VAX LISP/VMS User's Guide

Order Number: AA-Y921B-TE

May 1986

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

Operating System and Version:	VAX/VMS Version 4.2
Software Version:	VAX LISP/VMS Version 2.0

digital equipment corporation
maynard, massachusetts

First Printing, June 1984
Revised, May 1986

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.

Printed in U.S.A.

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

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

CONTENTS

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

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 seven 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.

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*
- *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.
...	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.

Convention	Meaning
. . .	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.
&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.

- *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.
...	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.

Convention	Meaning
. . .	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.
&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.

Convention	Meaning
<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

PART I

VAX LISP/VMS SYSTEM CONCEPTS AND FACILITIES

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 notes are kept on line in the file LISP$SYSTEM:VAXLISP*nnn*.MEM, where *nnn* is the VAX LISP version number.

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

These extensions are described in Sections 1.1.1 through 1.1.9.

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.

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 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
- Editor-specific objects
- System access-specific objects (pertaining to the call-out, alien structure, interrupt function, and program synchronization facilities)
- Graphics-specific objects

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 of this manual. 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*.

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.

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

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.

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
- Entering input
- Deleting input
- Entering the debugger
- Using control key characters
- 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 V2.0

Lisp>
```

See Section 2.10 for descriptions of the qualifiers you can use with the LISP command.

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 Chapter 3 for information on how to save modified buffers before exiting LISP.

You cannot exit the LISP system by typing CTRL/Z, as you can with many other interactive programs that run on VMS.

2.3 ENTERING 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:

```
/          *          +          -
//         **         ++
///        ***        +++
```

These 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>
```

2.4 DELETING AND EDITING INPUT

The DELETE key deletes characters to the left of the cursor on the current line of input. CTRL/U deletes the current line of input.

If you are using a video terminal, you can use control characters, function keys, and arrow keys on the terminal to edit the current line of input.

Table 2-1 lists the keys you can use to delete and edit input.

NOTE

You can use the BIND-KEYBOARD-FUNCTION function to bind most of the control characters listed in Table 2-1 to a LISP function. Binding a control character in this way cancels the effect listed for that control character in Table 2-1.

Table 2-1: Keys Used In Line Editing

Key	Effect
CTRL/A and F14*	Switches between overstrike and insert modes in the current line.
CTRL/B and Up Arrow	Recalls the last line entered.
CTRL/D and Left Arrow	Moves the cursor one character to the left.
CTRL/E	Moves the cursor to the end of the line.
CTRL/F and Right Arrow	Moves the cursor one character to the right.
CTRL/H and BACKSPACE and F12*	Moves the cursor to the beginning of the line.
CTRL/J and LINEFEED and F13*	Deletes the word to the left of the cursor.
CTRL/U	Deletes characters from the cursor position back to the beginning of the line.

* This key is available only on the LK201 keyboard.

2.5 ENTERING THE DEBUGGER

If you make an error during an interactive VAX LISP session, the error automatically invokes the debugger, which replaces the LISP prompt (Lisp>) with the debugger prompt (Debug 1>). For information on how to use the VAX LISP debugger, see Chapter 5.

Typing CTRL/C is 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 and starting over, type CTRL/C. CTRL/C returns you to the read-eval-print loop, which displays the LISP prompt (Lisp>).

2.6 USING CONTROL KEY CHARACTERS

Table 2-2 lists the control characters you can use in VAX LISP. CTRL/C is the only one whose listed function is specific to LISP. The other control characters perform standard VMS functions.

NOTE

You can use the BIND-KEYBOARD-FUNCTION function to bind most of the control characters listed in Table 2-2 to a LISP function. Binding a control character in this way cancels the effect listed for that control character in Table 2-2.

These control characters do not work in the VAX LISP Editor.

Table 2-2: Control Characters

Control Character	Function
CTRL/C	Returns you to the top-level loop from any other command level. In LISP, CTRL/C invokes the CLEAR-INPUT function on the *TERMINAL-IO* stream, then performs a throw to the catcher established for CANCEL-CHARACTER-TAG. If you want to recover from an error by discarding the expression you typed and starting over, type CTRL/C. (See CANCEL-CHARACTER-TAG in Part II for an example of changing the behavior of 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.

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 2.10.14.

- 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

using either the LISP COMPILE-FILE function or the DCL LISP /COMPILE qualifier.

2.9.1 Compiling Individual Functions and Macros

In LISP, the unit of compilation is normally either a function or a macro. You can compile a function or a macro in a currently running LISP session by using the COMPILE function. This function is described in *COMMON LISP: The Language*.

You normally call a LISP function first in interpreted form to see if the function works. Once it works as interpreted, you can test it in compiled form without having to write the function to a file. Use the COMPILE function for this purpose.

When you compile a function or a macro that is not in a file, the consequent compiled definition exists only in the current LISP; the definition is not in a file. However, you can use the VAX LISP UNCOMPILE function to retrieve the interpreted definition. This function, described in Part II, is useful when debugging programs. Because the interpreted code shows you more of your function's evaluation than the compiled code, you can find the error more easily. You can modify the function definition in the Editor to correct the error and also save your corrected version of the function in a file. See Chapter 3 for further information on using the Editor to write interpreted functions to files.

2.9.2 Compiling Files

Any collection of LISP expressions can make up a program and can be stored in a file. The compiler processes such a file by compiling the LISP expressions in the file and writing each compiled result to an output file.

You can compile VAX LISP files either at DCL level with the LISP command and the /COMPILE qualifier or in interactive VAX LISP with the LISP COMPILE-FILE function.

The /COMPILE qualifier is described in Section 2.10.2. The COMPILE-FILE function is described in Part II. The following example shows how the /COMPILE qualifier is used to compile the file MYPROG.LSP at the DCL level:

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

This example produces an output file named MYPROG.FAS.

The next example shows how the COMPILE-FILE function can be used to compile the file MYPROG.LSP from within the LISP system:

```
Lisp> (COMPILE-FILE "MYPROG.LSP")
Starting compilation of file DBA1:[JONES]MYPROG.LSP;1

FACTORIAL compiled.

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

Both methods of compiling LISP files are equivalent except in their defaults. The COMPILE-FILE function automatically lists the name of each function it compiles at the terminal, but the /COMPILE qualifier does not. Both methods produce fast-loading files (type FAS) that run more quickly than uncompiled files. Fast-loading files are automatically placed in the directory containing your source files.

The first method of compiling files, using the LISP /COMPILE qualifier, has the advantage that you can compile several files in one step. For example:

```
$ LISP/COMPILE FILE1.LSP, FILE2.LSP, FILE3.LSP
```

When you use the LISP COMPILE-FILE function, it takes several steps to compile several files, since you can only compile one file in each call to COMPILE-FILE.

The second method of compiling files, using the LISP COMPILE-FILE function, has the advantage of enabling you to stay in LISP both during compilation and afterwards. This method is convenient if you are using the LISP Editor to create a file and you do not want to leave the LISP environment. The method is also convenient if you are compiling a single function and want to quickly check for errors and correct them without leaving LISP. The method is necessary if the compilation depends on changes you have made to the LISP environment; that is, you have defined some macros or changed a package.

The COMPILE-FILE function returns a namestring corresponding to the output file it generates. Therefore, immediately after using the COMPILE-FILE function, you can load the resulting output file as follows:

```
Lisp> (LOAD *)
```

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 2.10.15 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.
/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 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

Table 2-3 (cont.)

Qualifier	Function
	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.
/[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.
/REMOVE=suspended-system-spec	Deletes global sections installed with the /INSTALL qualifier.
/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 Five 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 five ways called modes:

- INTERACTIVE -- to invoke an interactive LISP session (the default)
- COMPILE -- to compile LISP files

- RESUME -- to resume a suspended LISP system
- INSTALL -- to create a global section for the read-only code in a LISP suspended system
- REMOVE -- to delete a global section created with the /INSTALL qualifier

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
/INSTALL	INSTALL
/[NO]LIST	COMPILE
/[NO]MACHINE_CODE	COMPILE
/MEMORY	INTERACTIVE or COMPILE or RESUME
/[NO]OPTIMIZE	COMPILE
/[NO]OUTPUT_FILE	COMPILE
/REMOVE	REMOVE
/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 2.10.11 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 2.10.14). 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 2.10.14) 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 /INSTALL

The /INSTALL qualifier causes the read-only code in a LISP suspended system to be shareable, reducing the physical memory requirements in a multiuser system. Making the code shareable enables several people to simultaneously use the same read-only code. You need the SYSGBL (system global pages) and the PRMGBL (permanent global section) privileges to use the /INSTALL qualifier. A system manager generally uses this qualifier once when installing VAX LISP on a multiuser system. The default file type for a suspended system is SUS. For more information on this qualifier, see the *VAX LISP/VMS Installation Guide*.

Format

LISP/INSTALL=*suspended-system-spec*

Example

```
$ LISP/INSTALL=LISP$SYSTEM:LISPSUS.SUS
```

Mode

Install

2.10.7 /[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

2.10.8 /[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 2.10.7 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

2.10.9 /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 6000 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 6000 pages. If you specify fewer than 6000 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 6000 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 2.10.13 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

2.10.10 /[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

2.10.11 /[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 2.10.7) 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

2.10.12 /REMOVE

The /REMOVE qualifier deletes global sections installed by the /INSTALL qualifier. You can use the /REMOVE qualifier to remove outdated code when you add a new version of LISP to the system. You need the SYSGBL (system global pages) and the PRMGBL (permanent global section) privileges to use /REMOVE because it removes key system resources.

NOTE

If a new version of VAX LISP has been installed and you want to remove the old suspended system (the SUS file), be sure to specify an explicit version number with the /REMOVE qualifier.

Format

LISP/REMOVE=*suspended-system-spec*

Example

```
$ LISP/REMOVE=LISP$SYSTEM:LISPSUS.SUS;1
```

Mode

Remove

2.10.13 /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

2.10.14 /[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

2.10.15 /[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.

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 2.10.13 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.

CHAPTER 3

USING THE VAX LISP EDITOR

This chapter describes how to use the VAX LISP Editor to edit LISP objects and files containing LISP code. This chapter provides all the information you need to edit LISP and general text. If you want to learn more about the Editor, or wish to customize it in ways that are not covered in this chapter, refer to the *VAX LISP/VMS Editor Programming Guide*.

NOTE

This chapter assumes you are using the Editor in its default form to edit LISP objects or LISP files. That is, the Editor's major style is "EDT Emulation" and its minor style is "VAX LISP". If you are using or wish to use the "EMACS" style provided with the Editor, see Appendix B of this manual.

This chapter is divided as follows:

- Section 3.1 introduces the Editor and explains how to start it, how to get work into and out of it, and how to return to the LISP interpreter.
- Section 3.2 explains how to edit text, including special features for editing LISP objects and code.
- Section 3.3 shows how you can have more than one LISP object or file available for editing at one time and explains how to switch among the objects or files you are editing.
- Section 3.4 explains how to recover from problems while you are using the Editor.
- Section 3.5 shows how you can customize the Editor to suit your needs.

Each major section ends with a table of the commands and key bindings that are covered in that section.

> **Note to VAXstation users:** When you use the Editor on a VAXstation, screen behavior is different, and you can use the pointer to perform some editing operations. Throughout this chapter, these differences are noted at appropriate locations. Section 3.6 summarizes Editor behavior and use on a VAXstation.

3.1 INTRODUCTION TO THE EDITOR

The VAX LISP Editor is a general-purpose text editor. It includes some capabilities that make it particularly useful for editing LISP code. For example, the Editor matches parentheses and indents lines for you. It can also evaluate a LISP function definition or symbol value that you are editing.

You use the Editor directly from the LISP environment. The Editor is a part of LISP and cannot be used outside of LISP. You can move freely between the Editor and the LISP interpreter. When you go from the Editor to the interpreter, the Editor preserves the state of your work until you return to it.

The Editor is designed to work only on a video terminal or a VAXstation. It maintains the screen at all times to reflect the contents of the LISP object or file. When you insert text in the middle of lines or between lines, the Editor immediately adjusts the screen to show your modification.

You communicate with the Editor by using commands. Many commands are available. Keys or key sequences invoke the most useful commands, so you do not have to type the command names. Keys on the numeric keypad invoke a set of commands that emulate the EDT keypad editor, making the VAX LISP Editor similar to EDT.

The Editor allows you to have more than one LISP object or file available for editing at one time. Each object or file resides in its own buffer. Commands allow you to switch from one buffer to another, and you can view more than one buffer at a time, or more than one place in the same buffer.

The rest of this section describes the basics of using the Editor. Section 3.1.6 contains a table of the commands presented in this section.

3.1.1 Editing Cycle

An editing cycle starts when you are using the VAX LISP interpreter and you want to create or modify a LISP object or a file containing LISP code. The cycle is as follows:

- You start the Editor by calling the ED function, supplying as an argument the name of the object or the file specification of the file you wish to create or modify.
- You use Editor commands to edit the object or file. Most frequently used Editor commands are invoked by control characters or keys on the numeric keypad.
- If you are editing a LISP object, you use a command to make your edited version replace the function definition or value. If you are editing a file, you use a command to write the new or modified file out to the disk.
- You use a command to pause the Editor, returning you to the LISP interpreter.
- In the LISP interpreter, you can now use the new function definition or value of the object or you can load the new or modified file.
- If further modifications are required, you can use the ED function without arguments to return you to the Editor. Resuming the Editor in this way brings you back to the Editor state that existed when you paused the Editor.

This cycle can occur as many times and on as many objects or files as needed.

3.1.2 Invoking the Editor

The ED function invokes the VAX LISP editor. The first time you invoke the Editor during a LISP session, you should be sure to supply an argument to the ED function. The argument identifies the object or file you want to edit.

To edit a LISP object, give the object's symbol as the argument. For example, the following form edits the function definition of the symbol SHIP-ACCESSOR:

```
Lisp> (ED 'SHIP-ACCESSOR)
```

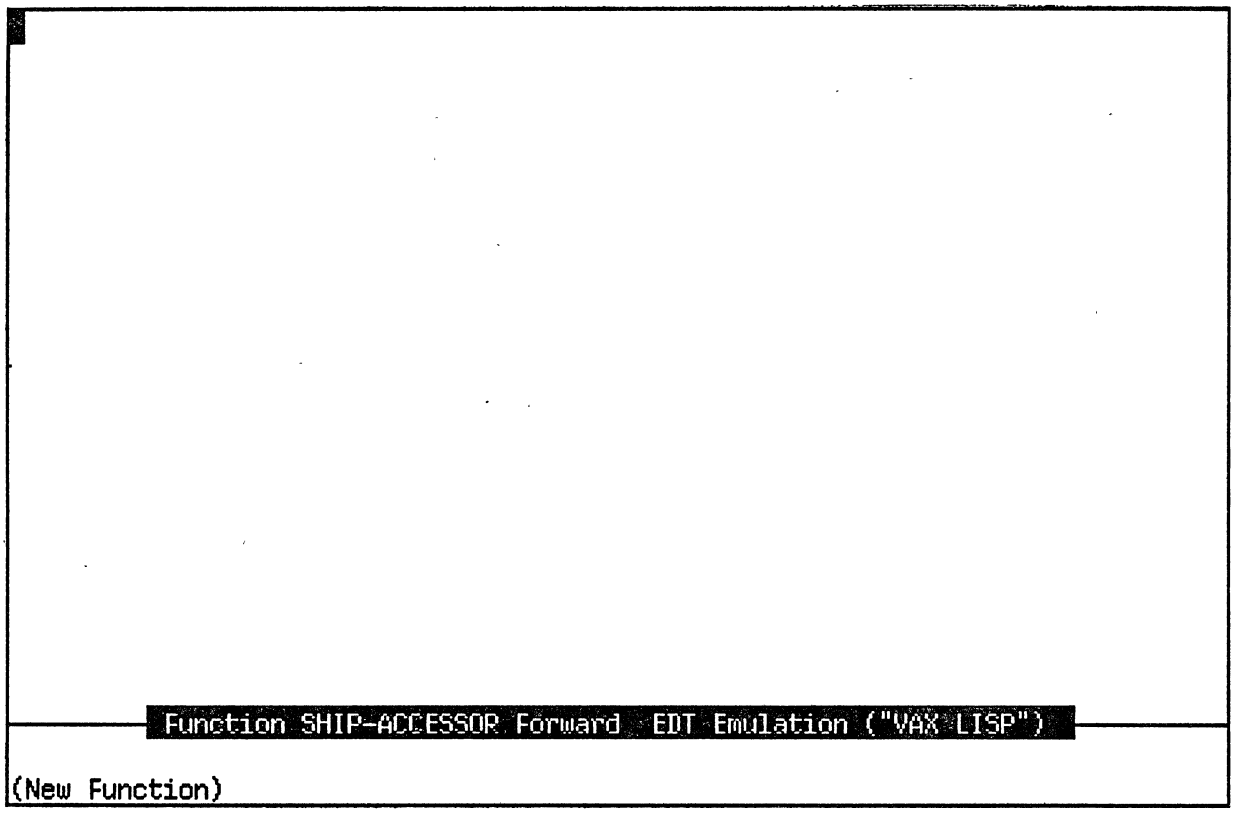
You can also edit the value of a LISP symbol, rather than its function definition, by using the :TYPE keyword with the ED function, as shown in this example:

```
Lisp> (ED 'SHIP-LIST :TYPE :VALUE)
```

To edit a file, give the file specification as the argument to the ED function. For example:

```
Lisp> (ED "CLOCK.LSP")
```

The first time you use the ED function, the screen clears. Then, after some initialization messages appear, the screen looks like this:



On a VAXstation: A new window appears; the window contains the Editor display. The window is taller than a standard 24-line screen, but otherwise the display is identical to that seen on a video terminal.

Note the following points about this screen display:

- The label strip near the bottom of the screen tells you that you are editing the function definition of SHIP-ACCESSOR, you are using the major style called "EDT Emulation" and the minor style called "VAX LISP", and your current movement direction is forward. The movement direction is useful to you while you are editing. You need not concern yourself with styles at this point.

- The information area at the bottom of the screen tells you that you are editing a new function definition. In general, information area contains short informational messages about Editor operations and errors.

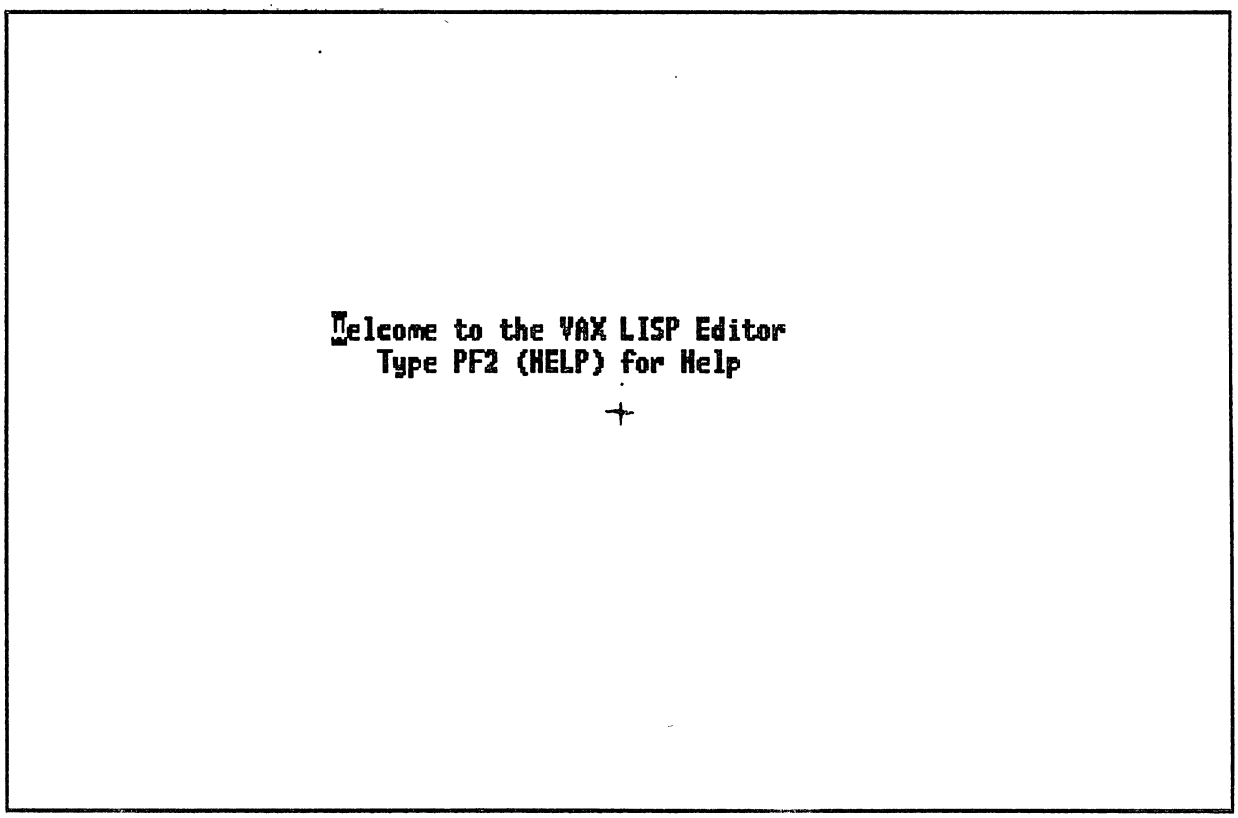
- The cursor is positioned at the upper left corner of the screen. The cursor shows where new text will be inserted.

After you have used the Editor, you can pause it (see Section 3.1.5) and return to the LISP interpreter. Later, you may want to resume your editing. If you want to return to the Editor state you left, simply call the ED function without arguments:

