAME)
        (UNLESS (SYMBOLP NAME)
                (ERROR "~S must be a symbol." NAME))
                (PPRINT-PLIST NAME))
SHOW-MY-STATISTICS
Lisp> (RECORD-MY-STATISTICS 'TOM 29 3 NIL)
TOM
Lisp> (SHOW-MY-STATISTICS 'TOM)
(IS-THIS-PERSON-MARRIED? NIL
 NUMBER-OF-SIBLINGS 3
 AGE 29)
```

- The first call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The second call to the DEFUN macro defines a function named SHOW-MY-STATISTICS. The definition includes a call to the PPRINT-PLIST function.
- The call to the RECORD-MY-STATISTICS function inputs the properties for the symbol TOM.
- The call to the SHOW-MY-STATISTICS function pretty-prints the property list for the symbol TOM.

PPRINT–RIGHT–MARGIN

Specifies the pretty printer's right margin. Columns are numbered starting at zero, so if you set this variable to n, the pretty printer inserts n spaces to the left of the right margin. If the variable's value is NIL and the stream being used goes to a terminal, the pretty printer uses the width of the terminal; if the value is NIL and the stream does not go to a terminal, the pretty printer uses a right margin of 72. The default value is NIL.

Example

```
Lisp> (DEFUN RECORD-MY-STATISTICS
(NAME AGE SIBLINGS MARRIED?)
(UNLESS (SYMBOLP NAME)
(ERROR "~S must be a symbol." NAME))
(SETF (GET NAME 'AGE) AGE
(GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
(GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED)
NAME)
RECORD-MY-STATISTICS
Lisp> (SETF *PPRINT-RIGHT-MARGIN* 40)
40
Lisp> (PPRINT-DEFINITION 'RECORD-MY-STATISTICS)
(DEFUN
 RECORD-MY-STATISTICS
 (NAME AGE SIBLINGS MARRIED?)
 (UNLESS
  (SYMBOLP NAME)
  (ERROR
   "~S must be a symbol."
   NAME))
 (SETF
  (GET NAME 'AGE) AGE
  (GET NAME 'NUMBER-OF-SIBLINGS)
  SIBLINGS
  (GET
   NAME
   'IS-THIS-PERSON-MARRIED?)
  MARRIED)
 NAME)
```

- The call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The call to the SETF macro sets the value of the *PPRINT-RIGHT-LINE* variable to 40.
- The call to the PPRINT function shows the effect the variable's value has on the output of the pretty printer. The pretty printer stops printing each line at column 40.

PPRINT-SPECIAL-FORMATTERS

A variable whose value is a list of pretty-printer formatting functions. This variable is the first variable that the pretty-printer dispatch routine checks. Before the pretty printer prints an object, the dispatch routine checks the applicability of each function stored in the value of the variable to the object, starting from the first function in the list.

You can add formatting functions to this variable's value by using the PUSH macro. The elements that you add to the list must be either formatting functions or symbols that have formatting-function definitions. The functions in the list must take one argument, the object to be pretty-printed, and must produce pretty-printer formatting code. The function must be able to take an argument of any type.

A formatting function should return NIL if the pretty printer is to perform additional formatting. If the pretty printer is not to perform additional formatting, the formatting function should end with either a call to the EXPAND-PPRINT-TEMPLATE macro or a value other than NIL.

The functions that are stored as the value of the *PPRINT-SPECIAL-FORMATTERS* variable are used as follows:

- If a function returns NIL, the system evaluates the next function. A call to the EXPAND-PPRINT-TEMPLATE macro results in translated pretty-printer code, which is used in the output.

- If a function returns a value other than NIL, the dispatch routine stops. Each call to the EXPAND-PPRINT-TEMPLATE macro produces translated pretty-printer code, which the pretty printer uses to format the object.

For more information about the use of the *PPRINT-SPECIAL-FORMATTERS* variable, see Section 5.3.2.3.

Example

```
Lisp> (DEFUN MY-PATHNAME-FORMATTER (OBJECT)
        (IF (PATHNAMEP OBJECT)
            (PATHNAME-FORMATTER OBJECT)
            NIL))
MY-PATHNAME-FORMATTER
Lisp> (DEFUN PATHNAME-FORMATTER (OBJECT)
        (LET (TEMP)
          (EXPAND-PPRINT-TEMPLATE "{3 '#S(PATHNAME'")
          (WHEN (SETQ TEMP (PATHNAME-HOST OBJECT))
                (EXPAND-PPRINT-TEMPLATE "! ':HOST' T12 P"
                                        TEMP))
          (WHEN (SETQ TEMP (PATHNAME-DEVICE OBJECT))
                (EXPAND-PPRINT-TEMPLATE "! ':DEVICE' T12 P"
                                        TEMP))
          (WHEN (SETQ TEMP (PATHNAME-DIRECTORY OBJECT))
                (EXPAND-PPRINT-TEMPLATE "!':DIRECTORY' T12 P"
                                        TEMP))
          (WHEN (SETQ TEMP (PATHNAME-NAME OBJECT))
                (EXPAND-PPRINT-TEMPLATE "! ':NAME' T12 P"
                                        TEMP))
          (WHEN (SETQ TEMP (PATHNAME-TYPE OBJECT))
                (EXPAND-PPRINT-TEMPLATE "! ':TYPE' T12 P"
                                        TEMP))
```

```
          (WHEN (SETQ TEMP (PATHNAME-VERSION OBJECT))
                (EXPAND-PPRINT-TEMPLATE "! ':VERSION' T12 P"
                                        TEMP))
          (EXPAND-PPRINT-TEMPLATE "')'}")))
PATHNAME-FORMATTER
Lisp> (PUSH 'MY-PATHNAME-FORMATTER *PPRINT-SPECIAL-FORMATTERS*)
(MY-PATHNAME-FORMATTER)
Lisp> (PPRINT (PATHNAME "HOME::[BASE]"))
#S(PATHNAME
   :HOST       "HOME"
   :DIRECTORY  "BASE")
```

- The two calls to the DEFUN macro define two formatting functions named MY-PATHNAME-FORMATTER and PATHNAME-FORMATTER. The functions are defined such that they are called if the object being pretty-printed is a pathname. If the object is a pathname, the fields for which values are specified are to be pretty-printed. Fields that have a value of NIL are not to be printed.

- The call to the PUSH macro pushes the formatting function MY-PATHNAME-FORMATTER onto the list that is bound to the *PPRINT-SPECIAL-FORMATTERS* variable.

- The call to the PPRINT function shows the pretty-printed output of the pathname represented by the namestring HOME::[BASE] after the formatting function MY-PATHNAME-FORMATTER is added to the list that is bound to the *PPRINT-SPECIAL-FORMATTERS* variable. The pretty printer prints the output in tabular format and prints only the fields that are specified in the namestring.

PPRINT-START-LINE

Specifies the line number at which the pretty printer is to start printing. Line numbers start at zero, so if you set the variable to n, the pretty printer starts printing after it skips n lines. The default value is NIL; NIL has the same affect as the value zero.

You can use the *PPRINT-START-LINE* variable with the *PPRINT-END-LINE* variable to pretty-print a specific section of a program. For example, if the value of the *PPRINT-START-LINE* variable is four and the value of the *PPRINT-END-LINE* variable is 10, the pretty printer prints lines 4 to 9 of an object.

For more information about the use of the *PPRINT-START-LINE* variable, see Section 5.2.1.1.

Example

```
Lisp> (DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
(UNLESS (SYMBOLP NAME)
(ERROR "~S must be a symbol." NAME))
(SETF (GET NAME 'AGE) AGE
(GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
(GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED) NAME)
RECORD-MY-STATISTICS
Lisp> (SETF *PPRINT-START-LINE* 3)
3
Lisp> (PPRINT-DEFINITION 'RECORD-MY-STATISTICS)
  (SETF (GET NAME 'AGE) AGE
        (GET NAME 'NUMBR-OF-SIBLINGS) SIBLINGS
        (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED)
       NAME)
```

- The call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The call to the SETF macro sets the value of the *PPRINT-START-LINE* variable to three.
- The call to the PPRINT function shows the effect the variable's value has on the output the pretty printer prints. The pretty printer skips three lines and then prints the rest of the function definition.

PRE-GC-MESSAGE

Controls the message the LISP system displays when a garbage collection starts. The value of this variable can be NIL, a string of message text, or the null string (""). If the value is NIL, the system displays a system message. If the value is a string of message text, the system displays the message text. If the variable's value is the null string, the system displays no output. The default value is NIL.

System messages appear in the following form:

```
; Starting garbage collection due to GC function.
```

Example

```
Lisp> (GC)
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
T
Lisp> (SETF *PRE-GC-MESSAGE* "")
""
Lisp> (GC)
; Finished garbage collection due to GC function.
T
Lisp> (SETF *PRE-GC-MESSAGE* "GC -- started")
"GC -- started"
Lisp> (GC)
GC -- started
; Finished garbage collection due to GC function.
T
```

- The first call to the GC function shows the garbage collection messages that are printed by default.
- The first call to the SETF macro sets the value of the *PRE-GC-MESSAGE* variable to the null string ("").
- The second call to the GC function causes the system not to display a message when the garbage collection starts.
- The second call to the SETF macro sets the value of the variable to the string "GC -- started".
- The third call to the GC function causes the system to display the new message text when the garbage collection starts.

PRINT-SIGNALED-ERROR

Used by the VAX LISP error handler to display a formatted error message when an error is signaled. The function prints all output to the stream bound to the *ERROR-OUTPUT* variable. The error message formats are described in Sections 3.2.1 to 3.2.3.

You can include a call to this function in an error handler that you create (see Section 3.3.1).

Format

```
PRINT-SIGNALED-ERROR function-name
     error-signaling-function &REST args
```

Arguments

function-name

The name of the function that is to call the specified error-signaling function.

error-signaling-function

The name of an error-signaling function. Valid function names are ERROR, CERROR, and WARN.

args

The specified error-signaling function's arguments.

Return Value

Undefined.

Example

```
Lisp> (DEFUN CONTINUING-ERROR-HANDLER (FUNCTION-NAME
                                       ERROR-SIGNALING-FUNCTION
                                       &REST ARGS)
        (IF (EQ ERROR-SIGNALING-FUNCTION 'CERROR)
            (PROGN
              (APPLY #'PRINT-SIGNALED-ERROR
                     FUNCTION-NAME
                     ERROR-SIGNALING-FUNCTION
                     ARGS)
              (FORMAT *ERROR-OUTPUT*
                     "~&It will be continued automatically.~2%.")
              NIL)
            (APPLY #'UNIVERSAL-ERROR-HANDLER
                   FUNCTION-NAME
                   ERROR-SIGNALING-FUNCTION
                   ARGS)))
CONTINUING-ERROR-HANDLER
```

Defines an error handler that automatically continues from a continuable error after displaying an error message. All other errors are passed to the system's error handler.

PRINT-SLOT-NAMES-AS-KEYWORDS

Determines how the slot names of a structure are formatted when they are displayed. The value can be either T or NIL. If the value is T, slot names are preceded with a colon (:). For example:

```
#S(SPACE :AREA 4 :COUNT 10)
```

If the value is NIL, slot names are not preceded with a colon. For example:

```
#S(SPACE AREA 4 COUNT 10)
```

The default value is T.

Example

```
Lisp> (DEFSTRUCT HOUSE
                 ROOMS
                 FLOORS)
HOUSE
Lisp> (MAKE-HOUSE :ROOMS 8 :FLOORS 2)
#S(HOUSE :ROOMS 8 :FLOORS 2)
Lisp> (SETF *PRINT-SLOT-NAMES-AS-KEYWORDS* NIL)
NIL
Lisp> (MAKE-HOUSE :ROOMS 8 :FLOORS 2)
#S(HOUSE ROOMS 8 FLOORS 2)
```

- The call to the DEFSTRUCT macro defines a structure named HOUSE.
- The first call to the constructor function MAKE-HOUSE creates a structure named HOUSE. Colons are included in the output because the value of the *PRINT-SLOT-NAMES-AS-KEYWORDS* variable is T.
- The call to the SETF macro changes the value of the *PRINT-SLOT-NAMES-AS-KEYWORDS* variable to NIL.
- The second call to the constructor function MAKE-HOUSE creates a structure named HOUSE. Colons are not included in the output because the value of the *PRINT-SLOT-NAMES-AS-KEYWORDS* variable is NIL.

SET-TERMINAL-MODES

Sets the terminal characteristics of the stream bound to the *TERMINAL-IO* variable when you invoke the LISP system. Changes to the stream affect all streams attached to the terminal.

You must be careful when you change the settings of terminal modes. A change to terminal modes affects all the streams that are open to the terminal. If you put a stream into pass-all mode, for example, all the streams open to the terminal are put into pass-all mode.

NOTE

Create an error handler to prevent your terminal from being placed in a nonstandard state. See Section 3.3 for information about how to create an error handler.

Format

SET-TERMINAL-MODES &KEY {keyword value}*

Argument

keyword value

Optional keyword-value pairs, which specify options that set the terminal characteristics of the stream bound to the *TERMINAL-IO* variable.

Table 11 lists the options that you can specify.

Table 11
SET-TERMINAL-MODES Options

Keyword-Value Pair	Description
:BROADCAST value	Specifies whether the terminal can receive broadcast messages such as MAIL notifications and REPLY messages. The value can be either T or NIL. If you specify T, the terminal can receive messages; if you specify NIL, the terminal cannot receive messages.
:ECHO value	Specifies whether the terminal displays the input characters it receives. The value can be either T or NIL. If you specify T, the terminal displays input characters; if you specify NIL, the terminal displays only data output from the system or from a user application program.

(Continued on next page)

Table 11 (Cont.)
SET-TERMINAL-MODES Options

Keyword-Value Pair	Description
:ESCAPE value	Specifies whether ANSI standard escape sequences transmitted from the terminal are handled as a single multiple-character terminator. The value can be either T or NIL. If you specify T, the escape sequences are handled as a single multiple-character terminator. The terminal driver checks the escape sequences for syntax before passing them to the program. For more information on escape sequences, see the VAX/VMS I/O User's Guide (Volume 1).
:HALF-DUPLEX value	Specifies the terminal's operating mode. The value can be either T or NIL. If you specify T, the terminal's operating mode is half-duplex. If you specify NIL, the operating mode is full-duplex. For a description of terminal operating modes, see the VAX/VMS I/O User's Guide (Volume 1).
:PASS-ALL value	Specifies whether the terminal is in pass-all mode. The value can be either T or NIL. If you specify T, the system does not expand tab characters to blanks, fill carriage return or line feed characters, recognize control characters, or receive broadcast messages. If you specify NIL, the system passes all data to an application program as binary data.
:TYPE-AHEAD value	Specifies whether the terminal accepts input that is typed when there is no outstanding read. The value can be either T or NIL. If you specify T, the terminal accepts input even if there is not outstanding read. If you specify NIL, the terminal is dedicated and accepts input only when a program or the system issues a read.

(Continued on next page)

Table 11 (Cont.)
SET-TERMINAL-MODES Options

Keyword-Value Pair	Description
:WRAP value	Specifies whether the terminal driver generates a carriage return and a line feed when the end of a line is reached. The value can be either T or NIL. If you specify T, the terminal driver generates a carriage return and a line feed when the end of a line is reached. The end of the line is determined by the terminal width setting.

Return Value

Undefined

Example