```
Lisp> (ED)
```

You can also supply an argument -- another LISP symbol or file -- when you resume the Editor. The LISP symbol or file you specify does not replace the symbol or file you were editing when you paused the Editor. The old symbol or file is made inactive, although it is still available for editing. See Section 3.3 for details.

If you use the ED function without arguments to start the Editor, you see the following screen display:



This means that the Editor is running but has nothing to edit. You can return to the LISP interpreter by typing CTRL/X CTRL/Z. Or, you can type CTRL/Z and enter an Editor command by name, as described in Section 3.1.3.

NOTE

You can use the BIND-KEYBOARD-FUNCTION function to bind a control character (such as CTRL/E) to the ED function, allowing you to invoke or resume the Editor asynchronously by typing the control character. If you do this, do not specify a value greater than 1 with BIND-KEYBOARD-FUNCTION's :LEVEL keyword. Using a value greater than 1 may disrupt the Editor's operation.

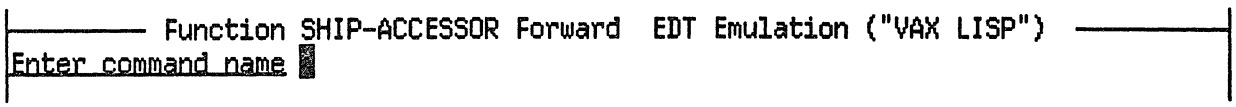
3.1.3 Interacting with the Editor

You interact with the Editor through commands. Commands do the following:

- Control the operation of the Editor: pause it, change from one buffer to another, set operating characteristics, and so on.
- Modify the LISP object or file that you are editing.

To enter a command to the Editor, you can type its name or type a key or sequence of keys that causes the command to be executed. The two ways are equivalent.

- To type a command by name, first type CTRL/Z, which causes a prompt to appear just below the label strip:



  Type the name of the command, then press RETURN. While you are typing, you can use any of the editing keys described in Section 3.2 to edit your input. You must supply the full name of the command. (However, once you have typed part of the command, the Editor can complete the name for you or display a list of command names that start that way; see Section 3.1.3.2.)

- If a key or key sequence is bound to the command, you can enter the command by typing that key or key sequence. Most

frequently used commands have keys or key sequences bound to them. You can use the "List Key Bindings" command to see which keys are currently bound to commands.

For example, to enter the "Pause Editor" command, you can type CTRL/Z, type "Pause Editor" in response to the command prompt, and press RETURN. Or, you can type CTRL/X CTRL/Z, which is bound to the "Pause Editor" command. Both methods cause the Editor to pause and return you to the LISP interpreter.

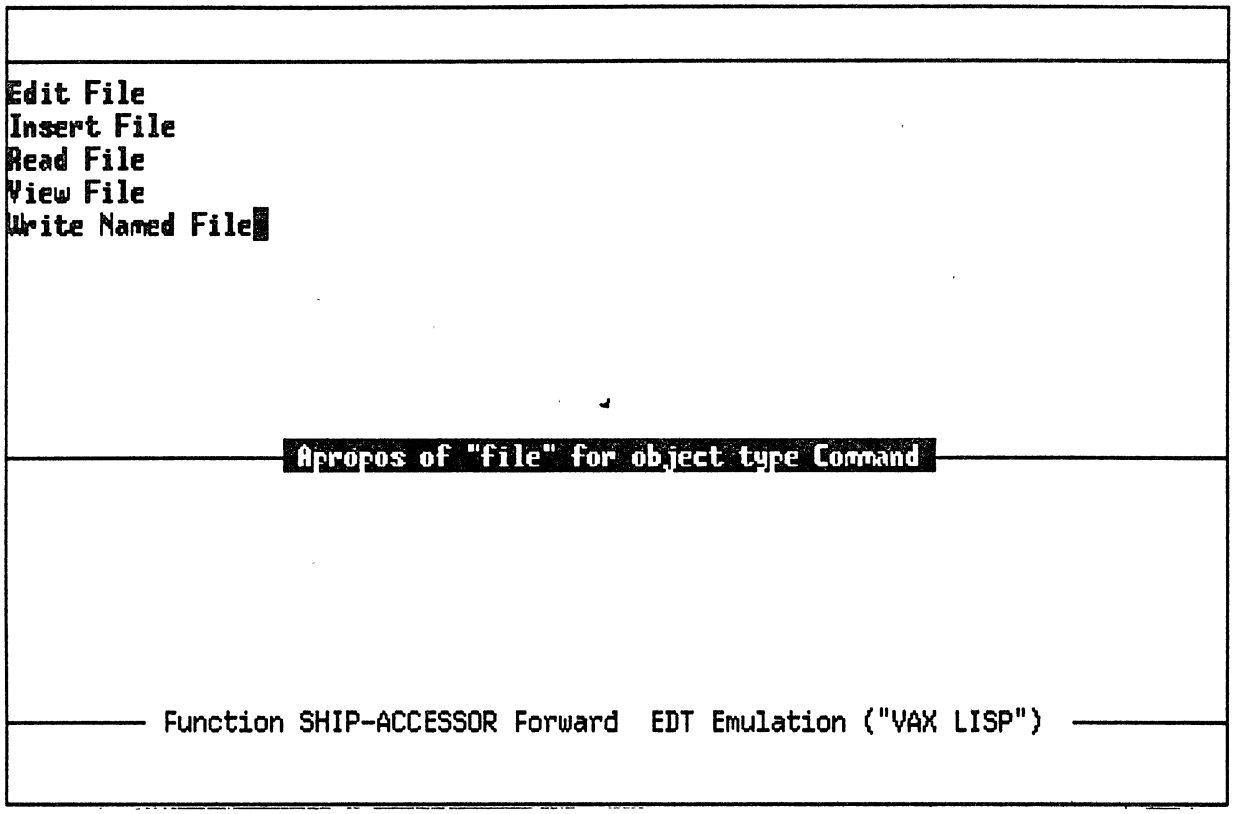
If you type CTRL/Z but then decide that you do not want to type a command, or if you decide to cancel a command in the middle of its execution, type CTRL/C. CTRL/C stops the current command and makes the Editor ready to accept other commands.

Commands are introduced throughout this chapter. Appendix C contains short descriptions of the available commands and their key bindings (if any).

3.1.3.1 **Getting Help** - The Editor provides different kinds of help. You can press the HELP key (either PF2 on the numeric keypad or the key labeled "Help" on the LK201 keyboard) at any time to get help on your current situation. A window called "VAX LISP Editor General Help" appears. It contains instructions on how to move around in a window and between windows and how to remove a window from the screen. To remove the window containing this help text from the screen, type CTRL/X CTRL/R.

If you press the HELP key while the Editor is displaying a prompt -- for example, after you have typed CTRL/Z -- the Editor displays help on the prompt. Typically, the help explains the prompt and describes the options you have. Press CTRL/V to scroll through this help text. The text will disappear from the screen when you have entered a response to the prompt and pressed RETURN.

The Editor also provides the "Describe" and "Apropos" commands to obtain information on Editor objects. These commands are similar to the LISP functions of the same names. The "Describe" command displays a description of an Editor command (by default) or other Editor object. The "Apropos" command lists all Editor commands or other specified Editor objects whose names contain a certain string. For example, using the "Apropos" command for the string "file" produces the following display:



You can also obtain descriptions of LISP symbols through the Editor when you are editing LISP code. The CTRL/? key invokes the LISP DESCRIBE function on the word at the current cursor position.

> **On a VAXstation:** You can also invoke the LISP DESCRIBE function by moving the pointer cursor to the symbol to be described and pressing the right pointer button.

You can use the cursor movement techniques described in Section 3.2.3 to move around in the window containing help text. When you are done, use the key sequence CTRL/X CTRL/R to remove this window and return to editing.

3.1.3.2 **Input Completion and Alternatives** - The Editor can help you enter responses to prompts in two ways. The first way is input completion. If you type CTRL/SPACE at any time while you are typing a response to a prompt, the Editor will attempt to complete your input for you. The Editor will complete as much of the input as it can, and display the status of the completion.

For example, if, to the "Enter command name" prompt, you type the string "pau" followed by CTRL/SPACE, the Editor will complete the command name "Pause Editor" and inform you that the input is complete. You can now press RETURN to execute the command. If, on the other hand, you type the string "new" followed by CTRL/SPACE, the Editor will be able to complete the input only as far as "New Li" and will then report that the input is ambiguous, because more than one command starts with the string "New Li".

At any point when entering information to a prompt, you can obtain a list of the available alternatives by typing PF1 PF2 on the numeric keypad. The Editor examines what you have typed so far and displays a list of all the commands starting that way. For example, when you have used input completion to get as far as "New Li", you can type PF1 PF2. The Editor will display a list of the commands beginning with "New Li". You can choose the command you want, enter enough of it to make the input unambiguous, and then use input completion (CTRL/SPACE) to complete the command name.

Input completion and alternatives are not restricted to command names. You can also use them to fill out file specifications and to obtain a list of all files matching a particular template. For example, assume you wish to edit an existing LISP file but are unsure of the name. You type CTRL/Z and enter the "Edit File" command, which then prompts you for a file name. You can type ".LSP" at this point, followed by PF1 PF2, to see a list of all files in your current directory having the file type "LSP". You can then edit your input by moving the cursor back to the beginning of the file specification and typing enough of the file name to distinguish it from other file names. Typing CTRL/SPACE at this point fills in the rest of the file specification.

3.1.3.3 **Errors and Other Problems** - If you make a minor error, the Editor displays a short error message in the information area. These error messages are usually sufficient to allow you to correct the problem. If the short message is not sufficient, the CTRL/X ? key sequence may display more information on the error.

If you make a major error, or if the Editor encounters an internal error from which it cannot recover, the Editor reports the error and asks if you wish to attempt to save your work. Depending on the nature and severity of the error, the Editor may not be able to save all your work. Section 3.4 contains more information on how to recover from these problems.

If the screen should become disrupted for some reason -- for example, MAIL messages arriving -- use the CTRL/W key to refresh the screen.

3.1.4 Moving Work Back to LISP

There are several ways to move your work back to the LISP environment. Use one of the methods described in this section if you want your work to be available in LISP.

Two commands, "Write Current Buffer" and "Write Modified Buffers", place your work back in a symbol or file:

- If you were editing the function definition or the value of a symbol, the commands cause the new function definition or value to replace the existing function definition or value.

- If you were editing a file, the commands write a new version of the file.

The difference between the two commands is that "Write Current Buffer" affects only the current buffer; that is, the buffer whose window contained the cursor when you entered the command. "Write Modified Buffers" affects any buffer you have worked on since the last time the buffer was written.

Neither of these commands pauses the Editor or alters the contents of your buffers. After using either command, you can immediately return to editing, or you can use the "Pause Editor" command to return to the LISP interpreter. If you were editing the function definition or value of a symbol, the new function definition or value is immediately available to you in LISP. If you were editing a file, you will have to load the file before you can use the modifications you made.

You can also move a function definition to the LISP environment by positioning the cursor in the function definition, then type CTRL/X CTRL/SPACE CTRL/X CTRL/A. This procedure causes the function definition containing the cursor to be evaluated. You can now return to the LISP interpreter and use the modified function definition. However, you must eventually include the definition in a file, or the modification will be lost when you exit LISP. This procdure is particularly useful when you are editing a file containing a number of definitions, and you want to modify only one of them.

If you are editing a function definition and you want to save it in a file, use the "Write Named File" command. This command prompts for the name of a file, and then writes the current buffer to the file.

3.1.5 Returning to the LISP Interpreter

When you have finished creating or modifying objects or files, you generally want to return to the LISP interpreter to test whatever you have written. The "Pause Editor" command returns control to the LISP interpreter; the key sequence CTRL/X CTRL/Z invokes "Pause Editor".

The "Pause Editor" command saves the state of your editing session. If you return to the session by calling the ED function without arguments, the Editor will be as you left it.

The "Pause Editor" command does not cause any of your buffers to be written. Before pausing the Editor, you must use one of the methods described in Section 3.1.4 to make your work available in the LISP environment. Also, if you pause the Editor without first writing your modified files and then exit LISP, the work you did on your files will be lost. (See Section 3.4 for information on partially recovering from this situation.)

> **On a VAXstation:** When you pause the Editor, the cursor returns to the LISP window.

In contrast to the "Pause Editor" command, the "Exit" command shuts down the Editor. If you use the "Exit" command, the Editor warns that changes will be lost and asks if you want to continue. If you type Y, the Editor allows you to save modified buffers on a buffer-by-buffer basis.

3.1.6 Summary of Commands

Table 3-1 provides a summary of the commands presented in this section and the keys (if any) that invoke those commands. These commands are useful for controlling the operation of the Editor. Subsequent sections in this chapter contain tables of commands that are useful in specific situations. Appendix C provides an alphabetic table of all the commands.

Table 3-1: General-Purpose Commands and Key Bindings

Name	Binding*	Description
Execute Named Command	CTRL/Z or keypad PF1 7 or DO*	Prompts for the name of a command to execute; type the name of the command, followed by RETURN
List Key Bindings	None	Displays a list of keys and key sequences currently bound to commands
Pause Editor	CTRL/X CTRL/Z	Pauses the Editor, saving its state, and returns to the LISP interpreter

* Keys marked with an asterisk (*) available only on LK201 keyboard

Table 3-1 (cont.)

Name	Binding*	Description
Write Current Buffer	None	Replaces a LISP symbol's function definition or value with the contents of the current buffer, or writes a new version of the file
Write Modified Buffers	None	Writes the contents of all modified buffers to the corresponding LISP object or new file version
Select Outermost Form	CTRL/X CTRL/SPACE	Highlights the outermost LISP form containing the cursor
Evaluate LISP Region	CTRL/X CTRL/A	Evaluates LISP code highlighted by "Select Outermost Form" or by other means, making the result available in the LISP interpreter
Write Named File	None	Prompts for a file name, then writes the current buffer to that file
Next Window	CTRL/X CTRL/N	Makes the next window be the current window; useful for moving into or out of help window
Remove Current Window	CTRL/X CTRL/R	Removes the current window from the screen; useful for getting rid of help window
Remove Other Windows	None	Removes windows other than the current window from the screen
Help	keypad PF2 or HELP*	Displays a window with help on your current situation
Prompt Scroll Help Window	CTRL/V (only while responding to prompt)	When used while responding to a prompt, causes the window containing help text to scroll

Table 3-1 (cont.)

Name	Binding*	Description
Prompt Show Alternatives	Keypad PF1 PF2 (only while responding to prompt)	When used while responding to a prompt, displays a list of input alternatives based on the context and what you have typed so far
Prompt Complete String	CTRL/SPACE (only while responding to prompt)	When used while responding to a prompt, attempts to complete the input based on the available alternatives and what you have typed so far
Describe	None	Displays a description of a command or other Editor object
Apropos	None	Displays a list of Editor commands or other Editor objects containing the string you supply
Describe Word	CTRL/?	Invokes the LISP DESCRIBE function for the word at the cursor location
Help on Editor Error	CTRL/X ?	Displays help on the last Editor error that occurred
Redisplay Screen	CTRL/W	Refreshes the screen

3.2 EDITING OPERATIONS

This section describes editing operations and how to perform them. The operations are those that you can perform in a single buffer; that is, while editing one LISP object or file. Section 3.3 explains how to deal with multiple buffers.

This section is divided as follows:

- Section 3.2.1 describes the numeric keypad you use to perform many editing operations.
- Section 3.2.2 describes how to insert text.
- Section 3.2.3 explains ways to move the cursor.
- Section 3.2.4 shows how you can modify text by deleting it and moving it.

- Section 3.2.5 explains how to cause an operation to occur more than once.
- Section 3.2.6 summarizes the commands described in this section.

3.2.1 Keypad

The Editor incorporates a set of commands and key bindings that cause it to behave like the EDT text editor. The keys on the numeric keypad are bound to the EDT-like commands. For the most part, you can use keypad keys as if you were using EDT, although there are some differences.

Figure 3-1 illustrates the numeric keypad. Each key has three items on it. Whatever appears on the actual key is shown in the lower right corner of each key in Figure 3-1. The meaning of the two names is as follows:

- The top name specifies the action that occurs if you press the key by itself.
- The bottom name specifies the action that occurs if you press and release the PF1 key (sometimes called the GOLD key) before pressing the key.

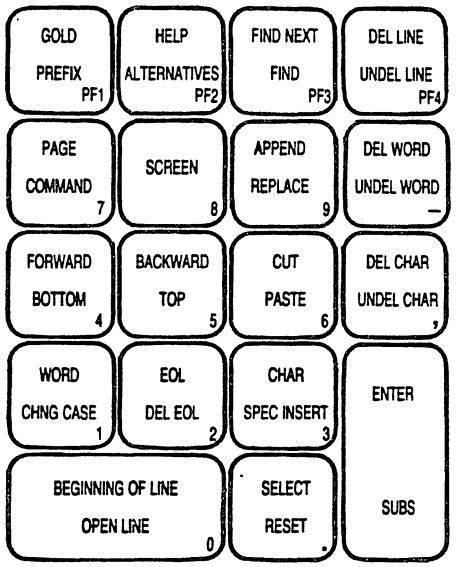
For example, if you press the 0 key by itself, the Editor moves the cursor to the beginning of a line. If you first press PF1 and then the 0 key, the Editor opens a new line at the cursor location.

For the rest of this section, keys will be referred to by the names of the actions the keys invoke. For example, the 0 key by itself is called the BEGINNING OF LINE key, while the sequence PF1 0 is called the OPEN LINE key.

3.2.2 Inserting and Formatting Text

This section describes ways you can create new text. Section 3.2.2.1 describes routine text insertion, Section 3.2.2.2 describes how to type and format LISP code, and Section 3.2.2.3 describes how to insert nongraphic characters in the text.

3.2.2.1 **Inserting Ordinary Text** - To insert text, simply type. All the keys corresponding to printing characters cause that character to appear preceding the character at the cursor location. If the cursor is in the middle of a line, the cursor and the characters to its right will be displaced further to the right to make room.



Note: The letters, numbers, and characters in the lower right corners of the keys are what actually appear on the keys of a VT100 keypad.

Figure 3-1: Numeric Keypad

You can start a new line by typing RETURN. If you type RETURN while the cursor is at the end of a line, you get a blank line with the cursor at the beginning. If the cursor is in the middle of a line, the line is broken in the middle.

The OPEN LINE key also starts a new line, but leaves the cursor at the end of the old line instead of the beginning of the new line.

3.2.2.2 **Typing and Formatting LISP Code** - The Editor includes several commands that help you to enter LISP code. Whenever you are editing a LISP object (either its function definition or value) or a file containing LISP code, the following key bindings are in effect:

- If you type a right parenthesis, the Editor highlights the corresponding left parenthesis for a moment. If the corresponding left parenthesis is not on the screen, the line that contains it is displayed in the information area with the parenthesis highlighted.

- The key sequence ESCAPE] closes the outermost form by inserting the correct number of parentheses at the cursor position. (Typing CTRL/[produces an ESCAPE.)

- The "New LISP Line" command is similar to RETURN, but it indents the new line properly with respect to the preceding

line. CTRL/J is bound to the "New LISP Line" command. (On a VT100 terminal, pressing the LINEFEED key results in CTRL/J.)

- The TAB key indents the line that currently contains the cursor, relative to the preceding LISP code.

- The CTRL/X TAB key sequence indents all the lines in the LISP outermost form that contains the cursor.

- Use CTRL/X ; to start a LISP comment at the end of a line of code. When you type CTRL/X ;, the Editor inserts enough spaces to move the cursor to the comment column, then inserts a semicolon and another space. If there is already a comment on the line, CTRL/X ; moves the cursor to the beginning of the comment.

On a VAXstation: Pressing the right pointer button when the pointer cursor is positioned at a close-parenthesis character highlights the matching open-parenthesis character.

3.2.2.3 **Inserting Nongraphic Characters** - You cannot insert some characters directly into your text. For example, you cannot insert a #\^X character by typing CTRL/X because the Editor interprets that character. The Editor provides two ways around this problem. In most cases, you can use the CTRL/X \ key sequence. After typing this sequence, the Editor will take the next character you type and insert it without interpretation. This procedure would handle the case of #\^X, for example; you would type:

CTRL/X \ CTRL/X

The Editor echo for this is

<^X>

In general, the Editor display for a nongraphic character is the LISP representation for the character, surrounded by angle brackets.

Some characters cannot be generated directly from the keyboard. You can use the SPEC(IAL) INSERT key to insert such characters. To use SPEC INSERT, you must first supply the decimal ASCII value of the character as a prefix argument (see Section 3.2.5). The SPEC INSERT key then inserts the character at the cursor location.

3.2.3 Moving the Cursor

To insert text where you want it to go, you have to move the cursor to that location first. A number of keys produce movements of various types. Section 3.2.3.1 describes how the keypad and arrow keys move the cursor. Section 3.2.3.2 describes how you can move the cursor within LISP code.

3.2.3.1 **Moving with the Keypad and Arrow Keys** - The keypad and arrow keys move the cursor in a way nearly identical to EDT. If you are familiar with EDT, you can skip this section; otherwise, the brief summary contained here should get you started.

The action produced by some keys depends on the current direction of movement. The current direction can be either forward or backward. The FORWARD key sets the current direction to forward, and the BACKWARD key sets it to backward. The label strip at the bottom of the window displays the current direction.

For other keys, the direction is always the same, regardless of the current direction.

The simplest way to move the cursor is with the arrow keys. Each arrow key moves the cursor one unit in the indicated direction. The left and right arrow keys move the cursor one character to the left or right, while the up and down arrow keys move the cursor up or down by one line in the current column. The arrow keys do not depend on the current direction.

Other keys allow you to move by line:

- The BEGINNING OF LINE key moves the cursor to the beginning of the next line (if the current direction is forward) or to the beginning of the current or previous line (if the direction is backward).

- The EOL (End Of Line) key moves to the end of the current or following line (if the current direction is forward) or to the end of the previous line (if the current direction is backward).

- CTRL/H moves the cursor to the beginning of the current or previous line, regardless of the current direction. (On a VT100, pressing the BACKSPACE key results in CTRL/H. On the LK201 keyboard, the F12 key produces the same action as CTRL/H.)

The WORD key moves to the beginning of the next word, if the current direction is forward, or to the beginning of the current or previous word, if the current direction is backward. In finding the beginning

of the word, the WORD key passes over most LISP syntax characters, such as parentheses, delimiters, single quote characters, and semicolons. #\NEWLINE characters are also passed over. In EDT, only spaces are passed over in finding the beginning of the next word.

The SCREEN key scrolls the text in the window. The text moves up if the current direction is forward and down if the current direction is backward. The amount of text that scrolls is equal to about two-thirds of the height of the window. The cursor moves only as much as is necesary to stay in the window. This behavior also differs from EDT.

The BOTTOM and TOP keys move the cursor to the end or the beginning of whatever you are editing.

You can also move the cursor to a specified text string. The FIND key prompts for a search string. You enter the string and terminate it with RETURN; the Editor then finds the first occurrence of that string in the current direction. (EDT allows you to terminate the search string with FORWARD or BACKWARD; the VAX LISP editor requires that you first set the current direction, then terminate the search string with RETURN or ENTER.) The FIND NEXT key finds the next occurrence of whatever string was last searched for.

3.2.3.2 **Moving in LISP Code** - Four bound commands allow you to move by LISP forms. The key sequence CTRL/X . is bound to the command "Next LISP Form", and the key sequence CTRL/X , is bound to the command "Previous LISP Form". These commands move the cursor from form to form within the current parentheses nesting level. The key sequence CTRL/X > (bound to the command "End of Outermost Form") moves the cursor to the end of the current or next outermost LISP form. The key sequence CTRL/X < (bound to the command "Beginning of Outermost Form") moves the cursor to the beginning of the current or previous outermost LISP form.

Four other commands allow you to move in lists. By default, no key sequences are bound to them. They are:

Backward Up List
Forward Up List
Beginning of List
End of List

See Appendix C for a brief description of these commands. Section 3.5 explains how you can bind keys to these (and other) commands.

3.2.3.3 **Moving with the Pointer (VAXstation Only)** - You can move the cursor by moving the pointer cursor to the desired spot in the text and pressing the left mouse button.

3.2.4 Modifying Text

In addition to inserting text, you can modify existing text by deleting it, moving it, and substituting in it. This section describes ways to modify text:

- Section 3.2.4.1 describes ways to delete portions of text.
- Section 3.2.4.2 shows how to undelete text you have just deleted.
- Section 3.2.4.3 explains how to move text from place to place by cutting and pasting it.
- Section 3.2.4.4 describes commands that modify text by changing its case.
- Section 3.2.4.5 shows two ways to substitute one text string for another.
- Section 3.2.4.6 explains how to insert text from a file or a buffer into your work.

3.2.4.1 **Deleting Text** - This section describes how to delete parts of your text by using keypad keys and keys on the main keyboard. Text that you delete disappears from the screen, but is not immediately discarded. The Editor maintains three areas for deleted text, one each for the last character, word, and line that you have deleted. The next section shows how to recover the contents of these areas and how to use them to move text from one place to another.

Two keys delete characters. The DELETE key (with the symbol <X] on the LK201 keyboard) deletes the character just before the cursor. The DEL CHAR key deletes the character at the cursor.

One or two keys delete words, depending on the terminal you are using. The DEL WORD key deletes from the current cursor position to the beginning of the next word. To find the beginning of the next word, the Editor passes over LISP syntax characters, such as parentheses, delimiters, single quote characters, and semicolons. Thus, DEL WORD deletes these syntax characters. DEL WORD also passes over and deletes a #\NEWLINE character at the end of a line.

If you are editing LISP code using a VT100, there is no key that deletes from the cursor position to the beginning of the current word. (In EDT, LINEFEED and CTRL/J do this; when editing LISP, however, CTRL/J is bound to "New LISP Line".) If you are using a terminal with the LK201 keyboard, the F13 key deletes to the beginning of the current word. It passes over LISP syntax characters and #\NEWLINE characters the same way the DEL WORD key does.

Three keys delete lines or portions of lines:

- The DEL LINE key deletes from the current cursor position to the end of the current line, including the #\NEWLINE character at the end of the line. If the cursor is positioned at the end of the line, DEL LINE simply deletes the #\NEWLINE character.

- The DEL EOL (End-Of-Line) key deletes from the current cursor position to the end of the current line, not including the #\NEWLINE character at the end of the line. If the cursor is positioned at the end of the line, DELETE LINE deletes the *next* line, including the #\NEWLINE.

- The CTRL/U key deletes from the current cursor position to the beginning of the current line. If the cursor is positioned at the beginning of the line, CTRL/U deletes the previous line.

3.2.4.2 **Undeleting Text** - Whenever you delete text, the Editor does not immediately discard it the text. Instead, the Editor temporarily saves the text, allowing you to put it back if you did not mean to delete it or to move it somewhere else.

- The UNDEL(ETE) CHAR key places the last character that was deleted at the cursor location.

- The UNDEL(ETE) WORD key places the last word or word portion that was deleted at the cursor location.

- The UNDEL(ETE) LINE key places the last line or line portion that was deleted at the cursor location.

Thus, if you want to move text from one place to another, you can first use the appropriate key to delete the text in its original location. Then move the cursor to the text's new location and use the appropriate UNDEL key to place the text there.

Copying text -- putting it in a new location while leaving it in its original location -- is similar to moving text, except that you undelete it in its original location before moving to the new location. You can undelete text as many times as you want.

3.2.4.3 **Cutting and Pasting Text** - Cutting and pasting consists of marking a block of text, removing ("cutting") it from its original location, then moving the cursor to a new location and inserting ("pasting") the text in the new location.

Before you cut text, you must mark the text to be cut. You use the SELECT key and the cursor movement keys to mark text. Move the cursor to one end of the text to be cut, then press SELECT. Move the cursor to the other end of the text to be cut. The Editor highlights the text between the character at which you pressed SELECT and the cursor. The highlighted text is called a select region. If you make a mistake while you are marking a select region, use the RESET key to cancel the select region.

You can mark a select region from the outermost LISP form containing the cursor by typing CTRL/X CTRL/SPACE.

> **On a VAXstation:** You can mark a select region by moving the pointer cursor to one end of the region, pressing and holding the left pointer button, moving the pointer cursor to the other end of the region, and releasing the button.

The CUT key removes all the text in the select region from the screen and places it in an Editor buffer called the paste buffer, replacing what was there. At this point, if you wish to replace the text, use PASTE. PASTE restores the text to its orginal location but does not remove the text from the paste buffer.

Now move the cursor to the desired location. Use the PASTE key to put the text there. You can paste text as often as needed, up until the time you cut more text.

> **On a VAXstation:** When a select region has been marked, you can cut it by pressing the middle pointer button. Then move the pointer cursor to the new location for the text and paste it by pressing and holding the left pointer button, then pressing the middle button.

The APPEND key is similar to the CUT key, except instead of replacing the contents of the paste buffer with the select region, the APPEND key appends the select region to the paste buffer contents. APPEND is convenient when you want to build a block of text by taking text from different locations.

The REPLACE key is similar to the PASTE key, except the REPLACE key requires that you have defined a select region before you use REPLACE. The REPLACE key deletes the select region and replaces it with the contents of the paste buffer.

3.2.4.4 **Changing Case** - One key and five commands provide ways to change the case of alphabetic characters in your text. Nonalphabetic characters are not affected.

The CHNG CASE key changes uppercase letters to lowercase and vice versa. It works as follows:

- If a select region is defined, CHNG CASE changes the case of all letters in the select region.

- If no select region is defined, CHNG CASE changes the case of the character at the cursor position and advances the cursor one character.

Four commands that allow you to make all the alphabetic characters in a select region or word be of one case are "Upcase Region", "Upcase Word", "Downcase Region", and "Downcase Word". To use the commands that affect a region, first define the select region, then type CTRL/Z and enter the command. To use the commands that affect a word, position the cursor anywhere in the word, then type CTRL/Z and enter the command.

Finally, the "Capitalize Word" command makes the first character of a word uppercase. Position the cursor anywhere in the word, then type CTRL/Z and enter the command.

3.2.4.5 **Substituting Text** - Two mechanisms are available for substituting one string for another throughout text. The first is simpler; the second is more powerful.

The SUBS(TITUTE) key substitutes the contents of the paste buffer for a search string. To use the SUBSTITUTE key, first load the paste buffer with the new string by typing SELECT, typing the new string, and then typing CUT. Next, search for the string to be replaced. If the first occurrence is in fact a string that you want to replace, type SUBS. The Editor deletes the search string and replaces it with the contents of the paste buffer, then automatically moves to the next occurrence of the search string. If you want to pass over an occurrence of the search string, type FIND NEXT to move to the next occurrence.

The "Query Search Replace" command is similar to SUBS but more versatile. The command prompts for a search string and a replacement string. At each occurrence of the search string, the Editor queries you. You can answer as follows:

- SPACE -- replace this occurrence and move to the next one.

- S -- replace this occurrence and stay here. This option allows you to see the results of the change before moving on. Use N to move to the next occurrence.

- . -- replace this occurrence and terminate the command.

- ! -- replace this occurrence and all remaining occurrences without further querying.

- N -- do not replace this occurrence and find the next occurrence.
- CTRL/C -- do not replace this occurrence and terminate the command.
- Q -- do not replace this occurrence and terminate the operation, returning the cursor to the point at which the search began.
- R -- enter a recursive edit, which you terminate with the "Exit Recursive Edit" command. The recursive edit allows you to clean up a replacement site without losing your place in the search cycle.
- ? -- display help on the possible responses to the query.

3.2.4.6 **Inserting a File or Buffer** - You can insert the contents of a file or a buffer at the cursor location, using the "Insert File" or "Insert Buffer" command. Each of these commands prompts for the name of a file or buffer and then inserts the contents of the file or buffer at the cursor location. You can use the ALTERNATIVES key or request input completion with CTRL/SPACE while responding to either prompt. (Section 3.3 contains more information about buffers.)

3.2.5 Repeating an Operation

You can cause the Editor to perform an action more than once by supplying a numeric prefix argument. The prefix argument causes the next command to be executed the number of times specified by the argument's value. For example, if the prefix argument is 3 and you press the BEGINNING OF LINE key, the cursor will move three lines instead of one.

You enter a prefix argument by using the PREFIX key and then typing the number in response to the prompt, followed by RETURN. The prefix affects only the next command you issue. You can issue the command either by typing CTRL/Z and the command name or by typing the key or key sequence bound to the command.

The prefix argument also causes printing characters to be inserted more than once. For example, to type 32 zeros, you could enter a prefix argument of 32, then type "0" once.

For some commands, you can supply a negative prefix argument. In general, the commands that are sensitive to the current direction will accept a negative prefix argument. They interpret a negative argument as an instruction to act in a direction opposite to the current

direction. For example, if the current direction is forward, a prefix argument of -3 followed by the WORD key causes the cursor to move three words backward. If the current direction is backward, a prefix argument of -3 causes the cursor to move three words forward.

3.2.6 Summary of Commands

Table 3-2 summarizes the commands presented in this section and their key bindings.

Table 3-2: Editing Commands And Key Bindings

Name	Binding*	Description
	Text Insertion Commands	
Open Line	OPEN LINE (keypad PF1 0)	Breaks a line at the cursor location
Insert Close Paren and Match	)	Inserts a close parenthesis at the cursor and highlights the matching open parenthesis
Close Outermost Form	ESCAPE]	Closes the outermost LISP form by inserting sufficient parentheses at the cursor position
Indent LISP Line	TAB or CTRL/I	Indents the current line to the appropriate position in relationship to preceding LISP code
New LISP Line	LINEFEED or CTRL/J	Starts a new line, indenting it in the proper LISP fashion
Indent Outermost Form	CTRL/X TAB	Indents all the lines in the outermost form containing the cursor
Move to LISP Comment	CTRL/X ;	Starts or moves to a comment on the current line

* Keys marked with an asterisk (*) available only on LK201 keyboard

Table 3-2 (cont.)

Name	Binding*	Description
Quoted Insert	CTRL/X \	Causes the next character typed to be inserted in the text without interpretation by the Editor
EDT Special Insert	SPEC INSERT (keypad PF1 3)	Inserts the character whose ASCII value is specified by the prefix argument
Insert File	None	Prompts for a file name, then inserts the contents of the file at the cursor location
Insert Buffer	None	Prompts for a buffer name, then inserts the contents of the buffer at the cursor location
Cursor Movement Commands		
EDT Set Direction Forward	FORWARD (keypad 4)	Sets the current direction to forward
EDT Set Direction Backward	BACKWARD (keypad 5)	Sets the current direction to backward
Forward Character	Right arrow	Moves the cursor to the next character
Backward Character	Left arrow	Moves the cursor to the previous character
Next Line	Down arrow	Moves the cursor to current column in next line
Previous Line	Up arrow	Moves the cursor to current column in previous line
EDT Move Character	CHAR (keypad 3)	Moves the cursor to the next or previous character, depending on current direction
EDT Move Word	WORD (keypad 1)	Moves the cursor to the beginning of the next, current, or previous word, depending on current direction and starting cursor location

Table 3-2 (cont.)

Name	Binding*	Description
EDT Beginning of Line	BEGINNING OF LINE (keypad 0)	Moves the cursor to the beginning of the next, current, or previous line, depending on current direction and starting cursor location
EDT End of Line	EOL (keypad 2)	Moves the cursor to the end of the next, current, or previous line, depending on current direction and starting cursor location
EDT Back to Start of Line	BACKSPACE or CTRL/H or F12*	Moves the cursor to the start of the current line or previous line, depending on starting cursor location
EDT Scroll Window	SCREEN (keypad 8)	Scrolls text up or down in the window, depending on current direction
Previous Screen	PREV SCREEN*	Moves the cursor up in the buffer by one screenful
Next Screen	NEXT SCREEN*	Moves the cursor down in the buffer by one screenful
End of Buffer	BOTTOM (keypad PF1 4)	Moves the cursor to the end of the current buffer
Beginning of Buffer	TOP (keypad PF1 5)	Moves the cursor to the beginning of the current buffer
EDT Move Page	PAGE (keypad 7)	Moves the cursor to the top of the next, current, or previous page, depending on current direction and starting cursor location
EDT Query Search	FIND (keypad PF1 PF3) or FIND*	Prompts for a string and moves the cursor to the first occurrence of that string in the current direction

Table 3-2 (cont.)

Name	Binding*	Description
EDT Search Again	FIND NEXT (keypad PF3)	Moves the cursor to the first occurrence in the current direction of the last string searched for
Moving by LISP Entities		
Previous Form	CTRL/X ,	Moves the cursor to beginning of current or previous LISP form in the current nesting level
Next Form	CTRL/X .	Moves the cursor to beginning of next LISP form in the current nesting level
Beginning of Outermost Form	CTRL/X <	Moves the cursor to beginning of enclosing outermost form, or to beginning of preceding outermost form if the cursor starts between outermost forms
End of Outermost Form	CTRL/X >	Moves the cursor to end of enclosing outermost form, or to end of following outermost form if the cursor starts between outermost forms
Text Modification Commands		
Deleting		
EDT Delete Character	DEL CHAR (keypad ,)	Deletes the character at the cursor location
EDT Delete Previous Character	DELETE (<X])	Deletes the character before the cursor location
EDT Delete Word	DEL WORD (keypad -)	Deletes from cursor location to beginning of next word
EDT Delete Previous Word	F13*	Deletes from cursor location to beginning of current word
EDT Delete Line	DEL LINE (keypad PF4)	Deletes from cursor location to beginning of next line

Table 3-2 (cont.)

Name	Binding*	Description
EDT Delete to End of Line	DEL EOL (keypad PF1 2)	Deletes from cursor location to end of line, or all of next line if cursor is at end of line
EDT Delete Previous Line	CTRL/U	Deletes from cursor location to beginning of current line, or all of previous line if cursor is at beginning of line
Undeleting		
EDT Undelete Character	UNDEL CHAR (keypad PF1 ,)	Inserts the last character deleted at the cursor location
EDT Undelete Word	UNDEL WORD (keypad PF1 -)	Inserts the last word deleted at the cursor location
EDT Undelete Line	UNDEL LINE (keypad PF1 PF4)	Inserts the last line deleted at the cursor location
Cutting, Pasting, and Substituting		
Set Select Mark	SELECT (keypad .) or SELECT*	Defines one end of a select region
Unset Select Mark	RESET (keypad PF1 .)	Cancels a select region
Select Outermost Form	CTRL/X CTRL/SPACE	Makes a select region from the outermost LISP form containing the cursor
EDT Cut	CUT (keypad 6) or REMOVE*	Removes the select region from the text and replaces the contents of the paste buffer with the select region
EDT Append	APPEND (keypad 9)	Removes the select region from the text and appends the select region to the contents of the paste buffer
EDT Paste	PASTE (keypad PF1 6) or INSERT*	Inserts the contents of the paste buffer at the cursor location

Table 3-2 (cont.)

Name	Binding*	Description
EDT Replace	REPLACE (keypad PF1 9)	Deletes the select region and replaces it with the contents of the paste buffer
EDT Substitute	SUBS (keypad PF1 ENTER)	Substitutes the contents of the paste buffer for the search string and moves to the next occurrence of the search string
Query Search Replace	None	Prompts for old and new strings, and at each occurrence of the old string prompts for an action; more versatile than "EDT Substitute"
Exit Recursive Edit	None	Terminates a recursive edit and returns to the editing level from which the recursive edit was initiated
Changing Case		
EDT Change Case	CHNG CASE (keypad PF1 1)	Changes the case of all characters in a select region or of an individual character
Upcase Region	None	Makes all characters in a select region uppercase
Downcase Region	None	Makes all characters in a select region lowercase
Upcase Word	None	Makes all characters in the word at the cursor location uppercase
Downcase Word	None	Makes all characters in the word at the cursor location lowercase
Capitalize Word	None	Makes the first character of the word at the cursor location uppercase
General		
Supply Prefix Argument	PREFIX (keypad PF1 PF1)	Prompts for a numeric prefix argument, which may cause the next command to repeat its action

3.3 USING MULTIPLE BUFFERS AND WINDOWS

The Editor can keep track of more than one LISP object or file at a time. The Editor holds each object or file that you are currently editing in a buffer. Commands let you move between buffers, create new buffers, and gain access to buffers through windows on the screen.

3.3.1 Introduction to Buffers and Windows

Buffers are Editor objects that contain the text of the symbol or file that you are editing and some information about the text -- for example, the position of the cursor when the text is displayed. The Editor displays the contents of buffers through windows on the screen. The Editor can keep track of many buffers at once, but normally displays the contents of no more than two buffers it has created for you at a time.

> **On a VAXstation:** It is important to distinguish between the VAXstation window that contains the Editor and the Editor windows that display the contents of buffers. The VAXstation window is equivalent to the screen of a video terminal. Editor windows appear in the VAXstation window.

The first time you use the ED function, you supply an argument specifying the symbol or file you want to edit. The Editor creates a buffer having the same name as this symbol or file and a window on the screen for this buffer. You can then enter and modify the text in the buffer.

If, after you return to the LISP interpreter, you use the ED function with a different symbol or file, the Editor creates another buffer and window. Both windows appear on the screen, but the window for the buffer most recently created is the current window. If you type characters or enter Editor commands, the buffer viewed through the current window will be affected.

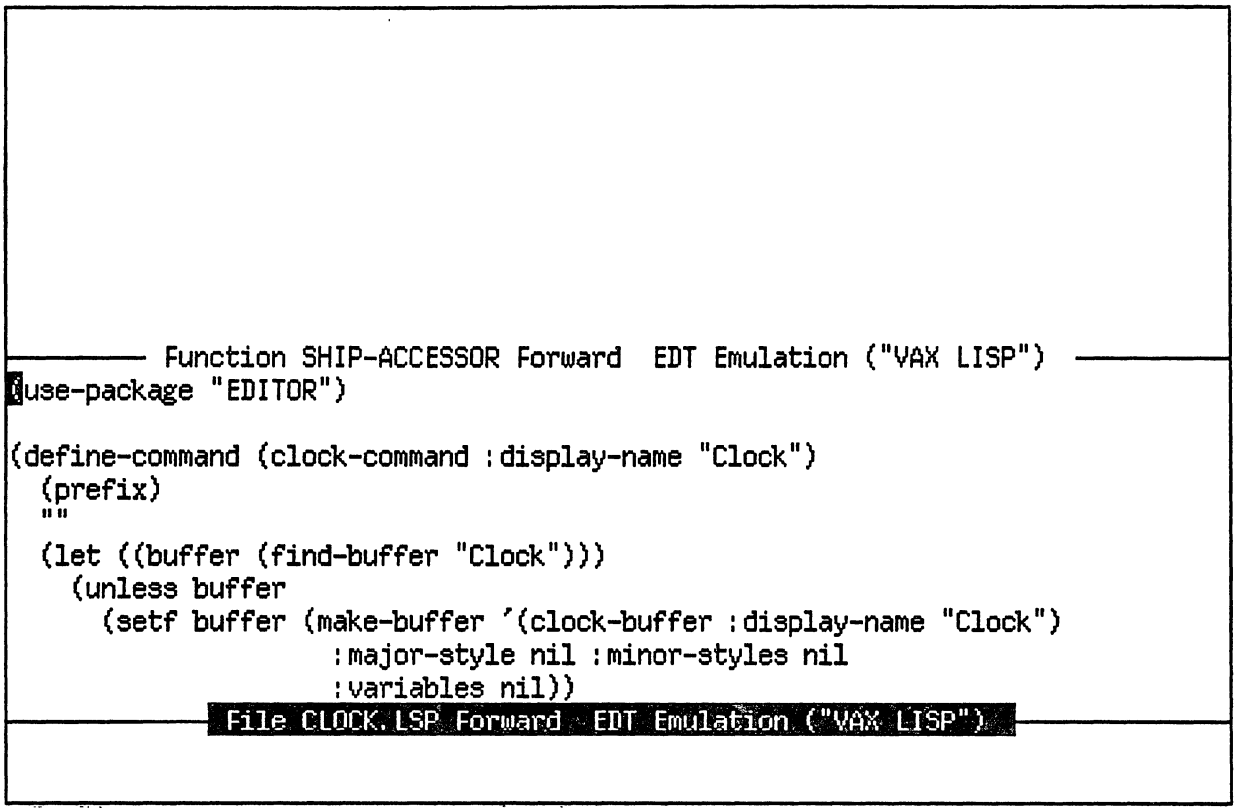
For example, if you first typed

```
Lisp> (ED 'SHIP-ACCESSOR)
```

and then, after pausing the Editor, typed

```
Lisp> (ED "CLOCK.LSP")
```

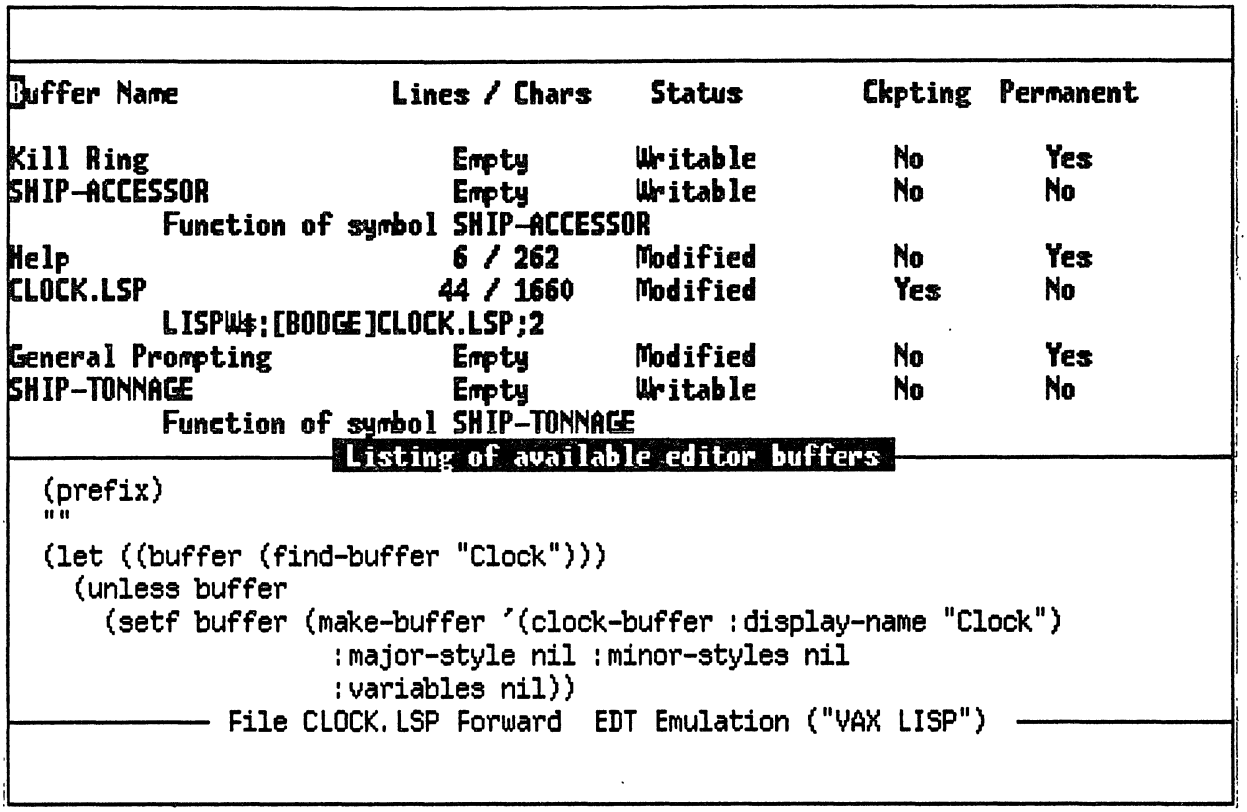
the screen might look like this:



Now, two label strips appear, one at the bottom of each window. The reverse-video label strip and the presence of the cursor in a window show you which window is current.

To change the current window, use the CTRL/X CTRL/N key sequence, making each window on the screen current in turn. When you change from one window to another, the cursor moves to the position it occupied when you last edited in that window.

Windows that display text you are editing are called anchored windows, because they are fixed at a particular spot on the screen. Unless you use the "Split Window" command, the Editor can by default display no more than two anchored windows at once. However, you can have more than two LISP objects or files available for editing at once, each occupying its own buffer. The "List Buffers" command displays a list of all the buffers in the Editor. For example, if you had used the ED function three times, "List Buffers" might result in the following display:



The buffers holding the objects and files that you are editing are identified by an additional line detailing the contents of the buffer. For example, the buffer named SHIP-ACCESSOR contains the "Function of symbol SHIP-ACCESSOR." The other buffers listed contain Editor information.

You can select a buffer for editing that is not currently on the screen with the "Select Buffer" command. This command prompts for the name of a buffer to edit. You can type part of the name, then use CTRL/SPACE to request that the Editor fill in the rest of the name. If you do not know which buffers are available, use ALTERNATIVES to see a list of their names.

When you select a buffer from among those not currently displayed, the Editor displays it in a new anchored window. If two anchored windows are already there, the Editor removes the least current one and replaces it with one displaying the contents of the buffer just selected.

"Removing" a window does not delete or modify the contents of the buffer. Removing a window simply causes the corresponding buffer to be no longer displayed, until the next time you select it. You can use the CTRL/X CTRL/R key sequence to remove the current window from the screen and the "Remove Other Windows" command to remove all windows other than the current window from the screen.

In addition to anchored windows, the Editor also has floating windows. Floating windows may be displayed anywhere on the screen, overlaying and obscuring the anchored windows that lie under the floating windows. The window in which help appears is a floating window. For the purpose of commands, these windows are just like anchored windows; CTRL/X CTRL/N moves the cursor to them in turn, CTRL/X CTRL/R removes them when they are current, and "Remove Other Windows" removes them when they are not current.

3.3.2 Creating New Buffers from Within the Editor

You do not need to return to the LISP interpreter to create a new buffer. Two commands allow you to start editing new LISP objects or files without leaving the Editor.

The "Ed" command works the same as the ED function. The "Ed" command prompts for each of the arguments that you would enter to the ED function. If you supply a symbol name, the Editor asks you to specify whether you want to edit the function definition or the value of the symbol. If you supply a character string containing a file specification, the Editor starts editing that file.

The "Edit File" command prompts you for a file to edit. The "Edit File" command differs from the "Ed" command in that the "Edit File" command allows you to request completion of the file name with CTRL/SPACE and a listing of possible file names with ALTERNATIVES. (See Section 3.1.3.2.)

3.3.3 Working with Buffers

Buffers generally take care of themselves. The only three common situations in which you need to deal with buffers directly are:

- When you need to save the contents of a buffer
- When you need to delete a buffer
- When two buffers have conflicting names

Buffers maintain some information about the state of your editing session with regard to the LISP object or file contained in the buffer. Specifically, a buffer keeps track of:

- The position of the cursor in the text
- The select region, if one is active

- Key bindings, if any keys are bound in the context of the buffer (see Section 3.5.1)

- The major and minor styles that are active in that buffer (see Section 3.5.1)

This information ensures that, when you select a buffer you worked on previously, it will be in the same state as it was when you left it.

3.3.3.1 **Saving Buffer Contents** - Three commands save buffer contents. The "Write Current Buffer" and "Write Modified Buffers" commands (discussed in Section 3.1) save the contents of the single current buffer and of all buffers that have been modified, respectively. The third way to save buffer contents is to use the "Exit" command and request that modified buffers be saved.

When you pause the Editor, your buffers are not written, but they are available to you when you resume the Editor. If, however, you should pause the Editor and then exit LISP, the contents of your buffers will be lost. (Section 3.4 explains how you can partially recover from this situation.)

3.3.3.2 **Deleting Buffers** - Two commands delete a buffer. "Delete Current Buffer" deletes the buffer you are currently working on; "Delete Named Buffer" prompts for a buffer name and deletes that buffer. Both commands check to see if the buffer has been modified and, if it has been, ask if you want to write the buffer before deleting it.

If you are editing an existing file and you delete the buffer associated with the file, the Editor does not delete the file. However, the Editor also does not create a new version of the file. For example, if you are editing the file CLOCK.LSP;1 and you delete the buffer "CLOCK.LSP", the file CLOCK.LSP;1 is not deleted. However, the file CLOCK.LSP;2, which would have been created if you had saved the buffer contents, is not created.

3.3.3.3 **Buffer Name Conflicts** - The Editor requires that buffer names be unique. This requirement can cause a problem in the following situations:

- You are trying to edit the function definition and the value of the same symbol

- You are trying to edit two files having the same name and type but differing in some other respect (version number, directory, and so on)

When your attempt to edit something creates a buffer name conflict, the Editor requests a new buffer name. You can type in any name you like. However, if you should type in no name and just type RETURN, the Editor deletes the current contents of the buffer, replacing them with whatever you are trying to edit.

3.3.4 Manipulating Windows

The commands most commonly used to manipulate windows have already been presented in this chapter:

- CTRL/X CTRL/N (bound to "Next Window") to make the next window the current window
- CTRL/X CTRL/R (bound to "Remove Current Window") to remove the current window from the screen
- "Remove Other Windows" to remove windows other than the current window from the screen

Other commands allow you to manipulate windows in other ways.

The "Grow Window" and "Shrink Window" commands make the current window larger and smaller, respectively. If no prefix argument is set, they make the window one line larger or smaller. If a prefix argument is set, they make the window larger or smaller by the number of lines specified in the prefix argument.

The "Split Window" command allows you to open two or more windows on a single buffer. The command causes the current window to be split in two, with identical text appearing in the two windows. Once created, the two windows can be treated as ordinary windows; the window-manipulation commands move between the two windows and remove them in the normal fashion. Each window maintains its own cursor position and scrolls separately from the other; but if you type or edit in one window, the change will appear in the other as well.

Split windows are useful if you want to examine two parts of the same buffer at one time, or if you want to move text from one place to another in a buffer. To move text, you would delete it or cut it in one window, move to the other window, and undelete the text or paste it.

You can have more than two windows on a buffer; just use "Split Window" repeatedly. The number of windows is limited only by the size of the screen; each window must have at least one line.

Although the "Split Window" command initially creates two windows on the same buffer, you can cause one of those windows to switch to another buffer. Use the "Select Buffer" command and specify a buffer

not currently displayed in a window. By repeatedly splitting windows and selecting new buffers, you can view as many buffers as you can fit windows on the screen.

3.3.5 Moving Text Between Buffers

It is frequently useful to be able to move or copy text from one buffer to another. For example, if you have worked on the definition of a function, you may want to move it to a buffer in which you are editing a file. Two general ways of doing this are:

- You can delete or cut the text from the source buffer, change to the destination buffer, and undelete or paste the text in the destination buffer
- To insert an entire buffer in another, use the "Insert Buffer" command

3.3.6 Summary of Commands

Table 3-3 summarizes the commands presented in this section and their key bindings. (Some of the general-purpose commands in Table 3-1 also pertain to buffers and windows.)

Table 3-3: Commands For Manipulating Buffers And Windows

Name	Binding	Description
Select Buffer	None	Prompts for a buffer name, then makes that buffer the current buffer and displays it in a window
List Buffers	None	Displays a list of all Editor buffers
Delete Current Buffer	None	Deletes the current buffer
Delete Named Buffer	None	Prompts for the name of a buffer, then deletes that buffer
Ed	None	Prompts for a symbol name or file specification to edit, then creates a new buffer for the symbol or file

Table 3-3 (cont.)

Name	Binding	Description
Edit File	None	Prompts for the name of a file, then creates a buffer for that file and a window into the buffer
Grow Window	None	Enlarges the current window by one line, or by the number of lines specified by the prefix argument
Shrink Window	None	Shrinks the current window by one line, or by the number of lines specified by the prefix argument
Split Window	None	Splits the current windows into two windows on the current buffer
Insert Buffer	None	Prompts for the name of a buffer, then inserts the contents of that buffer at the cursor location

3.4 RECOVERING FROM PROBLEMS

The Editor provides facilities that let you recover from problems with all or most of your work intact. Section 3.1.3.3 contains information on how the Editor responds to minor errors. This section describes checkpointing, by means of which the Editor protects work in progress.

Whenever you are editing a file, the Editor periodically makes a copy of the current state of that file. The copy is a separate disk file, called the checkpoint file. It has the same name as the file you are editing, and a file type composed as follows:

*type_version*_LSC

where *type* and *version* are the file type and version number, respectively, of the file you are editing. For example, if you are editing the file CLOCK.LSP;2, the associated checkpoint file will be named CLOCK.LSP_2_LSC.

While you are using the Editor or the LISP interpreter, an error may occur that returns you to DCL, or you may inadvertently exit LISP without first saving your Editor buffers, or the system may crash. In

any of these cases the current state of your Editor work is lost. However, the checkpoint files for any files you were editing still remain, reflecting the state of those buffers at the last time that checkpointing took place. To use a checkpoint file after you have lost the associated buffer, change its file type back to LSP. Then use the Editor to edit the file.

When checkpointing a file, the Editor displays the message "Checkpointing ..." in the information area. You can continue to type while checkpointing is taking place but whatever you type will not be displayed until checkpointing is complete. By default, the Editor checkpoints after every 350 commands that alter text in buffers. (Each keystroke that inserts a text character counts as a command.)

3.5 CUSTOMIZING THE EDITOR

You can customize the Editor to make it more convenient or comfortable to use. This section describes two ways to customize the Editor:

- Section 3.5.1 explains how to bind keys or key sequences to commands. You can bind keys to commands that have no keys bound to them by default, or you can change the default bindings.

- Section 3.5.2 describes keyboard macros. A keyboard macro is a sequence of keystrokes that the Editor captures for you; you can then replay the sequence at a later time.

The *VAX LISP/VMS Editor Programming Guide* explains how you can customize the Editor even further by creating new commands or new editing styles.

3.5.1 Binding Keys to Commands

As previously stated, you interact with the Editor by using commands. Many commands have keys or key sequences bound to them; others do not. One way you can customize the Editor is to bind a key or key sequence to a command. Once you have bound a key or key sequence to a command, typing that key or key sequence invokes the command.

The two ways to bind a key or key sequence to a command are:

- While using the Editor, you can use the "Bind Command" command.

- While using the LISP interpreter, you can use the BIND-COMMAND function. Your LISP initialization file can contain calls to BIND-COMMAND to set up the Editor.

These two methods are discussed in Sections 3.5.1.1 and 3.5.1.2, respectively.

No matter how you bind keys or key sequences to commands, there are two pieces of information you must supply and a third that you may supply:

- You must supply the name of the command to be invoked.
- You must supply the key or key sequence to bind to the command. Sections 3.5.1.1 and 3.5.1.2 describe how to specify the key or key sequence. Section 3.5.1.3 contains suggestions on how to select a key or key sequence to bind.
- You can optionally supply the context in which the binding is effective. Section 3.5.1.4 explains the key binding context.

3.5.1.1 **Binding Within the Editor** - The "Bind Command" command allows you to bind a key or key sequence to a command while using the Editor. This command prompts you for each of the three items you need to specify a complete binding.

The "Bind Command" command first prompts you for the name of the command you wish to have bound. You can use input completion and alternatives to get a complete command name.

The second prompt is for the key sequence. Type the actual key or key sequence that you want to bind to the command -- *not* a LISP representation of the characters. However, you cannot type control characters or function keys unless you use CTRL/X \ to quote them. Since most bindings involve control characters or function keys, you will tend to use CTRL/X \ most of the time.

For example, assume that you want to bind the key sequence CTRL/X CTRL/O to a command. Both CTRL/X and CTRL/O are control characters so they must both be quoted. In response to the "Enter key sequence" prompt, you would type:

```
CTRL/X \ CTRL/X CTRL/X \ CTRL/O
```

After you completed this sequence, the prompting area would appear like this:

```
Enter key sequence <^X><^O>
```

Function keys, arrow keys, and keys on the numeric keypad must also be quoted. Each of these keys generates more than one character when it is struck, so more than one character appears in the prompting area. For example, to bind the F12 key to a command, you would type:

```
CTRL/X \ F12
```

This sequence is echoed in the prompting area as:

```
Enter key sequence <ESCAPE>[24~
```

The third prompt is for the binding context. The context can be :GLOBAL (the default) or a particular style or buffer. Type :STYLE or :BUFFER, followed by RETURN, to specify one of these options. The Editor then prompts for the name of the style or the buffer. (See Section 3.5.1.4 for more information on binding context.)

3.5.1.2 **Binding from the LISP Interpreter** - The BIND-COMMAND function allows you to establish key bindings while you are using the LISP interpreter. BIND-COMMAND is especially useful in your LISP initialization file to set up the bindings you use all the time.

The BIND-COMMAND function takes three arguments. The first argument is the name of the command you wish to have bound, in the form of a character string.

The second argument is the key or key sequence that is to invoke the command. A single key may be given as a LISP character. A key sequence must be given as a vector or list of characters.

For all the keys on the main part of the keyboard -- those keys that produce letters, numbers, and other printing symbols -- you may use any valid LISP representation of the character. For example, "A" is #\A, "a" is #\a, and "CTRL/A" is #\^A. The character transmitted by the BACKSPACE key on a VT100 can be #\BACKSPACE, #\BS, or #\^H. The LISP function CHAR-NAME-TABLE displays a table of the LISP names for control characters.

The remaining keys on the keyboard -- the numeric keypad, arrow keys, editing keys, and function keys -- transmit more than one character when struck. Table 3-4 lists each key and the character sequence it generates. The VT100 keyboard lacks function and editing keypad keys, but the numeric keypad keys and arrow keys generate the same characters listed in Table 3-4.

Some of the function keys on the LK201 keyboard are commonly associated with particular characters. For example, the F12 key is associated with BACKSPACE and the F13 key with LINEFEED. However, these function keys do not actually transmit these characters, and the Editor does not treat them as having transmitted these characters.

Table 3-4: Characters Generated by Keys

Key	Characters Generated
Numeric Keypad Keys (LK201 and VT100)	
keypad 0	#\ESCAPE # O # p
keypad 1	#\ESCAPE # O # q
keypad 2	#\ESCAPE # O # r
keypad 3	#\ESCAPE # O # s
keypad 4	#\ESCAPE # O # t
keypad 5	#\ESCAPE # O # u
keypad 6	#\ESCAPE # O # v
keypad 7	#\ESCAPE # O # w
keypad 8	#\ESCAPE # O # x
keypad 9	#\ESCAPE # O # y
keypad -	#\ESCAPE # O # m
keypad ,	#\ESCAPE # O # l
keypad .	#\ESCAPE # O # n
keypad ENTER	#\ESCAPE # O # M
keypad PF1	#\ESCAPE # O # P
keypad PF2	#\ESCAPE # O # Q
keypad PF3	#\ESCAPE # O # R
keypad PF4	#\ESCAPE # O # S
Arrow Keys (LK201 and VT100)	
Up Arrow	#\ESCAPE #\[#\A
Down Arrow	#\ESCAPE #\[#\B
Right Arrow	#\ESCAPE #\[#\C
Left Arrow	#\ESCAPE #\[#\D
Function, HELP, and DO keys (LK201)	
F6	#\ESCAPE #\[#\1 #\7 #\~
F7	#\ESCAPE #\[#\1 #\8 #\~
F8	#\ESCAPE #\[#\1 #\9 #\~
F9	#\ESCAPE #\[#\2 #\0 #\~
F10	#\ESCAPE #\[#\2 #\1 #\~
F11	#\ESCAPE #\[#\2 #\3 #\~
F12	#\ESCAPE #\[#\2 #\4 #\~
F13	#\ESCAPE #\[#\2 #\5 #\~
F14	#\ESCAPE #\[#\2 #\6 #\~
HELP (F15)	#\ESCAPE #\[#\2 #\8 #\~
DO (F16)	#\ESCAPE #\[#\2 #\9 #\~
F17	#\ESCAPE #\[#\3 #\1 #\~
F18	#\ESCAPE #\[#\3 #\2 #\~
F19	#\ESCAPE #\[#\3 #\3 #\~
F20	#\ESCAPE #\[#\3 #\4 #\~

Table 3-4 (cont.)

Key	Characters Generated
Editing Keys (LK201)	
FIND (E1)	#\ESCAPE #\[#\1 #\~
INSERT HERE (E2)	#\ESCAPE #\[#\2 #\~
REMOVE (E3)	#\ESCAPE #\[#\3 #\~
SELECT (E4)	#\ESCAPE #\[#\4 #\~
PREV SCREEN (E5)	#\ESCAPE #\[#\5 #\~
NEXT SCREEN (E6)	#\ESCAPE #\[#\6 #\~

The third argument to BIND-COMMAND, which is optional, specifies the binding context. If you omit this argument, the context is global; that is, the key binding is effective everywhere in the Editor. If you include this argument, supply it in the form

```
'(:STYLE "style-name")
```

or

```
'(:BUFFER "buffer-name")
```

Section 3.5.1.4 describes binding context in more detail.

The following example binds the key sequence CTRL/X CTRL/O to the "Remove Other Windows" command globally:

```
(BIND-COMMAND "Remove Other Windows" '#(#\^X #\^O))
```

Alternatively, you could globally bind the key sequence PF1 REMOVE (the REMOVE key is on the LK201's editing keypad) to "Remove Other Windows" as shown here:

```
(BIND-COMMAND "Remove Other Windows"
              '#(#\ESCAPE #\O #\P #\ESCAPE #\[ #\3 #\~))
```

To bind the F12 key on an LK201 keyboard to the "EDT Back to Start of Line" command in the "EDT Emulation" style, you would use the following function:

```
(BIND-COMMAND "EDT Back to Start of Line"
              '#(#\ESCAPE #\[ #\2 #\4 #\~)
              '(:STYLE "EDT Emulation"))
```

Following execution of this function, the F12 key moves the cursor to the beginning of the line, but only if the "EDT Emulation" style is active. (This binding is in effect by default.)

On a VAXstation: You can also bind actions of the pointing device (movement and buttons) to commands. See the description of the BIND-POINTER-COMMAND function in the *VAX LISP/VMS Editor Programming Guide*.

3.5.1.3 **Selecting a Key or Key Sequence** - You can bind almost any key or key sequence to a command, but you should be careful that your selection does not interfere with Editor operation. This section explains restrictions and provides hints to help you make a selection.

The three control characters you must not include anywhere in a key sequence are:

- The cancel character, CTRL/C by default, which terminates an Editor operation. You cannot include CTRL/C in a key sequence because typing CTRL/C at any time stops the collection of keystrokes and returns the Editor to the end of the last completed command.

- CTRL/S and CTRL/Q, which are interpreted by the operating system (they stop output to the terminal and resume it, respectively) and therefore never reach the Editor for interpretation.

You should not use any graphic (printing) character to start a key sequence, although you can use graphic characters elsewhere in the sequence. If you start a key sequence with, say, the letter A, you will never be able to type the letter A as part of a word. The Editor, as soon as it sees the A, will recognize it as the beginning of a key sequence; unless the next character(s) complete the sequence, the Editor will signal an error and discard the A.

When you include an alphabetic character in a key sequence, remember that the Editor differentiates between upper- and lower-case. For example, the following two key sequences are different:

```
'#(#\^X #\A)
'#(#\^X #\a)
```

By convention, the three keys used to start a key sequence are CTRL/X, ESCAPE, and keypad PF1. You can, of course, use others if you choose, as long as they are nonprinting. (On terminals that do not have an ESCAPE key, CTRL/[transmits the #\ESCAPE character.)

Finally, be careful not to select a key or key sequence that is already bound to a useful command. Appendix C contains a list of all the key bindings supplied with the Editor. Section 3.5.1.4 explains how a single key or key sequence can be bound to two different commands in different contexts.

3.5.1.4 **Key Binding Context and Shadowing** - When you bind a key or key sequence to a command, you can specify the context in which that binding is effective. Specifying a context means that the key or key sequence invokes the command only in that particular context.

The three general types of context are:

- The buffer context. If the context is a particular buffer, the key or key sequence invokes the command only if that buffer is current.
- The style context. If the context is a particular style, the key or key sequence invokes the command only if that style is the major style or one of the minor styles that is active in the current buffer.
- The global context. If the context is global, the key or key sequence always invokes the command. The default context is global.

Styles

A style is a collection of key bindings and of other Editor characteristics that cause the Editor to behave in a certain way. The two styles that you encounter in the default Editor are named "EDT Emulation" and "VAX LISP". The "EDT Emulation" style causes the numeric keypad to generate editing actions similar to those of EDT. The "VAX LISP" style provides access to the Editor's ability to edit LISP code easily.

An Editor buffer can have one major style and one or more minor styles active at any time. You can tell which styles are active by looking at the label strip for the buffer:

The major style is generally established before the Editor is started. Minor styles are activated automatically, depending on what is being edited. For example, whenever you edit a LISP object or a file having the type LSP, the "VAX LISP" style is activated for that buffer as a minor style.

Shadowing

It is possible to bind the same key or key sequence to two different commands. If the contexts of the two bindings are the same, then the second binding replaces the first one. If, however, the two bindings have different contexts, then the key or key sequence may invoke

either command, depending on the situation at the time. To locate a command to execute when a key is pressed, the Editor:

1. First checks to see if that key is bound in the context of the current buffer.

2. Next checks to see if that key is bound in the context of one of the current minor styles, examining the most recently activated style first.

3. Next checks to see if that key is bound in the context of the current major style.

4. Next checks to see if that key is bound in the global context.

As soon as the Editor finds a command to execute, it does so. Therefore, if the same key or key sequence is bound in, say, the current minor style and the current major style, the binding in the minor style "shadows," or takes precedence over, the binding in the major style.

For example, the CTRL/J key is bound to "EDT Delete Previous Word" in the "EDT Emulation" style and to "New LISP Line" in the "VAX LISP" style. When you are editing LISP code, "EDT Emulation" is the major style and "VAX LISP" is the minor style. Therefore, the binding of CTRL/J to "New LISP Line" shadows the binding to "EDT Delete Previous Word".

3.5.2 Keyboard Macros

A keyboard macro is a series of keystrokes that you ask the Editor to "remember" for future use. The keystrokes can be keys that insert characters, keys or key sequences that invoke editing commands, or even commands that you type in and that issue additional prompts. A keyboard macro is useful whenever you have a series of identical, complicated operations to perform.

To begin a keyboard macro, type CTRL/X (. Everything you type from that point is executed normally, but is also stored for future use. Typing CTRL/X) stops the storage of keystrokes. To execute a keyboard macro, type CTRL/X CTRL/E. This sequence causes the current keyboard macro to be "played back" starting at the current cursor location. A keyboard macro that you define in this way lasts until you define another keyboard macro.

You can also use the "Start Named Keyboard Macro" command to define a keyboard macro having a name. Use the "Start Named Keyboard Macro" command as you would the CTRL/X (key sequence. The command prompts you for a name. After you enter the name, the Editor starts

remembering keystrokes. Terminate the macro with CTRL/X). The macro thus defined is the current keyboard macro (you can invoke it with CTRL/X CTRL/E) but it is also a named entity that you can treat like a command. You can execute it as a named command or bind a key to it. A named keyboard macro remains accessible by name even after another keyboard macro has been defined.

A keyboard macro may not work properly if the context changes between the time the macro is created and the time it is executed. For example, if you switch to a buffer that has a different minor style active, the commands invoked by the keyboard macro may fail.

3.5.3 Summary of Commands

Table 3-5 summarizes the commands presented in this section and their key bindings.

Table 3-5: Commands For Customizing The Editor

Name	Binding	Description
Bind Command	None	Prompts for a command name and a key sequence to bind to it
Start Keyboard Macro	CTRL/X (	Starts collecting keystrokes for a keyboard macro
Start Named Keyboard Macro	None	Prompts for a name, then starts collecting keystrokes for a keyboard macro having that name
End Keyboard Macro	CTRL/X)	Terminates the collection of keystrokes for a keyboard macro
Execute Keyboard Macro	CTRL/X CTRL/E	Executes the current keyboard macro

3.6 USING THE EDITOR ON A VAXSTATION

The behavior and capabilities of the Editor when operating on a VAXstation are similar to its operation on an ordinary video terminal. The same commands are available and the same key bindings are in effect. However, the Editor incorporates several features that make use of VAXstation capabilities. This section summarizes those features.

3.6.1 Screen Appearance and Behavior

The most obvious difference in Editor behavior on a VAXstation is that the Editor occupies a separate VAXstation window from the LISP interpreter. This window has the title "VAX LISP Editor". It is created the first time you start the Editor, and the cursor is shifted to it from the window containing the LISP prompt. The Editor window is taller than the 24 lines normally contained in a video terminal but is otherwise identical to the Editor display described throughout this chapter.

When you pause the Editor, the cursor returns to the LISP prompt. When you resume the Editor, the cursor moves to the spot it occupied in the Editor window when you paused the Editor.

If you select the Delete option from the Window Options menu, the Editor exits, giving you the opportunity to save buffers first.

While you are using the Editor, you cannot use the LISP interpreter, even though the window containing the LISP prompt is still on the screen. You cannot use the LISP interpreter until you pause or exit from the Editor.

3.6.2 Editing with the Pointer

You can use the mouse or other pointing device to perform some editing tasks. You can select a new window to be the current window, move the text insertion cursor within the current window, and cut and paste text. When you are editing LISP code, you can also select LISP forms, describe LISP symbols, and match parentheses.

3.6.2.1 **The Pointer Cursor** - The pointer cursor is the cursor that you move around the screen by moving the pointing device. By contrast, the text insertion cursor is the blinking cursor that you move around the Editor windows using conventional Editor commands.

The appearance of the pointer cursor changes when it is in the LISP Editor window. When the pointer cursor is in the window that represents the current buffer, it looks like this:

When the pointer cursor is in a window that represents a buffer other than the current buffer, it looks like this:

When the pointer cursor is in the information area, it looks like this:

3.6.2.2 **Selecting and Removing Windows** - When the pointer cursor is in a window other than the current window, press the left pointer button to make that window the current window. The text insertion cursor is placed where it was the last time that window was current. Press the middle pointer button in a window other than the current window to remove that window from the screen.

3.6.2.3 **Moving the Text Insertion Cursor and Marking Text** - When the pointer cursor is in the current window, press the left pointer button to move the text insertion cursor to the pointer cursor. If you release the left button without moving the pointer cursor, the text insertion cursor stays in the same place. However, you can also leave the left button down and move the pointer cursor. If you do this, the text insertion cursor also moves. The text between where you first pressed the left button and where you finally release it is marked as a select region.

If you are editing LISP code, you can use the left pointer button to mark LISP forms as select regions. The first time you press the button, the text insertion cursor moves to the pointer cursor. Each time you press the button without moving the pointer cursor, a LISP form that encloses the pointer cursor is marked. Enclosing forms are marked until the outermost form is reached.

3.6.2.4 **Cutting and Pasting** - In the current window, press the middle pointer button to cut text that has been marked as a select region. Press the middle pointer button with the left pointer button depressed to paste text from the paste buffer at the pointer cursor position. To cut and paste text, follow these steps:

1. Mark the text to be cut using any method.
2. Press the middle pointer button to cut the text.
3. Move the pointer cursor to the position at which you want to paste the text.
4. Press and hold the left pointer button, then press the middle button.

3.6.2.5 **Invoking the DESCRIBE Function And Matching Parentheses** - When you are editing LISP code, you can use the right pointer button to invoke the LISP DESCRIBE function. Move the pointer cursor to a symbol, then press the right button. The Editor's help window appears and displays the results of using the DESCRIBE function on that symbol.

If you move the pointer cursor to a right parenthesis and press the right pointer button, the matching left parenthesis is highlighted.

3.6.2.6 **Information About Pointer Effects** - You can find out what action a pointer button invokes by moving the pointer cursor to the information area. The pointer cursor then appears as a large question mark. When you press any pointer button, the name of the command invoked by that button is displayed in the information area. Note that the command displayed is the one invoked by that button when the pointer cursor is in the current window. When you release the button, the command (if any) invoked by releasing the button is displayed.

Moving the pointer cursor in the information area with the buttons held a certain way displays the command that is invoked by pointer movement with the buttons in that state.

3.6.3 Binding Pointer Buttons to Commands

Binding pointer buttons to commands is analogous to binding keys or key sequences to commands. See the *VAX LISP/VMS Editor Programming Guide* for information.

CHAPTER 4

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.

4.1 ERROR HANDLER

The VAX LISP error handler function, UNIVERSAL-ERROR-HANDLER, 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 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.

4.2 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
- The name of the function that was used to signal the error -- ERROR, CERROR, or WARN
- 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 at which 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.

4.2.1 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 4-1). The error description is a message telling why the error occurred. 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 qualifier you use

with the LISP command sets the value of the *ERROR-ACTION* variable when you invoke the LISP system (see Chapter 2). When the value is :EXIT (you used the ERROR_ACTION=EXIT form of the qualifier), the error handler causes the LISP system to exit on an error; when the value is :DEBUG (you used the ERROR_ACTION=DEBUG form of the qualifier), the default in an interactive session), 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 at which a fatal error occurred.

The *ERROR-ACTION* variable is described in Part II and the debugger is described in Chapter 5.

4.2.2 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 4-1). The error description is a message telling why the error occurred. 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 is:

```
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 in the same way it checks the value after a fatal error (see Section 3.2.1).

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 Chapter 5.

4.2.3 Warnings

A warning is an error condition that exists in your program, which may or may 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 4-1). The error description is a message telling why the error occurred. 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 is:

```
Warning in function TE (signaled with WARN).
3 is not a symbol.
```

After the message is displayed, the error handler checks the value of the *BREAK-ON-WARNINGS* variable in the same way it checks the value *ERROR-ACTION* variable after a fatal error (see Section 3.2.1).

NOTE

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

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 Chapter 5.

4.3 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:

1. Define the error handler.
2. Bind the *UNIVERSAL-ERROR-HANDLER* variable to your defined handler.

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

4.3.1 Defining an Error Handler

To define an error handler, you must define an error handler function. This function must be able to accept two or more arguments since the LISP system passes at least two arguments to the error handler each time an error occurs in a program. Therefore, 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 is:

```
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 of the variable in the handler's definition.

If you want an error handler to display error messages in the formats described in Sections 4.2.1 to 4.2.3, 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.

4.3.1.1 **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 the handler displays.

4.3.1.2 **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 4-1 lists the functions that can be passed as the error-signaling-function argument and briefly describes each function.

Table 4-1: 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 Part II for a description of the WARN function.

4.3.1.3 **Arguments** - The *args* argument is the list of arguments 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.

4.3.2 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 4.3.1 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 5

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 correct the programming error. Table 5-1 lists the debugging functions and macros with a brief description of each. See Part II for more detailed descriptions.

Table 5-1: 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.

Table 5-1 (cont.)

Name	Function or Macro	Description
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.
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 interactively to step through a form's 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.

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.

The LISP system can have several open 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.

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

5.4.1 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:

- 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 in your form calling the BREAK function) 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 CHECK-INPUT (signaled with BREAK).
Values are too high.
```

After the message is displayed, a prompt is displayed at the left margin of your terminal:

```
Break>
```

5.4.2 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> (CONTINUE)
```

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

If you are in a nested break loop and you invoke the CONTINUE function, you are placed in the previous break-loop level. A description of the CONTINUE function is provided in Part II.

5.4.3 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 *FIRST*.

```
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>
```

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

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

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

```
Break> (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> *FIRST*
93
Break>
```

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 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 from and prints its output to the stream 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, 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 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 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.
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	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.

Table 5-2 (cont.)

Command	Description
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 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.

5.5.3.1 **Arguments** - Some debugger commands require an argument; other debugger commands accept optional arguments. 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.

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

```
Debug 1> DOWN ALL<RET>
```

The types of arguments you can specify with debugger commands are:

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

- Modifier

NOTE

Only parenthesized expressions and arguments to evaluate (that is, arguments specified with the EVALUATE command) 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.

Table 5-3 provides a summary of the modifier arguments you can specify with debugger commands.

Table 5-3: Debugger Command Modifiers

Modifier	Command Modification
ALL	Operates on both significant and insignificant stack frames.
ARGUMENTS	Operates on the arguments specified with the function in the current stack frame.
CALL	Operates on the call to the current stack frame.
DOWN	Moves the pointer down the control stack.
FUNCTION	Operates on the function object in the current stack frame.
HERE	Operates on the current stack frame.
NORMAL	Displays the function name and the argument list in the control stack frames.
QUICK	Displays the function name in the control stack frames.

Table 5-3 (cont.)

Modifier	Command Modification
TOP	Starts a backtrace at the top of the control stack.
UP	Moves the pointer up the control stack.
VERBOSE	Displays the function name, argument list, local variable bindings, and special variable bindings in the control stack frames.

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

Help Command

HELP
?

The HELP command displays help text about the debugger commands. You can specify this command with one argument, which is 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 parenthesized 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 5.4.

EVALUATE

The EVALUATE command explicitly evaluates a specified form. You must specify the command with an argument that is 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 deals with errors that invoke the debugger. Each of the following debugger commands deals with errors in a different way.

CONTINUE The CONTINUE command causes the debugger to return NIL, letting you 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. See Chapter 3 for a description of error types.

QUIT The QUIT command lets you exit to the previous command level. If the current level of the debugger is 1, the command causes the debugger to exit to the LISP prompt (Lisp>). 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. To do so, first bind the variables or define the function with the SET command, then use the REDO command.

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 that is 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. If you specify the ALL modifier with the BOTTOM command, the command moves the pointer to the first (oldest) stack frame on the control stack whether the frame is significant or insignificant.

DOWN The DOWN command moves the pointer toward the bottom of the control stack, one frame at a time. You can specify this command with optional arguments. One of the optional arguments is the ALL modifier. If you specify ALL, the command moves the pointer down the significant and insignificant stack frames on the control stack.

You can also specify an optional integer argument, which 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 that specifies the number of the stack frame.

SEARCH The SEARCH command searches the control stack for a specified function name. You must specify this command with two arguments. The first argument must be either the UP or the DOWN modifier to specify 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 (newest) significant stack frame on the control stack. If you specify the ALL modifier with the TOP command, the command moves the pointer to the last stack frame on the control stack whether the frame is significant or insignificant.

UP The UP command moves the pointer toward the top of 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. The first argument must be either the ARGUMENTS or the FUNCTION modifier. 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 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 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 in 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> (SETF THIS-STRING "abcd")
"abcd"
Lisp> (FUNCTION-Y THIS-STRING 4)
.... Error in function FUNCTION-Y
Frame #4 (FUNCTION-Y "abcd" 4)
Debug 1> (SETF STRING (SECOND (DEBUG-CALL)))
"abcd"
Debug 1> (EQ "abcd" STRING)
NIL
Debug 1> (EQ THIS-STRING STRING)
T
```

In this case, the function in the current stack frame is FUNCTION-Y. The call to (DEBUG-CALL) returns the list (FUNCTION-Y "abcd" 4). The form (SECOND (DEBUG-CALL)) evaluates "abcd", the first argument to FUNCTION-Y in the current stack frame. Note that the string returned by the call (SECOND (DEBUG-CALL)) is the same string passed to the function FUNCTION-Y. See the description of the TRACE macro for another example of the use of the DEBUG-CALL function.

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
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>
```

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>
```

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.

5.6.2 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, use 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> QUIT
Lisp>
```

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

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

If you type YES, the stepper exits and returns control to the command level that was active when the stepper was invoked.

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

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

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

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

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

A description of the QUIT command is provided in Section 5.6.4.2.

5.6.3 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:

- 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 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 the descriptions in a line of text that includes 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 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 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))
#4: (FACTORIAL 3)
Step> STEP
: #10: (BLOCK FACTORIAL (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
Step> STEP
: : #14: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step> STEP
: : : #18: (<= N 1)
Step> STEP
: : : : #22: N => 3
: : : #18 => NIL
: : : #17: (* N (FACTORIAL (- N 1)))
Step> STEP
: : : : #21: N => 3
: : : : #21: (FACTORIAL (- N 1))
Step> STEP
: : : : : #25: (- N 1)
Step> STEP
: : : : : : #29: N => 3
: : : : : #25 => 2
: : : : : #27: (BLOCK FACTORIAL (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
Step> OVER
: : : : : #27 => 2
: : : : #21 => 2
: : : #17 => 6
: : #14 => 6
: #10 => 6
#4 => 6
6
```

Note that the FACTORIAL function is a recursive function and, in the preceding example, has 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 onto which the LISP system pushes the value 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.

5.6.4 Using Stepper Commands

Stepper commands let you use the stepper to step through the evaluation of a LISP expression, form by form. You must specify some commands with arguments. 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 no ambiguity is 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> STEP<RET>
: : : #22: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step>
```

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

Table 5-4 provides a summary of the stepper commands. Descriptions of the stepper commands are provided in Section 5.5.4.2.

Table 5-4: Stepper Commands

Command	Description
BACKTRACE	Displays a backtrace of a form's evaluation.
DEBUG	Invokes the debugger.
EVALUATE	Evaluates a specified form with the stepper disabled.

Table 5-4 (cont.)

Command	Description
FINISH	Finishes the evaluation of the form that was specified in the call to the STEP macro with the stepper disabled.
HELP (or) ?	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.

5.6.4.1 **Arguments** - Stepper command arguments modify the operations the stepper commands perform. Some stepper commands require an argument, and some commands accept optional arguments. The arguments you can specify with the stepper commands are:

- Integer
- Form
- Stepper command

NOTE

Only parenthesized expressions and arguments to evaluate (that is, arguments specified with the EVALUATE command) are evaluated.

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

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

If an argument is required and you omit it, the stepper prompts you for the argument. For example:

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

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

5.6.4.2 **Stepper Commands** - The stepper provides commands that let you control how it steps through a form's evaluation.

Help Command

HELP
?

The HELP command displays help text about the stepper commands. You can specify this command with one argument, 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 parenthesized 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, which 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 in the current stack frame.

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 command with an optional integer, which 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 Through

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

FINISH The FINISH command evaluates the form you specified in the call to the STEP macro. You can specify the command with an optional argument that is 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 the RETURN command's 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 evaluates 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 1.

5.6.5 Using Stepper Variables

The stepper facility has two special variables that are useful debugging tools when in the stepper: *STEP-FORM* and *STEP-ENVIRONMENT*.

5.6.5.1 ***STEP-FORM*** - The *STEP-FORM* variable 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.

5.6.5.2 ***STEP-ENVIRONMENT*** - The *STEP-ENVIRONMENT* variable 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.

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 FUNCTION-X (X)
        (IF (< X 3) 1
            (+ (FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))))
FUNCTION-X
Lisp> (STEP (FUNCTION-X 5))
#4: (FUNCTION-X 5)
Step> STEP
: #10: (BLOCK FUNCTION-X (IF (< X 3) 1
                             (+ (FUNCTION-X (- X 1))
                                (FUNCTION-X (- X 2)))))
Step> STEP
: : #14: (IF (< X 3) 1 (+ (FUNCTION-X (- X 1))
                          (FUNCTION-X (- X 2))))
Step> STEP
: : : #18: (< X 3)
Step> STEP
: : : : #22: X => 5
: : : #18 => NIL
: : : #17: (+ (FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))
Step> STEP
: : : : #21: (FUNCTION-X (- X 1))
Step> STEP
: : : : : #25: (- X 1)
Step> STEP
: : : : : : #29: X => 5
: : : : : #25 => 4
: : : : : #27: (BLOCK FUNCTION-X (IF (< X 3) 1
                                     (+ (FUNCTION-X (- X 1))
                                        (FUNCTION-X (- X 2)))))
Step> STEP
: : : : : : #31: (IF (< X 3) 1
                     (+ (FUNCTION-X (- X 1))
                        (FUNCTION-X (- X 2))))
Step> STEP
: : : : : : : #35: (< X 3)
```

```
Step> STEP
: : : : : : : : #39: X => 4
: : : : : : : : #35 => NIL
: : : : : : : : #34: (+ (FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))
Step> STEP
: : : : : : : : #38: (FUNCTION-X (- X 1))
Step> EVAL *STEP-FORM*
(FUNCTION-X (- X 1))
Step> STEP
: : : : : : : : : #42: (- X 1)
Step> STEP
: : : : : : : : : : #46: X => 4
: : : : : : : : : : #42 => 3
: : : : : : : : : : #44: (BLOCK FUNCTION-X
                           (IF (< X 3) 1
                             (+ (FUNCTION-X (- X 1))
                                (FUNCTION-X (- X 2)))))
Step> EVAL *STEP-FORM*
(BLOCK FUNCTION-X
  (IF (< X 3) 1 (+ (FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))))
Step> STEP
: : : : : : : : : : #48: (IF (< X 3) 1
                             (+ (FUNCTION-X (- X 1))
                                (FUNCTION-X (- X 2))))
Step> STEP
: : : : : : : : : : : #52: (< X 3)
Step> STEP
: : : : : : : : : : : : #56: X => 3
: : : : : : : : : : : : #52 => NIL
: : : : : : : : : : : : #51: (+ (FUNCTION-X (- X 1))
                                (FUNCTION-X (- X 2)))
Step> STEP
: : : : : : : : : : : : #55: (FUNCTION-X (- X 1))
Step> EVAL X
3
Step> (EVAL 'X)
"Top level value of X"
Step> EVAL *STEP-FORM*
(FUNCTION-X (- 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: (BLOCK FUNCTION-X
                                   (IF (< X 3) 1
```

```
                                        (+ (FUNCTION-X (- X 1))
                                           (FUNCTION-X (- 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))
: #9: (FIRST-ELEMENT MY-LIST)
Step> STEP
: : #14: MY-LIST => (FIRST SECOND THIRD)
: : #15: (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))
: #9: (PLUS-Y 10)
Step> STEP
: : #15: (BLOCK PLUS-Y (+ X Y))
Step> EVALUATE
Evaluate: (+ X Y)
15
Step> STEP
: : : #22: (+ X Y)
```

```
Step> BACKTRACE
(PLUS-Y 10)
: (BLOCK PLUS-Y (+ X Y))
: : (+ X Y)
Step> SHOW
(+ X Y)
Step> 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> STEP
: : #15: (BLOCK ADDITION (+ X Y))
Step> STEP
: : : #22: (+ X Y)
Step> BACKTRACE
(ADDITION 4)
: (BLOCK ADDITION (+ X Y))
: : (+ X Y)
Step> EVALUATE
Evaluate: (+ X Y)
9
Step> STEP
: : : : #27: X => 4
: : : : #26: Y => 5
: : : #22 => 9
: : #15 => 9
: #9 => 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.

NOTE

You cannot trace special forms.

5.7.1 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 Chapter 8.

5.7.2 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*.

5.7.3 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:

- 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 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 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
```

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 the list of forms and the tracer displays the 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 FUNCTION-X (X)
        (IF (< X 3) 1
            (+ (FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))))
FUNCTION-X

Lisp> (TRACE (FUNCTION-X
                :PRE-DEBUG-IF (< (SECOND *TRACE-CALL*) 2)
                :SUPPRESS-IF T))
(FUNCTION-X)
Lisp> (FUNCTION-X 5)
Control Stack Debugger
Frame #26: (DEBUG)
Debug 1> DOWN
Frame #21: (BLOCK FUNCTION-X
             (IF (< X 3) 1
                 (+ (FUNCTION-X (- X 1))
                    (FUNCTION-X (- X 2)))))
```

```
Debug 1> DOWN
Frame #19: (FUNCTION-X 3)
Debug 1> (CADR (DEBUG-CALL))
3
Debug 1> CONTINUE
Control Stack Debugger
Frame #19: (DEBUG)
Debug 1> CONTINUE
5
```

- In this example, FUNCTION-X is first defined.
- Then the TRACE macro is called for FUNCTION-X. TRACE is specified to invoke the debugger if the first argument to FUNCTION-X (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 FUNCTION-X. As the :SUPPRESS-IF option has a value of T, calls to FUNCTION-X 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 FUNCTION-X.

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 (FUNCTION-X
                :POST-DEBUG-IF (> (FIRST *TRACE-VALUES*) 2)))
(FUNCTION-X)
Lisp> (FUNCTION-X 5)
#4: (FUNCTION-X 5)
. #11: (FUNCTION-X 4)
. . #18: (FUNCTION-X 3)
. . . #25: (FUNCTION-X 2)
. . . #25=> 1
. . . #25: (FUNCTION-X 1)
. . . #25=> 1
. . #18=> 2
. . #18: (FUNCTION-X 2)
. . #18=> 1
```

```
Control Stack Debugger
Frame #12: (DEBUG)
Debug 1> BACKTRACE
-- Backtrace start --
Frame #12: (DEBUG)
Frame #7: (BLOCK FUNCTION-X
             (IF (< X 3) 1
                 (+ (FUNCTION-X (- X 1))
                    (FUNCTION-X (- X 2)))))
Frame #5: (FUNCTION-X 5)
Frame #1: (EVAL (FUNCTION-X 5))
-- Backtrace ends --
Frame #12: (DEBUG)
Debug 1> CONTINUE
. #11=> 3
. #11: (FUNCTION-X 3)
. . #18: (FUNCTION-X 2)
. . #18=> 1
. . #18: (FUNCTION-X 1)
. . #18=> 1
. #11=> 2
Control Stack Debugger
Frame #5: (DEBUG)
Debug 1> CONTINUE
#4=> 5
```

TRACE is called for FUNCTION-X (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*.

CHAPTER 6

PRETTY PRINTING AND USING EXTENSIONS TO FORMAT

Pretty printing clarifies the meanings of LISP objects by modifying their printed representations. It inserts indentation and line breaks at appropriate places, making pretty-printed output easier to read than output produced with standard print functions. Pretty printing is an alternative to standard printing for all LISP objects, but is particularly useful for printing LISP code, complex data lists, and arrays.*

When pretty printing is enabled, any output function that prints output can potentially perform pretty printing. The following example contrasts the standard and pretty-printed treatments of a COND structure:

```
Lisp> (SETF T-QUESTION '(COND ((EQUAL TERMINAL
'VT240) START) (T (PRIN1 '(WHAT TERMINAL TYPE ARE YOU
USING?)))))
(COND ((EQUAL TERMINAL (QUOTE VT240)) START) (T (PRIN1
(QUOTE (WHAT TERMINAL TYPE ARE YOU USING?)))))
Lisp> (PPRINT T-QUESTION)
(COND ((EQUAL TERMINAL 'VT240) START)
      (T (PRIN1 '(WHAT TERMINAL TYPE ARE YOU USING?))))
```

The first version (produced by the standard read-eval-print loop) breaks the line at an awkward place and provides no indentation. Only one line is being printed. The line is either wrapped or truncated, depending on the operating system (VMS or ULTRIX-32) and the setting of the terminal. The pretty-printed (PPRINT) version is more readable because it starts a new line at the beginning of a nested list, indenting the list to line up with the structure nested to the equivalent level in the first line.

---

* VAX LISP pretty printing and the extensions to FORMAT are based on a program described in the paper *PP: A Lisp Pretty Printing System*, A.I. Memo No. 816, December, 1984. The paper and the program were written by Richard C. Waters, Ph.D., of the MIT Artificial Intelligence Laboratory.

PRETTY PRINTING AND USING EXTENSIONS TO FORMAT

This chapter describes four ways to print LISP objects:

- Section 6.1 tells how to pretty-print objects.
- Section 6.2 tells how to control the format of pretty-printed objects using print control variables.
- Section 6.3 tells how to use the VAX LISP FORMAT directives that support pretty-printing.
- Sections 6.4 through 6.9 tell how you can extend the VAX LISP print functions to handle specific structures and types of structures by defining new print functions.