```
Lisp> (DEFVAR *OLD-TERMINAL-STATE*)
*OLD-TERMINAL-STATE*
Lisp> (DEFUN PASS-ALL-HANDLER (FUNCTION ERROR &REST ARGS)
        (LET ((CURRENT-SETTINGS (GET-TERMINAL-MODES)))
          (APPLY #'SET-TERMINAL-MODES *OLD-TERMINAL-STATE*)
          (APPLY #'UNIVERSAL-ERROR-HANDLER FUNCTION ERROR ARGS)
          (APPLY #'SET-TERMINAL-MODES CURRENT-SETTINGS)))
PASS-ALL-HANDLER
Lisp> (DEFUN UNUSUAL-INPUT NIL
        (LET ((*OLD-TERMINAL-STATE* (GET-TERMINAL-MODES))
              (*UNIVERSAL-ERROR-HANDLER* #'PASS-ALL-HANDLER))
          (UNWIND-PROTECT (PROGN
                            (SET-TERMINAL-MODES
                            :PASS-ALL
                            T
                            :ECHO
                            NIL)
                           (GET-INPUT))
                          (APPLY #'SET-TERMINAL-MODES
                                 *OLD-TERMINAL-STATE*))))
UNUSUAL-INPUT
```

- The call to the DEFVAR macro informs the LISP system that *OLD-TERMINAL-STATE* is a special variable.
- The first call to the DEFUN macro defines an error handler named PASS-ALL-HANDLER, which is used when the terminal is placed in an unusual state. The handler assumes that the normal terminal modes are stored as the value of the *OLD-TERMINAL-STATE* variable.
- The second call to the DEFUN macro defines a function named UNUSUAL-INPUT, which causes the function PASS-ALL-HANDLER to be the error handler while the function GET-INPUT is being executed. The GET-INPUT function is inside a call to the UNWIND-PROTECT function so an error or throw puts the terminal back in its original state.

SPAWN

Creates a subprocess for executing Command Language Interpreter (CLI) commands. This function causes the LISP system to interrupt execution of a LISP process and to optionally execute the specified CLI command. If you specify the :PARALLEL keyword with a value of T, the LISP process continues to execute while the subprocess is executing. If you do not specify this keyword or if you specify it with NIL, the LISP process is put into a hibernation state until the subprocess completes its execution.

This function is equivalent to the DCL SPAWN command. For more information on the SPAWN command, see the VAX/VMS Command Language User's Guide.

Format

SPAWN &KEY {keyword value}*

Arguments

keyword value

Optional keyword-value pairs that specify options that modify the spawn operation.

Table 12 lists the options that you can specify.

Return Value

Undefined

Table 12
SPAWN Options

Keyword-Value Pair	Description
:COMMAND-STRING string	Specifies a DCL command the specified subprocess is to process. The value must be a DCL command. By default, the SPAWN function does not process a command.
:DCL-SYMBOLS value	Specifies whether the spawned subprocess is to acquire the currently defined CLI symbols from the LISP process. The value can be either T or NIL. If you specify T, the subprocess acquires the CLI symbols. If you specify NIL, the subprocess does not acquire the CLI symbols. The default value is T.

(Continued on next page)

Table 12 (Cont.)
SPAWN Options

Keyword-Value Pair	Description
:INPUT-FILE pathname	Specifies a pathname, namestring, symbol, or stream that names an input file containing one or more DCL commands to be associated with the logical name SYS$INPUT and to be executed by the spawned subprocess. If you specify both a command string and an input file, the command string is processed before the commands in the input file. The subprocess is terminated when processing is complete.
:LOGICAL-NAMES value	Specifies whether the spawned subprocess is to acquire the currently defined logical names. The value can be either T or NIL. If you specify T, the subprocess acquires the logical names; if you specify NIL, the subprocess does not acquire the logical names. The default value is T.
:OUTPUT-FILE pathname	Specifies a pathname, namestring, symbol, or stream that names the output file to be associated with the logical name SYS$OUTPUT and to which the results of the spawned subprocess are to be written.
:PARALLEL value	Specifies whether the execution of the LISP system and the created subprocess are to be parallel. The value can be either T or NIL. If you specify T, the execution of the system and the subprocess are parallel. If you specify NIL, the LISP system remains in a hibernation state until the created subprocess completes its execution and exits. The default value is NIL.
:PROCESS-NAME string	Specifies the name of the subprocess to be created. If you omit this keyword, the system generates a unique name.

Examples

1.
```
Lisp> (SPAWN)
$
```

   Creates a uniquely named subprocess and attaches the terminal to it. The commands typed at the terminal are directed to the subprocess until the subprocess exits.

2.
```
Lisp> (SPAWN :INPUT-FILE "START.COM"
             :OUTPUT-FILE "START.LOG"
             :PARALLEL T)
Lisp>
```

   Creates a subprocess that will execute the contents of START.COM.

SUSPEND

Writes information about a LISP system to a file, making it possible to resume the LISP system at a later time. The function does not stop the current system, but copies the state of the LISP system when the function is invoked to the specified file. When you reinvoke the LISP system with the /RESUME qualifier and the file name that was specified with the SUSPEND function, program execution continues from the point where the SUSPEND function was called.

Only the static and dynamic portions of LISP memory are written to the specified file. When you resume a suspended system, the read-only sections of LISP memory are taken from LISP$SYSTEM:LISPSUS.SUS. You must make sure that your original LISP system is in LISP$SYSTEM:LISPSUS.SUS; if it is not, you will not be able to resume the system.

When a suspended system is resumed, the LISP environment is identical to the environment that existed when the suspend operation occurred with the following exceptions:

- All streams except the standard streams are closed.
- The *DEFAULT-PATHNAME-DEFAULTS* variable is set to the current directory.
- Call-out state might be lost (see Section 6.9).
- Some Editor state is changed (see the VAX LISP Editor Manual).

Format

SUSPEND pathname

Argument

pathname

A pathname, namestring, or symbol that represents the file name to which the function is to write the LISP system state.

Return Value

T when the LISP system is resumed at a later time and NIL when execution continues after a resume operation.

Example

```
Lisp> (DEFUN PROGRAM-MAIN-LOOP NIL
        (LOOP (PRINC "Enter number> ")
              (SETF X (READ *STANDARD-INPUT*))
              (FORMAT *STANDARD-OUTPUT*
                      "~%The square root of ~F is ~F. ~%"
                      X
                      (SQRT X))))
PROGRAM-MAIN-LOOP
Lisp> (DEFUN DUMP-PROGRAM NIL
        (SUSPEND "MYPROG.SUS")
        (FRESH-LINE)
        (PRINC "Welcome to my program!")
        (TERPRI)
        (PROGRAM-MAIN-LOOP))
DUMP-PROGRAM
Lisp> (DUMP-PROGRAM)
```

```
; Starting garbage collection due to GC function.
; Finished garbage collection due to GC function.
; Starting garbage collection due to SUSPEND function.
; Finished garbage collection due to SUSPEND function.
Welcome to my program
Enter number> 25
The square root of 25.0 is 5.0.
Enter number> 5
The square root of 5.0 is 2.236038.
Enter number>
      .
      .
      .
CTRL/C
Lisp> (EXIT)
$ LISP/RESUME=MYPROG.SUS
Welcome to my program
Enter number>
```

- The first call to the DEFUN macro defines a function named PROGRAM-MAIN-LOOP.
- The second call to the DEFUN macro defines a function named DUMP-PROGRAM.
- The call to the DUMP-PROGRAM function copies the current state of the LISP environment to the file MYPROG.SUS. The LISP system continues to run, displaying the message "Welcome to my program" and then executes the PROGRAM-MAIN-LOOP function.
- The call to the EXIT function exits the LISP system.
- The LISP/RESUME=MYPROG.SUS specification reinvokes the LISP system, displays the message, and executes the PROGRAM-MAIN-LOOP function.

THROW-TO-COMMAND-LEVEL

Throws you to a command level.

Interactive LISP can have several command levels or levels of control. The top-level loop is the highest command level. The break loop and the debugger are nested command levels within the top-level loop.

If you are in the break loop or debugger, the LISP system can place you in other nested loops. For example, if you are in a break loop, you can invoke another break loop. Or, if you are in a break loop and the system signals an error, the debugger is invoked and you are placed in a debug loop. While you are in the debug loop, you can invoke another break loop, or if the system signals another error, you can be placed in another debug loop. Each loop is a nested command level. The n in the break loop and debugger prompts (Break n>, Debug n>) indicates the command level.

See COMMON LISP: The Language or Section 2.3 for information on using command levels. For more information on the break loop and the debugger, see Sections 4.3 and 4.4.

Format

THROW-TO-COMMAND-LEVEL level

Argument

level

The command level to be thrown to. The value of this argument can be either an integer or a keyword. The keywords you can specify and the position each keyword throws to are the following:

:CURRENT	Current command level
:PREVIOUS	Previous command level
:TOP	Top level

NOTE

CTRL/C is bound to the form (THROW-TO-COMMAND-LEVEL :TOP).

Return Value

No value.

Example

```
Lisp> (FACTORIAL M)

Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Symbol has no value: M

Control Stack Debugger
Frame #3: (EVAL (FACTORIAL M))
Debug 1> (THROW-TO-COMMAND-LEVEL :TOP)
Lisp>
```

THROW-TO-COMMAND-LEVEL Function

- The debugger is invoked because an error was signaled when the FACTORIAL function was called. The number 1 in the debugger prompt indicates the command level after the error.
- The call to the THROW-TO-COMMAND-LEVEL function returns control to the top-level loop.

TOP-LEVEL-PROMPT

Enables you to change the top-level prompt. The value of this variable can be one of the following:

- A string
- A function of no arguments that returns a string
- NIL

If you specify NIL, the default prompt (Lisp>) is used.

Example

```
Lisp> (SETF *TOP-LEVEL-PROMPT*  "TOP> ")
"TOP> "
TOP>
```

Sets the value of the variable *TOP-LEVEL-PROMPT* to TOP>.

TRANSLATE-LOGICAL-NAME

Searches a logical name table for a logical name, translates it, and returns it as a 1-element list. If the function does not find the logical name during its first pass through the logical name table, it changes the logical name to all uppercase characters and searches the table again. As a result, the function is faster if you specify logical names in uppercase characters.

The TRANSLATE-LOGICAL-NAME function performs only one level of logical-name translation.

This function is equivalent to the DCL SHOW TRANSLATION command. For additional information about the SHOW TRANSLATION command or about using logical names, see the VAX/VMS Command Language User's Guide.

Format

TRANSLATE-LOGICAL-NAME string &KEY :TABLE keyword

Arguments

string

The logical name for which the function is to search.

keyword

A keyword whose value indicates the logical name table that the function is to search. If you do not specify a table name, the process, group, and system name tables are searched respectively. The values you can specify with the :TABLE keyword are the following:

:SYSTEM	System name table
:GROUP	Group name table
:PROCESS	Process name table
:ALL	All three tables (default)

Return Value

Returns the logical name as a 1-element list if a match is found. Returns NIL if a match is not found.

Example

```
Lisp> (DEFUN SHOW-WHERE-I-AM (&OPTIONAL
                              (STREAM *STANDARD-OUTPUT*))
        (FORMAT STREAM
                "~&Current host is ~A ~
                 ~%Current device is ~A ~
                 ~%Current directory is ~A ~%"
                (CAR (TRANSLATE-LOGICAL-NAME "SYS$NODE"))
                (CAR (TRANSLATE-LOGICAL-NAME "SYS$DISK"))
                (CONCATENATE 'STRING
                             "["
                             (PATHNAME-DIRECTORY
                              (DEFAULT-DIRECTORY))
                             "]"))
        (VALUES))
SHOW-WHERE-I-AM
Lisp> (SHOW-WHERE-I-AM)
Current host is MIAMI::
Current device is DBA1:
```

```
Current directory is [VAXLISP]
Lisp> (SETF (DEFAULT-DIRECTORY) "SYS$LIBRARY")
"SYS$LIBRARY"
Lisp> (SHOW-WHERE-I-AM)
Current host is MIAMI::
Current device is SYS$SYSROOT:
Current directory is [SYSLIB]
```

- The call to the DEFUN macro defines a function named SHOW-WHERE-I-AM, which displays the current host, device, and directory.
- The first call to the function SHOW-WHERE-I-AM displays the current host, device, and directory.
- The call to the SETF macro changes the directory to SYSLIB.
- The second call to the function SHOW-WHERE-I-AM includes the new directory in the output it displays.

ROOM Function (cont.)

Examples

1. Lisp> (ROOM)

```
Read-Only Storage   Total Size: 4352, Current Allocation: 4113, Free: 5%
Static Storage      Total Size: 2176, Current Allocation: 2146, Free: 1%
Dynamic-0 Storage   Total Size: 3065, Current Allocation: 1292, Free: 58%
```

   Displays a list of the current memory storage information.

2. Lisp> (ROOM T)

```
Read-Only Storage   Total Size: 4352, Current Allocation: 4113, Free: 5%
    (reserved)     0  Functions:   191  Arrays:         0  B-Vectors:     6
    Strings:     381  U-Vectors:  3403  S Flo Vecs:     0  D Flo Vecs:    0
    L Flo Vecs:    0  L Wrd Vecs:    0  Bignums:        1  (reserved)     0
    Sngl Flos:     1  Dbl Flos:      1  Long Flos:      1  Ratios:        0
    Complexes:     0  Symbols:       0  Conses:       128  (reserved)     0
    Ctrl Stack:    0  Bind Stack:    0
Static Storage      Total Size: 2176, Current Allocation: 2146, Free: 1%
    (reserved)     0  Functions:   322  Arrays:         1  B-Vectors:    81
    Strings:     576  U-Vectors:   257  S Flo Vecs:     0  D Flo Vecs:    0
    L Flo Vecs:    0  L Wrd Vecs:    0  Bignums:        1  (reserved)     0
    Sngl Flos:     2  Dbl Flos:      2  Long Flos:      0  Ratios:        0
    Complexes:     0  Symbols:     360  Conses:       544  (reserved)     0
    Ctrl Stack:    0  Bind Stack:    0
Dynamic-0 Storage   Total Size: 3065, Current Allocation: 1280, Free: 58%
    (reserved)     0  Functions:     3  Arrays:         1  B-Vectors:   214
    Strings:     254  U-Vectors:    12  S Flo Vecs:     1  D Flo Vecs:    0
    L Flo Vecs:    0  L Wrd Vecs:    0  Bignums:        3  (reserved)     0
    Sngl Flos:     1  Dbl Flos:      1  Long Flos:      1  Ratios:        0
    Complexes:     0  Symbols:       4  Conses:       656  (reserved)     0
    Ctrl Stack:  129  Bind Stack:   36

Read-Only Storage   Total Size: 4352, Current Allocation: 4113, Free: 5%
```

   Displays a detailed list of the current memory storage information.

ROOM-ALLOCATION Function

Returns multiple values indicating the number of bytes allocated in a specified space and the total number of bytes available in that space. You can obtain space information for dynamic, static, and read-only space. Unlike the ROOM function, the ROOM-ALLOCATION function does not create garbage when used.

Format

ROOM-ALLOCATION &OPTIONAL *space*

Argument

space

One of :DYNAMIC, :STATIC, or :READ-ONLY. The default is :DYNAMIC.

Return Value

Multiple values:

1. A fixnum indicating the number of bytes currently allocated in the specified space.
2. A fixnum indicating the total number of bytes (allocated and free) available in the specified space.

Example

```
Lisp> (ROOM-ALLOCATION)
28672 ;
2038784
Lisp> (ROOM-ALLOCATION :STATIC)
2009088 ;
2042880
```

SET-TERMINAL-MODES Function

Sets the terminal characteristics of the stream bound to the *TERMINAL-IO* variable when you invoke the LISP system. Changes to the stream affect all streams attached to the terminal.

Be careful when you change the settings of terminal modes. A change to terminal modes affects all the streams that are open to the terminal. If you put a stream into pass-through mode, for example, all the streams open to the terminal are put into pass-through mode.

NOTE

> Create an error handle to prevent your terminal from being placed in a nonstandard state. See Section 3.3 for information about how to create an error handler.

Format

```
SET-TERMINAL-MODES
     &KEY :BROADCAST :ECHO :ESCAPE :HALF-DUPLEX
          :PASS-ALL :TYPE-AHEAD :WRAP :PASS-THROUGH
```

Arguments

:BROADCAST

Specifies whether the terminal can receive broadcast messages such as MAIL notifications and REPLY messages. The value can be either T or NIL. If you specify T, the terminal can receive messages; if you specify NIL, the terminal cannot receive messages.

:ECHO

Specifies whether the terminal displays the input characters it receives. The value can be either T or NIL. If you specify T, the terminal displays input characters; if you specify NIL, the terminal displays only data output from the system or from a user application program.

:ESCAPE

Specifies whether ANSI standard escape sequences transmitted from the terminal are handled as a single multicharacter terminator. The value can be either T or NIL. If you specify T, the escape sequences are handled as a single multicharacter terminator. The terminal driver checks the escape sequences for syntax before passing them to the program. For more information on escape sequences, see the *VAX/VMS I/O User's Reference Manual: Part I*.

UNIVERSAL-ERROR-HANDLER

The function to which the VAX LISP system sends all errors that are signaled during program execution. By default, this function is bound to the VAX LISP *UNIVERSAL-ERROR-HANDLER* variable.

The VAX LISP error handler is described in Chapter 3.

Format

```
UNIVERSAL-ERROR-HANDLER function-name
     error-signaling-function &REST args
```

Arguments

function-name

The name of the function that produced or signaled the error.

error-signaling-function

The name of an error-signaling function. Valid function names are ERROR, CERROR, and WARN.

args

The specified error-signaling function's arguments.

Return Value

Invokes the VAX LISP debugger, exits the LISP system, or returns NIL.

Example

```
Lisp> (DEFUN CRITICAL-ERROR-HANDLER (FUNCTION-NAME
                                     ERROR-SIGNALING-FUNCTION
                                     &REST ARGS)
        (WHEN (OR (EQ ERROR-SIGNALING-FUNCTION 'ERROR)
                  (EQ ERROR-SIGNALING-FUNCTION 'CERROR))
              (FLASH-ALARM-LIGHT))
        (APPLY #'UNIVERSAL-ERROR-HANDLER
               FUNCTION-NAME
               ERROR-SIGNALING-FUNCTION
               ARGS))
CRITICAL-ERROR-HANDLER
```

Defines an error handler that checks whether a fatal or continuable error is signaled. If either type of error is signaled, the handler flashes an alarm light and then passes the error signal information to the universal error handler. For information on how to create an error handler, see Section 3.3.

UNIVERSAL-ERROR-HANDLER

Determines the function to be called when an error is signaled. By default, this variable is bound to the VAX LISP error handler, the UNIVERSAL-ERROR-HANDLER function. If you create an error handler you must bind the *UNIVERSAL-ERROR-HANDLER* to it.

Example

```
Lisp> (DEFUN CRITICAL-ERROR-HANDLER (FUNCTION-NAME
                                     ERROR-SIGNALING-FUNCTION
                                     &REST ARGS)
        (WHEN (OR (EQ ERROR-SIGNALING-FUNCTION 'ERROR)
                  (EQ ERROR-SIGNALING-FUNCTION 'CERROR))
              (FLASH-ALARM-LIGHT))
        (APPLY #'UNIVERSAL-ERROR-HANDLER
               FUNCTION-NAME
               ERROR-SIGNALING-FUNCTION
               ARGS))
CRITICAL-ERROR-HANDLER
Lisp> (LET ((*UNIVERSAL-ERROR-HANDLER*
             #'CRITICAL-ERROR-HANDLER))
        (PERFORM-CRITICAL-OPERATION))
```

- The call to the DEFUN macro defines an error handler named CRITICAL-ERROR-HANDLER.
- The call to the LET special form binds the *UNIVERSAL-ERROR-HANDLER* variable to the error handler named CRITICAL-ERROR-HANDLER, while the PERFORM-CRITICAL-OPERATION function is evaluated.

SHORT-SITE-NAME Function

Translates the logical name LISP$SHORT_SITE_NAME.

Format

```
SHORT-SITE-NAME
```

Return Value

The translation of the logical name LISP$SHORT_SITE_NAME is returned as a string. If the logical name is not defined, NIL is returned.

Example

```
Lisp> (SHORT-SITE-NAME)
"Smith's Computer Company"
```

SOFTWARE-VERSION-NUMBER Function

Returns as multiple values the version number of the specified software component.

Format

SOFTWARE-VERSION-NUMBER *component*

Argument

component

A string indicating the software component. Possible values are "VAX LISP", "VMS", and "UIS".

Return Value

Multiple values. For a software version number in the form x.*y*:

1. A fixnum designating x.
2. A fixnum designating *y*.

Example

```
Lisp> (SOFTWARE-VERSION-NUMBER "VAX LISP")
2 ;
2
Lisp>
```

SOURCE-CODE Function

Returns a lambda-expression that is the source code for an interpreted function.

Format

SOURCE-CODE *function*

Argument

function

An interpreted function or a symbol designating an interpreted function.

Return Value

A lambda-expression.

Example

```
Lisp>(DEFUN F (X Y)
       (* (+ X Y) (* X Y)))
F
Lisp>(PPRINT (SYMBOL-FUNCTION 'F))
#<Interpreted Function
   (LAMBDA (X Y) (BLOCK F (* (+ X Y) (* X Y))))
   4980172>
Lisp>(SOURCE-CODE (SYMBOL-FUNCTION 'F))
(LAMBDA (X Y) (BLOCK F (* (+ X Y) (* X Y))))
Lisp>
```

SPAWN Function

Creates a subprocess for executing Command Language Interpreter (CLI) commands. This function causes the LISP system to interrupt execution of a LISP process and to optionally execute the specified CLI command. If you specify the :PARALLEL keyword with a value of T, the LISP process continues to execute while the subprocess is executing. If you do not specify this keyword or if you specify it with NIL, the LISP process is put into a hibernation state until the subprocess completes its execution.

This function is equivalent to the DCL SPAWN command. For more information on the SPAWN command, see the *VAX/VMS DCL Dictionary*.

Format

```
SPAWN
     &KEY :COMMAND-STRING :DCL-SYMBOLS :INPUT-FILE
          :LOGICAL-NAMES :OUTPUT-FILE :PARALLEL
          :PROCESS-NAME
```

Arguments

:COMMAND-STRING

A string that specifies a DCL command the specified subprocess is to process. The value must be a DCL command. By default, the SPAWN function does not process a command.

:DCL-SYMBOLS

Specifies whether the spawned subprocess is to acquire the currently defined CLI symbols from the LISP process. The value can be either T or NIL. If you specify T, the subprocess acquires the CLI symbols. If you specify NIL, the subprocess does not acquire the CLI symbols. The default value is T.

:INPUT-FILE

A pathname, namestring, symbol, or stream that specifies an input file containing one or more DCL commands to be associated with the logical name SYS$INPUT and to be executed by the spawned subprocess. If you specify both a command string and an input file, the command string is processed before the commands in the input file. The subprocess is terminated when processing is complete.

STEP Macro

Invokes the VAX LISP stepper.

The STEP macro evaluates the form that is its argument and returns what the form returns. In the process, you can interactively step through the evaluation of the form. Entering a question mark (?) in response to the stepper prompt displays helpful information. The stepper is command oriented rather than expression oriented - do not surround commands with parentheses. For further information on using the VAX LISP stepper, see Chapter 5.

Format

STEP *form*

Argument

form

A form to be evaluated.

Return Value

The value returned by *form*.

Example

```
Lisp> (STEP (FACTORIAL 3))
: #9: (FACTORIAL 3)
Step >
```

Invokes the VAX LISP stepper for the function call (FACTORIAL 3).

STEP-ENVIRONMENT Variable

The *STEP-ENVIRONMENT* variable, a debugging tool, is bound to the lexical environment in which *STEP-FORM* is being evaluated. By default in the stepper, the lexical environment is used if you use the EVALUATE command. See *COMMON LISP: The Language* for a description of dynamic and lexical environment variables.

Some COMMON LISP functions (for example, EVALHOOK, APPLYHOOK, and MACROEXPAND) take an optional environment argument. The value bound to the *STEP-ENVIRONMENT* variable can be passed as an environment to these functions to allow evaluaton of forms in the context of the stepped form.

Example

```
Step> EVAL *STEP-FORM*
(FIBONACCI (- X 1))
Step> (EVALHOOK 'X NIL NIL NIL)
"Top level value of X"
Step> (EVALHOOK 'X NIL NIL *STEP-ENVIRONMENT*)
3
```

The use of the *STEP-ENVIRONMENT* variable in this call to the EVALHOOK function causes the local value of X to be used in the evaluation of the form (- X 1). See Chapter 5 for the full stepper sessions from which this excerpt is taken.

STEP-FORM Variable

The *STEP-FORM* variable, a debugging tool, is bound to the form being evaluated while stepping. For example, while executing the form

```
(STEP (FUNCTION-Z ARG1 ARG2))
```

the value of *STEP-FORM* is the list (FUNCTION-Z ARG1 ARG2). When not stepping, the value is undefined.

Example

```
Step> STEP
: : : : : : : : #39: X => 4
: : : : : : : #35: => NIL
: : : : : : : #34: (+ FIBONACCI (- X 1)) (FIBONACCI (- X 2)))
Step> STEP
: : : : : : : : #38: (FIBONACCI (- X 1))
Step> EVAL *STEP-FORM*
(FIBONACCI (- X 1))
```

See Chapter 5 for the full stepper session from which this excerpt is taken. In this case, the *STEP-FORM* variable is bound to (FIBONACCI (- X 1)).

SUSPEND Function

Writes information about a LISP system to a file, making it possible to resume the LISP system at a later time. The function does not stop the current system, but copies the state of the LISP system when the function is invoked to the specified file. When you reinvoke the LISP system with the /RESUME qualifier and the file name that was specified with the SUSPEND function, program execution continues from the point where the SUSPEND function was called.

Only the static and dynamic portions of the LISP environment are written to the specified file. When you resume a suspended system, the read-only sections of the LISP environment are taken from LISP$SYSTEM:LISPSUS.SUS. You must make sure that your original LISP system is in LISP$SYSTEM:LISPSUS.SUS; if it is not, you will not be able to resume the system.

When a suspended system is resumed, the LISP environment is identical to the environment that existed when the suspend operation occurred, with the following exceptions:

- All streams except the standard streams are closed.
- The *DEFAULT-PATHNAME-DEFAULTS* variable is set to the current directory.
- Call-out state might be lost (see Chapter 2 of the *VAX LISP/VMS System Acess Programming Guide*).
- Any interrupt functions are uninstated (see Chapter 4 of the *VAX LISP/VMS System Access Programming Guide*). They are not automatically reinstated upon resuming.
- For all workstation-related functions that take an *action* argument, the *action* is reset to the system default state. An *action* that you have established is not automatically reestablished upon resuming.
- Some Editor state is changed (see the *VAX LISP Editor Programming Guide*).
- On a workstation, windows, displays, and display lists are lost.

Format

SUSPEND *pathname*

TRACE Macro (cont.)

Table 7 (cont.)

Keyword-Value Pair	Description
	(directly or indirectly) from within one of the functions or macros specified by the :DURING keyword.

Return Value

A list of the functions currently being traced.

Examples

1. ```
   Lisp> (TRACE FACTORIAL COUNT1 COUNT2)
   (FACTORIAL COUNT1 COUNT2)
   ```

   Enables the tracer for the functions FACTORIAL, COUNT1, and COUNT2.

2. ```
   Lisp> (TRACE)
   (FACTORIAL COUNT1 COUNT2)
   ```

   Returns a list of the functions for which the tracer is enabled.

3. ```
   Lisp> (DEFUN REVERSE-COUNT (N)
           (DECLARE (SPECIAL *GO-INTO-DEBUGGER*))
           (IF (> N 3)
               (SETQ *GO-INTO-DEBUGGER* T)
               (SETQ *GO-INTO-DEBUGGER* NIL))
           (COND ((= N 0) 0)
                 (T (PRINT N) (+ 1 (REVERSE-COUNT (- N 1))))))
   Lisp> (SETQ *GO-INTO-DEBUGGER* NIL)
   NIL
   Lisp> (REVERSE-COUNT 3)
   3
   2
   1
   3
   Lisp> (TRACE (REVERSE-COUNT :DEBUG-IF *GO-INTO-DEBUGGER*))
   (REVERSE-COUNT)
   Lisp> (REVERSE-COUNT 3)
   #4: (REVERSE-COUNT 3)
   3
   . #16: (REVERSE-COUNT 2)
   2
   . . #28: (REVERSE-COUNT 1)
   ```
   ```

TRACE Macro (cont.)

```
1
. . . #40: (REVERSE-COUNT 0)
. . . #40=> 0
. . #28=> 1
. #16=> 2
#4=> 3
3
Lisp> (REVERSE-COUNT 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
Control Stack Debugger
Apply #17: (DEBUG)
Debug 1> CONTINUE
3
. . #28: (REVERSE-COUNT 2)
2
. . . #40: (REVERSE-COUNT 1)
1
. . . . #52: (REVERSE-COUNT 0)
. . . . #52=> 0
. . . #40=> 1
. . #28=> 2
. #16=> 3
#4=> 4
4
Lisp>
```

The recursive function REVERSE-COUNT is defined to count down from the number it is given and to return that number after the function is evaluated. If, however, the number given is greater than 3 (set low to simplify the example), the global variable *GO-INTO-DEBUGGER* (preset to NIL) is set to T.

The first time the REVERSE-COUNT function is traced using the DEBUG-IF keyword, the argument is 3. The second time the function is traced, the argument is over 3. This sets the global variable *GO-INTO-DEBUGGER* to T, which causes the debugger to be invoked during a trace of the REVERSE-COUNT function. The debugger is invoked after the function's argument is evaluated.

To reset the global variable *GO-INTO-DEBUGGER* to NIL, the REVERSE-COUNT function must be completed. So, the evaluation of the function was continued with the Debug command CONTINUE.

4.
```
Lisp> (TRACE (REVERSE-COUNT
               :PRE-DEBUG-IF *GO-INTO-DEBUGGER*))
(REVERSE-COUNT)
```

TRACE Macro (cont.)

```
Lisp> (REVERSE-COUNT 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
Control Stack Debugger
Apply #17:
Debug 1>
```

The 4 argument to the REVERSE-COUNT function causes the *GO-INTO-DEBUGGER* variable to be set to T, which in turn causes the debugger to be invoked before the first recursive call to the REVERSE-COUNT function.

5.

```
Lisp> (TRACE (REVERSE-COUNT
               :POST-DEBUG-IF *GO-INTO-DEBUGGER*))
(REVERSE-COUNT)
Lisp> (REVERSE-COUNT 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
3
. . #28: (REVERSE-COUNT 2)
2
. . . #40: (REVERSE-COUNT 1)
1
. . . . #52: (REVERSE-COUNT 0)
. . . . #52=> 0
. . . #40=> 1
. . #28=> 2
. #16=> 3
#4=> 4
4
Lisp> (TRACE (REVERSE-COUNT
               :POST-DEBUG-IF (NOT *GO-INTO-DEBUGGER*)))
(REVERSE-COUNT)
Lisp> (REVERSE-COUNT 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
3
. . #28: (REVERSE-COUNT 2)
2
. . . #40: (REVERSE-COUNT 1)
1
. . . . #52: (REVERSE-COUNT 0)
Control Stack Debugger
Apply #53: (DEBUG)
Debug 1> CONTINUE

. . . . #52=> 0
```

TRACE Macro (cont.)

```
Control Stack Debugger
Apply #41: (DEBUG)
Debug 1> CONTINUE

. . . #40=> 1
Control Stack Debugger
Apply #29: (DEBUG)
Debug 1> CONTINUE

. . #28=> 2
Control Stack Debugger
Apply #17: (DEBUG)
Debug 1> CONTINUE

. #16=> 3
Control Stack Debugger
Apply #5: (DEBUG)
Debug 1> CONTINUE

#4=> 4
4
Lisp>
```

Here, the first time the REVERSE-COUNT function is evaluated, the debugger is not invoked despite the :POST-DEBUG-IF keyword, because the keyword invokes the debugger only if its condition is met after the function is evaluated. However, after the function is evaluated, the *GO-INTO-DEBUGGER* variable is reset back to NIL. If the form (SETQ *GO-INTO-DEBUGGER* NIL) were removed from the definition of the REVERSE-COUNT function, the variable would not have been reset to NIL, and the debugger would have been invoked.

The second time the REVERSE-COUNT function is invoked, the form (NOT *GO-INTO-DEBUGGER*) evaluates to T, since the value of its argument is NIL. This gives the :POST-DEBUG-IF keyword a T value, which in turn fulfills the condition of invoking the debugger after the function is evaluated.

In this situation, the Debug CONTINUE command causes only one evaluation. Here, the CONTINUE command must be repeated to evaluate all the recursive calls. This example differs from example 1, where the CONTINUE command did not have to be repeated.

6.
```
Lisp> (SETF *L* 5 *M* 6 *N* 7)
7
Lisp> (TRACE (* :PRINT (*L* *M* *N*)))
(*)
Lisp> (+ 2 3 *L* *M* *N*)
```

TRACE Macro (cont.)