6.1 PRETTY PRINTING WITH DEFAULTS

Three print functions let you pretty-print without explicitly using print control variables:

- PPRINT formats an object and prints it to a stream.
- PPRINT-DEFINITION formats the function object of a symbol and prints it to a stream.
- PPRINT-PLIST formats the property list of a symbol and prints it to a stream.

Use PPRINT when you want to let the system decide how best to format an object. PPRINT prints whatever object is given as its argument. The COND structure at the beginning of this chapter is an example of the output format specified for lists starting with a particular symbol.

You can use PPRINT-DEFINITION to print the definition of a LISP function. Supply the function name as the argument, as follows:

```
Lisp> (DEFUN BELONGS (THIS PILE) (COND ((NULL PILE) NIL) ((EQUAL
THIS (CAR PILE)) PILE) (T (BELONGS THIS (CDR PILE)))))
BELONGS
Lisp> (PPRINT-DEFINITION 'BELONGS)
(DEFUN BELONGS (THIS PILE)
  (COND ((NULL PILE) NIL)
        ((EQUAL THIS (CAR PILE)) PILE)
        (T (BELONGS THIS (CDR PILE)))))
```

If the object to be printed is the property list of a symbol, use PPRINT-PLIST, as shown in the following example:

```
Lisp> (SETF (GET 'PLACES 'CITIES) '(AUGUSTA SACRAMENTO))
(AUGUSTA SACRAMENTO)
```

```
Lisp> (SETF (GET 'PLACES 'STATES) '(MAINE CALIFORNIA))
(MAINE CALIFORNIA)
Lisp> (PPRINT-PLIST 'PLACES)
(STATES (MAINE CALIFORNIA)
 CITIES (AUGUSTA SACRAMENTO))
```

PPRINT-PLIST prints only indicator-value pairs for which the indicator is accessible in the current package. PPRINT-PLIST emphasizes the relationships between the indicator-value pairs.

6.2 HOW TO PRETTY-PRINT USING CONTROL VARIABLES

VAX LISP supports the global print control variables included in COMMON LISP. In addition, VAX LISP provides three variables that affect only pretty-printed output:

- *PRINT-RIGHT-MARGIN*
- *PRINT-MISER-WIDTH*
- *PRINT-LINES*

By changing the values of these variables, you can adjust pretty-printed output to suit a variety of situations.

You can also specify values for these three variables in calls to the WRITE and WRITE-TO-STRING functions. These functions have been extended to accept the following keyword arguments:

```
:RIGHT-MARGIN
:MISER-WIDTH
:LINES
```

If you specify any of these arguments, the corresponding special variable is bound to the value you supply with the argument before any output is produced.

6.2.1 Explicitly Enabling Pretty Printing

When the COMMON LISP variable *PRINT-PRETTY* is non-NIL, it enables pretty printing. If you set *PRINT-PRETTY* to T, you can pretty print by calling any print function. The LISP read-eval-print loop will also pretty-print when *PRINT-PRETTY* is non-NIL.

The following example shows the effect of a PRIN1 function call when pretty printing is enabled:

```
Lisp> (SETF *PRINT-PRETTY* T)
T
Lisp> (PRIN1 '((TIGER TIGER BURNING BRIGHT) (IN THE FORESTS OF
THE NIGHT) (WHAT IMMORTAL HAND OR EYE) (COULD FRAME THY FEARFUL
SYMMETRY)))
((TIGER TIGER BURNING BRIGHT)
 (IN THE FORESTS OF THE NIGHT)
 (WHAT IMMORTAL HAND OR EYE)
 (COULD FRAME THY FEARFUL SYMMETRY))
```

You can also enable pretty printing by specifying a non-NIL value for the :PRETTY keyword in functions such as WRITE and WRITE-TO-STRING.

6.2.2 Limiting Output by Lines

Pretty printing lets you abbreviate output by controlling the number of lines printed. With the variable *PRINT-LINES* set to any integer value, the print function you use stops after printing the specified number of lines. The output stream replaces omitted output with the characters " ...". Abbreviation by number of lines occurs only when pretty printing is enabled. See Section 6.7 for more details on abbreviating output.

The following example shows pretty-printed output with *PRINT-LINES* set to 2.

```
Lisp> (SETF *PRINT-LINES* 2)
2
Lisp> (SETF *PRINT-PRETTY* T)
T
Lisp> (PRINT '((IN WHAT DISTANT DEEPS OR SKIES) (BURNT THE FIRE
OF THINE EYES) (ON WHAT WINGS DARE HE ASPIRE) (WHAT THE HAND
DARE SEIZE THE FIRE)))
((IN WHAT DISTANT DEEPS OR SKIES)
 (BURNT THE FIRE OF THINE E ...
```

6.2.3 Controlling Margins

The *PRINT-RIGHT-MARGIN* variable lets you adjust the width of pretty-printed output. The value should be an integer; it specifies the exclusive upper limit on column numbers. With the left margin at 0, *PRINT-RIGHT-MARGIN* specifies the number of columns in which you can print. The default value, NIL, causes the print functions to query the output stream for the right margin value. The default varies, but is always appropriate to the output device.

Output may exceed the right margin if the printer encounters a long symbol name or string. The left margin is normally 0, but you can

change it by using logical blocks with the FORMAT function to indent (see Section 6.3).

6.2.4 Conserving Space with Miser Mode

Miser mode can help you avoid running out of horizontal space when you print complicated structures. Pretty printing adds line breaks and indentation to output to indicate levels of nesting, so that deeply nested structures often use up much of the line width. Miser mode conserves line width by minimizing indentation and inserting new lines where possible. You can use this feature by setting the variable *PRINT-MISER-WIDTH* to an integer value two or three times the length of the longest symbol in the output (usually a value between 20 and 40 is appropriate).

The system subtracts the value of *PRINT-MISER-WIDTH* from the right margin of the output stream to determine the column at which miser mode takes effect. In other words, miser mode becomes effective when the total line width available for printing after indentation is less than the value of *PRINT-MISER-WIDTH*. You can set *PRINT-MISER-WIDTH* to NIL to disable miser mode. See Section 6.8 for more details.

The default value of *PRINT-MISER-WIDTH* is 40.

6.3 EXTENSIONS TO THE FORMAT FUNCTION

VAX LISP provides eight FORMAT directives in addition to those specified in COMMON LISP. The added directives allow you to specify:

- Logical blocks, which are groupings of related output tokens
- Multiline mode new lines, which result in new lines if output cannot fit on one line
- Indentation, which aids in indenting portions of a form

Table 6-1 lists and briefly describes the FORMAT directives that VAX LISP provides. This section provides a guide to their use. The section presupposes a thorough knowledge of the LISP FORMAT function. See *COMMON LISP: The Language* for a full description of FORMAT.

Use the FORMAT function as follows:

FORMAT *destination control-string* &REST *arguments*

This function formats the arguments according to the format you specify with directives in the control string. *destination* specifies

the output stream. The *arguments* identify the objects to be operated on by the *control string*. The sections that follow describe the application of these directives and the effects of the colon and at-sign modifiers on them.

Table 6-1: Format Directives Provided by VAX LISP

Directive	Effect
~W	Prints the corresponding argument under direction of the current print variable values.
~!	Begins a logical block. Depending on modifiers, this directive causes FORMAT to print one or more of the arguments following the control string.
~.	Ends a logical block.
~_	Specifies a multiline mode new line. This directive is effective only in a logical block.
~*n*I	Sets indentation to *n* columns after the logical block or after the prefix. This directive is effective only in a logical block.
~*n*/FILL/	Prints the elements of a list with as many elements as possible on each line. If *n* is 1, FORMAT encloses the printed list in parentheses. This directive is effective only in a logical block.
~*n*/LINEAR/	If the elements of the list to be printed cannot be printed on a single line, this directive prints each element on a separate line. If *n* is 1, FORMAT encloses the printed list in parentheses. This directive is effective only in a logical block.
~*n*,*m*/TABULAR/	Prints the list in tabular form. If *n* is 1, FORMAT encloses the list in parentheses; *m* specifies the column spacing. This directive is effective only in a logical block.

These FORMAT directives provide the sole means of performing pretty printing in VAX LISP. All functions that explicitly perform pretty printing (for example, PPRINT and PPRINT-DEFINITION) do so by using these directives. Objects printed with FORMAT are printed normally unless pretty printing is enabled. Pretty printing is enabled when both the following conditions exist:

1. A logical block is started.

2. *PRINT-PRETTY* is non-NIL, or the colon modifier is specified in the logical block directive (~:!).

Nothing prevents you from starting a logical block when *PRINT-PRETTY* is NIL. However, any conditional new lines or indentation specified within the logical block will be ignored. This feature results in normal-looking output, as opposed to pretty-printed output. By allowing this flexibility, FORMAT lets you use one control string to format data, and the data is either printed normally or pretty-printed, according to the value of *PRINT-PRETTY*.

6.3.1 Using the WRITE FORMAT Directive

Use the ~W FORMAT directive to print an element when you want to use the current values of the print control variables. The argument for ~W can be any LISP object. In contrast, ~A and ~S specify the values of print control variables.

You can use up to four prefix parameters with ~W to pad the printed object:

~mincol,colinc,minpad,padcharW

For an explanation of these parameters, see the description under "FORMAT Directives Provided with VAX LISP" in Part II of this manual.

The colon modifier (~:W) binds the following print control variables for the duration of the WRITE: *PRINT-ESCAPE* to T, *PRINT-PRETTY* to T, *PRINT-LENGTH* to NIL, *PRINT-LEVEL* to NIL, and *PRINT-LINES* to NIL. The following example contrasts the effects of using ~W and ~:W.

```
Lisp> (SETF *PRINT-PRETTY* NIL)
NIL
Lisp> (SETF *PRINT-ESCAPE* NIL)
NIL
Lisp> (SETF *PRINT-LENGTH* 2)
2
Lisp> (SETF COLORS '(("Yellow" "Purple" "Orange" "Green") ("Aqua"
"Pink" "Beige" "Buff") ("Peach" "Violet" "Chartreuse")))
  .
  .
Lisp> (FORMAT T "~W" COLORS)
((Yellow Purple ...) (Aqua Pink ...) ...)
NIL
Lisp> (FORMAT T "~:W" COLORS)
(("Yellow" "Purple" "Orange" "Green")
 ("Aqua" "Pink" "Beige" "Buff")
 ("Peach" "Violet" "Chartreuse"))
```

The first FORMAT call truncates the first two sublists to two colors and truncates the outer list to two sublists. This truncation occurs because *PRINT-LENGTH* is 2. The first FORMAT call omits quotation marks because *PRINT-ESCAPE* is NIL. The second FORMAT call produces the full list of colors and includes quotation marks, because it implicitly sets *PRINT-LENGTH* to NIL and *PRINT-ESCAPE* to T. The second FORMAT call also indents the lists because it implicitly sets *PRINT-PRETTY* to T.

6.3.2 Controlling the Arrangement of Output

Two concepts support the dynamic arrangement of output for pretty printing: logical blocks and conditional new lines. Logical block directives divide the total output into hierarchical groupings, which are referred to as logical blocks or subblocks. The goal of FORMAT is to print an entire logical block (including all its subblocks) on one line. If pretty printing is enabled, the logical block is printed on one line only if the logical block fits between the current left and right margins. Printing all the output on one line is referred to as single-line mode printing.

The output for a logical block may not fit on one line when pretty printing. In this case, the block must be subdivided into sections at points where it may be split into multiple lines. Conditional new line directives specify these points. Multiline mode printing is the name given to the condition where a logical block must occupy multiple lines.

When pretty printing is enabled, FORMAT buffers the contents of a logical block until it can decide whether to use single-line mode or multiline mode printing.

A third mode, miser mode, is described briefly in Section 6.2.4 and in detail in Section 6.8.

Use the ~! and ~. directives to specify a logical block in the form:

~!*block*~.

where *block* can include any FORMAT directives. A logical block takes one argument from the FORMAT argument list. If that argument is a list, any directives within the logical block that take arguments take them from that list, as shown in the following example:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 40)
40
Lisp> (SETF *PRINT-PRETTY* T)
T
```

```
Lisp> (FORMAT T "~!~W~." '((STARS (BETELGEUSE
DENEB SIRIUS)) (PLANETS (MERCURY VENUS EARTH
MARS JUPITER SATURN NEPTUNE PLUTO))))
(STARS (BETELGEUSE DENEB SIRIUS))
(PLANETS (MERCURY VENUS EARTH MARS
          JUPITER SATURN URANUS NEPTUNE
          PLUTO))
```

The logical block takes the entire list as its argument. The ~W directive within the logical block causes FORMAT to pretty-print the list because *PRINT-PRETTY* is set to T.

If the argument is not a list, the logical block is effectively replaced by the ~W directive.

You can alter the directive to start a logical block (~!) by adding two modifiers. When the directive includes a colon (~:!), the directive sets *PRINT-PRETTY* and *PRINT-ESCAPE* to T and *PRINT-LENGTH*, *PRINT-LEVEL*, and *PRINT-LINES* to NIL for all the printing controlled by the logical block.

When the ~! directive includes an at-sign (~@!), the directives within the logical block take successive arguments from the FORMAT argument list. The logical block uses up all the arguments, not just a single list argument. Therefore, no directives that take arguments from the argument list can appear after a logical block modified by an at-sign in the logical block directive (see the last example in this section). You can use the ~^ directive inside a logical block to check whether the logical block arguments have been reduced to a non-NIL atom. See Section 6.9 for information on handling improperly formed argument lists.

The output associated with any FORMAT directive is subject to pretty printing when the directive occurs within a logical block and *PRINT-PRETTY* is non-NIL.

A logical block defines an indentation level and can define a prefix and a suffix. By default, when pretty printing is enabled, the indentation level is the position of the first character in the logical block. Each line following the first line in the logical block is printed preserving indentation and per-line prefixes, so that the first character in the line normally lines up with the first character in the block following the prefix. However, no default prefix or suffix is associated with a logical block.

You can create nested logical blocks within a logical block, using the ~!*block*~. directive. For example:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 70)
70
Lisp> (SETF *PRINT-PRETTY* T)
T
Lisp> (FORMAT T "~!Stars: ~!~S ~S~. Planets: ~!~S ~S~.~."
          '((BETELGEUSE DENEB) (MARS JUPITER)))
Stars: BETELGEUSE DENEB Planets: MARS JUPITER
```

In this example, two logical blocks are created within the principal logical block. Each logical block uses the next argument for printing:

- The enclosing logical block uses the elements of the principal list ((BETELGEUSE DENEB) (MARS JUPITER)) as its arguments.
- The first inner logical block uses the elements of the list (BETELGEUSE DENEB) as its arguments.
- The second inner logical block uses the elements of the list (MARS JUPITER) as its arguments.

```
Lisp> (FORMAT T "~:!Stars: ~!~S ~S~. Planets: ~!~S ~S~.~."
          '((BETELGEUSE DENEB) (MARS JUPITER)))
Stars: BETELGEUSE DENEB Planets: MARS JUPITER
```

In this example, the colon in the ~:! directive enables pretty printing implicitly, producing the same output as the previous example.

```
Lisp> (SETF *PRINT-PRETTY* T)
T
Lisp> (FORMAT T "~@!~S ~%~S ~%~S ~%~S~."
          '(BETELGEUSE DENEB SIRIUS) 'POLARIS 'VEGA 'ALGOL
           'ALDEBERAN)
(BETELGEUSE DENEB SIRIUS)
POLARIS
VEGA
ALGOL
```

In this example, the at-sign causes the logical block to use all following arguments. Unneeded arguments are used up by the logical block but not printed. The first ~S applies to the first argument (the list (BETELGEUSE DENEB SIRIUS)). The remaining three ~S directives apply to POLARIS, VEGA, and ALGOL. ALDEBERAN goes unprinted, because there is no corresponding directive.

```
Lisp> (FORMAT T "~@!Stars: ~!~S ~S~. Planets: ~!~S ~S~.~."
          '(BETELGEUSE DENEB) '(MARS JUPITER))
Stars: BETELGEUSE DENEB Planets: MARS JUPITER
```

In this example the at-sign in the outermost logical block directive (~@!) directs the logical block to use all the arguments. The first

inner logical block uses the elements of the list (BETELGEUSE DENEB); the second inner logical block uses the elements of the list (MARS JUPITER).

6.3.3 Controlling Where New Lines Begin

Five FORMAT directives let you specify places where new lines can start according to the demands of the situation. Each directive delimits a section in a logical block.

- The ~% directive produces an unconditional new line. When used within a logical block, the directive preserves indentation and per-line prefixes.
- The ~& directive produces a fresh line. When used within a logical block, the directive preserves indentation and per-line prefixes.
- The ~_ directive produces a multiline mode new line when used within a logical block.
- The ~:_ directive produces an if-needed new line when used within a logical block.
- The ~@_ directive produces a miser-mode new line when used within a logical block.

You can specify unconditional new lines (~%) and fresh lines (~&) if you know in advance how the text should be laid out. If a new line is produced by one of these directives when the FORMAT function is printing a logical block, FORMAT prints the logical block in the multiline mode, preserving indentation and per-line prefixes.

The ~& directive specifies a fresh line, whether or not pretty printing is enabled. If the ~& directive occurs inside a logical block when pretty printing is enabled and any output is on the line other than prefixes and indentation, the FORMAT call starts a fresh line, preserving indentation and per-line prefixes. The following examples show the use of the ~% and ~& directives:

```
Lisp> (FORMAT T "Stars~:!;~@;~%~S ~%~S ~%~S~."
          '(BETELGEUSE DENEB SIRIUS))
Stars;
     ; BETELGEUSE
     ; DENEB
     ; SIRIUS
NIL
Lisp> (FORMAT T "Stars~:!;~@;~&~S ~&~S ~&~S~."
          '(BETELGEUSE DENEB SIRIUS))
Stars; BETELGEUSE
     ; DENEB
     ; SIRIUS
```

The first FORMAT call starts a new line after the prefix ";", because the ~% directive starts a new line wherever the directive occurs. Replacing the ~% directive with the ~& directive changes the output, because the fresh line is not needed after the prefix.

The remaining three new line directives offer flexibility because they are conditional. However, they have no effect on output (except length abbreviation -- see Section 6.7.1) when pretty printing is not enabled.

The ~_ directive (multiline mode new line) starts a new line if the output for the enclosing logical block is too long to fit on one line or if any other directive in the logical block causes a new line. When the output is too long, FORMAT uses multiline mode, and every ~_ directive in a logical block starts a new line. The ~:_ directive (if-needed new line) produces a new line if it is needed: if the following section of output is too long to fit on the current line. The ~@_ directive (miser-mode new line) produces a new line if pretty printing is enabled with miser mode in effect (see Section 6.8 for details). The FORMAT function ignores the three conditional new line directives when they occur outside a logical block.

The following example shows how you can specify a multiline mode new line and an if-needed new line:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 16)
16
Lisp> (FORMAT T "~:!~S ~_~S ~:_~S ~_~S~."
          '(BETELGEUSE ALDEBERAN MERCURY JUPITER))
BETELGEUSE
ALDEBERAN
MERCURY
JUPITER
```

This FORMAT function produces output in the multiline mode, because the output will not fit on one line. The multiline mode new line directives (~_) produce a new line for each element. The ~:_ directive directs FORMAT to start a new line before MERCURY if needed (and a new line is needed).

You can produce printed output that fills up the space available in each line by using the at-sign (@) modifier with the directive that ends the logical block (~!*block*~@.). This modifier causes FORMAT to start a new line if needed following every blank space or tab and is equivalent to inserting a ~:_ directive after each element to be printed, as shown in the following example:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 25)
25
Lisp> (FORMAT T "~@:!ANTARES ALPHECCA ALBIREO CANOPUS CASTOR
                    POLLUX MIRZAM ALGOL BELLATRIX CAPELLA MIRA
                    MIRFAK DUBHE POLARIS ~@." )
ANTARES ALPHECCA ALBIREO
CANOPUS CASTOR POLLUX
MIRZAM ALGOL BELLATRIX
CAPELLA MIRA MIRFAK DUBHE
POLARIS
```

6.3.4 Controlling Indentation

With pretty printing enabled, a call to FORMAT indents the output for a logical block so that the first character in each succeeding line falls under the first character following the prefix in the first line. When pretty printing is not enabled, the FORMAT call does not produce indentation, and the indentation directive has no effect.

Use the ~*n*I directive or the ~*n*:I directive if you want to change the standard pretty-printed indentation. The ~*n*I directive causes FORMAT to indent subsequent lines *n* spaces from the position of the first character in the logical block. The ~*n*:I directive, on the other hand, causes FORMAT to indent subsequent lines *n* spaces from the output column corresponding to the position of the directive. If you omit the parameter *n*, the default is 0. Although this parameter can be less than 0 when used with the colon, the indentation cannot move to the left of the first character in the logical block. An indentation directive affects only indentation produced on subsequent new lines.

The following example shows several variations of the indentation directive:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 15)
15
Lisp> (FORMAT T "~:!~S ~2I~:_~S ~:I~S ~_~S ~1I~_~S~."
          '(BETELGEUSE DENEB SIRIUS VEGA ALDEBERAN))
BETELGEUSE
  DENEB SIRIUS
        VEGA
 ALDEBERAN
```

DENEB lines up under the T in BETELGEUSE, because the ~:_ directive produces a new line and ~2I causes an indentation of 2 spaces past the beginning of the block. The ~:I directive for the third argument sets the indentation to the column of the first S in SIRIUS, so that the V of VEGA lines up with the S. ALDEBERAN lines up with the first E in BETELGEUSE, because the ~1I directive resets the indentation to one column past the first character in the logical block.

The ~I directives only set the indentation. They do not start new lines and they do not take effect until new lines begin. Therefore, in the directives for DENEB and ALDEBERAN, the indentation directives precede the new line directives.

6.3.5 Producing Prefixes and Suffixes

You can specify FORMAT control strings that add prefixes and suffixes to the printed output produced for a logical block. Several options are available.

If you divide the format control string into three sections by inserting the ~; directive twice, the string will specify a prefix and a suffix, as follows: ~!*prefix*~;*body*~;*suffix*~.. The first ~; directive marks the end of the prefix; the second marks the beginning of the suffix. If you omit the second ~; directive, no suffix is specified. Although the body can be any FORMAT control string, the prefix and suffix cannot include FORMAT directives.

When a FORMAT call prints output for a logical block that includes a prefix and pretty printing is enabled, the second line of the output is indented so that the second line lines up with the first character in the block following the prefix. When the logical block includes a suffix, the FORMAT call always prints the suffix at the end, even if abbreviation directives eliminate some of the body of the block.

In the following examples, "Stars <" forms the prefix, and ">" forms the suffix.

```
Lisp> (FORMAT T "~!Stars <~;~S ~%~S ~_~S~;>~."
          '(SIRIUS VEGA DENEB))
Stars <SIRIUS
       VEGA
       DENEB>
NIL
Lisp> (SETF *PRINT-LENGTH* 2)
2
Lisp> (FORMAT T "~!Stars <~;~S ~%~S ~_~S~;>~."
          '(SIRIUS VEGA DENEB))
Stars <SIRIUS
       VEGA...>
```

In the second example, FORMAT truncates the list to two elements, because *PRINT-LENGTH* is set to 2 (see Section 6.7), but it still adds the suffix after the last list element. VEGA lines up under SIRIUS in the first column for the body of the logical block.

You can specify the prefix parameter 1 in the logical block directive (~1!*block*~.), causing the FORMAT call to use parentheses for the prefix and suffix, as shown:

```
Lisp> (FORMAT T "~1:!~S ~%~S~."
'(CASTOR POLLUX))
(CASTOR
 POLLUX)
```

You can create per-line prefixes in a logical block by specifying the at-sign modifier in the ~; directive used to indicate the end of the prefix (~@;). This modifier causes FORMAT to repeat the prefix at the beginning of each line, as shown in the following example:

```
Lisp> (FORMAT T "~:!<<~@;~S ~%~S ~_~S ~_~S~;>>~."
'(ALGOL ANTARES ALBIREO ALPHECCA))
<<ALGOL
<<ANTARES
<<ALBIREO
<<ALPHECCA>>
```

The prefixes and the list elements line up.

If you nest logical blocks, you can specify a prefix with each block, as shown:

```
Lisp> (FORMAT T "~:!Bright stars~; ~@!<<~@;~S ~S ~%~S ~
~S~;>>~.~;
still twinkle.~."
'(SIRIUS VEGA DENEB ALGOL))
Bright stars <<SIRIUS VEGA
             <<DENEB ALGOL>> still twinkle.
```

The prefix and suffix for the outer logical block are "Bright stars" and "still twinkle". The prefix for the inner logical block, "<<", is printed on each line after the indentation required by the prefix for the first logical block. The suffix for the inner logical block, ">>", is printed once at the end of the block.

6.3.6 Using Tabs

You can use the tab directive to arrange output in columns. When pretty printing is enabled, the ~*n*,*m*T tab directive counts spaces, beginning with the indentation of the immediately enclosing logical block. The integer *n* specifies a number of columns. The integer *m*

specifies an increment: the number of columns to be added at one time until the column width is at least *n* columns. The at-sign modifier makes the tab directive relative, so that ~*n*,*m*@T counts spaces beginning with the current output column. When pretty printing is not enabled, on the other hand, the ~*n*,*m*T directive counts spaces from the beginning of the line, as specified in COMMON LISP. The defaults for *n* and *m* are 1 (see *COMMON LISP: The Language* for details).

In the iterative example that follows, the tab directive precedes the if-needed new line directive:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 29)
29
Lisp> (FORMAT T "Stars: ~:@!~{~S~^ ~11T~S ~^ ~:_~}~."
          '(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
            MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
            ANTARES))
Stars: POLARIS    DUBHE
       MIRA       MIRFAK
       BELLATRIX  CAPELLA
       ALGOL      MIRZAM
       POLLUX     CANOPUS
       ALBIREO    CASTOR
       ALPHECCA   ANTARES
```

Since the tabs are counted from the indentation of the logical block, the tab directives do not have to account for the fact that the whole block is shifted seven columns to the right.

6.3.7 Directives for Handling Lists

VAX LISP provides three FORMAT directives that simplify the printing of lists. Each implicitly uses the ~W directive repeatedly to print elements.

- If pretty printing is enabled, the ~*n*/FILL/ directive causes FORMAT to fill the available line width by inserting a space and an if-needed new line after each list element except the last. FORMAT encloses the list in parentheses if *n* is 1. If pretty printing is not enabled, ~*n*/FILL/ causes FORMAT to print the output in single-line mode.

- If pretty printing is enabled, the ~*n*/LINEAR/ directive causes FORMAT to print the list on a single line if the list fits. Otherwise, FORMAT prints each element on a separate line. FORMAT encloses the list in parentheses if *n* is 1. If pretty printing is not enabled, ~*n*/LINEAR/ causes FORMAT to print the output in single-line mode.

- If pretty printing is enabled, the ~*n*,*m*/TABULAR/ directive causes FORMAT to print the list as a table, using columns of *m* spaces for list elements. The default value for *m* is 16. FORMAT encloses the list in parentheses if *n* is 1. If pretty printing is not enabled, ~*n*,*m*/TABULAR causes FORMAT to print the output in single-line mode.

The following examples show the kinds of formats you can produce with the list-handling directives:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 36)
36
Lisp> (FORMAT T "Stars: ~@:!~/FILL/~."
          '(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
            MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
            ANTARES))
Stars: POLARIS DUBHE MIRA MIRFAK
       BELLATRIX CAPELLA ALGOL
       MIRZAM POLLUX CANOPUS
       ALBIREO CASTOR ALPHECCA
       ANTARES
NIL
Lisp> (SETF *PRINT-RIGHT-MARGIN* NIL)
NIL
Lisp> (FORMAT T "Stars: ~@:!~/LINEAR/~."
'(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
ANTARES))
Stars: POLARIS
       DUBHE
       MIRA
       MIRFAK
       BELLATRIX
       CAPELLA
       ALGOL
       MIRZAM
       POLLUX
       CANOPUS
       ALBIREO
       CASTOR
       ALPHECCA
       ANTARES
NIL
Lisp> (FORMAT T "Stars: ~@:!~0,20/TABULAR/~."
'(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
ANTARES))
Stars: POLARIS              DUBHE              MIRA
       MIRFAK               BELLATRIX          CAPELLA
       ALGOL                MIRZAM             POLLUX
       CANOPUS              ALBIREO            CASTOR
       ALPHECCA             ANTARES
```

6.4 DEFINING YOUR OWN FORMAT DIRECTIVES

VAX LISP lets you define your own FORMAT directives to supplement the directives supplied with the system. Any FORMAT directive that you define you can use in the control string argument to a FORMAT call.