```
23
Lisp> (* 2 3 *L* *M* *N*)
#4: (* 2 3 5 6 7)
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
#4=> 1260
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
1260
```

The + function is not traced, but the * function is traced. The values of the global variables *L*, *M*, and *N* are displayed before and after the call to the * function is evaluated.

7.
```
Lisp> (TRACE (* :PRE-PRINT (*L* *M* *N*)))
(*)
Lisp> (* 2 3 *L* *M* *N*)
#4: (* 2 3 5 6 7)
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
#4=> 1260
1260
```

The values of the global variables *L*, *M*, and *N* are displayed before the call to the * function is evaluated.

8.
```
Lisp> (TRACE (* :POST-PRINT (*L* *M* *N*)))
(*)
Lisp> (* 2 3 *L* *M* *N*)
#4: (* 2 3 5 6 7)
#4=> 1260
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
1260
```

The values of the global variables *L*, *M*, and *N* are displayed after the call to the * function is evaluated.

9.
```
Lisp> (TRACE +)
(+)
Lisp> (+ 2 3 (SQUARE 4) (SQRT 25))
#4: (+ 2 3 16 5.0)
#4=> 26.0
26.0
Lisp> (SETQ *STOP-TRACING* T)
```

TRACE Macro (cont.)

```
T
Lisp> (TRACE (+ :SUPPRESS-IF *STOP-TRACING*))
(+)
Lisp> (+ 2 3 (SQUARE 4) (SQRT 25))
26.0
```

In the first example, the call to the + function is traced. In the second example, the call to the + function is not traced because of the form (+ :SUPPRESS-IF *STOP-TRACING*).

10.
```
Lisp> (TRACE (FACTORIAL :STEP-IF T))
(FACTORIAL)
Lisp> (+ (FACTORIAL 2) 3)
#6: (FACTORIAL 2)
#10: (BLOCK FACTORIAL (IF (<= N 1) 1
                          (* N (FACTORIAL (- N 1)))))
Step>
: #15: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step>
: : #20: (<= N 1)
Step>
   .
   .
   .
```

The call to the FACTORIAL function invokes the stepper.

11.
```
Lisp> (TRACE (LIST-LENGTH :DURING PRINT-LENGTH))
(LIST-LENGTH)
Lisp> (PRINT-LENGTH '(CAT DOG PONY))
#13: (LIST-LENGTH (CAT DOG PONY))
#13=> 3

The length of (CAT DOG PONY) is 3.
NIL
```

The PRINT-LENGTH function has been defined to find the length of its argument with the function LIST-LENGTH. The LIST-LENGTH function is traced during the call to the PRINT-LENGTH function.

12.
```
Lisp> (DEFUN FIBONACCI (X)
        (IF (< X 3) 1
            (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))))
FIBONACCI

Lisp> (TRACE (FIBONACCI
               :PRE-DEBUG-IF (< (SECOND *TRACE-CALL*) 2)
               :SUPPRESS-IF T))
(FIBONACCI)
```

TRACE Macro (cont.)

```
Lisp> (FIBONACCI 5)
Control Stack Debugger
Apply #30: (DEBUG)
Debug 1> DOWN
Eval  #27: (FIBONACCI (- X 2))
Debug 1> DOWN
Eval  #26: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
Debug 1> DOWN
Eval  #25: (IF (< X 3) 1
               (+ (FIBONACCI (- X 1))
                  (FIBONACCI (- X 2))))
Debug 1> DOWN
Eval  #24: (BLOCK FIBONACCI
               (IF (< X 3) 1
                   (+ (FIBONACCI (- X 1))
                      (FIBONACCI (- X 2)))))
Debug 1> DOWN
Apply #22: (FIBONACCI 3)
Debug 1> (CADR (DEBUG-CALL))
3
Debug 1> CONTINUE
Control Stack Debugger
Apply #22: (DEBUG)
Debug 1> CONTINUE
5
```

- In this example, FIBONACCI is first defined.
- Then the TRACE macro is called for FIBONACCI. TRACE is specified to invoke the debugger if the first argument to FIBONACCI (the function call being traced) is less than 2. Since the PRE-DEBUG-IF option is specified, the debugger is invoked before the call to FIBONACCI. As the :SUPPRESS-IF option has a value of T, calls to FIBONACCI do not cause any trace output.
- The DOWN command moves the pointer down the control stack.
- The DEBUG-CALL function returns a list representing the current debug frame function call. In this case, the CADR of the list is 3. This accesses the first argument to the function in the current stack frame.
- Finally the CONTINUE command continues the evaluation of FIBONACCI.

13.
```
Lisp> (TRACE (FIBONACCI
               :POST-DEBUG-IF (> (FIRST *TRACE-VALUES*) 2)))
```

TRACE Macro (cont.)

```
(FIBONACCI)
Lisp> (FIBONACCI 5)
#5: (FIBONACCI 5)
. #13: (FIBONACCI 4)
. . #21: (FIBONACCI 3)
. . . #29: (FIBONACCI 2)
. . . #29=> 1
. . . #29: (FIBONACCI 1)
. . . #29=> 1
. . #21=> 2
. . #21: (FIBONACCI 2)
. . #21=> 1
Control Stack Debugger
Apply #14: (DEBUG)
Debug 1> BACKTRACE
-- Backtrace start --
Apply #14: (DEBUG)
Eval  #11: (FIBONACCI (- X 1))
Eval  #10: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
Eval  #9: (IF (< X 3) 1
              (+ (FIBONACCI (- X 1))
                 (FIBONACCI (- X 2))))
Eval  #8: (BLOCK FIBONACCI
             (IF (< X 3) 1
                 (+ (FIBONACCI (- X 1))
                    (FIBONACCI (- X 2)))))
Apply #6: (FIBONACCI 5)
Eval  #3: (FIBONACCI 5)
Apply #1: (EVAL (FIBONACCI 5))
-- Backtrace end --
Apply #14: (DEBUG)
Debug 1> CONTINUE
. #13=> 3
. #13: (FIBONACCI 3)
. . #21: (FIBONACCI 2)
. . #21=> 1
. . #21: (FIBONACCI 1)
. . #21=> 1
. #13=> 2
Control Stack Debugger
Apply #6: (DEBUG)
Debug 1> CONTINUE
#5=> 5
5
```

TRACE is called for FIBONACCI (the same function as in the previous example) to start the debugger if the value returned exceeds 2. The value returned exceeds 2 twice -- once when it returns 3 and at the end when it returns 5.

TRACE-CALL Variable

The *TRACE-CALL* variable, a debugging tool, is bound to the function or macro call being traced.

Examples

1. ```
   Lisp> (TRACE (FIBONACCI
                  :SUPPRESS-IF (> (SECOND *TRACE-CALL*) 1)))
   ```

   This causes FIBONACCI to be traced only if its first argument is 1 or less.

2. ```
   Lisp> (TRACE (FIBONACCI
                  :SUPPRESS-IF (<= (LENGTH *TRACE-CALL*) 2)))
   ```

   This causes FIBONACCI to be traced if it is called with more than 1 argument.

3. ```
   Lisp> (TRACE (FIBONACCI
                  :PREDEBUG-IF (< (SECOND *TRACE-CALL*) 2)
                  :SUPPRESS-IF (< (SECOND *TRACE-CALL*) 2)))
   FIBONACCI
   ```

   In this case, the TRACE macro is enabled for FIBONACCI. The debugger will be invoked and tracing suppressed if the first argument to FIBONACCI (the SECOND of the value of the *TRACE-CALL* variable) is less than 2. So, for example, if FIBONACCI is called with the arguments 3 and 5, *TRACE-CALL* is bound to the form (FIBONACCI 3 5); as 3 is greater than 2, the call is traced and the debugger not entered. See the description of the TRACE macro for further examples of the use of *TRACE-CALL*.
   ```

TRACE-VALUES Variable

The *TRACE-VALUES* variable, a debugging tool, is bound to the list of values returned by the traced function. You can use the value bound to this variable in the forms used with the trace option keywords such as :DEBUG-IF.

Example