```
DEFINE-FORMAT-DIRECTIVE name
                        (arg stream colon at-sign
                        &OPTIONAL (parameter1 default)
                                  (parameter2 default)
                                  ...)
                        &BODY forms
```

This macro defines a directive named *name*. After you define a FORMAT directive, you can use it (whether or not pretty printing is enabled) by including ~/*name*/ in a FORMAT control string.

NOTE

If you do not specify a package with *name* when you define the directive, *name* is placed in the current package. If you do not specify a package when you refer to the directive, the FORMAT directive looks in the USER package for the directive definition.

For the body of the macro call, the symbols you supply for *arg*, *stream*, *colon*, and *at-sign* are bound as follows:

- *arg* is bound to the argument list for the FORMAT directive you define.
- *stream* is bound to the stream on which the printing is to be done.
- The *colon* and *at-sign* arguments are bound to NIL unless the colon and at-sign modifiers are used with the directive.

There must be one optional argument for each prefix *parameter* that is allowed in the directive. A parameter argument will receive the corresponding prefix parameter if it was specified in the directive. Otherwise, the default value will be used, as with all optional arguments.

The body is evaluated to print the argument *arg* on the output *stream*. A user-defined FORMAT directive can be useful because it provides a level of indirection. In addition, you can call the directive repeatedly, which may save you some time coding and debugging. The following example shows a format directive used to produce error messages:

```
Lisp> (DEFINE-FORMAT-DIRECTIVE EVALUATION-ERROR
                               (SYMBOL STREAM COLON-P ATSIGN-P
                               &OPTIONAL (SEVERITY 0))
        (DECLARE (IGNORE ATSIGN-P))
        (FRESH-LINE STREAM)
        (PRINC (CASE SEVERITY
                 (0     "Warning: ")
                 (1     "Error: ")
                 (2     "Severe Error: "))
               STREAM)
        (FORMAT STREAM "~:!The symbol ~S ~:_does not have an ~
                        integer value.~%Its value is: ~:_~S~."
                        SYMBOL (SYMBOL-VALUE SYMBOL))
        (WHEN COLON-P
          (WRITE-CHAR #\BELL STREAM)))
EVALUATION-ERROR
Lisp> (SETF PROCESS NIL)
NIL
Lisp> (FORMAT T "~1:/EVALUATION-ERROR/" 'PROCESS)
Error: The symbol PROCESS does not have an integer value.
       Its value is: NIL
<BEEP>
```

This example shows the definition of a FORMAT directive, an application of the directive, and the printed output. It assumes that the current package is USER. The prefix parameter 1 in "~:/EVALUATION-ERROR/" indicates the severity of the error being signaled. The colon in the FORMAT call produces a beep on the terminal.

6.5 DEFINING PRINT FUNCTIONS FOR LISTS

You can use DEFINE-LIST-PRINT-FUNCTION to define functions to print specific kinds of lists in formats of your choice. Functions that you define are effective only if pretty printing is enabled. The printer checks the first element of each list that it prints. If the first element of a list matches the name of a list-print function, the list is printed according to the format you have specified. Create a list-print function according to the following format:

```
DEFINE-LIST-PRINT-FUNCTION  symbol (list stream)
                            &BODY forms
```

This macro defines or redefines a print function for lists for which the first element is *symbol*. *list* is bound to the list to be printed and *stream* is bound to the stream on which the printing is to be done. The *forms* are evaluated to output *list*.

For example, if you define a list-print function for the symbol MY-SETQ, any list beginning with MY-SETQ will be printed in your

format when pretty-printing is enabled:

```
Lisp> (DEFINE-LIST-PRINT-FUNCTION MY-SETQ (LIST STREAM)
        (FORMAT STREAM
                "~1!~W~^ ~:I~@{~W~^ ~:_~W~^~%~}~."
                LIST))
MY-SETQ
Lisp> (SETF BASE '(MY-SETQ HI 3 BYE 4))
(MY-SETQ HI 3 BYE 4)
Lisp> (PRINT BASE)
(MY-SETQ HI 3 BYE 4)
(MY-SETQ HI 3 BYE 4)
Lisp> (PPRINT BASE)
(MY-SETQ HI 3
         BYE 4)
```

When pretty printing is not enabled, the value of BASE is printed without regard to the list-print function defined for MY-SETQ. PPRINT enables pretty printing, producing a representation of the value of BASE using the specified list-print function.

VAX LISP pretty printing incorporates predefined list-print functions for many standard LISP functions. However, if you define a list-print function for a LISP keyword, your function will override the one built into the system.

NOTE

When you use DEFINE-LIST-PRINT-FUNCTION, you may encounter two kinds of output that you do not expect:

- In most cases, a list whose first element is the symbol for a defined list-print function will be printed in the format specified, even if the context and meaning of the list are irregular and the format is inappropriate. For example, if your data says (LET IT BE) and LET is the symbol of a defined list-print function, the resulting output may be inappropriate.

- List-print functions are not used when you print a list under control of a user-defined FORMAT directive.

You can disable any defined list-print function by using the UNDEFINE-LIST-PRINT-FUNCTION macro. Its format is:

UNDEFINE-LIST-PRINT-FUNCTION *symbol*

This macro disables the list-print function defined for *symbol*. The following example disables the LET list-print function defined in the example at the beginning of this section:

```
Lisp> (UNDEFINE-LIST-PRINT-FUNCTION MY-SETQ)
MY-SETQ
```

6.6 DEFINING GENERALIZED PRINT FUNCTIONS

Using generalized print functions, you can specify how any object is pretty-printed, regardless of its form. Functions that you define and enable are effective only if pretty printing is enabled. First you define a function with DEFINE-GENERALIZED-PRINT-FUNCTION. Then you enable the function. You can enable it globally, using GENERALIZED-PRINT-FUNCTION-ENABLED-P. Or you can enable it locally, using WITH-GENERALIZED-PRINT-FUNCTION.

Use the following format when you define a generalized print function:

```
DEFINE-GENERALIZED-PRINT-FUNCTION name (object stream)
                                  predicate
                                  &BODY forms
```

This macro defines or redefines a print function with the name *name*. *object* is bound to the object to be printed. *stream* is bound to the stream to which output is to be sent. *predicate* governs the application of the generalized print function. The predicate is operative on any LISP object. A generalized print function will be used if it is enabled and the predicate evaluates to true on the object to be printed. (NULL OBJECT) is the predicate in the sample generalized print function shown at the end of this section. The output stream can use your generalized print function to print any object for which the predicate does not evaluate to NIL. *forms* identifies arguments to be evaluated in the call to FORMAT.

If a generalized print function and a list-print function for the same symbol are both enabled, the generalized print function will be used.

A related function lets you test whether a specific generalized print function is enabled:

```
GENERALIZED-PRINT-FUNCTION-ENABLED-P name
```

You can also use this function to globally change the status of the function, using SETF as shown:

(SETF (GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*) T)

or

(SETF (GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*) NIL)

Use the WITH-GENERALIZED-PRINT-FUNCTION macro to locally enable a generalized print function in the following format:

WITH-GENERALIZED-PRINT-FUNCTION *name* &BODY *forms*

This macro locally enables the generalized print function named *name* when it evaluates the specified *forms*.

The printer checks generalized print functions that have been enabled in reverse order from the order of their enabling. This means that in cases where two or more generalized print functions apply, the most recently enabled function is used.

Enabling a generalized print function globally is less efficient than enabling it locally, because the printer must check the predicate of globally enabled print functions against every object to be printed. If you enable the generalized print function locally, the printer checks the function's predicate against the object being printed only during execution of the code within the macro, instead of on every call to a print function. Since the read-eval-print loop is used often, the difference in efficiency can be significant.

Consider the following examples:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 25)
25
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> (PRINT NIL)
NIL
NIL
Lisp>(PPRINT NIL)
NIL
Lisp> (WITH-GENERALIZED-PRINT-FUNCTION 'PRINT-NIL-AS-LIST
        (PRINT NIL)
        (PPRINT NIL))
NIL
( )
```

```
Lisp> (SETF (GENERALIZED-PRINT-FUNCTION-ENABLED-P
                    'PRINT-NIL-AS-LIST) T)
T
LISP> (PPRINT NIL)
( )
```

The first PRINT call prints NIL, because pretty printing is not enabled. The first PPRINT call prints NIL, because the generalized print function PRINT-NIL-AS-LIST is not enabled. The second PRINT call prints NIL, because pretty printing is again not enabled. The second PPRINT call prints ( ), because the generalized print function is enabled locally and pretty printing is enabled. The third PPRINT call prints ( ), because the generalized print function is enabled globally and pretty printing is enabled.

NOTE

A generalized print function controls the printing of an object only if the following conditions exist:

1. The generalized print function is enabled globally or locally.

2. The predicate specified with DEFINE-GENERALIZED-PRINT-FUNCTION is true.

3. The object to be printed does not come under control of a user-defined FORMAT directive.

In cases where two or more generalized print functions are applicable, only one is chosen. The one chosen is the most recently enabled (globally or locally) generalized print function for which the predicate specified with DEFINE-GENERALIZED-PRINT-FUNCTION is true.

Generalized print functions are not used when you print an object under control of a user-defined FORMAT directive.

6.7 ABBREVIATING PRINTED OUTPUT

You can abbreviate printed output according to:

- The length of the object to be printed
- The depth of nested logical blocks
- The number of lines in the output

Length and depth abbreviation are supported in COMMON LISP and are effective whether or not pretty printing is enabled. In addition, abbreviation based on the number of lines of output is supported in VAX LISP; this is effective only when pretty printing is enabled.

6.7.1 Abbreviating Output Length

You can control the number of sections of printed output by setting the *PRINT-LENGTH* variable. The value you supply specifies the number of sections to be printed for any affected logical block. The directives ~_, ~%, and ~& mark the sections of a logical block (see Section 6.3.3 for details). After the output stream prints *PRINT-LENGTH* sections of a logical block, it prints an ellipsis (...) and stops processing the logical block. If the logical block is nested with other logical blocks, the output stream terminates only the processing of the immediately enclosing logical block. Output is not truncated if the value of *PRINT-LENGTH* is NIL.

The following example shows output abbreviation based on length:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 47)
47
Lisp> (SETF *PRINT-LENGTH* 11)
11
Lisp> (SETF *PRINT-PRETTY* T)
T
Lisp> (FORMAT T "Stars: ~@!~{~W~^ ~:_~}~."
          '(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
            MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
            ANTARES))
Stars: POLARIS DUBHE MIRA MIRFAK BELLATRIX
       CAPELLA ALGOL MIRZAM POLLUX CANOPUS
       ALBIREO ...
```

Each star name in the list constitutes a separate logical block section. FORMAT prints " ..." after the eleventh star name to indicate that the list has been abbreviated at that point.

6.7.2 Abbreviating Output Depth

Use the variable *PRINT-LEVEL* to control the depth of printed output. *PRINT-LEVEL* specifies the lowest level of dynamically nested logical blocks to be printed. When your program calls FORMAT recursively, the output stream keeps track of the actual nesting level and abbreviates output when the level reaches *PRINT-LEVEL*. The printed character # indicates where the stream has truncated the output. You can prevent depth abbreviation by setting *PRINT-LEVEL* to NIL.

Dynamic nesting of logical blocks occurs frequently when you print complicated structures. This nesting may not be obvious as you read the program. For example, if you have defined list-print functions for the primitives IF and PROGN, printing a program that uses a combination of these primitives would involve dynamic nesting of logical blocks, since each list print function uses the ~W directive implicitly. The following example shows how the output stream abbreviates the printing of a structure in accord with the value of *PRINT-LEVEL*:

```
Lisp> (SETF *PRINT-LEVEL* 3)
3
Lisp> (PPRINT '(LEVEL1 (LEVEL2 (LEVEL3 (LEVEL4 (LEVEL5))))))
(LEVEL1 (LEVEL2 (LEVEL3 #)))
Lisp> (SETF *PRINT-LEVEL* 2)
2
Lisp> (PPRINT '(LEVEL1 (LEVEL2 (LEVEL3 (LEVEL4 (LEVEL5))))))
(LEVEL1 (LEVEL2 #))
Lisp> (PPRINT '(LEVEL1 4 5 6 (LEVEL2 (LEVEL3 (LEVEL4
                (LEVEL5))))))
(LEVEL1 4 5 6 (LEVEL2 #))
```

6.7.3 Abbreviating Output by Lines

You can control the number of lines printed in the output by setting the *PRINT-LINES* variable. The value you supply specifies the number of lines to be printed for the outermost logical block. The output stream prints " ..." at the end of the last line to indicate where it has truncated the output. If *PRINT-LINES* is NIL, the output stream will not abbreviate the number of lines printed. This abbreviation mechanism is effective only when pretty printing is enabled.

In the following example, printing stops at the end of the fourth line:

```
Lisp> (SETF *PRINT-LINES* 4)
4
Lisp> (FORMAT T "Stars: ~:!~/LINEAR/~."
          '(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
            MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
            ANTARES))
Stars: POLARIS
       DUBHE
       MIRA
       MIRFAK ...
```

6.8 USING MISER MODE

If you print large structures with deeply nested logical blocks, you may find the miser mode useful. Indentation produced in the output by the nesting of logical blocks, prefixes, and the ~*n*I directive reduces the line length available for printing. Miser mode helps you avoid running out of space and printing beyond the right margin. Miser mode does not, however, guarantee the elimination of these problems.

Pretty printing uses single-line mode if the output fits on one line. If the FORMAT control string permits new lines and the output requires two or more lines, pretty printing normally uses multiline mode. The printer determines whether to print a logical block in miser mode according to the current column of the output at the beginning of the logical block and the values of two variables:

- *PRINT-RIGHT-MARGIN*
- *PRINT-MISER-WIDTH*

PRINT-RIGHT-MARGIN specifies the location of the right margin. *PRINT-MISER-WIDTH* specifies a number of columns before the right margin. When the current output column at the beginning of a logical block is equal to or greater than the difference between *PRINT-RIGHT-MARGIN* and *PRINT-MISER-WIDTH*, then the logical block is printed in miser mode. This condition occurs when the total available line width is less than the value of *PRINT-MISER-WIDTH*, as shown in Figure 6-1.

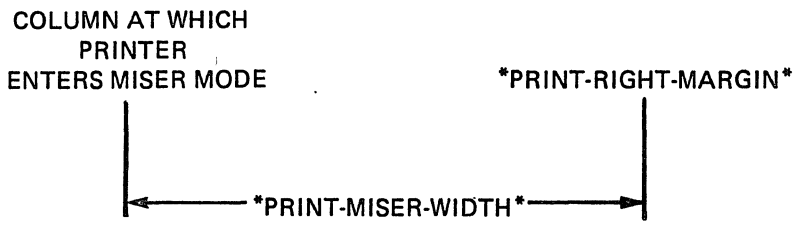


Figure 6-1: Variables Governing Miser Mode

You can disable miser mode by setting *PRINT-MISER-WIDTH* to NIL.

Miser mode saves space by:

- Ignoring indentation FORMAT directives

- Starting a new line at every conditional new line directive:

  Multiline mode new line (~_)

  If-needed new line (~:_)

  Miser mode new line (~@_)

The two examples that follow contrast pretty printing in multiline mode and miser mode:

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 60)
60
Lisp> (SETF *PRINT-MISER-WIDTH* 35)
35
Lisp> (FORMAT T "~:!Stars with Arabic names: ~S ~S ~27I~:_~S ~
                 ~:I~@_~S ~_~S ~1I~_~S~."
           '(BETELGEUSE (DENEB SIRIUS VEGA)
             ALDEBERAN ALGOL (CASTOR POLLUX) BELLATRIX)
Stars with Arabic names: BETELGEUSE (DENEB SIRIUS VEGA)
                            ALDEBERAN ALGOL
                                      (CASTOR POLLUX)
 BELLATRIX
NIL
Lisp> (FORMAT T "~!Stars with Arabic names: ~:@!~S ~:_~S ~
                 ~27I~:_~S ~:I~@_~S ~_~S ~1I~_~S~.~."
           '(BETELGEUSE (DENEB SIRIUS VEGA)
             ALDEBERAN ALGOL (CASTOR POLLUX) BELLATRIX)
Stars with Arabic names: BETELGEUSE
                         (DENEB SIRIUS VEGA)
                         ALDEBERAN
                         ALGOL
                         (CASTOR POLLUX)
                         BELLATRIX
```

In the first output sample, FORMAT uses multiline mode. Miser mode is never enabled, because the logical block begins at column 0 and miser mode takes effect only if the column begins at column 25 (60 - 35). ALDEBERAN lines up with the T in BETELGEUSE, because the ~27I directive sets the indentation for following lines at column 27 and the ~:_ directive produces a new line. The ~:I~@_~S directive sets the column for the next line at the level of the A in ALGOL. The ~1I directive controls the last argument, BELLATRIX, setting the indentation to column 1.

The second output example shows the effects of miser mode, because the text in the outer logical block, "Stars with Arabic names:", causes the inner logical block to begin at column 26. With *PRINT-MISER-WIDTH* set to 35, FORMAT enables miser mode when the logical block begins past column 25. FORMAT conserves space by starting a new line at every multiline mode new line directive (~_) and every if-needed new line directive (~:_). FORMAT also inserts a

new line at the miser mode new line directive (~@_) and ignores the indentation directives (~*n*I).

6.9 HANDLING IMPROPERLY FORMED ARGUMENT LISTS

VAX LISP provides a method for gracefully handling argument lists that are improperly formed. The function of the ~^ directive, when used in a logical block, differs slightly from the corresponding function in COMMON LISP.

In COMMON LISP the ~^ directive is used with the iteration directives ~{ and ~} to check whether the argument list has been reduced to NIL. If the list is NIL, iteration stops.

You can also use the ~^ directive to check whether the argument list for a logical block has been reduced to a non-NIL atom. If the check shows that the argument list is a non-NIL atom, the printer prints space-dot-space (.) and uses the ~W directive to print the value of the atom. FORMAT then stops processing the immediately enclosing logical block, after printing the suffix (if one is there). No error condition results. The following example shows the use of FORMAT to print a dotted pair:

```
Lisp> (FORMAT T "~1:!~@{~S~^ ~}~."
          '(CASTOR POLLUX DENEB . ALDEBERAN))
(CASTOR POLLUX DENEB . ALDEBERAN)
```

This feature serves as a useful debugging tool, because it lets the FORMAT function work even when the argument list is improperly formed.

NOTE

When the ~^ directive is included in a logical block, the FORMAT function checks whether the argument list is a non-NIL atom, even when pretty printing is not enabled.

CHAPTER 7

VAX LISP/VMS IMPLEMENTATION NOTES

VAX LISP is an implementation of LISP that is based on COMMON LISP as described in *COMMON LISP: The Language*. This chapter describes how implementaton-dependent aspects of COMMON LISP are implemented on the VMS operating system. This chapter does not describe implementation differences between VAX LISP/VMS (VAX LISP as implemented on VMS) and VAX LISP/ULTRIX (VAX LISP as implemented on ULTRIX). For such differences, see the *VAX LISP/VMS Release Notes*. For example, LISP020.MEM is the file containing the release notes for Version V2.0.

Most of the information in this chapter refers to subjects that *COMMON LISP: The Language* refers to as implementation dependent. The purpose of this chapter is to clarify the implementation specifics for the following topics:

- Data representation
- Pathnames
- The garbage collector
- Input and output
- Interrupt functions, including keyboard functions that execute asynchronously when you type a control character
- The compiler
- Functions and macros

NOTE

Complex numbers are documented in *COMMON LISP: The Language*, but they are not implemented in VAX LISP.

T, NIL, and keywords are not legal function names in VAX LISP.

VAX LISP supports only symbols that are in the packages named LISP, EDITOR, and UIS.

7.1 DATA REPRESENTATION

COMMON LISP defines the data types implemented in VAX LISP but COMMON LISP does not define implementation-dependent information related to the data types. This section provides data type information specific to VAX LISP. Complete descriptions of data types are provided in *COMMON LISP: The Language*. The following data types require VAX LISP implementation information:

- Numbers
- Characters
- Arrays
- Strings

7.1.1 Numbers

Sections 7.1.1.1 and 7.1.1.2 provide implementation information about the integer and floating-point number data types.

7.1.1.1 **Integers** - COMMON LISP defines two subtypes of integers: fixnums and bignums. The ranges of these two integer types depend on the implementation. In VAX LISP, the integers in the range -2**29 to 2**29-1 are represented as fixnums; integers not in the fixnum range are represented as bignums. VAX LISP stores bignums as two's complement bit sequences.

In VAX LISP, the EQ function returns T when it is called with two fixnums having the same value.

The values of the COMMON LISP integer constants are implementation dependent. The names of the constants and the corresponding VAX LISP values follow:

- MOST-POSITIVE-FIXNUM -- 536870911
- MOST-NEGATIVE-FIXNUM -- -536870912

NOTE

The range of integers represented as fixnums will likely be cut in half in VAX LISP Version 3.0. That is, integers in the range -268,435,456 to +268,435,456 (-2**28 to 2**28-1) will be represented as fixnums. The current range for fixnums is -2**29 to 2**29-1. Remember this note when placing FIXNUM declarations in your programs.

Descriptions of these constants are provided in *COMMON LISP: The Language*.

7.1.1.2 **Floating-Point Numbers** - COMMON LISP defines the following types of floating-point numbers:

- Short floating-point numbers
- Single floating-point numbers
- Double floating-point numbers
- Long floating-point numbers

In VAX LISP, these four types are implemented with VAX floating data types. Both the short and single floating-point numbers are implemented as VAX F_floating data. Double floating-point numbers are implemented as VAX G_floating data. Long floating-point numbers are implemented as VAX H_floating data. For information on the VAX floating data types, see the *VAX Architecture Handbook*.

Table 7-1 lists the types of COMMON LISP floating-point numbers, the corresponding VAX data types, and the number of bits allocated for the exponent and significand of each floating-point type.

Table 7-1: VAX LISP Floating-Point Numbers

COMMON LISP Type	VAX Type	Exponent	Significand
SHORT-FLOAT	F_floating	8	24
SINGLE-FLOAT	F_floating	8	24
DOUBLE-FLOAT	G_floating	11	53
LONG-FLOAT	H_floating	15	113

The values of the COMMON LISP floating-point constants are implementation dependent. You can use the values of these constants to compare the range of values and the degrees of precision of the VAX LISP floating-point types. Table 7-2 lists the names of the constants and provides the actual hexadecimal values and the decimal approximations for VAX LISP.

Table 7-2: Floating-Point Constants

Constant	Hexadecimal Representation	Approximate Decimal Value
DOUBLE-FLOAT-EPSILON	00000000 00003CC0	1.11d-16
DOUBLE-FLOAT-NEGATIVE-EPSILON	00000000 00003CC0	1.11d-16
LEAST-NEGATIVE-DOUBLE-FLOAT	00000000 00008010	-5.56d-309
LEAST-NEGATIVE-LONG-FLOAT	00000000 00000000 00000000 00008001	-8.41L-4933
LEAST-NEGATIVE-SHORT-FLOAT	00008080	-2.94e-39
LEAST-NEGATIVE-SINGLE-FLOAT	00008080	-2.94e-39
LEAST-POSITIVE-DOUBLE-FLOAT	00000000 00000010	5.56d-309
LEAST-POSITIVE-LONG-FLOAT	00000000 00000000 00000000 00000001	8.41L-4933
LEAST-POSITIVE-SHORT-FLOAT	00000080	2.94e-39
LEAST-POSITIVE-SINGLE-FLOAT	00000080	2.94e-39
LONG-FLOAT-EPSILON	00000000 00000000 00000000 00003F90	9.63L-35
LONG-FLOAT-NEGATIVE-EPSILON	00000000 00000000 00000000 00003F90	9.63L-35
MOST-NEGATIVE-DOUBLE-FLOAT	FFFFFFFF FFFFFFFF	-8.99d307
MOST-NEGATIVE-LONG-FLOAT	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF	-5.95L4931
MOST-NEGATIVE-SHORT-FLOAT	FFFFFFFF	-1.70e38
MOST-NEGATIVE-SINGLE-FLOAT	FFFFFFFF	-1.70e38
MOST-POSITIVE-DOUBLE-FLOAT	FFFFFFFF FFFF7FFF	8.99d307
MOST-POSITIVE-LONG-FLOAT	FFFFFFFF FFFFFFFF FFFFFFFF FFFF7FFF	5.95L4931
MOST-POSITIVE-SHORT-FLOAT	FFFF7FFF	1.70e38
MOST-POSITIVE-SINGLE-FLOAT	FFFF7FFF	1.70e38
SHORT-FLOAT-EPSILON	00003480	5.96e-8
SHORT-FLOAT-NEGATIVE-EPSILON	00003480	5.96e-8
SINGLE-FLOAT-EPSILON	00003480	5.96e-8
SINGLE-FLOAT-NEGATIVE-EPSILON	00003480	5.96e-8

Descriptions of these constants are provided in *COMMON LISP: The Language*.

COMMON LISP allows an implementation to define a floating-point minus zero. In VAX LISP, floating-point minus zero does not exist.

7.1.2 Characters

COMMON LISP defines characters as objects that have three attributes: code, bits, and font. The code attribute specifies the way a character is printed or formatted. The bits and font attributes specify extra flags to be associated with a character.

In VAX LISP, the character attributes are defined as follows:

- The code attribute consists of eight bits and is encoded using the extended ASCII character set.
- The bits attribute consists of the four COMMON LISP bits: CONTROL, HYPER, META, and SUPER.
- The font attribute consists of four bits.

NOTE

The CONTROL attribute bit has no association with control characters in the ASCII character set.

The VAX LISP implementation of COMMON LISP functions that perform character comparisons bases its comparisons on the numeric values that correspond to the extended 8-bit ASCII character set. The character predicate functions and the rules that the functions use to compare characters are described in *COMMON LISP: The Language*.

The ordering of two characters that have different bits and font attributes and the same character code is undefined in VAX LISP.

The COMMON LISP character constants that are the exclusive upper limits on the code, bits, and font attributes have implementation-dependent values. The names of the constants and the corresponding VAX LISP values are:

- CHAR-CODE-LIMIT -- 256
- CHAR-BITS-LIMIT -- 16
- CHAR-FONT-LIMIT -- 16

NOTE

The values of these constants might change in future releases of VAX LISP.

Descriptions of these constants are provided in *COMMON LISP: The Language*.

You can obtain a table of valid VAX LISP character names by calling the VAX LISP CHAR-NAME-TABLE function described in Part II.

7.1.3 Arrays

COMMON LISP defines an array as an object whose components are arranged according to a Cartesian coordinate system and whose number of dimensions is called its rank. The limits on an array's rank, dimensions, and total size are implementation dependent.

The names of the array constants and the corresponding VAX LISP values are:

- ARRAY-DIMENSION-LIMIT -- 536870911
- ARRAY-RANK-LIMIT -- 536870911
- ARRAY-TOTAL-SIZE-LIMIT -- 536870911

These constants are described in *COMMON LISP: The Language*.

COMMON LISP defines a specialized array as an array that can contain only elements of a specific type. VAX LISP creates a more efficient specialized array when an array's element type is STRING-CHAR, (SIGNED-BYTE 32), or a subtype of FLOAT or (UNSIGNED-BYTE 1-29). If an array does not have one of these element types, VAX LISP creates a general array (element type is T).

7.1.4 Strings

COMMON LISP defines a string to be a vector of string characters. In VAX LISP, a string can be composed of as many as 65,535 characters.

A string character is a character that can be stored in a string object. In VAX LISP, the characters that compose the 8-bit ASCII character set are string characters. String characters cannot have a bits or font attribute.

7.2 PATHNAMES

In COMMON LISP, a pathname is a LISP data object that represents a file specification. This section describes how VAX LISP implements COMMON LISP pathnames as VMS file specifications. The section is divided as follows:

- Namestrings
- Logical names and pathnames
- When to use pathnames
- Fields of a COMMON LISP pathname
- Field values of a VAX LISP pathname
- Three ways to create pathnames
- Comparing similar pathnames
- Converting pathnames into namestrings
- Functions that use pathnames
- Using the *DEFAULT-PATHNAME-DEFAULTS* variable

7.2.1 Namestrings

In VAX LISP, file names can be represented by pathnames, namestrings, symbols, or streams. Besides the term PATHNAME, COMMON LISP uses the term NAMESTRING. Since computer systems (for example, VMS and ULTRIX) have different ways of formatting file names, COMMON LISP uses namestrings to translate between pathnames (implementation-independent names) and file names (implementation-dependent names).

A namestring is a string naming a file in an implementation-dependent form customary for the file system. A VAX LISP namestring is a string containing a valid VMS file specification. For example, if a file in the VMS file system is called SYS$LOGIN:LOGIN.COM;4, the equivalent namestring would be displayed as "SYS$LOGIN:LOGIN.COM;4".

File system functions, such as LOAD, accept pathnames but internally convert pathnames to namestrings. For more information on namestrings, see Section 7.2.8.

7.2.2 Logical Names and Pathnames

In VAX LISP/VMS, logical names are translated at the time a pathname is created. This is to allow pathnames to be merged properly. Translation of logical names is not normally a problem unless the logical name has multiple translations. In general, use of strings (rather than pathnames) for file specifications improves the utility of logical names with multiple translations. Some functions that accept pathnames or strings as arguments (such as OPEN and PROBE-FILE)

can be passed a string including a reference to such a logical name, and the appropriate translation will be used. Other functions, however, (such as LOAD and COMPILE-FILE) convert string arguments to pathnames in order to merge in the file type and directory, if not specified. In that case, the first translation for which the device and directory exist is used. Providing a complete file specification in the string argument to LOAD or COMPILE-FILE allows all the translations of included logical names to be used.

If a file specification includes a reference to a remote node, logical names are not translated in the resulting pathname.

7.2.3 When to Use Pathnames

Pathnames do not replace the traditional ways of representing a file in LISP. Instead, the pathnames add a new way of representing a file to make LISP programs portable between systems with different file-naming conventions.

Pathnames, however, do not have to refer to an existing file or give complete file specifications; pathnames can exist as data objects in themselves and are used as arguments to pathname functions (see Section 7.2.9 and *COMMON LISP: The Language*).