```
Lisp> (FACTORIAL 4)
#4: (FACTORIAL 4)
. #11: (FACTORIAL 3)
. . #18: (FACTORIAL 2)
. . . #25: (FACTORIAL 1)
. . . #25=> 1
. . . #25=> *TRACE-VALUES* is (1)
. . #18=> 2
. . #18=> *TRACE-VALUES* is (2)
. #11=> 6
. #11=> *TRACE-VALUES* is (6)
#4=> 24
#4=> *TRACE-VALUES* is (24)
24
```

In this case, the values returned by the FACTORIAL function and bound to the *TRACE-VALUES* variable are displayed as (1), (2), (6), and (24). Since the *TRACE-VALUES* variable is bound to the list of values returned by a function, it can be used only in the :POST- options to the TRACE macro. Before being bound to the return values, it returns NIL. See the description of the TRACE macro for further examples of the use of the *TRACE-VALUES* variable.

INDEX

Page numbers in the Index in the form c-n (for example, 2-13) refer to a page in Part I. Page numbers in the form n (for example, 25) refer to a page in Part II.

?
  debugger command
    description, 5-13
    (table), 5-10
  stepper command
    description, 5-26
    (table), 5-25

-A-

Abbreviating output by lines, 6-25
Abbreviating output depth, 6-24
Abbreviating output length, 6-24
Abbreviating printed output, 6-23
Access control string, 7-10, 7-15
:ACCOUNT keyword
  GET-PROCESS-INFORMATION function, 65
:ACP-PID keyword
  GET-DEVICE-INFORMATION function, 51
:ACP-TYPE keyword
  GET-DEVICE-INFORMATION function, 51
"Activate Minor Style" Editor command
  using, B-3
Active stack frame, 5-4
:ACTIVE-PAGE-TABLE-COUNT keyword
  GET-PROCESS-INFORMATION function, 65
Alien structure facility, 1-5
ALL debugger command modifier, 5-12
  with BACKTRACE command, 5-17
  with BOTTOM command, 5-15
  with DOWN command, 5-15
  with TOP command, 5-15
  with UP command, 5-16
:ALL keyword
  TRANSLATE-LOGICAL-NAME function, 137
:ALLOCATION keyword
  MAKE-ARRAY function, 7-18, 86
:ALLOCATION-QUANTITY keyword
  GET-FILE-INFORMATION function, 55
Alternatives
  Editor prompt input, 3-9
    files, 3-9
Anchored windows, 3-31
APROPOS function
  debugging information, 5-1
  description, 1
  help, 1-8
  (table), 7-30
"Apropos" Editor command, 3-13
  using, 3-7
APROPOS-LIST function
  debugging information, 5-1
  description, 3
  (table), 7-30
ARGUMENTS debugger command modifier, 5-12
  with SET command, 5-16
  with SHOW command, 5-17
ARRAY-DIMENSION-LIMIT constant, 7-7
ARRAY-RANK-LIMIT constant, 7-7
ARRAY-TOTAL-SIZE-LIMIT constant, 7-7
Arrays, 7-7
  constants, 7-7
  creating, 86
  specialized, 7-7, 86
Arrow keys
  Editor usage, 3-17
  specifying in BIND-COMMAND function, 3-40
:AST-ACTIVE keyword
  GET-PROCESS-INFORMATION function, 65
:AST-COUNT keyword
  GET-PROCESS-INFORMATION function, 65

:AST-ENABLED keyword
  GET-PROCESS-INFORMATION
    function, 66
:AST-QUOTA keyword
  GET-PROCESS-INFORMATION
    function, 66
ATTACH function
  description, 4
:AUTHORIZED-PRIVILEGES keyword
  GET-PROCESS-INFORMATION
    function, 66

-B-

BACKTRACE
  debugger command
    description, 5-17
    (table), 5-10
  stepper command
    description, 5-27
    (table), 5-24
:BACKUP-DATE keyword
  GET-FILE-INFORMATION function,
    55
"Backward Character" Editor
    command, 3-25
  "EMACS" style binding, B-4
"Backward Word" Editor command
  "EMACS" style binding, B-4
:BASE-PRIORITY keyword
  GET-PROCESS-INFORMATION
    function, 66
:BATCH keyword
  GET-PROCESS-INFORMATION
    function, 66
"Beginning of Buffer" Editor
    command, 3-26
  "EMACS" style binding, B-5
"Beginning of Line" Editor
    command
  "EMACS" style binding, B-4
"Beginning of Outermost Form"
    Editor command, 3-27
"Beginning of Paragraph" Editor
    command
  "EMACS" style binding, B-4
"Beginning of Window" Editor
    command
  "EMACS" style binding, B-5
"Bind Command" Editor command,
    3-46
  specifying context, 3-40
"Bind Command" Editor command
    (Cont.)
  specifying keys, 3-39
  using, 3-39
BIND-COMMAND function
  specifying context, 3-42
  specifying keys, 3-40
  using, 3-40
BIND-KEYBOARD-FUNCTION
  and ED, 3-6
BIND-KEYBOARD-FUNCTION function
  description, 6
  garbage collector, 7-19
  interrupt functions, 7-25
  invoking the break loop, 5-5
Binding stack, 105
:BIO-BYTE-COUNT keyword
  GET-PROCESS-INFORMATION
    function, 66
:BIO-BYTE-QUOTA keyword
  GET-PROCESS-INFORMATION
    function, 7-23, 66
:BIO-COUNT keyword
  GET-PROCESS-INFORMATION
    function, 66
:BIO-OPERATIONS keyword
  GET-PROCESS-INFORMATION
    function, 66
:BIO-QUOTA keyword
  GET-PROCESS-INFORMATION
    function, 66
Bits attribute, 7-5
:BLOCK-SIZE keyword
  GET-FILE-INFORMATION function,
    55
BOTTOM debugger command
  description, 5-15
  (table), 5-10
BREAK function, 20
  binding control character to, 6
  debugging information, 5-1
  description, 9
  invoking the break loop, 5-5
  (table), 7-30
Break loop, 1-5, 5-4 to 5-7
  exiting, 5-5, 9, 20
  invoking, 5-5, 9
  message, 5-5
  prompt, 5-5
  using, 5-6
  variables, 5-7

BREAK-ON-WARNINGS variable, 5-14
  defining an error handler, 4-6
  WARN function, 144
:BROADCAST keyword
  GET-TERMINAL-MODES function, 73
  SET-TERMINAL-MODES function, 108.1
BROADCAST-STREAM data type, 8-6
BROADCAST-STREAM-P function, 8-6
:BUFFER-SIZE keyword
  GET-DEVICE-INFORMATION function, 51
Buffers
  Editor
    see Editor buffers

-C-

CALL debugger command modifier, 5-12
  with SHOW command, 5-17
Call-out facility, 1-5
Cancel character, 10
CANCEL-CHARACTER-TAG tag
  description, 10
"Capitalize Word" Editor command, 3-29
  "EMACS" style binding, B-5
CERROR function, 142
  defining an error handler, 4-7
  error messages, 4-3
CHAR-BITS-LIMIT constant, 7-6
CHAR-CODE-LIMIT constant, 7-6
CHAR-FONT-LIMIT constant, 7-6
CHAR-NAME-TABLE function, 7-6
  description, 11
Characters, 7-5
  attributes, 7-5
  changing case with editor, 3-21
  comparisons, 7-6
  constants, 7-6
  names, 11
  nongraphic
    Editor representation, 3-16
    inserting with Editor, 3-16
    specifying in "Bind Command", 3-39
Checkpoint file, 3-37
:CLEAR-INPUT
  I/O request specifier, 8-4
:CLEAR-OUTPUT
  I/O request specifier, 8-4
:CLOSE
  I/O request specifier, 8-4
"Close Outermost Form" Editor command, 3-24
:CLUSTER-SIZE keyword
  GET-DEVICE-INFORMATION function, 51
Code attribute, 7-5
Command Language Interpreter (CLI) commands, 112.2
Command levels, 121
  debugger, 5-8
  stepper, 5-27
  tracer, 5-34
Command modifiers
  See Debugger
:COMMAND-STRING keyword
  SPAWN function, 112.2
Commands
  Editor
    see Editor commands
Comments
  LISP
    Inserting with Editor, 3-16
COMMON LISP, 1-2
  VAX LISP I/O extensions to, 8-1
COMPILE function, 1-4, 13, 140
  compiler restrictions, 7-26
  compiling functions and macros, 2-7
/COMPILE qualifier, 1-3
  compiling files, 2-7
  description, 2-13
  modes, 2-13
  optimizing compiler, 7-27
  (table), 2-10
  with /ERROR_ACTION qualifier, 2-14
  with /INITIALIZE qualifier, 2-15
  with /LIST qualifier, 2-17
  with /MACHINE_CODE qualifier, 2-18
  with /NOOUTPUT_FILE qualifier, 2-20
  with /OPTIMIZE qualifier, 2-20
  with /OUTPUT_FILE qualifier, 2-20
  with /VERBOSE qualifier, 2-21
  with /WARNINGS qualifier, 2-23

COMPILE-FILE function, 1-4, 17, 18
  compiler restrictions, 7-27
  compiling files, 2-7
  description, 14 to 16
  (table), 7-30
COMPILE-VERBOSE variable
  default for :VERBOSE keyword, 15
  description, 17
COMPILE-WARNINGS variable
  default for :WARNINGS keyword, 15
  description, 18
COMPILEDP function
  description, 13
Compiler, 1-4, 7-26 to 7-29
  optimizations, 2-19, 7-27 to 7-29, 14
    fast code, 7-28
    safe code, 7-28
  restrictions, 7-26
    COMPILE function, 7-26
    COMPILE-FILE function, 7-27
Completion
  Editor prompt input, 3-8
    files, 3-9
CONCATENATED-STREAM data type, 8-6
CONCATENATED-STREAM-P function, 8-6
Conditional new line directives, 6-8
Constructor function
  allocating static space, 7-18
CONTINUE
  DCL command, 1-11
  debugger command
    description, 5-14
    (table), 5-10
  function
    description, 20
    exiting the break loop, 5-5, 9
Control characters
  binding to functions, 7-25, 6
  Editor representation, 3-16
  inserting with Editor, 3-16
  returning information about bindings, 7-26, 64
  specifying in "Bind Command", 3-39
Control characters (Cont.)
  specifying in BIND-COMMAND function, 3-40
  (table), 2-4
  unbinding from functions, 7-26, 139
Control stack, 5-3
  debugger, 5-7
  overflow, 7-19
  stack frame
    See Stack frame
  storage allocation, 105
Controlling indentation, 6-13
Controlling margins, 6-4
Controlling where new lines begin, 6-11
CPU time
  displaying, 122
  garbage collector, 61
  getting, 63
:CPU-LIMIT keyword
  GET-PROCESS-INFORMATION function, 66
:CPU-TIME keyword
  GET-PROCESS-INFORMATION function, 67
:CREATION-DATE keyword
  GET-FILE-INFORMATION function, 55
CTRL/C
  and CANCEL-CHARACTER-TAG, 10
  prohibition in Editor key binding, 3-43
  recovering from an error, 2-4
  to cancel Editor command, 3-7
CTRL/O, 2-4
CTRL/Q, 2-4
  prohibition in Editor key binding, 3-43
CTRL/R, 2-4
CTRL/S, 2-4
  prohibition in Editor key binding, 3-43
CTRL/T, 2-4
CTRL/U, 1-11, 2-4
CTRL/X, 2-4
CTRL/Y, 1-11, 2-4
Current direction
  Editor, 3-17
:CURRENT keyword
  THROW-TO-COMMAND-LEVEL function, 121

Current package, 92
Current stack frame, 5-7
:CURRENT-PRIORITY keyword
  GET-PROCESS-INFORMATION
      function, 67
:CURRENT-PRIVILEGES keyword
  GET-PROCESS-INFORMATION
      function, 67
:CYLINDERS keyword
  GET-DEVICE-INFORMATION function,
      51

-D-

Data
  representation, 7-2 to 7-7
  structure, 1-1
Data types
  arrays, 7-7, 86
    constants, 7-7
    specialized, 7-7
  characters, 7-5
    attributes, 7-5
    comparisons, 7-6
    constants, 7-6
    names, 11
  floating-point numbers, 7-3
    constants, 7-5
  integers, 7-3
    constants, 7-3
  numbers, 7-2
  package, 3
  packages, 1
  pathnames, 37
    See Pathnames
  strings, 7-7, 86
  vectors, 86
DCL commands
  CONTINUE, 1-11
  entering, 1-10
  LISP, 1-3, 2-1
  STOP, 1-11
:DCL-SYMBOLS keyword
  SPAWN function, 112.2
DEBUG
  function
    debugging information, 5-1
    description, 21
    invoking the debugger, 5-8
  stepper command
    description, 5-26
    (table), 5-24
DEBUG function
  binding control character to, 6
:DEBUG keyword
  See *ERROR-ACTION* variable
DEBUG-CALL
  function, 5-18
    description, 22
:DEBUG-IF keyword
  TRACE macro, 5-36, 125
DEBUG-IO variable
  debugger, 5-8
  stepper, 5-20
DEBUG-PRINT-LENGTH variable
  controlling output, 5-3
  description, 23
DEBUG-PRINT-LEVEL variable
  controlling output, 5-3
  description, 24
Debugger, 1-5, 5-7 to 5-20
  commands
    arguments, 5-11
      entering, 5-11
    descriptions, 5-13 to 5-17
    modifiers (table), 5-12
    (table), 5-10
  controlling output, 23, 24
  error handler, 4-2 to 4-4
  exiting, 5-9, 5-14
  invoking, 5-8, 5-26, 5-36, 21,
      125
  prompt, 5-8
  sample sessions, 5-18
  using, 5-10
Debugging facilities, 1-5
  See also Break loop, Debugger,
      Stepper, Tracer, Editor
Debugging functions and macros
    (table), 5-1
Declarations, 7-28
DECnet-VAX
  network operations, 7-14
Default directory
  changing, 25
DEFAULT-DIRECTORY function, 25
  See also
      DEFAULT-PATHNAME-DEFAULTS
      variable
  description, 25
:DEFAULT-EXTENSION keyword
  GET-FILE-INFORMATION function,
      55

:DEFAULT-PAGE-FAULT-CLUSTER keyword
  GET-PROCESS-INFORMATION function, 67
DEFAULT-PATHNAME-DEFAULTS variable
  default directory, 25
  DIRECTORY function, 7-16, 37
  filling file specification components, 14
  resuming a suspended system, 118
  using, 7-16, 7-17
:DEFAULT-PRIVILEGES keyword
  GET-PROCESS-INFORMATION function, 67
DEFINE-ALIEN-STRUCTURE macro
  allocating static space, 7-18
DEFINE-FORMAT-DIRECTIVE macro
  description, 27
DEFINE-GENERALIZED-PRINT -FUNCTION
  macro, 6-21
DEFINE-GENERALIZED-PRINT-FUNCTION macro
  description, 30
DEFINE-LIST-PRINT- FUNCTION
  macro, 6-19
DEFINE-LIST-PRINT-FUNCTION macro
  description, 32
Defining list-print functions, 6-19
DEFMACRO macro
  creating programs, 2-5
DEFUN macro
  creating programs, 2-5
"Delete Current Buffer" Editor command, 3-36
  "EMACS" style binding, B-6
  using, 3-34
DELETE key, 2-4
"Delete Named Buffer" Editor command, 3-36
  using, 3-34
"Delete Next Character" Editor command
  "EMACS" style binding, B-5
"Delete Next Word" Editor command
  "EMACS" style binding, B-5
"Delete Previous Character" Editor command
  "EMACS" style binding, B-5
"Delete Previous Word" Editor command
  "EMACS" style binding, B-5
"Delete Whitespace" Editor command
  "EMACS" style binding, B-5
DELETE-PACKAGE
  function
    description, 34
DESCRIBE function
  debugging information, 5-1
  description, 35
  help, 1-8
  invoking from Editor, 3-8
    using pointer, 3-49
  (table), 7-30
"Describe Word" Editor command, 3-13
"Describe" Editor command, 3-13
  using, 3-7
Device, 1-8
  getting information, 51
:DEVICE keyword
  pathname field, 7-10
:DEVICE-CHARACTERISTICS keyword
  GET-DEVICE-INFORMATION function, 52
:DEVICE-CLASS keyword
  GET-DEVICE-INFORMATION function, 52
:DEVICE-DEPENDENT-0 keyword
  GET-DEVICE-INFORMATION function, 52
:DEVICE-DEPENDENT-1 keyword
  GET-DEVICE-INFORMATION function, 52
:DEVICE-NAME keyword
  GET-DEVICE-INFORMATION function, 52
:DEVICE-TYPE keyword
  GET-DEVICE-INFORMATION function, 52
:DIO-COUNT keyword
  GET-PROCESS-INFORMATION function, 67
:DIO-OPERATIONS keyword
  GET-PROCESS-INFORMATION function, 67
:DIO-QUOTA keyword
  GET-PROCESS-INFORMATION function, 67

:DIRECTION keyword
  OPEN function, 7-24
~! directive, 6-6
~% directive, 6-11
~& directive, 6-11
~. directive, 6-6
~:_ directive, 6-11
~@_ directive, 6-11
~^ directive, 6-28
~_ directive, 6-6, 6-11
Directives for handling lists, 6-16
Directory, 1-8
DIRECTORY function
  description, 37
  pathnames, 7-16
  (table), 7-30
:DIRECTORY keyword
  pathname field, 7-10
DISPATCH-FUNCTION slot
  STREAM structure, 8-2
DO-ALL-SYMBOLS macro, 1, 3
DO-SYMBOLS macro, 1, 3
Documentation string, 35
DOES-INPUT-P slot
  STREAM structure, 8-2
DOES-OUTPUT-P slot
  STREAM structure, 8-2
Double floating-point numbers, 7-3
DOUBLE-FLOAT-EPSILON constant, 7-5
DOUBLE-FLOAT-NEGATIVE-EPSILON constant, 7-5
DOWN
  debugger command
    description, 5-15
    (table), 5-10
  debugger command modifier, 5-12
    with SEARCH command, 5-15
"Downcase Region" Editor command, 3-29
"Downcase Word" Editor command, 3-29
  "EMACS" style binding, B-5
DRIBBLE function
  debugging information, 5-2
  description, 40
  (table), 7-30
DRIBBLE-STREAM data type, 8-6
DRIBBLE-STREAM-P function, 8-6
:DURING keyword
  TRACE macro, 5-37, 126
Dynamic memory, 2-18, 105, 118
  garbage collector, 7-18, 7-19

-E-

:ECHO keyword
  GET-TERMINAL-MODES function, 73
  SET-TERMINAL-MODES function, 108.1
ECHO-STREAM data type, 8-6
ECHO-STREAM-INPUT-STREAM function, 8-6
ECHO-STREAM-OUTPUT-STREAM function, 8-6
ECHO-STREAM-P function, 8-6
ED function
  and BIND-KEYBOARD-FUNCTION, 3-6
  binding control character to, 6
  debugging information, 5-2
  description, 41
  resuming Editor with, 3-5
  starting Editor with, 3-3
  (table), 7-30
"Ed" Editor command, 3-36
  "EMACS" style binding, B-6
  using, 3-33
"Edit File" Editor command, 3-37
  "EMACS" style binding, B-6
  using, 3-33
Editing keys
  specifying in BIND-COMMAND function, 3-40
Editor, 1-4
  checkpointing, 3-37
  checkpointing file
    file type, 1-10
  copying text, 3-20, 3-21
  creating programs, 2-5
  cursor movement, 3-17
    by LISP entities, 3-18
    current direction, 3-17
    moving by lines, 3-17
    moving by words, 3-17
    searching, 3-18
    using pointer, 3-48
  customizing, 3-38
  debugging facility, 5-40
  errors while using, 3-9
  exiting, 3-11

Editor
  exiting (Cont.)
    by deleting VAXstation window, 3-47
  getting help, 3-7
  help window, 3-7
    removing, 3-7
    scrolling, 3-7
  information area, 3-5
  invoking, 3-3, 41
  invoking with control character, 6
  keyboard macros, 3-45
  label strip, 3-4
  loading files, 2-6
  modifying function and macro definitions, 2-7
  moving text, 3-20
    using pointer, 3-48
  overview of operation, 3-3
  pausing, 3-10
    on VAXstation, 3-47
  protection against work loss, 3-37
  refreshing the screen, 3-9
  repeating operations, 3-23
  restoring deleted text, 3-20
  resuming, 3-5
  saving work, 3-10
  searching, 3-18
  substituting in text, 3-22
  table of commands, C-2
  text deletion, 3-19
    by characters, 3-19
    by lines, 3-20
    by words, 3-19
  text insertion, 3-14
  typing LISP code, 3-15
  undeleting text, 3-20
  using on VAXstation, 3-46
    editing with pointer, 3-47
Editor buffers, 3-30
  as context, 3-44
  creating, 3-30
    from within Editor, 3-33
  current buffer, 3-30
    changing, 3-31
  deleting, 3-34
  displaying more than two, 3-35
  "General Prompting", C-14
  information maintained by, 3-33
Editor buffers (Cont.)
  inserting into other buffers, 3-23
  listing, 3-31
  moving text between, 3-36
  moving to endpoints, 3-18
  name conflicts, 3-34
  saving contents, 3-34
  selecting, 3-32
Editor commands, 3-6
  binding keys to, 3-38
    conflicts in "EMACS" style, B-2
    from LISP interpreter, 3-40
    key binding shadowing, 3-44
    multiple bindings, C-14
    table of bindings, C-2
    table of bindings by key, C-14
    within Editor, 3-39
  buffer and window
    summary, 3-36
  cancelling, 3-7
  capturing sequences of, 3-45
  creating
    with "Start Named Keyboard Macro", 3-45
  customizing
    summary, 3-46
  descriptions, C-2
  editing
    summary, 3-24
  general-purpose
    summary, 3-11
  invoking with keys, 3-6
  issuing, 3-6
  repeating, 3-23
  typing, 3-6
Editor context
  buffer, 3-44
  effect on key bindings, 3-44
  effect on keyboard macro execution, 3-46
  global, 3-44
  order of search, 3-45
  specifying
    in "Bind Command", 3-40
  styles, 3-44
Editor styles, 3-44
  as context, 3-44
  major style, 3-44
  minor style, 3-44

Editor styles (Cont.)
  order of search, 3-45
Editor windows, 3-30
  anchored windows, 3-31
  changing size, 3-35
  creating, 3-30
  current window, 3-30
    changing, 3-31
    changing with pointer, 3-48
    indicated by pointer cursor, 3-47
  floating windows, 3-33
  noncurrent window
    indicated by pointer cursor, 3-47
  removing, 3-32
    with pointer, 3-48
  scrolling text in, 3-18
  splitting, 3-35
"EDT Append" Editor command, 3-28
"EDT Back to Start of Line" Editor command, 3-26
"EDT Beginning of Line" Editor command, 3-26
"EDT Change Case" Editor command, 3-29
"EDT Cut" Editor command, 3-28
"EDT Delete Character" Editor command, 3-27
"EDT Delete Line" Editor command, 3-27
"EDT Delete Previous Character" Editor command, 3-27
"EDT Delete Previous Line" Editor command, 3-28
"EDT Delete Previous Word" Editor command, 3-27
"EDT Delete to End of Line" Editor command, 3-28
"EDT Delete Word" Editor command, 3-27
"EDT Emulation" Editor style, 3-44
"EDT End of Line" Editor command, 3-26
"EDT Move Character" Editor command, 3-25
"EDT Move Page" Editor command, 3-26
"EDT Move Word" Editor command, 3-25
"EDT Paste" Editor command, 3-28
"EDT Query Search" Editor command, 3-26
"EDT Replace" Editor command, 3-29
"EDT Scroll Window" Editor command, 3-26
"EDT Search Again" Editor command, 3-27
"EDT Set Direction Backward" Editor command, 3-25
"EDT Set Direction Forward" Editor command, 3-25
"EDT Special Insert" Editor command, 3-25
"EDT Substitute" Editor command, 3-29
"EDT Undelete Character" Editor command, 3-28
"EDT Undelete Line" Editor command, 3-28
"EDT Undelete Word" Editor command, 3-28
:ELEMENT-TYPE
  I/O request specifier, 8-4
:ELEMENT-TYPE keyword
  OPEN function, 7-23, 7-24
"EMACS Backward Search" Editor command
  "EMACS" style binding, B-5
"EMACS Forward Search" Editor command
  "EMACS" style binding, B-5
"EMACS" Editor style, B-1
  activating, B-3
    as major style, B-4
    as minor style, B-3
  key binding conflicts, B-2
  key bindings, B-4
Enabling pretty printing, 6-3
"End Keyboard Macro" Editor command, 3-46
"End of Buffer" Editor command, 3-26
  "EMACS" style binding, B-5
"End of Line" Editor command
  "EMACS" style binding, B-4
"End of Outermost Form" Editor command, 3-27
"End of Paragraph" Editor command
  "EMACS" style binding, B-4
"End of Window" Editor command
  "EMACS" style binding, B-5

End-of-file operations, 7-22
:END-OF-FILE-BLOCK keyword
  GET-FILE-INFORMATION function, 56
END-OF-FILE-BLOCK keyword
  GET-FILE-INFORMATION function, 7-23
ENLARGE-BINDING-STACK function
  description, 42.1
ENLARGE-CONTROL-STACK function
  description, 42.2
:ENQUEUE-COUNT keyword
  GET-PROCESS-INFORMATION function, 67
:ENQUEUE-QUOTA keyword
  GET-PROCESS-INFORMATION function, 67
EQ function, 7-3
EQUAL function, 7-13
ERROR
  debugger command
    description, 5-16
    (table), 5-10
  function, 142
    defining an error handler, 4-7
    error messages, 4-2
Error
  listing
    file type, 1-10
  messages
    compiler, 18
    debugger, 5-16
    error handler, 100
    error-handler definition, 4-6
    format, 4-2
    warnings, 2-23, 18
  types, 4-2 to 4-5
    continuable, 4-3
    fatal, 4-2
    warning, 4-4, 144
Error handler, 1-4, 43
  binding *UNIVERSAL-ERROR-HANDLER* variable, 4-7
  creating, 143
  debugging information, 5-1
  defining, 4-5
  description, 4-1
  error message, 100
  invoking, 144
  UNIVERSAL-ERROR-HANDLER function, 142
:ERROR keyword
  EXIT function, 44
Error messages
  Editor, 3-9
ERROR-ACTION variable, 43
  See also /ERROR_ACTION qualifier
  continuable error, 4-3
  defining an error handler, 4-6
  description, 43
  fatal error, 4-3
  WARN function, 144
  warning, 4-4
:ERROR-COUNT keyword
  GET-DEVICE-INFORMATION function, 52
ERROR-OUTPUT variable
  PRINT-SIGNALED-ERROR function, 100
Error-signaling functions, 142
  (table), 4-7
/ERROR_ACTION qualifier, 2-14
  See also *ERROR-ACTION* variable
  description, 2-14
  fatal error, 4-3
  modes, 2-13
  (table), 2-10
  with /INITIALIZE qualifier, 2-15
Errors
  Editor protection against file loss, 3-37
  while using Editor, 3-9
ESCAPE character
  transmitting, 3-43
:ESCAPE keyword
  GET-TERMINAL-MODES function, 73
  SET-TERMINAL-MODES function, 108.1
EVAL function, 1-1
EVALUATE
  debugger command
    description, 5-13
    (table), 5-10
  stepper command
    description, 5-26
    (table), 5-24
"Evaluate LISP Region" Editor command, 3-12

:EVENT-FLAG-WAIT-MASK keyword
  GET-PROCESS-INFORMATION
      function, 67
"Exchange Point and Select Mark"
    Editor command
  "EMACS" style binding, B-5
"Execute Keyboard Macro" Editor
    command, 3-46
  "EMACS" style binding, B-6
"Execute Named Command" Editor
    command, 3-11
  "EMACS" style binding, B-6
EXIT function
  description, 44
  exiting LISP, 2-2
:EXIT keyword
  See *ERROR-ACTION* variable
"Exit Recursive Edit" Editor
    command, 3-29
  "EMACS" style binding, B-6
"Exit" Editor command
  using, 3-11, 3-34
:EXPIRATION-DATE keyword
  GET-FILE-INFORMATION function,
      56
Extended attribute block (XAB),
    55
Extensions to the FORMAT function,
    6-5 to 6-17

-F-

Fast-loading file, 2-8, 2-13
  file type, 1-10
  loading, 81
  locating, 81
  producing, 14, 15
File
  compiling, 2-7
  getting information, 55
  loading, 2-6
  name, 1-9
    representation, 7-9
  organization, 7-23
  specification
    See also Pathnames,
        Namestrings
    components, 1-8 to 1-9
    defaults (table), 1-10
    format, 1-8
  type, 1-9
  version number 1-9
File access block (FAB), 55
File name representation
  See File
FILE-LENGTH function, 7-23
FILE-POSITION function, 7-24
FILE-STREAM data type, 8-6
FILE-STREAM-P function, 8-6
Files
  creating
    from Editor, 3-10
  editing with Editor, 3-4
    saving edited version, 3-10
  Editor checkpoint file, 3-37
  Editor input completion, 3-9
  Editor protection against loss,
      3-37
  inserting in Editor buffer,
      3-23
~/FILL directive, 6-6
FINISH stepper command
  description, 5-27
  (table), 5-25
:FINISH-OUTPUT
  I/O request specifier, 8-4
:FIRST-FREE-BYTE keyword
  GET-FILE-INFORMATION function,
      7-23, 56
:FIRST-FREE-P0-PAGE keyword
  GET-PROCESS-INFORMATION
      function, 67
:FIRST-FREE-P1-PAGE keyword
  GET-PROCESS-INFORMATION
      function, 67
:FIXED-CONTROL-SIZE keyword
  GET-FILE-INFORMATION function,
      56
Floating windows, 3-33
Floating-point numbers, 7-3
  constants (table), 7-5
  (table), 7-4
Font attribute, 7-5
:FORCE-OUTPUT
  I/O request specifier, 8-4
FORMAT
  function, 6-5 to 6-17
FORMAT directives
  user defined, 6-18
FORMAT directives in VAX LISP,
    6-6
Format Directives Provided with
    VAX LISP, 45

FORMAT function
  break-loop messages, 9
  error messages, 4-7
  warning messages, 144
"Forward Character" Editor
    command, 3-25
  "EMACS" style binding, B-4
"Forward Word" Editor command
  "EMACS" style binding, B-4
:FREE-BLOCKS keyword
  GET-DEVICE-INFORMATION function,
      52
Fresh line directive, 6-11
:FRESH-LINE
  I/O request specifier, 8-4
Function
  compiled, 13
  compiling, 2-7
  defining, 2-5
  definition
    editing, 140
    pretty printing, 90
  implementation-dependent
      (table), 7-30
  interpreted, 13
  interrupt, 7-19, 7-25
    garbage collector, 7-25
    suspended systems, 7-26
  keyboard
    creating, 6
  modifying, 2-7
FUNCTION debugger command
    modifier, 5-12
  with SET command, 5-16
  with SHOW command, 5-17
Function keys
  specifying
    in "Bind Command", 3-39
  specifying in BIND-COMMAND
      function, 3-40
:FUNCTION keyword
  ED function, 42
Functions
  editing definition, 3-3
    moving back to LISP, 3-10
  evaluating
    in Editor, 3-10

-G-

Garbage collector, 7-17 to 7-19
  available space, 7-19
Garbage collector (Cont.)
  changing messages, 7-19
  control stack overflow, 7-19
  CPU time, 61
  displaying time, 122
  dynamic memory, 7-18, 7-19
  elapsed time, 59
  failure, 7-19
  frequency of use, 7-18
  interrupt functions, 7-19, 7-25
  invoking, 48
  message, 95
    See also *POST-GC-MESSAGE*
        variable
  messages, 49, 89
    See also *PRE-GC-MESSAGE*
        variable, *POST-GC
        -MESSAGE* variable
  run-time efficiency, 7-18
  static memory, 7-18, 86
  suspended systems, 2-24
GC function
  description, 48
GC-VERBOSE variable
  changing garbage collector
      messages, 7-19
  description, 49
"General Prompting" Editor buffer,
    C-14
Generalized print functions, 6-21
GENERALIZED-PRINT-FUNCTION-
    ENABLED-P
  function, 6-21
GENERALIZED-PRINT-FUNCTION-
    ENABLED-P function
  description, 50
GET-DEVICE-INFORMATION function
  description, 51 to 54
  keywords (table), 51 to 54
GET-FILE-INFORMATION function
  description, 55 to 58
  keywords (table), 55
  number of bytes in a file, 7-23
GET-GC-REAL-TIME function
  description, 59
GET-GC-RUN-TIME function
  description, 61
GET-INTERNAL-RUN-TIME function
  description, 63
  (table), 7-30
GET-KEYBOARD-FUNCTION function, 6
  description, 64

GET-KEYBOARD-FUNCTION function (Cont.)
  returning information about key bindings, 7-26
GET-PROCESS-INFORMATION function
  description, 65 to 72
  keywords (table), 65 to 71
  record length, 7-23
GET-TERMINAL-MODES function
  description, 73 to 75
  keywords (table), 73
GET-VMS-MESSAGE function
  description, 76
Global
  definitions, 5-7
  variables, 5-7
:GLOBAL-PAGES keyword
  GET-PROCESS-INFORMATION function, 68
GOTO debugger command
  description, 5-15
  (table), 5-10
Graphics interface, 1-6
:GROUP keyword
  GET-FILE-INFORMATION function, 56
  GET-PROCESS-INFORMATION function, 68
  TRANSLATE-LOGICAL-NAME function, 137
"Grow Window" Editor command, 3-37
  "EMACS" style binding, B-6
  using, 3-35

-H-

:HALF-DUPLEX keyword
  GET-TERMINAL-MODES function, 74
  SET-TERMINAL-MODES function, 109
Handling lists, 6-16
Hash table
  comparing keys, 80
  initial size, 79
  rehash size, 77
  rehash threshold, 78
HASH-TABLE-REHASH-SIZE function
  description, 77
HASH-TABLE-REHASH-THRESHOLD function
  description, 78
HASH-TABLE-SIZE function
  description, 79
HASH-TABLE-TEST function
  description, 80
HELP
  debugger command
    description, 5-13
    (table), 5-10
  stepper command
    description, 5-26
    (table), 5-25
Help
  Editor, 3-7
Help facilities
  DCL, 1-7
  debugger, 5-13
  LISP, 1-8
  stepper, 5-26
"Help on Editor Error" Editor command, 3-13
"Help" Editor command, 3-12
HERE debugger command modifier, 5-12
  with BACKTRACE command, 5-17
  with SHOW command, 5-17
Hibernation state, 112.2
:HOST keyword
  pathname field, 7-10

-I-

~I directive, 6-6
I/O request specifiers, 8-3
  table, 8-4
:IF-DOES-NOT-EXIST keyword
  LOAD function, 81
  OPEN function, 7-24
:IF-EXISTS keyword
  OPEN function, 7-24
If-needed new line directive, 6-11
:IMAGE-NAME keyword
  GET-PROCESS-INFORMATION function, 68
:IMAGE-PRIVILEGES keyword
  GET-PROCESS-INFORMATION function, 68
:IMMEDIATE-OUTPUT-P
  I/O request specifier, 8-4
IMMEDIATE-OUTPUT-P function
  description, 8-7
Implementation notes, 7-1 to 7-31

Improperly formed argument lists, 6-28
"Indent LISP Line" Editor command, 3-24
"Indent Outermost Form" Editor command, 3-24
Indentation, 6-13
  preserving, 6-9
Information area, 3-5
  pointer cursor in, 3-48
/INITIALIZE qualifier
  description, 2-15
  modes, 2-13
  (table), 2-11
  with /COMPILE qualifier, 2-14
  with /RESUME qualifier, 2-21
  with /VERBOSE qualifier, 2-21
:INPUT-FILE keyword
  SPAWN function, 112.2
Input/Output, 7-20 to 7-25
  end-of-file operations, 7-22