Several pathname functions and most functions that deal with the file system can take either pathnames, namestrings, symbols, or streams as their arguments. However, the values of the following variable and arguments must be pathnames:

- The *DEFAULT-PATHNAME-DEFAULTS* variable
- The defaults argument in a call to the PARSE-NAMESTRING function

See Section 7.2.10 and *COMMON LISP: The Language* for a description of the preceding variable and function.

7.2.4 Fields of a COMMON LISP Pathname

A COMMON LISP pathname is a LISP data object composed of six fields. Each field represents one of the following aspects of a file specification:

- Host - file system
- Device - file structure or a (physical or logical) device on which files are stored

- Directory - group of related files
- Name - file name
- Type - file extension
- Version - number incremented every time the file is modified

7.2.5 Field Values of a VAX LISP Pathname

For a description of a VAX LISP file specification, see Chapter 1. The following examples show how the components of a VAX LISP file specification are mapped into the fields of a VAX LISP pathname. The first example shows a VAX LISP file specification:

```
MIAMI::DBA1:[SMITH]LOGIN.COM;4
```

The second example shows the pathname that represents the preceding file specification:

```
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME "LOGIN" :TYPE "COM" :VERSION 4)
```

Table 7-3 names the fields of a VAX LISP pathname, the VMS file components that correspond to those fields, and the VAX LISP data type each field accepts.

Table 7-3: VAX LISP Pathname Fields

Pathname Field	VMS Component	Value
:HOST	node	String, integer, or NIL. If you specify a string, the field value can include an access control string, and you must omit the final double colon (::). Examples of host field values are 0, "0", "HOST", "A::B::C", and "A\"NAME password\"".
:DEVICE	device	String or NIL. If you specify a string, you must omit the final colon (:). An example of a device field value is "DBA1".
:DIRECTORY	directory	String, NIL, or the :WILD keyword. The :WILD keyword is translated to the VMS wildcard symbol --

Table 7-3 (cont.)

Pathname Field	VMS Component	Value
		asterisk (*). If you specify a string, you must omit the square brackets ([ ]) or angle brackets (< >). Examples of directory field values are "SMITH", "SMITH.COMMAND", and "SMITH...".
:NAME	filename	String, NIL, or the :WILD keyword. The :WILD keyword is translated to the VMS wildcard symbol -- asterisk (*). If you specify a string, you must omit the period (.) that follows the name. Examples of name field values are "LISP" and "L*SP".
:TYPE	filetype	String, NIL, or the :WILD keyword. The :WILD keyword is translated to the VMS wildcard symbol -- asterisk (*). If you specify a string, you must omit the period (.) that precedes the type. Examples of type field values are "LSP" and "FAS".
:VERSION	version	String, integer, NIL, or keyword. An integer can be positive, negative, or zero. Zero represents the newest version of a file, and minus one (-1) represents the previous version of a file. The following keywords can be specified: :NEWEST equivalent to 0 :PREVIOUS equivalent to -1 :WILD equivalent to "*" If you specify a string, you must omit the initial semicolon (;). Examples of version field values are 0, -14, "2%", and "4*".

7.2.6 Three Ways to Create Pathnames

You can create a pathname in any one of three ways, depending on which of the following functions you use:

- The MAKE-PATHNAME function

```
Lisp> (MAKE-PATHNAME :HOST "MIAMI"
                     :DEVICE "DBAI"
                     :DIRECTORY "SMITH"
                     :NAME "TEST"
                     :TYPE "LSP"
                     :VERSION 0)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME  "TEST" :TYPE "LSP" :VERSION 0)
```

- The PATHNAME function

```
Lisp> (PATHNAME "MIAMI::DBA1:[SMITH]LOGIN.COM;4")
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME "LOGIN" :TYPE "COM" :VERSION 4)
```

- The PARSE-NAMESTRING function

```
Lisp> (PARSE-NAMESTRING "MIAMI::DBA1:[SMITH]LOGIN.COM")
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBAI" :DIRECTORY "SMITH"
            :NAME  "LOGIN" :TYPE "COM" :VERSION 4)
```

The MAKE-PATHNAME function directly creates a pathname from the user-input keywords :HOST, :DIRECTORY, and so on. On the other hand, the PATHNAME function and the PARSE-NAMESTRING function create a pathname by:

- Using a pathname, namestring, symbol, or stream as an argument.
- Parsing the argument.
- Returning a pathname, if the parse operation is a success.

NOTE

The LISP system does not check that you enter an existing or a complete file specification when you create a pathname. So, you can create a pathname that is not usable in VMS. If that situation occurs, and you perform a file operation, the operation will not succeed. To correct the problem, you must change the pathname to conform with a VMS file specification. See Chapter 1 for a description of VMS file specifications and see Section 7.2.5 for a description of the field values in a VAX LISP pathname.

You can specify any valid DECnet-VAX node specification in the host field of a pathname when you are calling a parsing function. Each host name in the specification must be followed by two colons (::) as shown in the following example:

```
Lisp> (PATHNAME "FIRST::SECOND::THIRD::DBA1:[SMITH]PATHNAME")
#S(PATHNAME :HOST "FIRST::SECOND::THIRD" :DEVICE "DBA1"
            :DIRECTORY "SMITH" :NAME "PATHNAME" :TYPE NIL
            :VERSION NIL)
```

The PATHNAME function concatenated the three nodes, FIRST, SECOND, and THIRD, into a single string in the pathname's host field.

If the namestring argument in a call to the PATHNAME or the PARSE-NAMESTRING function is a logical name, the logical name is translated.

The values that the PATHNAME and PARSE-NAMESTRING functions return make the functions different from each other. The PATHNAME function returns a pathname if the parse operation succeeds and returns an error signal if the operation fails. The PARSE-NAMESTRING function also returns a pathname if the parse operation succeeds; if the operation fails, the function either returns NIL or signals an error, depending on the value of the :JUNK-ALLOWED keyword.

Descriptions of the MAKE-PATHNAME, PATHNAME, and PARSE-NAMESTRING functions are provided in *COMMON LISP: The Language*.

7.2.7 Comparing Similar Pathnames

You should use the EQUAL function to compare pathnames with the same field entries. This function is sensitive to keywords and their equivalent symbols (that is, :WILD is equivalent to "*"), and case is not considered in comparisons. For example, if the MAKE-PATHNAME and PARSE-NAMESTRING functions create different pathnames for the file TEST.*;, you can use the EQUAL function to compare the pathname that is returned by each function (see *COMMON LISP: The Language*). The following calls to the SETF macro set the pathnames created by the MAKE-PATHNAME and PARSE-NAMESTRING functions to the variables X and Y:

```
Lisp> (SETF X (MAKE-PATHNAME :NAME "Test"
                             :TYPE :WILD
                             :VERSION 0))
#S(PATHNAME :HOST "MIAMI" :DEVICE NIL :DIRECTORY NIL
            :NAME "Test" :TYPE :WILD :VERSION 0)
Lisp> (SETF Y (PARSE-NAMESTRING "Test.*;"))
#S(PATHNAME :HOST "MIAMI" :DEVICE NIL :DIRECTORY NIL
            :NAME "TEST" :TYPE "*" :VERSION :NEWEST)
```

The EQUAL function can be used to compare the variables X and Y, even though the case of the characters is not the same and the keyword :WILD and its string equivalent ("*") are used.

```
Lisp> (EQUAL X Y)
T
```

The function returns T, indicating that the pathname values of X and Y are equal.

7.2.8 Converting Pathnames into Namestrings

You can convert a pathname into a namestring by specifying the pathname in a call to the NAMESTRING function.

If the argument you specify contains the name of a host, the function invokes DECnet-VAX to perform network operations whether or not the specified host is the current host. To avoid using DECnet-VAX, the VAX LISP implementation of the NAMESTRING function removes the host value if the value is the same as the translation value of SYS$NODE. The following call to the TRANSLATE-LOGICAL-NAME function shows that the current node is MIAMI:

```
Lisp> (TRANSLATE-LOGICAL-NAME "SYS$NODE")
("_MIAMI::")
```

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"
            :NAME "LOGIN" :TYPE "COM" :VERSION NIL)
Lisp> (NAMESTRING THAT-PATHNAME)
"0::THATDEVICE:[SMITH]LOGIN.COM"
```

Table 7-3 noted 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 mypassword\"" :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 mypassword\"::THISDEVICE:[SMITH]FILE"
```

7.2.9 Functions That Use Pathnames

Most of the functions you can use to create and manipulate VAX LISP pathnames are described in *COMMON LISP: The Language*. These functions have at least one required argument and some have optional arguments. In VAX LISP, the value of a pathname function's required argument can be a pathname, namestring, symbol, or stream.

The following two functions need further explanation than is given in *COMMON LISP: The Language*.

- The DIRECTORY function

  The DIRECTORY function (described in Part II) converts its argument to a pathname and merges that pathname with the following VMS file specification:

  host::*device*:[*directory*]*.*;*

  The values for the *host*, *device*, and *directory* fields are supplied by the *DEFAULT-PATHNAME-DEFAULTS* variable (see next section).

- The DEFAULT-DIRECTORY function

  The DEFAULT-DIRECTORY function (described in Part II) is supplied by VAX LISP in addition to the pathname functions described in *COMMON LISP: The Language*. This function returns a pathname that refers to the current directory.

7.2.10 Using the *DEFAULT-PATHNAME-DEFAULTS* Variable

The value of the *DEFAULT-PATHNAME-DEFAULTS* variable is used by some pathname functions to fill pathname fields not specified in their arguments. The default value of this variable is a pathname whose host and directory fields indicate the current directory and whose device, name, type, and version fields contain NIL.

In VAX LISP, you can change the value of the *DEFAULT-PATHNAME-DEFAULTS* variable in two ways:

- With the SETF macro

  The following example illustrates using the SETF macro to change a pathname's directory from [SMITH] to [SMITH.TEST]:

```
Lisp> (SETF *DEFAULT-PATHNAME-DEFAULTS*
       (MAKE-PATHNAME :DIRECTORY "[SMITH.TEST]"))
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
            :DIRECTORY "[SMITH.TEST]" :NAME NIL
            :TYPE NIL :VERSION NIL)
```

- With the DEFAULT-DIRECTORY function

The value of the *DEFAULT-PATHNAME-DEFAULTS* variable is set to the value of your default directory when LISP starts and when you change your directory with the form (SETF (DEFAULT-DIRECTORY) ...). To check the value of your default directory, call the DEFAULT-DIRECTORY function. For example:

```
Lisp> (DEFAULT-DIRECTORY)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
            :DIRECTORY "SMITH" :NAME NIL
            :TYPE NIL :VERSION NIL)
```

The pathname returned in this example indicates that the default directory is SMITH on host MIAMI. In this case, each time a pathname function fills a pathname field with a default value, the corresponding value in the directory SMITH is used.

To change the value of your default directory, set it with the SETF macro. For example, the following illustrates how to change a default directory from SMITH to SMITH.TEST:

```
Lisp> (SETF (DEFAULT-DIRECTORY) "[.TEST]")
"[.TEST]"
```

The next example illustrates that when the directory is changed, the DEFAULT-DIRECTORY function returns a new pathname referring to the new default directory:

```
Lisp> (DEFAULT-DIRECTORY)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
            :DIRECTORY "SMITH.TEST" :NAME NIL
            :TYPE NIL :VERSION NIL)
```

NOTE

The value of the *DEFAULT-PATHNAME-DEFAULTS* variable must be a pathname. Do not set this variable to a namestring, symbol, or stream.

7.3 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-0 space and dynamic-1 space. At a given time, LISP objects are allocated in either dynamic-0 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 referred to are copied to the other space. The objects that cannot be referred to 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 7.3.1 through 7.3.6 provide information about the VAX LISP garbage collector.

7.3.1 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 Chapter 2) 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 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.

7.3.2 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 :ALLOCATION keyword with the MAKE-ARRAY function (see Part II) or with the constructor functions that are defined by the DEFINE-ALIEN-STRUCTURE macro for alien structures. (See the description of the DEFINE-ALIEN-STRUCTURE macro in Part II.).

7.3.3 LISP Processing

LISP processing is suspended during a garbage collection. The VMS operating system queues interrupt functions, such as those defined by the VAX LISP BIND-KEYBOARD-FUNCTION and INSTATE-INTERRUPT-FUNCTION functions, for delivery after garbage collection is finished. Interrupt functions are discussed in Section 7.5.

7.3.4 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, to determine whether your program is in a recursive loop is difficult. Therefore, you should not suppress or change the messages before you debug your program.

7.3.5 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 Chapter 2.

7.3.6 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.

7.4 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 Record Management Services 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 7.4.1 through 7.4.6.

7.4.1 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 displayed 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.

7.4.2 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 key.

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 interrupt function.

See the *VAX/VMS I/O Reference Manual: Part I* for information on terminal control characters, and see Section 7.5 for information about interrupt functions.

You can change the mode in which your terminal performs input operations by invoking the VAX LISP SET-TERMINAL-MODES function with the :PASS-THROUGH keyword (see Part II). For example:

```
Lisp> (SET-TERMINAL-MODES :PASS-THROUGH T)
T
```

If the value of the :PASS-THROUGH keyword is T, the SET-TERMINAL-MODES pass-through mode, control characters processed by the VMS system and characters defined to invoke interrupt 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 pass-through mode, that function returns a character as soon as the character is typed. See *COMMON LISP: The Language* 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-THROUGH keyword set to NIL.

```
Lisp> (SET-TERMINAL-MODES :PASS-THROUGH NIL)
T
```

7.4.3 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 *TERMINAL-IO*, no action is performed.

7.4.4 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 9995 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.

7.4.5 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.

7.4.6 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

7.4.6.1 **FILE-LENGTH Function** - The length of a file is measured in units of the OPEN function's :ELEMENT-TYPE keyword. In VAX LISP/VMS, 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.

7.4.6.2 **FILE-POSITION Function** - The FILE-POSITION function returns or sets the current position within a random-access file. VAX LISP/VMS does not support random-access files; therefore, the function returns NIL.

7.4.6.3 **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 restricts the values you can specify for the preceding keywords. The rest of this section explains the restrictions.

In VAX LISP/VMS, 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 funciton 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 :OVERWRITE option is not supported in VAX LISP/VMS.

For the :IF-EXISTS keyword values of :RENAME, :RENAME-AND-DELETE, and :SUPERSEDE, the old file is renamed to the same name with the string "old" appended to the file type. On closing files opened with any of these three values, and specifying :ABORT T, the new version is deleted and the old is restored to its former name. On closing files with :ABORT NIL, on :RENAME, there is no action; with :RENAME-AND-DELETE or :SUPERSEDE, the old file is deleted.

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.

7.4.6.4 **WRITE-CHAR Function** - The WRITE-CHAR function disregards the bit and font attributes of characters.

7.5 INTERRUPT FUNCTIONS AND KEYBOARD FUNCTIONS

An interrupt function is a function that is invoked when a specific event occurs. If an interrupt function is defined for an event, the VAX LISP system interrupts the current LISP processing and invokes the interrupt function when the event occurs. When the interrupt function exits, the VAX LISP system resumes processing at the point where it was interrupted.

VAX LISP provides two functions you can use to define interrupt functions: INSTATE-INTERRUPT-FUNCTION and BIND-KEYBOARD-FUNCTION. The INSTATE-INTERRUPT-FUNCTION function is part of a general mechanism that lets your program respond to asynchronous events (ASTs) in the VMS operating system. This mechanism is described in the *VAX LISP/VMS System Access Programming Guide*.

The BIND-KEYBOARD-FUNCTION function is a more specialized function that binds an ASCII control character to an interrupt 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. Functions bound using BIND-KEYBOARD-FUNCTION are also called keyboard functions.

Interrupt functions are not always called as soon as the defined event occurs. If a low-level LISP function, such as CDR or CONS, is being evaluated or a garbage collection is being performed, interrupt functions are placed in a queue until they can be evaluated. Delays in interrupt function evaluation are generally not perceptible. An example of when you might perceive a delay is when the system performs a garbage collection.

VAX LISP also provides a means by which you can assign different priorities for interrupt and keyboard functions. These priorities, called interrupt levels, are described in the *VAX LISP/VMS System Access Programming Guide*.

If you suspend the LISP system when interrupt 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. Any other interrupt functions that you may have defined are uninstated when the system is suspended and are not reinstated when the system is resumed.

Besides 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.

7.6 COMPILER

For information on how to compile LISP expressions and the advantages and disadvantages of compiling LISP expressions, see Chapter 2. This section describes two compiler restrictions (one with the COMPILE function and one with the COMPILE-FILE function) and compiler optimizations.

7.6.1 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.

7.6.1.1 **COMPILE Function** - The compiler cannot compile pieces of code unless they are function definitions defined at top level. 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 0))
           (COMPILE NIL #'(LAMBDA () (INCF COUNTER))))
```

The COMPILE function cannot compile the function object in the preceding example because the object 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 0))
           (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.

7.6.1.2 **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> (SETF 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 SETF special form. The preceding call to the SETF special form can be compiled but the list cannot be put into the output file.

7.6.2 Compiler Optimizations

In VAX LISP, you can control two qualities of compiled code: the speed of the generated code and whether run-time safety checking is to be performed. The default value for these qualities is 1. 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 Chapter 2) or specify the OPTIMIZE declaration in a call to the PROCLAIM function (see *COMMON LISP: The Language*). Both 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 (speed of object code) to 3 and the safety quality (run-time error checking) to 2, 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.

All proclamations are put into the fastload file so that they also occur when fastloaded. However, the compiler observes INLINE proclamations only when the OPTIMIZE SPEED quality is greater than the OPTIMIZE SPACE quality, and does not check for stack overflow.

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 1, or the value of the speed quality must be less than 2. 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 2, and the value of the safety quality must be less than or equal to 1. 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 1 and the speed quality is 2:

```
(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 2 and the value of the speed quality is greater than 1, the compiler regards the type declarations. In this example, the types FIXNUM and LIST are declared with 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.

The value of the *FINAL-VALUE* variable and the return value of the GET-INITIAL-VALUE function must be 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.

If you want a function to be compiled inline, you must proclaim it INLINE. Declaring a function INLINE has no effect. However, once a function has been proclaimed INLINE, it will be compiled inline unless specifically declared NOTINLINE.

For more information on making LISP compiled code run fast, see the release notes.

7.7 FUNCTIONS AND MACROS

Several functions and macros described in *COMMON LISP: The Language* have implementation dependencies. Table 7-4 lists the names of these functions and macros and provides a brief explanation of the type of information that is implementation dependent. For a summary description of these functions and macros, see Part II. 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 further information regarding these functions and macros.

Table 7-4: 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	Argument merged with wildcards
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

Table 7-4 (cont.)

Name	Function or Macro	Implementation-Dependent Information
LOAD	Function	Finds latest file
LONG-SITE-NAME	Function	Logical name and return value
MACHINE-INSTANCE	Function	Logical name and return value
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

PART II

VAX LISP/VMS FUNCTION, MACRO, AND VARIABLE DESCRIPTIONS

APROPOS Function

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 by default only symbols that are accessible from the current or specified 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.

APROPOS Function (cont.)

Example

```
Lisp> (APROPOS "*PRINT")

Symbols in package USER containing the string "*PRINT":
  *PRINT-CIRCLE*, has a value
  *PRINT-SLOT-NAMES-AS-KEYWORDS*, has a value
  *PRINT-RADIX*, has a value
  *PRINT-ESCAPE*, has a value
  *PRINT-ARRAY*, has a value
  *PRINT-GENSYM*, has a value
  *PRINT-LEVEL*, has a value
  *PRINT-PRETTY*, has a value
  *PRINT-LENGTH*, has a value
  *PRINT-RIGHT-MARGIN*, has a value
  *PRINT-MISER-WIDTH*, has a value
  *PRINT-BASE*, has a value
  *PRINT-CASE*, has a value
  *PRINT-LINES*, 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 Function

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 by default 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 string.

Example

```
Lisp> (APROPOS-LIST "ARRAY")
(ARRAY-TOTAL-SIZE ARRAY-DIMENSION ARRAY-DIMENSIONS
SIMPLE-ARRAY ARRAY-DIMENSION-LIMIT ARRAY-ELEMENT-TYPE
ARRAYP *PRINT-ARRAY* ARRAY-RANK ARRAY-RANK-LIMIT
MAKE-ARRAY ARRAY-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.

ATTACH Function

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 DCL Dictionary*.

NOTE

The ATTACH function can be used only if LISP is invoked from DCL; it cannot be used if LISP is invoked from another command language interpreter (CLI).

Format

ATTACH *process*

Argument

process

The name or identification (PID) of the process 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.

ATTACH Function (cont.)

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 Function

Binds an ASCII keyboard control character (characters of codes 0 to 31) 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. A function bound in this way is called a keyboard function.

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.

You can specify an interrupt level (an integer in the range 0 through 7) for a keyboard function by using the :LEVEL keyword. A keyboard function can only interrupt code that is executing at an interrupt level below its own. Keep the following guidelines in mind when specifying an interrupt level:

- The default interrupt level for keyboard functions is 1.
- Interrupt level 6 is used by LISP to handle keyboard input; therefore, a keyboard function executing at interrupt level 6 cannot receive input from the keyboard. For this reason, be careful when using interrupt level 6.
- Interrupt level 7 can interrupt any code that is not in the body of a CRITICAL-SECTION macro. A keyboard function executing at interrupt level 7 *must* terminate by executing a THROW to a tag, such as CANCEL-CHARACTER-TAG.
- If you bind a control character to the BREAK or DEBUG functions, use a level that is high enough to interrupt your other keyboard and interrupt functions but that is less than 6.
- If you bind a control character to the ED function, use the default interrupt level (1) or a lower level.

The *VAX LISP/VMS System Access Programming Guide* contains more information about using interrupt levels and about the CRITICAL-SECTION macro and interrupt functions.

BIND-KEYBOARD-FUNCTION Function (cont.)

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 Chapter 7 for an explanation about calling functions asynchronously.

Format

BIND-KEYBOARD-FUNCTION *control-character function*
&KEY :ARGUMENTS :LEVEL

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.

:ARGUMENTS

A list containing 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.

:LEVEL

An integer in the range 0-7, specifying the interrupt level for the keyboard function. The default is 1.

Return Value

T.

Examples

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

Binds CTRL/B to the BREAK function. You can then invoke a break loop by typing CTRL/B.

BIND-KEYBOARD-FUNCTION Function (cont.)

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.

BREAK Function

Invokes a break loop. A break loop is a nested read-eval-print loop. For more information about break loops, see Chapter 5.

Format

BREAK &OPTIONAL *format-string* &REST *args*

Arguments

format-string

The string of characters that is passed to the FORMAT function to create the break-loop message.

args

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 (UNUSUAL-SITUATION-P STATUS)
    (BREAK "Unusual situation ~D encountered. Please investigate"
      STATUS))
```

Calls the BREAK function if the value of the UNUSUAL-SITUATION-P function is not NIL. The break message contains the condition code.

CANCEL-CHARACTER-TAG Tag

CANCEL-CHARACTER-TAG, when used in a CATCH construct, catches the throw that occurs whenever the cancel character is typed at the keyboard. In VAX LISP/VMS, CTRL/C is the cancel character. Thus, you can use CANCEL-CHARACTER-TAG in a CATCH construct to alter the behavior when a user types CTRL/C.

You can also use CANCEL-CHARACTER-TAG in a THROW construct to cause an exit to the VAX LISP read-eval-print loop. In this way, you can partially simulate the action of the cancel character from within your code. (The cancel character also invokes the CLEAR-INPUT function on the *TERMINAL-IO* stream.)

Format

```
CANCEL-CHARACTER-TAG
```

Example

```
Lisp> (DEFUN TRAPPER ()
        (CATCH 'CANCEL-CHARACTER-TAG
               (LOOP))
        (PRINC "Execution came through here"))
TRAPPER
Lisp> (TRAPPER)
<CTRL/C>
Execution came through here
"Execution came through here"
Lisp>
```

- The TRAPPER function sets up a catcher for CANCEL-CHARACTER-TAG, then enters an infinite loop.
- The user types CTRL/C.
- The PRINC function prints a string, indicating that execution continued following the CATCH form rather than returning directly to the Lisp> prompt.

CHAR-NAME-TABLE Function

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
```

CHAR-NAME-TABLE Function (cont.)

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 Function

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, macro, or special form definition is to be checked.

Return Value

The interpreted function, macro, or special form definition, if the symbol has an interpreted 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 previously compiled.

COMPILE-FILE Function

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 :LISTING :MACHINE-CODE :OPTIMIZE
          :OUTPUT-FILE :VERBOSE :WARNINGS
```

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. The file type defaults to LSP.

:LISTING

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 that contains the source file.

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 *input-pathname* to fill the components of the file specification that are not specified.

:MACHINE-CODE

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 T or NIL. If you specify T, the listing file contains the machine code. If you specify NIL, the listing file does not contain the machine code. The default value is NIL.

:OPTIMIZE

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

COMPILE-FILE Function (cont.)

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 of compiler optimizations, see Chapter 7.

:OUTPUT-FILE

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 that contains the source file. The default value is T.

If you specify NIL, no fast-loading file is produced.

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 *input-pathname* to fill the components of the file specification that are not specified.

:VERBOSE

Specifies whether the compiler is to display the name of functions and macros it compiles. The value can be 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 (By default, T).

:WARNINGS

Specifies whether the compiler is to display warning messages. The value can be 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 (By default, T).

Return Value

If the compiler generated an output file, a namestring is returned. Otherwise, NIL is returned.

COMPILE-FILE Function (cont.)

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.

COMPILE-VERBOSE Variable

Controls the amount of information that the compiler displays.

The COMPILE-FILE function binds the *COMPILE-VERBOSE* variable to the value supplied by the :VERBOSE keyword. If the :VERBOSE keyword is not specified, the function uses the existing value of the *COMPILE-VERBOSE* variable. 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.FAS;1"
Lisp> (SETF *COMPILE-VERBOSE* NIL)
NIL
Lisp> (COMPILE-FILE 'MATH)
"DBA1:[SMITH]MATH.FAS;2"
```

- 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 Variable

Controls whether the compiler displays warning messages during a compilation.
The COMPILE-FILE function binds the *COMPILE-WARNINGS* variable to the value supplied with the :WARNINGS keyword. If the :WARNINGS keyword is not specified, the function uses the existing value of the *COMPILE-WARNINGS* variable. 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 value not used.
FACTORIAL compiled.
Warning in FIBONACCI
  N bound but value not used.
FIBONACCI compiled.

Finished compilation of file DBA1:[SMITH]MATH.LSP;2
0 Errors, 2 Warnings
"DBA1:[SMITH]MATH.FAS;3"
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.FAS;4"
```

- 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.

COMPILE-WARNINGS Variable (cont.)

- 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 Function

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 at which the break loop was entered.

Format

```
CONTINUE
```

Return Value

NIL.

Example

```
Lisp> (BIND-KEYBOARD-FUNCTION #\^B #'BREAK)
Lisp> (LOAD "FILEB.LSP")
; Loading contents of file LISPW$:[SMI...
^B
Break> (LOAD "FILEA.LSP")
; Loading contents of file LISPW$:[SMITH]FILEA.LSP;1
;   FUNCTION-A
; Finished loading LISPW$:[SMITH]FILEA.LSP;1
T
Break> (CONTINUE)
Continuing from break loop...
;   FUNCTION-B
; Finished loading LISPW$:[SMITH]FILEB.LSP;1
T
Lisp>
```

- The BREAK function is bound to CTRL/B.
- FILEB.LSP is loaded.
- The programmer, realizing that FILEA.LSP (which is needed to initialize an environment for FILEB.LSP) is not yet loaded, invokes the BREAK loop.
- FILEA.LSP is then loaded.
- Finally, the call to the CONTINUE function continues the loading of FILEB.LSP and then returns the programmer to the top-level loop.

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
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-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> (SETF THIS-STRING "abcd")
"abcd"
Lisp> (FUNCTION-Y THIS-STRING 4)
.... Error in function FUNCTION-Y
Frame #4 (FUNCTION-Y "abcd" 4)
Debug 1> (SETF STRING (SECOND (DEBUG-CALL)))
"abcd"
Debug 1> (EQ "abcd" STRING)
NIL
Debug 1> (EQ THIS-STRING STRING)
T
```

In this case, the function in the current stack frame is FUNCTION-Y. The call to (DEBUG-CALL) returns the list (FUNCTION-Y "abcd 4). The form (SECOND (DEBUG-CALL)) evaluates "abcd", the first argument to FUNCTION-Y in the current stack frame. Note that the string returned by the call (SECOND (DEBUG-CALL)) is the same string passed to the function FUNCTION-Y. See the description of the TRACE macro 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
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 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
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 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.

DEFAULT-DIRECTORY Function

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 DCL Dictionary*.

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 Chapter 7 and *COMMON LISP: The Language* for information about pathnames and the *DEFAULT-PATHNAME-DEFAULTS* variable.

Note that the directory must exist for the change of directory to succeed.

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.TESTS" :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.

DEFAULT-DIRECTORY Function (cont.)