  file organization, 7-23
  FILE-LENGTH function, 7-23
  FILE-POSITION function, 7-24
  functions, 7-23, 7-24
  #\NEWLINE character, 7-20
  record length, 7-22
  terminal input, 7-21
  WRITE-CHAR function, 7-25
"Insert Buffer" Editor command, 3-25, 3-37
  using, 3-23, 3-36
"Insert Close Paren and Match" Editor command, 3-24
"Insert File" Editor command, 3-25
  "EMACS" style binding, B-5
  using, 3-23
Insignificant stack frame, 5-4
Integers, 7-3
  constants, 7-3
/INTERACTIVE qualifier, 1-3
  description, 2-16
  modes, 2-13
  (table), 2-11
INTERNAL-TIME-UNITS-PER-SECOND constant, 59, 61, 63
Interpreted function definition
  restoring, 140
Interpreter, 1-3
  creating programs, 2-5
Interrupt function facility, 1-6
Interrupt functions, 7-19, 7-25
  garbage collector, 7-25
  suspended systems, 7-26
  terminal input, 7-21
Interrupt levels
  keyboard functions, 6

-J-

:JOB-SUBPROCESS-COUNT keyword
  GET-PROCESS-INFORMATION function, 68

-K-

Keyboard functions
  creating, 6
  interrupt level, 6
    specifying, 7
  passing arguments to, 7
Keyboard macros, 3-45
  named, 3-45
Keypad
  numeric
    see Numeric keypad
Keys
  binding to commands, 3-38
  binding to Editor commands
    conflicts in "EMACS" style, B-2
    from LISP interpreter, 3-40
    key binding shadowing, 3-44
    multiple bindings, C-14
    selecting key or sequence, 3-43
    specifying in "Bind Command", 3-39
    specifying in BIND-COMMAND function, 3-40
    table of bindings, C-14
    table of bindings by command, C-2
    within Editor, 3-39
  function
    see Function keys
"Kill Line" Editor command
  "EMACS" style binding, B-5
"Kill Paragraph" Editor command
  "EMACS" style binding, B-5
"Kill Region" Editor command
  "EMACS" style binding, B-5

-L-

Label strip, 3-4
LEAST-NEGATIVE-DOUBLE-FLOAT
    constant, 7-5
LEAST-NEGATIVE-LONG-FLOAT
    constant, 7-5
LEAST-NEGATIVE-SHORT-FLOAT
    constant, 7-5
LEAST-NEGATIVE-SINGLE-FLOAT
    constant, 7-5
LEAST-POSITIVE-DOUBLE-FLOAT
    constant, 7-5
LEAST-POSITIVE-LONG-FLOAT
    constant, 7-5
LEAST-POSITIVE-SHORT-FLOAT
    constant, 7-5
LEAST-POSITIVE-SINGLE-FLOAT
    constant, 7-5
:LEVEL keyword
  BIND-KEYBOARD-FUNCTION function,
      6
Lexical environment
  compiler restrictions, 7-26
Limiting output by lines, 6-4,
    6-25
"Line to Top of Window" Editor
    command
  "EMACS" style binding, B-5
:LINE-POSITION
  I/O request specifier, 8-4
LINE-POSITION function
  description, 8-7
~/LINEAR/ directive, 6-6
:LINES keyword
  WRITE and WRITE-TO-STRING, 6-3
LISP
  command, 1-3, 2-1
    qualifier descriptions, 2-10
        to 2-23
    qualifier modes (table), 2-12
    qualifiers (table), 2-10
  exiting, 2-2, 44
  implementation notes, 7-1 to
      7-31
  input/output
    See Input/Output
  invoking, 2-1
  processing during garbage
      collection, 7-19
  program, 1-1
    compiling, 2-7
LISP
  program (Cont.)
    creating, 2-5
    loading
      See File
  programming language, 1-1
  prompt, 2-1
  storage allocation, 1-1
    See also Memory
LISP code
  indenting with Editor, 3-15
  typing and formatting with
      Editor, 3-15
"List Buffers" Editor command,
    3-36
  "EMACS" style binding, B-6
  using, 3-31
"List Key Bindings" Editor
    command, 3-11
  using, 3-7
/LIST qualifier
  description, 2-17
  modes, 2-13
  (table), 2-11
  with /COMPILE qualifier, 2-14
List-print functions, 6-19
:LISTEN2
  I/O request specifier, 8-4
LISTEN2 function
  description, 8-8
Listing file, 2-17
  producing, 14
:LISTING keyword
  COMPILE-FILE function, 14
LOAD function, 2-6, 2-15
  description, 81
  (table), 7-30
LOAD-VERBOSE variable
  load message, 81
:LOCAL-EVENT-FLAGS keyword
  GET-PROCESS-INFORMATION
      function, 68
Logical block, 6-5
Logical name table, 137
Logical names, 1-10, 137
  translating, 83, 84
:LOGICAL-NAMES keyword
  SPAWN function, 113
:LOGICAL-VOLUME-NAME keyword
  GET-DEVICE-INFORMATION function,
      52

:LOGIN-TIME keyword
  GET-PROCESS-INFORMATION
      function, 68
Long floating-point numbers, 7-3
LONG-FLOAT-EPSILON constant, 7-5
LONG-FLOAT-NEGATIVE-EPSILON
    constant, 7-5
LONG-SITE-NAME function
  description, 83
  (table), 7-30
:LONGEST-RECORD-LENGTH keyword
  GET-FILE-INFORMATION function,
      56

-M-

:MACHINE-CODE keyword
  COMPILE-FILE function, 14
Machine-code listing, 2-18
MACHINE-INSTANCE function
  description, 84
  (table), 7-30
MACHINE-VERSION function
  description, 85
  (table), 7-30
/MACHINE_CODE qualifier, 2-18
  modes, 2-13
  (table), 2-11
  with /COMPILE qualifier, 2-14
Macro
  compiling, 2-7
  defining, 2-5
  implementation-dependent
      (table), 7-30
  modifying, 2-7
Major style, 3-44
  default, B-4
  establishing default, B-4
MAKE-ARRAY function
  allocating static space, 7-18
  description, 86
  (table), 7-30
MAKE-HASH-TABLE function, 77 to
    80
MAKE-PATHNAME function
  constructing pathnames, 7-12
  creating pathnames, 7-11
  setting pathnames, 7-13
:MAX-BLOCKS keyword
  GET-DEVICE-INFORMATION function,
      52
:MAX-FILES keyword
  GET-DEVICE-INFORMATION function,
      52
:MAX-RECORD-SIZE keyword
  GET-FILE-INFORMATION function,
      56
:MEMBER keyword
  GET-FILE-INFORMATION function,
      56
  GET-PROCESS-INFORMATION
      function, 68
Memory, 105
  control stack, 5-3
  dynamic, 2-18, 105, 118
    garbage collector, 7-18, 7-19
  read-only, 2-18, 105, 118
  static, 2-18, 86, 105, 118
    garbage collector, 7-18
/MEMORY qualifier
  description, 2-18
  garbage collector, 7-18
  modes, 2-13
  (table), 2-11
Minor style, 3-44
  activating
    from Editor, B-3
    from LISP interpreter, B-3
  activation, 3-44
  default, B-3
  determining most recently
      activated, C-14
Miser mode, 6-5, 6-26, 97
Miser-mode new line directive,
    6-11
:MISER-WIDTH keyword
  WRITE and WRITE-TO-STRING, 6-3
Modifiers
  See Debugger
Module, 103
MODULE-DIRECTORY variable, 103
  description, 88
Modules, 88
MOST-NEGATIVE-DOUBLE-FLOAT
    constant, 7-5
MOST-NEGATIVE-FIXNUM constant,
    7-3
MOST-NEGATIVE-LONG-FLOAT constant,
    7-5
MOST-NEGATIVE-SHORT-FLOAT
    constant, 7-5
MOST-NEGATIVE-SINGLE-FLOAT
    constant, 7-5

MOST-POSITIVE-DOUBLE-FLOAT
    constant, 7-5
MOST-POSITIVE-FIXNUM constant,
    7-3
MOST-POSITIVE-LONG-FLOAT constant,
    7-5
MOST-POSITIVE-SHORT-FLOAT
    constant, 7-5
MOST-POSITIVE-SINGLE-FLOAT
    constant, 7-5
:MOUNT-COUNT keyword
  GET-DEVICE-INFORMATION function,
      52
:MOUNTED-VOLUMES keyword
  GET-PROCESS-INFORMATION
      function, 68
"Move to LISP Comment" Editor
    command, 3-24
Multiline mode, 6-8
Multiline mode new line directive,
    6-11

-N-

~*n*,*m*/TABULAR/ directive, 6-6
~*n*/FILL/ directive, 6-6, 6-16
~*n*/LINEAR/ directive, 6-6, 6-16
~*n*/TABULAR/ directive, 6-17
:NAME keyword
  pathname field, 7-11
NAMESTRING function
  creating namestrings, 7-14
Namestrings, 7-8, 7-10, 7-14
  See also File
  creating, 7-14
New lines, 6-11
"New LISP Line" Editor command,
    3-24
  using, 3-15
:NEWEST keyword
  See :VERSION keyword
#\NEWLINE character
  description, 7-20
"Next Form" Editor command, 3-27
"Next Line" Editor command, 3-25
  "EMACS" style binding, B-4
"Next Paragraph" Editor command
  "EMACS" style binding, B-4
"Next Screen" Editor command,
    3-26
  "EMACS" style binding, B-4
"Next Window" Editor command,
    3-12
  "EMACS" style binding, B-6
:NEXT-DEVICE-NAME keyword
  GET-DEVICE-INFORMATION function,
      53
~*n*I directive, 6-6
Node, 1-8
  pathnames, 7-13
/NOINITIALIZE qualifier
  modes, 2-13
/NOLIST qualifier
  description, 2-17
  modes, 2-13
  (table), 2-11
  with /COMPILE qualifier, 2-14
  with /MACHINE_CODE qualifier,
      2-18
/NOMACHINE_CODE qualifier
  description, 2-18
  modes, 2-13
  (table), 2-11
  with /COMPILE qualifier, 2-14
/NOOPTIMIZE qualifier
  modes, 2-13
/NOOUTPUT_FILE qualifier
  description, 2-20
  modes, 2-13
  (table), 2-12
  with /COMPILE qualifier, 2-13
NORMAL debugger command modifier,
    5-12
  with BACKTRACE command, 5-17
/NOVERBOSE qualifier
  description, 2-21
  modes, 2-13
  (table), 2-12
  with /COMPILE qualifier, 2-14
/NOWARNINGS qualifier
  description, 2-23
  modes, 2-13
  (table), 2-12
  with /COMPILE qualifier, 2-14
:NREAD-LINE
  I/O request specifier, 8-4
NREAD-LINE function
  description, 8-8
Null lexical environment
  break loop, 5-7
  compiler restrictions, 7-26
  tracer, 5-36, 125
Numbers, 7-2

Numeric keypad
  Editor use of, 3-14
  illustration, 3-15
Numeric keypad keys
  specifying in BIND-COMMAND
      function, 3-40

-O-

OPEN function, 7-23, 7-24
"Open Line" Editor command, 3-24
  "EMACS" style binding, B-5
:OPEN-FILE-COUNT keyword
  GET-PROCESS-INFORMATION
      function, 68
:OPEN-FILE-QUOTA keyword
  GET-PROCESS-INFORMATION
      function, 68
OPEN-STREAM-P function
  description, 8-9
:OPERATION-COUNT keyword
  GET-DEVICE-INFORMATION function,
      53
Optimization qualities
  See Compiler
OPTIMIZE declaration, 7-27
:OPTIMIZE keyword
  COMPILE-FILE function, 14
/OPTIMIZE qualifier
  description, 2-19
  modes, 2-13
  optimizing compiler, 7-27
  (table), 2-11
  with /COMPILE qualifier, 2-14
:ORGANIZATION keyword
  GET-FILE-INFORMATION function,
      56
Outermost form
  making select region from, 3-21
:OUTPUT-FILE keyword
  COMPILE-FILE function, 15
  SPAWN function, 113
/OUTPUT_FILE qualifier
  description, 2-20
  modes, 2-13
  (table), 2-12
  with /COMPILE qualifier, 2-14
OVER stepper command
  description, 5-28
  (table), 5-25
:OWNER-PID keyword
  GET-PROCESS-INFORMATION
      function, 68
:OWNER-UIC keyword
  GET-DEVICE-INFORMATION function,
      53

-P-

Packages, 1, 3
  current, 1, 3, 92
"Page Next Window" Editor command
  "EMACS" style binding, B-6
:PAGE-FAULTS keyword
  GET-PROCESS-INFORMATION
      function, 68
:PAGE-FILE-COUNT keyword
  GET-PROCESS-INFORMATION
      function, 69
:PAGE-FILE-QUOTA keyword
  GET-PROCESS-INFORMATION
      function, 69
:PAGES-AVAILABLE keyword
  GET-PROCESS-INFORMATION
      function, 69
:PARALLEL keyword
  SPAWN function, 113
Parentheses
  matching with Editor, 3-15
    using pointer, 3-49
PARSE-NAMESTRING function
  constructing pathnames, 7-12
  creating pathnames, 7-11
  setting pathnames, 7-13
:PASS-ALL keyword
  GET-TERMINAL-MODES function, 74
  SET-TERMINAL-MODES function,
      109
Pass-all mode, 7-22, 109
:PASS-THROUGH keyword
  GET-TERMINAL-MODES function, 74
  SET-TERMINAL-MODES function,
      7-21, 109
Pass-through mode, 108.1
Paste buffer, 3-21
  appending text to, 3-21
PATHNAME function
  constructing pathnames, 7-12
  creating pathnames, 7-11
Pathnames, 7-7 to 7-15
  See also File
  constructing, 7-12

Pathnames (Cont.)
  creating, 7-11
  default directory, 25
  description, 7-10
  DIRECTORY function, 37
  fields, 7-10, 7-11
    (table), 7-10
  functions, 7-15
"Pause Editor" Editor command,
    3-11
  effect on buffers, 3-34
  "EMACS" style binding, B-6
  using, 3-10
Per-line prefix, 6-15
Per-line prefixes
  preserving, 6-9
:PID keyword
  GET-DEVICE-INFORMATION function,
      53
  GET-PROCESS-INFORMATION
      function, 69
Pointer
  determining Editor commands
      bound to, 3-49
Pointer cursor
  VAXstation
    appearance in Editor, 3-47
Pointing device
  VAXstation
    using in Editor, 3-47
:POST-DEBUG-IF keyword
  TRACE macro, 5-36, 125
POST-GC-MESSAGE variable, 49
  changing garbage collector
      messages, 7-19
  description, 89
:POST-PRINT keyword
  TRACE macro, 5-36, 126
PPRINT
  function, 6-2
PPRINT-DEFINITION
  function, 6-2
PPRINT-DEFINITION function
  description, 90
PPRINT-PLIST
  function, 6-2
PPRINT-PLIST function
  description, 92
:PRE-DEBUG-IF keyword
  TRACE macro, 5-36, 125
PRE-GC-MESSAGE variable, 49
  changing garbage collector
      messages, 7-19
  description, 95
:PRE-PRINT keyword
  TRACE macro, 5-36, 126
Prefix, 6-14
  per-line, 6-15
Prefix argument, 3-23
  entering, 3-23
  negative, 3-23
Preserving indentation, 6-9
Preserving per-line prefixes, 6-9
Pretty printer, 1-5
  controlling margins, 98
  miser mode, 97
Pretty printing, 6-1 to 6-28
"Previous Form" Editor command,
    3-27
:PREVIOUS keyword
  See :VERSION keyword
  THROW-TO-COMMAND-LEVEL function,
      121
"Previous Line" Editor command,
    3-25
  "EMACS" style binding, B-4
"Previous Paragraph" Editor
    command
  "EMACS" style binding, B-4
"Previous Screen" Editor command,
    3-26
  "EMACS" style binding, B-4
"Previous Window" Editor command
  "EMACS" style binding, B-6
Print control variables, 6-3
:PRINT keyword
  LOAD function, 81
  TRACE macro, 5-36, 125
PRINT-LENGTH, 6-24
PRINT-LEVEL, 6-24
PRINT-LINES, 6-4, 6-25
PRINT-LINES variable
  description, 96
PRINT-MISER-WIDTH, 6-26
  variable, 6-5
PRINT-MISER-WIDTH variable
  description, 97
PRINT-RIGHT-MARGIN, 6-26
  variable, 6-4
PRINT-RIGHT-MARGIN variable
  description, 98

PRINT-SIGNALED-ERROR function
  defining an error handler, 4-6
  description, 100
PRINT-SLOT-NAMES-AS-KEYWORDS
    variable
  description, 102
Process
  connecting to, 4
  getting information, 65
  identification, 4
:PROCESS keyword
  TRANSLATE-LOGICAL-NAME function,
      137
:PROCESS-NAME keyword
  GET-PROCESS-INFORMATION
      function, 69
  SPAWN function, 113
PROCLAIM function, 7-27
Prompt
  break loop, 5-5
  debugger, 5-8
  Editor
    completing input, 3-8
    displaying alternative
        choices, 3-9
    help on, 3-7
  LISP, 2-1
  stepper, 5-20.1
  top-level, 2-1
    changing, 123
"Prompt Complete String" Editor
    command, 3-13
"Prompt Scroll Help Window"
    Editor command, 3-12
"Prompt Show Alternatives" Editor
    command, 3-13
Property list
  pretty-print, 92
:PROTECTION keyword
  GET-FILE-INFORMATION function,
      56

-Q-

"Query Search Replace" Editor
    command, 3-29
  "EMACS" style binding, B-5
  using, 3-22
QUICK debugger command modifier,
    5-12
  with BACKTRACE command, 5-17
QUIT
  debugger command, 5-9
    description, 5-14
    (table), 5-10
  stepper command
    description, 5-27
    exiting stepper, 5-21
    (table), 5-25
"Quoted Insert" Editor command,
    3-25
  "EMACS" style binding, B-5

-R-

"Read File" Editor command
  "EMACS" style binding, B-6
:READ-CHAR
  I/O request specifier, 8-4
READ-CHAR function
  #\NEWLINE character, 7-20
  terminal input, 7-21
:READ-LINE
  I/O request specifier, 8-4
Read-only memory, 2-18, 105, 118
Real time
  displaying, 122
  garbage collector, 59
Record length, 7-22
Record Management Services (RMS)
  input/output, 7-20
  record length, 7-22
:RECORD-ATTRIBUTES keyword
  GET-FILE-INFORMATION function,
      56
:RECORD-FORMAT keyword
  GET-FILE-INFORMATION function,
      56
:RECORD-SIZE keyword
  GET-DEVICE-INFORMATION function,
      53
"Redisplay Screen" Editor command,
    3-13
  "EMACS" style binding, B-6
REDO debugger command
  description, 5-14
  (table), 5-10
:REFERENCE-COUNT keyword
  GET-DEVICE-INFORMATION function,
      53
Relative tabbing, 6-16

"Remove Current Window" Editor command, 3-12
  "EMACS" style binding, B-6
"Remove Other Windows" Editor command, 3-12
  "EMACS" style binding, B-6
  using, 3-32
REQUIRE function, 88
  description, 103
  (table), 7-30
/RESUME qualifier, 2-25, 118
  description, 2-21
  modes, 2-13
  (table), 2-12
  with /INITIALIZE qualifier, 2-15
  with /MEMORY qualifier, 2-19
RETURN
  debugger command
    description, 5-14
    (table), 5-10
  key
    as a stepper command, 5-28
    entering
      debugger command arguments, 5-11
      debugger commands, 5-10
      stepper commands, 5-24
    terminal input, 7-21
  stepper command
    description, 5-28
    (table), 5-25
:REVISION keyword
  GET-FILE-INFORMATION function, 56
:REVISION-DATE keyword
  GET-FILE-INFORMATION function, 56
:RIGHT-MARGIN
  I/O request specifier, 8-4
RIGHT-MARGIN function
  description, 8-10
:RIGHT-MARGIN keyword
  WRITE and WRITE-TO-STRING, 6-3
ROOM function
  debugging information, 5-2
  description, 105
  specifying memory, 2-19
  (table), 7-30
ROOM-ALLOCATION function
  description, 108
:ROOT-DEVICE-NAME keyword
  GET-DEVICE-INFORMATION function, 53
Run-time efficiency, 7-18

-S-

Screen
  refreshing, in Editor, 3-9
"Scroll Window Down" Editor command
  "EMACS" style binding, B-5
"Scroll Window Up" Editor command
  "EMACS" style binding, B-5
SEARCH debugger command
  description, 5-15
  (table), 5-10
:SECTORS keyword
  GET-DEVICE-INFORMATION function, 53
"Select Buffer" Editor command, 3-36
  "EMACS" style binding, B-6
  using, 3-32
"Select Outermost Form" Editor command, 3-12, 3-28
Select region
  cancelling, 3-21
  changing case of, 3-22
  defining, in Editor, 3-21
  from outermost form, 3-21
  marking with pointer, 3-48
  replacing with paste buffer, 3-21
:SERIAL-NUMBER keyword
  GET-DEVICE-INFORMATION function, 53
SET debugger command
  description, 5-16
  (table), 5-11
"Set Select Mark" Editor command, 3-28
  "EMACS" style binding, B-5
SET-TERMINAL-MODES function
  changing terminal input mode, 7-21
  description, 108.1
SETF macro
  changing the default directory, 25
  setting pathnames, 7-13
Short floating-point numbers, 7-3

SHORT-FLOAT-EPSILON constant, 7-5
SHORT-FLOAT-NEGATIVE-EPSILON
    constant, 7-5
SHORT-SITE-NAME function
  description, 111
  (table), 7-31
SHOW
  debugger command
    description, 5-17
    (table), 5-11
  stepper command
    description, 5-27
    (table), 5-25
"Show Time" Editor command
  "EMACS" style binding, B-6
"Shrink Window" Editor command,
    3-37
  "EMACS" style binding, B-6
  using, 3-35
Significant stack frame, 5-4
Single floating-point numbers,
    7-3
SINGLE-FLOAT-EPSILON constant,
    7-5
SINGLE-FLOAT-NEGATIVE-EPSILON
    constant, 7-5
:SITE-SPECIFIC keyword
  GET-PROCESS-INFORMATION
      function, 69
SOFTWARE-VERSION-NUMBER function
  description, 112
Source file
  compiling, 14
  file type, 1-10
  loading, 81
  locating, 81
SOURCE-CODE function
  description, 112.1
SPAWN function
  description, 112.2
Specialized arrays, 7-7
"Split Window" Editor command,
    3-37
  "EMACS" style binding, B-6
  using, 3-35
Stack frame, 5-3
  active, 5-4
  current, 5-7
  insignificant, 5-4
  number
    debugger command argument,
        5-12
Stack frame
  number (Cont.)
    stepper output, 5-22
    tracer output, 5-34
  significant, 5-4
STANDARD-OUTPUT variable
  LOAD function, 81
  PPRINT-DEFINITION function, 90
  PPRINT-PLIST function, 93
"Start Keyboard Macro" Editor
    command, 3-46
"Start Named Keyboard Macro"
    Editor command, 3-46
  using, 3-45
:STATE keyword
  GET-PROCESS-INFORMATION
      function, 69
:STATIC keyword
  See :ALLOCATION keyword
Static memory, 2-18, 86, 105, 118
  garbage collector, 7-18
Status code, 76
:STATUS keyword
  GET-PROCESS-INFORMATION
      function, 69
Status return, 44
STEP
  debugger command
    description, 5-14
    (table), 5-11
  macro
    debugging information, 5-2
    invoking stepper, 5-20.1
  stepper command
    description, 5-28
    (table), 5-25
Step
  macro
    description, 115
STEP-ENVIRONMENT
  variable, 5-28
    description, 116
STEP-FORM
  variable, 5-28
    description, 117
:STEP-IF keyword
  TRACE macro, 5-36, 126
Stepper, 1-5, 5-20 to 5-32
  commands
    arguments, 5-25
    descriptions, 5-26 to 5-28
    (table), 5-24

Stepper (Cont.)
  exiting, 5-21, 5-27
  invoking, 5-14, 5-20.1, 5-36, 115, 126
  output, 5-21
    controlling, 23, 24
  prompt, 5-20.1
  sample sessions, 5-31
  using, 5-24
STOP command, 1-11
Storage allocation, 1-1
  See also Memory
Stream dispatch function, 8-3
  arguments, 8-3
STREAM structure, 8-2
Streams, 118
  defining new types, 8-1
  information about, 8-5
String
  searching for
    with Editor, 3-18
STRING-STREAM data type, 8-6
STRING-STREAM-P function, 8-6
Strings, 7-7
  creating, 86
Subprocess, 112.2
:SUBPROCESS-COUNT keyword
  GET-PROCESS-INFORMATION function, 70
:SUBPROCESS-QUOTA keyword
  GET-PROCESS-INFORMATION function, 70
:SUCCESS keyword
  EXIT function, 44
Suffix, 6-14
"Supply EMACS Prefix" Editor command
  "EMACS" style binding, B-6
"Supply Prefix Argument" Editor command, 3-29
  "EMACS" style binding, B-6
:SUPPRESS-IF keyword
  TRACE macro, 5-37, 126
SUSPEND function
  creating suspended systems, 2-24
  description, 118
Suspended systems, 118
  creating, 2-24
  file type, 1-10
  garbage collector, 2-24
  Internal time, 61
Suspended systems (Cont.)
  interrupt functions, 7-26
  real time, 59
  resuming, 2-21, 2-25
Symbolic expressions, 1-1
Symbols
  editing function definition, 3-3
    moving back to LISP, 3-10
  editing value, 3-3
    moving back to LISP, 3-10
SYNONYM-STREAM data type, 8-6
SYNONYM-STREAM-P function, 8-6
SYNONYM-STREAM-SYMBOL function, 8-6
System identification (SID) register, 85
:SYSTEM keyword
  TRANSLATE-LOGICAL-NAME function, 137

-T-

~T directive, 6-15
Tab directive, 6-15
Tabs, 6-15
~/TABULAR/ directive, 6-6
Terminal
  getting information, 73
  input, 7-21
    changing modes, 7-21
    pass-all mode, 7-22
:TERMINAL keyword
  GET-PROCESS-INFORMATION function, 70
TERMINAL-IO variable
  BIND-KEYBOARD-FUNCTION function, 7
  end-of-file operations, 7-22
  GET-TERMINAL-MODES function, 73
  SET-TERMINAL-MODES function, 108.1
TERMINAL-STREAM data type, 8-6
TERMINAL-STREAM-P function, 8-6
:TERMINATION-MAILBOX keyword
  GET-PROCESS-INFORMATION function, 70
TERPRI function
  #\NEWLINE character, 7-20
  record length, 7-22

Text
  changing case of characters, 3-21
  copying with Editor, 3-20, 3-21
  cutting and pasting, 3-20
  deleting with Editor, 3-19
    restoring deleted, 3-20
  inserting with Editor, 3-14
    starting new line, 3-15
  moving between Editor buffers, 3-36
  moving with Editor, 3-20
  substituting in, 3-22
THROW-TO-COMMAND-LEVEL function
  description, 121
TIME macro
  debugging information, 5-2
  description, 122
  (table), 7-31
:TIMER-QUEUE-COUNT keyword
  GET-PROCESS-INFORMATION function, 70
:TIMER-QUEUE-QUOTA keyword
  GET-PROCESS-INFORMATION function, 70
TOP
  debugger command
    description, 5-15
    (table), 5-11
  debugger command modifier, 5-13
    with BACKTRACE command, 5-17
:TOP keyword
  THROW-TO-COMMAND-LEVEL function, 121
Top-level loop
  prompt, 2-1
  variables, 2-2
TOP-LEVEL-PROMPT variable
  description, 123
TRACE macro
  debugging information, 5-2
  description, 124
  enabling the tracer, 5-33
  options, 5-35
  (table), 7-31
TRACE-CALL
  Variable
    description, 135
  variable, 5-37
TRACE-OUTPUT variable
  stepper, 5-20
  tracer, 5-32
TRACE-VALUES
  variable, 5-38
    description, 136
Tracer, 1-5, 5-32 to 5-39
  disabling, 5-33
  enabling, 5-33, 124
  options
    adding to output, 5-36
    defining when to trace a function, 5-37
    invoking the debugger, 5-36
    invoking the stepper, 5-36
    removing information from output, 5-37
  options (table), 125
  output, 5-34
    controlling, 23, 24
:TRACKS keyword
  GET-DEVICE-INFORMATION function, 53
:TRANSACTION-COUNT keyword
  GET-DEVICE-INFORMATION function, 53
TRANSLATE-LOGICAL-NAME function
  description, 137
  using, 7-14
"Transpose Previous Characters" Editor command
  "EMACS" style binding, B-5
"Transpose Previous Words" Editor command
  "EMACS" style binding, B-5
TWO-WAY-STREAM data type, 8-6
TWO-WAY-STREAM-INPUT-STREAM function, 8-6
TWO-WAY-STREAM-OUTPUT-STREAM function, 8-6
TWO-WAY-STREAM-P function, 8-6
:TYPE keyword
  pathname field, 7-11
:TYPE-AHEAD keyword
  GET-TERMINAL-MODES function, 75
  SET-TERMINAL-MODES function, 109

-U-

:UIC keyword
  GET-FILE-INFORMATION function, 57
  GET-PROCESS-INFORMATION function, 70

UNBIND-KEYBOARD-FUNCTION function, 6
  description, 139
  unbinding control characters, 7-26
UNCOMPILE function
  description, 140
  retrieving interpreted definitions, 2-7
Unconditional new line directive, 6-11
UNDEFINE-LIST-PRINT-FUNCTION macro, 6-20
UNDEFINE-LIST-PRINT-FUNCTION macro
  description, 141
"Undo Previous Yank" Editor command
  "EMACS" style binding, B-5
:UNIT keyword
  GET-DEVICE-INFORMATION function, 53
UNIVERSAL-ERROR-HANDLER function, 4-1
  defining an error handler, 4-6
  description, 142
UNIVERSAL-ERROR-HANDLER variable, 4-5, 142
  description, 143
:UNREAD-CHAR
  I/O request specifier, 8-4
"Unset Select Mark" Editor command, 3-28
  "EMACS" style binding, B-5
UNTRACE macro
  debugging information, 5-2
  disabling the tracer, 5-33
UP
  debugger command
    description, 5-16
    (table), 5-11
  debugger command modifier, 5-13
    SEARCH debugger command, 5-15
  stepper command
    description, 5-28
    (table), 5-25
"Upcase Region" Editor command, 3-29
"Upcase Word" Editor command, 3-29
  "EMACS" style binding, B-5
User defined FORMAT directives, 6-18
:USERNAME keyword
  GET-PROCESS-INFORMATION function, 70

-V-

:VALUE keyword
  ED function, 42
Variable
  print control, 6-3
"VAX LISP" Editor style, 3-44
  automatic activation, 3-44
VAX/VMS file specification
  See File
VAXstation
  pointing device
    using in Editor, 3-47
  using Editor on, 3-46
Vectors
  creating, 86
VERBOSE debugger command modifier, 5-13
  with BACKTRACE command, 5-17
:VERBOSE keyword
  COMPILE-FILE function, 15, 17
  LOAD function, 81
/VERBOSE Qqualifier
  loading files, 2-6
/VERBOSE qualifier
  description, 2-21
  modes, 2-13
  (table), 2-12
  with /COMPILE qualifier, 2-14
  with /INITIALIZE qualifier, 2-15
  with /LIST qualifier, 2-17
:VERSION keyword
  pathname field, 7-11
Version number, 1-9
:VERSION-LIMIT keyword
  GET-FILE-INFORMATION function, 57
"View File" Editor command
  "EMACS" style binding, B-6
:VIRTUAL-ADDRESS-PEAK keyword
  GET-PROCESS-INFORMATION function, 70
VMS
  hibernation state, 4

:VOLUME-COUNT keyword
GET-DEVICE-INFORMATION function, 53
:VOLUME-NAME keyword
GET-DEVICE-INFORMATION function, 54
:VOLUME-NUMBER keyword
GET-DEVICE-INFORMATION function, 54
:VOLUME-PROTECTION keyword
GET-DEVICE-INFORMATION function, 54

-W-

~W directive, 6-6
WARN function, 142
description, 144
error messages, 4-4
(table), 7-31
WARNING function
defining an error handler, 4-7
:WARNING keyword
EXIT function, 44
:WARNINGS keyword
COMPILE-FILE function, 15, 18
/WARNINGS qualifier
description, 2-23
modes, 2-13
(table), 2-12
with /COMPILE qualifier, 2-14
"What Cursor Position" Editor command
"EMACS" style binding, B-6
WHERE debugger command
description, 5-16
(table), 5-11
:WILD keyword
See :VERSION keyword
Windows
Editor
see Editor windows
WITH-GENERALIZED-PRINT-FUNCTION macro, 6-22
WITH-GENERALIZED-PRINT-FUNCTION macro
description, 145
:WORKING-SET-AUTHORIZED-EXTENT keyword
GET-PROCESS-INFORMATION function, 70
:WORKING-SET-AUTHORIZED-QUOTA keyword
GET-PROCESS-INFORMATION function, 70
:WORKING-SET-COUNT keyword
GET-PROCESS-INFORMATION function, 70
:WORKING-SET-DEFAULT keyword
GET-PROCESS-INFORMATION function, 71
:WORKING-SET-EXTENT keyword
GET-PROCESS-INFORMATION function, 71
:WORKING-SET-PEAK keyword
GET-PROCESS-INFORMATION function, 71
:WORKING-SET-QUOTA keyword
GET-PROCESS-INFORMATION function, 71
:WORKING-SET-SIZE keyword
GET-PROCESS-INFORMATION function, 71
:WRAP keyword
GET-TERMINAL-MODES function, 75
SET-TERMINAL-MODES function, 109
WRITE
FORMAT directive, 6-7
"Write Current Buffer" Editor command, 3-12
"EMACS" style binding, B-6
using, 3-10, 3-34
WRITE function
pretty-printing control keywords, 6-3
"Write Modified Buffers" Editor command, 3-12
"EMACS" style binding, B-6
using, 3-10, 3-34
"Write Named File" Editor command, 3-12
"EMACS" style binding, B-6
using, 3-10
:WRITE-CHAR
I/O request specifier, 8-4
WRITE-CHAR function, 7-25
#\NEWLINE character, 7-20
record length, 7-22
:WRITE-STRING
I/O request specifier, 8-4
WRITE-STRING function, 7-20

WRITE-TO-STRING function
pretty-printing control
keywords, 6-3

-Y-

"Yank Previous" Editor command
"EMACS" style binding, B-5
"Yank Replace Previous" Editor command
"EMACS" style binding, B-5
"Yank" Editor command
"EMACS" style binding, B-5

VAX LISP/VMS
User's Guide
AD–Y921B–T1

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____
____
____
____
____
____
____
____
____

Did you find errors in this manual? If so, specify the error and the page number.

____
____
____
____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____

Name ____ Date ______________________________

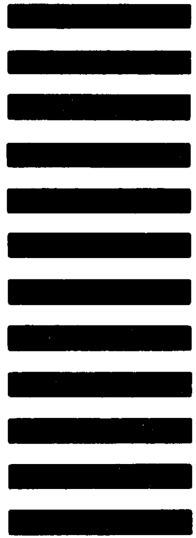
Organization ____

Street ____

City ____ State ______________ Zip Code or Country __________

— — Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — — — — —

digital



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
ML05–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

— — Do Not Tear — Fold Here —

VAX LISP
User's Guide
AA–Y921A–TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

Did you find errors in this manual? If so, specify the error and the page number.

____

Please indicate the type of user/reader that you most nearly represent.

- __ Assembly language programmer
- __ Higher-level language programmer
- __ Occasional programmer (experienced)
- __ User with little programming experience
- __ Student programmer
- __ Other (please specify) ________________________

Name ________________________ Date ________________

Organization ____________________ Telephone ______________

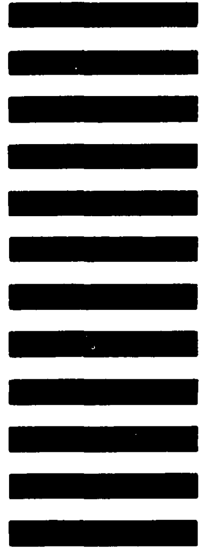
Street ____

City ________________________ State ________ Zip Code or Country ________

— — Do Not Tear — Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG/ML PUBLICATIONS, MLO5–5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754–2571

— — Do Not Tear — Fold Here —