2.
```
Lisp> (DEFAULT-DIRECTORY)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
:DIRECTORY "SMITH.TESTS" :NAME NIL :TYPE NIL
:VERSION NIL)
Lisp> *DEFAULT-PATHNAME-DEFAULTS*
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
:DIRECTORY "SMITH.TESTS" :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.
- 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 string.
- 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-FORMAT-DIRECTIVE Macro

Defines a directive for use in a FORMAT control string, supplementing directives supplied with VAX LISP. In a call to FORMAT, specify a directive you have defined in the form:

~/*name*/

You can also specify colon and at-sign modifiers:

~@:/*name*/

You can also specify one or more parameters:

~*n*,*n*/*name*/

DEFINE-FORMAT-DIRECTIVE provides means for the body of the format directive you define to receive the value of parameters and the presence or absence of colon and at-sign modifiers.

See Section 6.4 for more information about defining format directives.

Format

```
DEFINE-FORMAT-DIRECTIVE name
     (arg stream colon-p atsign-p
     &OPTIONAL (parameter1 default)
               (parameter2 default)...)
     &BODY forms
```

Arguments

name

The name of the FORMAT directive defined with this macro.

NOTE

If you do not specify a package with *name* when you define the directive, *name* is placed in the current package. If you do not specify a package when you refer to the directive, the FORMAT directive looks in the USER package for the directive definition.

arg

A symbol that is bound to the argument to be formatted by the directive.

DEFINE-FORMAT-DIRECTIVE Macro (cont.)

stream

A symbol that is bound to the stream to which the printing is to be done.

colon-p

A symbol that is bound to T or NIL, indicating whether a colon was specified in the directive.

atsign-p

A symbol that is bound to T or NIL, indicating whether an at-sign was specified in the directive.

parameters

There must be one optional argument for each prefix parameter that is allowed in the directive. A symbol supplied as a *parameter* argument will be bound to the corresponding prefix parameter if it was specified in the directive. Otherwise, the default value will be used, as with all optional arguments.

forms

Forms which are evaluated to print *argument* to *stream*. The body can begin with a declaration and/or documentation string.

Return Value

The name of the FORMAT directive that has been defined.

Example

```
Lisp> (DEFINE-FORMAT-DIRECTIVE EVALUATION-ERROR
                               (SYMBOL STREAM COLON-P ATSIGN-P
                               &OPTIONAL (SEVERITY 0))
        (DECLARE (IGNORE ATSIGN-P))
        (FRESH-LINE STREAM)
        (PRINC (CASE SEVERITY
                 (0     "Warning: ")
                 (1     "Error: ")
                 (2     "Severe Error: "))
               STREAM)
        (FORMAT STREAM "~:!The symbol ~S ~:_does not have an ~
                        integer value.~%Its value is: ~:_~S~."
                        SYMBOL (SYMBOL-VALUE SYMBOL))
        (WHEN COLON-P
          (WRITE-CHAR #\BELL STREAM)))
EVALUATION-ERROR
```

DEFINE-FORMAT-DIRECTIVE Macro (cont.)

```
Lisp> (SETF PROCESS NIL)
NIL
Lisp> (FORMAT T "~1:/EVALUATION-ERROR/" 'PROCESS)
Error: The symbol PROCESS does not have an integer value.
       Its value is: NIL
<BEEP>
```

- This example shows the definition of a FORMAT directive, a use of the directive, and the printed output.

- The prefix parameter 1 in "~1:/EVALUATION-ERROR/" indicates the severity of the error being signaled. The colon produces a beep on the terminal.

DEFINE-GENERALIZED-PRINT-FUNCTION Macro

Defines a function that specifies how any object is to be pretty printed, regardless of its form. Generalized print functions are effective only when they are enabled (globally or locally) and when pretty printing is enabled. You can enable a generalized print function globally, using GENERALIZED-PRINT-FUNCTION-ENABLED-P. Or, you can enable it locally, using WITH-GENERALIZED-PRINT-FUNCTION. An enabled generalized print function is used if its predicate evaluates to a non-NIL value.

See Section 6.6 for more information about generalized print functions.

Format

DEFINE-GENERALIZED-PRINT-FUNCTION *name* (*object stream*) *predicate*
&BODY *forms*

Arguments

name

The name of the generalized print function being defined.

object

A symbol that is bound to the object to be printed.

stream

A symbol that is bound to the stream to which output is to be sent.

predicate

A form. When the generalized print function has been enabled (globally or locally), the system evaluates this form for every object to be pretty printed. If the form evaluates to non-NIL on the object to be pretty printed, the generalized print function will be used.

forms

Forms that print *object* to *stream*, or take any other action. These forms can refer to the object and stream by means of the symbols used for *object* and *stream*. The body can begin with a declaration and/or documentation string.

Return Value

The name of the generalized print function that has been defined.

DEFINE-GENERALIZED-PRINT-FUNCTION Macro (cont.)

Example

```
Lisp> (DEFINE-GENERALIZED-PRINT-FUNCTION PRINT-NIL-AS-LIST
         (OBJECT STREAM)
         (NULL OBJECT)
         (PRINC "( )" STREAM))
PRINT-NIL-AS-LIST
Lisp> (PRINT NIL)
NIL
NIL
Lisp>(PPRINT NIL)
NIL
Lisp> (WITH-GENERALIZED-PRINT-FUNCTION 'PRINT-NIL-AS-LIST
         (PRINT NIL)
         (PPRINT NIL))
NIL
( )
Lisp> (SETF (GENERALIZED-PRINT-FUNCTION-ENABLED-P
               'PRINT-NIL-AS-LIST)
            T)
T
Lisp> (PPRINT NIL)
( )
```

- The first PRINT call prints NIL, because the generalized print function PRINT-NIL-AS-LIST is not enabled.
- The first PPRINT call prints NIL, because PRINT-NIL-AS-LIST is still not enabled.
- The second PRINT call prints NIL, because pretty printing is not enabled.
- The second PPRINT call prints ( ), because the generalized print function is enabled locally.
- The third PPRINT call prints ( ), because the generalized print function is enabled globally.

DEFINE-LIST-PRINT-FUNCTION Macro

Defines and enables a function to print lists that begin with a specified element. Defined functions are effective only when pretty printing is enabled. The system checks the first element of each list to be printed for a match. If the first element of a list matches the name of a list-print function, the list is printed according to the format you have defined.

See Section 6.5 for more information about list-print functions.

Format

DEFINE-LIST-PRINT-FUNCTION *symbol* (*list stream*) &BODY *forms*

Arguments

symbol

The first element of any list to be printed in the defined format.

list

A symbol that is bound to the list to be printed.

stream

A symbol that is bound to the stream on which printing is to be done.

forms

Forms to be evaluated. The forms refer to the list to be printed and the stream by means of the symbols you supply for *list* and *stream*. The body can include declarations. Calls to FORMAT may also be included.

Return Value

The name of the list-print function that has been defined.

Example

```
Lisp> (DEFINE-LIST-PRINT-FUNCTION MY-SETQ (LIST STREAM)
        (FORMAT STREAM
                "~1!~W~^ ~:I~@{~W~^ ~:_~W~^~%~}~."
                LIST))
MY-SETQ
Lisp> (SETF BASE '(MY-SETQ HI 3 BYE 4))
(MY-SETQ HI 3 BYE 4)
```

DEFINE-LIST-PRINT-FUNCTION Macro (cont.)

```
Lisp> (PRINT BASE)
(MY-SETQ HI 3 BYE 4)
(MY-SETQ HI 3 BYE 4)
Lisp> (PPRINT BASE)
(MY-SETQ HI 3
         BYE 4)
```

- The list-print function MY-SETQ is defined.
- The call to PRINT does not use the list-print function MY-SETQ to print the value of BASE, because pretty-printing is not enabled.
- The call to PPRINT does use the list-print function MY-SETQ to print the value of BASE.

DELETE-PACKAGE Function

Uninterns all the symbols interned in the package, unuses all the packages the function uses, and deletes the package. An error is signaled if the package is used by any other package.

Format

DELETE-PACKAGE *package*

Argument

package

A package, or a string or symbol naming a package

Return Value

T.

Example

```
Lisp> (DELETE-PACKAGE "TEST-PACKAGE")
T
Lisp> (FIND-PACKAGE "TEST-PACKAGE")
NIL
```

DESCRIBE Function

Displays information about a specified object. If the specified object has a documentation string, this function displays the string in addition to the other information the function 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
```

2.
```
Lisp> (DESCRIBE 'FACTORIAL)

It is the symbol FACTORIAL
Package:  USER
Value:    unbound
Function: a compiled-function
    FACTORIAL n
```

3.
```
Lisp> (DESCRIBE PI)

It is the long-float 3.14159265358979323846264338327950L0
Sign:         +
Exponent:     2 (radix 2)
Significand:  0.78539816339744830961566084581988L0
```

DESCRIBE Function (cont.)

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 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 (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.

DIRECTORY Function (cont.)

- 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 Function

Echoes the input and output of an interactive LISP session to a specified file, 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 echoing input and output to the pathname, close the file by calling the DRIBBLE function without an argument.

In VAX LISP/VMS, the two restrictions on the use of the DRIBBLE function are:

- 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 an argument is specified with the function, no value is returned and dribbling is turned on. If debugging is on and the function is called with no arguments, then T is returned and dribbling is turned off. If dribbling is off and is called without an argument, NIL is returned.

Examples

1.
```
Lisp> (DRIBBLE "NEWFUNCTION.LSP")
Dribbling to DBA1:[SMITH]NEWFUNCTION.LSP;1
Lisp>
```

   Creates a dribble file named NEWFUNCTION.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 Function

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.

NOTE

If you bind a control character (such as CTRL/E) to the ED function using BIND-KEYBOARD-FUNCTION, specify an interrupt level of 1 (the default) or 0 with the :LEVEL keyword. Do not specify a higher interrupt level.

See Chapter 3 for information on using the VAX LISP Editor.

Format

ED &OPTIONAL *x* &KEY :TYPE

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.

:TYPE

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.

ED Function (cont.)

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 'GAMEBOARD-ARRAY :TYPE :VALUE)

   Invokes the Editor to edit the value of the symbol GAMEBOARD-ARRAY.

4.
```
Lisp> (DEFSTRUCT ROOM
        DOORS
        WINDOWS
        OUTLETS
        COLOR)
ROOM
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.

ERROR-ACTION Variable

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.

In addition to setting this variable within a LISP form, you can also set it on LISP initialization with the /ERROR_ACTION qualifier (see Chapter 2).

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 Function

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

The EXIT function does not return to LISP.

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.

Format Directives Provided with VAX LISP

VAX LISP provides eight directives for the FORMAT function, in addition to those described in *COMMON LISP: The Language*. Table 1 lists and describes these directives. See Section 6.3 for more information about using these directives.

Table 1: Format Directives Provided with VAX LISP

Directive	Effect
~W	Prints the corresponding argument under direction of the current print variable values. The argument for ~W can be any LISP object. This directive takes a colon modifier and four prefix parameters. Use the colon modifier (~:W) when you want to set *PRINT-PRETTY* and *PRINT-ESCAPE* to T, and set *PRINT-LENGTH*, *PRINT-LEVEL*, and *PRINT-LINES* to NIL. The prefix parameters specify padding. These parameters are identical to those used with the ~A directive. ~*mincol*,*colinc*,*minpad*,*padchar*W *mincol* specifies the minimum width of the printed representation of the object. FORMAT inserts padding characters on the right, until the width is at least *mincol* columns. Use the at-sign with *minpad* to insert the padding characters on the left instead. The default for *mincol* is 0. *colinc* specifies an increment: the number of padding characters to be inserted at one time until the width is at least *mincol* columns. The default is 1. *minpad* specifies the minimum number of padding characters to be inserted. The default is 0. *padchar*, preceded by a single quote, specifies the padding character. The default is the space character.
~!	Begins a logical block. A logical block is a hierarchical grouping of FORMAT directives treated as a unit. FORMAT must be processing a logical block with *PRINT-PRETTY* true to enable pretty printing. Directives inside a logical block refer to elements of a single list taken from the argument list to FORMAT.

Format Directives Provided with VAX LISP (cont.)

Table 1 (cont.)

Directive	Effect
	(If the argument supplied to the logical block is not a list, then the logical block is skipped and the argument is printed as if with ~W.) The logical block directive takes colon and at-sign modifiers. When the directive is modified by a colon (~:!), the directive sets *PRINT-PRETTY* and *PRINT-ESCAPE* to T and *PRINT-LENGTH*, *PRINT-LEVEL*, and *PRINT-LINES* to NIL. When the directive is modified by an at-sign (~@!), the directives within the logical block take successive arguments from the FORMAT argument list. The logical block uses up all the arguments, not just a single list argument. Arguments not needed by the logical block are used up as well, so that they are not available for subsequent directives. Specify a parameter of 1 (~1!) to enclose the output in parentheses.
~.	Ends a logical block. If modified by an at-sign (~@!), the directive inserts a new line if needed after every blank space character.
~_	Specifies a multiline mode new line and marks a logical block section. This directive takes colon and at-sign modifiers. When modified by a colon (~:_), the directive starts a new line if not enough space is on the line to print the next logical block section. When modified by an at-sign (~@_), the directive starts a new line if miser mode is enabled. The ~_ directive and its variants are effective only when used within a logical block with pretty printing enabled.
~*n*I	Sets indentation for subsequent lines to *n* columns after the beginning of the logical block or after the prefix. When modified by a colon (~*n*:I), the directive causes FORMAT to indent subsequent lines *n* spaces from the column corresponding to the position of the directive. The ~*n*I directive and the ~*n*:I variant are effective only when used within a logical block with pretty printing enabled.

Format Directives Provided with VAX LISP (cont.)

Table 1 (cont.)

Directive	Effect
~*n*/FILL/	Prints the elements of a list with as many elements as possible on each line. If *n* is 1, FORMAT encloses the printed list in parentheses. If pretty printing is not enabled, the directive causes FORMAT to print the output on a single line.
~*n*/LINEAR/	If the elements of the list to be printed cannot be printed on a single line, this directive prints each element on a separate line. If *n* is 1, FORMAT encloses the printed list in parentheses. If pretty printing is not enabled, this directive causes FORMAT to print the output on a single line.
~*n*,*m*/TABULAR/	Prints the list in tabular form. If *n* is 1, FORMAT encloses the list in parentheses; *m* specifies the column spacing. If pretty printing is not enabled, this directive causes FORMAT to print the output on a single line.

GC Function

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 Chapter 7 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 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 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-DEVICE-INFORMATION Function

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 Reference Manual: Part I*.

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 2 lists the keywords that you can specify and the values they return.

Table 2: 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.

GET-DEVICE-INFORMATION Function (cont.)

Table 1 (cont.)

Keyword	Return Value
:DEVICE-CHARACTERISTICS	A vector of 32 bits that specifies the device characteristics. See the *VAX/VMS I/O User's Reference Manual* for information about device characteristics.
:DEVICE-CLASS	An integer that specifies the device class.
:DEVICE-DEPENDENT-0	A bit vector that specifies device-dependent information.
: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 has 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.

GET-DEVICE-INFORMATION Function (cont.)

Table 1 (cont.)

Keyword	Return Value
: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 has 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.
: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.

GET-DEVICE-INFORMATION Function (cont.)

Table 1 (cont.)

Keyword	Return Value
: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 property 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 0 :MOUNT-COUNT 1)
```

Returns the device name, the error count, and the mount count for the device DBA1.

GET-FILE-INFORMATION Function

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 Record Management Services 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 3 lists the keywords that you can specify and the values they return.

Table 3: GET-FILE-INFORMATION Keywords

Keyword	Return Value
:ALLOCATION-QUANTITY	An integer that specifies the number of blocks that is 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 was added to the file's size when the file was extended.

GET-FILE-INFORMATION Function (cont.)

Table 3 (cont.)

Keyword	Return Value
: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.
:FIRST-FREE-BYTE	An integer that specifies the offset of the first byte in the file's end-of-file block.
:FIXED-CONTROL-SIZE	An integer that specifies the fixed control area size.
:GROUP	An integer that specifies the owner group number.
:LONGEST-RECORD-LENGTH	An integer that specifies the length of the longest record in the file.
:MAX-RECORD-SIZE	An integer that specifies the maximum size a record in the file can be.
:MEMBER	An integer that specifies the owner member number.
:ORGANIZATION	An integer that specifies the organization.
:PROTECTION	A vector of 16 bits that specifies the protection code.
:RECORD-ATTRIBUTES	An integer that specifies the record attributes.
:RECORD-FORMAT	An integer that specifies the record format.
:REVISION	An integer that specifies the revision number.
:REVISION-DATE	The last universal date and time the file was revised.

GET-FILE-INFORMATION Function (cont.)

Table 3 (cont.)

Keyword	Return Value
:UIC	An integer that specifies the owner UIC.
:VERSION-LIMIT	An integer that specifies the maximum version number the file can have.

Return Value

The keywords and their values are returned as a property 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 file does not exist, the function returns NIL.

Examples

1.
```
Lisp> (GET-FILE-INFORMATION "IMPORTANT.DAT"
                            :ALLOCATION-QUANTITY
                            :BACKUP-DATE)
(:ALLOCATION-QUANTITY 252 :BACKUP-DATE 2654202351)
```

   Returns the allocation quantity and backup date for the file IMPORTANT.DAT.

2.
```
Lisp> (DEFUN SHOW-FILE-SIZE (FILE)
        (LET ((SIZE-LIST
               (GET-FILE-INFORMATION FILE
                                     :ALLOCATION-QUANTITY
                                     :END-OF-FILE-BLOCK)))
          (FORMAT T
                  "~A ~%~
                   ~3T Blocks allocated: ~D~%~
                   ~3T Blocks used:      ~D~%"
                  (NAMESTRING (TRUENAME FILE))
                  (GETF SIZE-LIST :ALLOCATION-QUANTITY)
                  (GETF SIZE-LIST :END-OF-FILE-BLOCK))))
SHOW-FILE-SIZE
```

GET-FILE-INFORMATION Function (cont.)

```
Lisp> (SHOW-FILE-SIZE "MYFILE.TXT")
DBA1:[SMITH]MYFILE.TXT;4
   Blocks allocated: 240
   Blocks used:      239
NIL
```

- The call to the DEFUN macro defines a function named SHOW-FILE-SIZE, which displays the amount of space that is allocated for a specified file and the amount of space the file uses.

- The call to SHOW-FILE-SIZE displays the amount of space that is allocated for the file MYFILE.TXT and the amount of space the file uses.

GET-GC-REAL-TIME Function

Lets you 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 0.

For more information on the garbage collector, see Chapter 7.

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)
3485700000
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.

GET-GC-REAL-TIME Function (cont.)

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.SUS"))
; 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.SUS").

GET-GC-RUN-TIME Function

Lets you 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 INTERNAL-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 INTERNAL-TIME-UNITS-PER-SECOND constant is provided in *COMMON LISP: The Language*.

When a suspended system is resumed, the CPU time is set to 0.

For more information on the garbage collector, see Chapter 7.

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.

GET-GC-RUN-TIME Function (cont.)

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.SUS"))
; 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: 10 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.SUS").

GET-INTERNAL-RUN-TIME Function

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.

GET-KEYBOARD-FUNCTION 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

Three values:

1. The function that is bound to the control character.
2. The function's argument list.
3. The function's interrupt level.

If a function is not bound to the specified control character, the function returns NIL for all three values.

Examples

1.
```
Lisp> (BIND-KEYBOARD-FUNCTION #\^B #'BREAK)
T
Lisp> (GET-KEYBOARD-FUNCTION #\^B)
#<Compiled Function BREAK #x261510> ;
NIL ;
1
```

   - 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 to which CTRL/B is bound, the function's argument list (which is NIL), and the function's interrupt level (which is 1).

2.
```
Lisp> (GET-KEYBOARD-FUNCTION #\^S)
NIL ;
NIL ;
NIL
```

   All three values returned are NIL, because CTRL/S is not bound to a function.

GET-PROCESS-INFORMATION Function

Returns information about a process. If the process is nonexistent, this function returns NIL. 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*.

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 4 lists the keywords that you can specify and the values they return.

Table 4: 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 has active asynchronous system traps (ASTs) for the process.
:AST-COUNT	An integer that specifies the remaining AST quota.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
:AST-ENABLED	A vector of four bits that specifies the number of access modes that has enabled ASTs for the process.
:AST-QUOTA	An integer that specifies the AST quota.
: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.
:CLI-TABLENAME	A string that specifies the file name of the current command language interpreter table.
:CPU-LIMIT	An integer that specifies the CPU time limit of the process in 10-millisecond units.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
: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.
: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-P0-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.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
: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.
: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.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
: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.
: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.

GET-PROCESS-INFORMATION Function (cont.)

Table 4 (cont.)

Keyword	Return Value
: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.

GET-PROCESS-INFORMATION Function (cont.)

2.
```
Lisp> (DEFUN PARENT NIL
        (LET ((PID
               (SECOND (GET-PROCESS-INFORMATION
                        NIL
                        :OWNER-PID))))
          (IF (ZEROP PID) NIL (ATTACH PID))))
PARENT
```

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 Function

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 DCL Dictionary*.

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 5 lists the keywords that you can specify and the values they return.

Table 5: 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.
:ESCAPE	Either T or NIL. The function returns T if ANSI standard escape sequences transmitted

GET-TERMINAL-MODES Function (cont.)

Table 5 (cont.)

Keyword	Return Value
	from the terminal are handled as a single multicharacter 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 Reference Manual: Part I*.
:HALF-DUPLEX	Either T or NIL. The function returns T if the terminal's operating mode is half-duplex, and the function returns NIL if the operating mode is full-duplex. For a description of terminal operating modes, see the *VAX/VMS I/O User's Reference Manual: Part I*.
: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.

NOTE

:PASS-ALL has been kept for the sake of compatibility with Version 1 of VAX LISP, but it is not recommended that you use :PASS-ALL.

:PASS-THROUGH	Either T or NIL. This mode is the same as the :PASS-ALL mode, except that "TTSYNC" protocol (CTRL/S, CTRL/Q) is still used.

GET-TERMINAL-MODES Function (cont.)

Table 5 (cont.)

Keyword	Return Value
: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 the function 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. Otherwise, the function returns NIL. 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 the list 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 :PASS-THROUGH NIL)
```

Returns a list of all the keyword-value pairs.

GET-VMS-MESSAGE Function

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 Reference 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 Function

Returns the rehash size of a hash table. The rehash size indicates how much a hash table is to increase when it is full. You specify that value 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 0 or a floating-point number greater than 1. 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 TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-REHASH-SIZE function returns the rehash size of the hash table, TABLE-1.

HASH-TABLE-REHASH-THRESHOLD Function

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. You specify that value 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 hash table whose rehash threshold is to be returned.

Return Value

An integer greater than 0 and less than hash table's rehash size or a floating-point number greater than 0 and less than 1.

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 TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-REHASH-THRESHOLD function returns the rehash threshold of the hash table, TABLE-1.

HASH-TABLE-SIZE Function

Returns the current size of a hash table. You specify that value 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 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)
233
```

- 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 TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-SIZE function returns the initial size of the hash table, TABLE-1.

HASH-TABLE-TEST Function

Returns a value or a symbol that indicates how a hash table's keys are compared. 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-TEST *hash-table*

Argument

hash-table

The hash table whose test value is to be returned.

Return Value

Either a function (#'EQ, #'EQL, or #'EQUAL) or a symbol (EQ, EQL, or EQUAL). EQL is the default when creating a 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-TEST TABLE-1)
EQUAL
```

- 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 TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-TEST function returns the test for the hash table, TABLE-1.

LOAD Function

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 :IF-DOES-NOT-EXIST :PRINT :VERBOSE

Arguments

filename

The name of the file to be loaded.

:IF-DOES-NOT-EXIST

Specifies whether the LOAD function signals an error if the file does not exist. The value can be 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

Specifies whether the value of each form that is loaded is printed to the stream bound to the *STANDARD-OUTPUT* variable. The value can be T or NIL. If you specify T, the value of each form in the file is printed to the stream. If you specify NIL, no action is taken. The default value is NIL. This keyword is useful for debugging.

:VERBOSE

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 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 T.

Return Value

A value other than NIL if the load operation is successful.

LOAD Function (cont.)

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
0 Errors, 0 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 Function

Translates the logical name LISP$LONG_SITE_NAME. If the first character of the resulting string is an at sign (@), the rest 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"
```

MACHINE-INSTANCE Function

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"
```

MACHINE-VERSION Function

Returns the content of the system identification (SID) register as a string that represents the version of computer hardware on which the VAX LISPsystem 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"
```

MAKE-ARRAY Function

Creates and returns an array. VAX LISP has added the :ALLOCATION keyword to this COMMON LISP function. When the function is used with the :ALLOCATION keyword and the value :STATIC, the function creates a statically allocated array.

During system usage, the garbage collector moves LISP objects. You can prevent the garbage collector from moving an object by allocating it in static space. Arrays, vectors, and strings can be statically allocated if you use the :ALLOCATION keyword and :STATIC value in a call to the MAKE-ARRAY function. Once an object is statically allocated, its virtual address does not change. Note that such objects ar never garbage collected and their space cannot be reclaimed. By default, LISP objects are allocated in dynamic space.

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 other keywords that can be used. See *COMMON LISP: The Language* for information on the other MAKE-ARRAY keywords.

VAX LISP creates a specialized array when the array's element type is STRING-CHAR, (SIGNED-BYTE 32), or a subtype of FLOAT or (UNSIGNED BYTE 1-29). For all other element types, VAX LISP creates a generalized array, with the element type T. For compatibility of VAX types with LISP types when calling external routines, see the tables on data conversion in the call-out chapter of the *VAX LISP/VMS System Access Programming Guide*.

Format

```
MAKE-ARRAY dimensions
     &KEY :ALLOCATION other-keywords
```

Arguments

dimensions

A list of positive integers that are to be the dimensions of the array.

MAKE-ARRAY Function (cont.)

:ALLOCATION

Specifies whether the LISP object is to be statically allocated. You can specify one of the following values with the :ALLOCATION keyword:

:DYNAMIC	The LISP object is not to be statically allocated. This value is the default.
:STATIC	The LISP object is to be statically allocated.

other-keywords

See *COMMON LISP: The Language*.

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. The :ELEMENT-TYPE keyword is one of the other keywords (described in *COMMON LISP: The Language*) that this function accepts.

MODULE-DIRECTORY Variable

A variable whose value refers to the directory containing the module that is being loaded into the LISP environment due to a call to the REQUIRE function. The value is a pathname.

This variable is useful to determine the location of a module if additional files from the same directory must be loaded by the module. For example, consider the following contents of a file called REQUIREDFILE1.LSP:

```
(PROVIDE "REQUIREDFILE1")
(LOAD (MERGE-PATHNAMES "REQUIREDFILE2" *MODULE-DIRECTORY*))
(DEFUN TEST
  ...)
```

When you specify the preceding module with the REQUIRE function, you do not have to identify the module's directory if it is in one of the places the REQUIRE function searches (see the description of the REQUIRE function later in Part II). Furthermore, using the *MODULE-DIRECTORY* variable as in this example ensures that the file REQUIREDFILE2 will be loaded from the same directory. After the module is loaded, the *MODULE-DIRECTORY* variable is rebound to NIL.

NOTE

As this variable is bound during calls to the REQUIRE function, nested calls to the function cause its value to be updated appropriately.

POST-GC-MESSAGE Variable

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.
```

System messages differ according to the cause of the garbage collection. If you set the *POST-GC-MESSAGE* variable, the message you establish supersedes all system messages displayed after a garbage collection, regardless of cause.

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-DEFINITION Function

Pretty prints to a stream the function definition of a symbol.

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
```

PPRINT-DEFINITION Function (cont.)

```
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-PLIST Function

Pretty-prints to a stream the property list of a symbol. 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.

PPRINT-PLIST prints only the symbol-value pairs for which a symbol is accessible in the current package. (For information on packages, see *COMMON LISP: The Language*.) On the other hand, SYMBOL-PLIST returns all the symbol-value pairs (the entire property list) of a symbol, even those not accessible in the current package. So, the form (PPRINT-PLIST 'ME) is not equivalent to the form (PPRINT (SYMBOL-PLIST 'ME). The following example shows the differences between the two forms:

```
Lisp> (MAKE-PACKAGE 'PLANET)
Lisp> (SETF (SYMBOL-PLIST 'ME)
            '(GIRL "SAMANTHA" BOY "DANIEL"
              PLANET::INHABITANT-OF "EARTH"))
(GIRL "SAMANTHA" BOY "DANIEL" PLANET::INHABITANT-OF "EARTH")
Lisp> (PPRINT (SYMBOL-PLIST 'ME))
(GIRL "SAMANTHA" BOY "DANIEL" PLANET::INHABITANT-OF "EARTH")
Lisp> (PPRINT-PLIST 'ME)
(GIRL "SAMANTHA"
 BOY "DANIEL")
```

The form (PPRINT (SYMBOL-PLIST 'ME)) prints the symbol-value pair PLANET::INHABITANT-OF "EARTH", but the form (PPRINT-PLIST 'ME) does not print that pair. This is because the symbol INHABITANT-OF in the package PLANET is not accessible in the current package (a symbol can be in another package but still be accessible in the current package). The symbol ME in the current package is associated with the symbol-value pair INHABITANT-OF "EARTH" in the PLANET package, but the PPRINT-PLIST function does not print that symbol-value pair because it is not accessible in the current package.

Format

PPRINT-PLIST *symbol* &OPTIONAL *stream*

Arguments

symbol

The symbol whose property list is to be pretty-printed.

PPRINT-PLIST Function (cont.)

stream

The stream to which the pretty-printed output is to be sent. 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)
```

PPRINT-PLIST Function (cont.)

- 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.

PRE-GC-MESSAGE Variable

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.
```

VAX LISP messages preceding garbage collection differ depending on the cause of the garbage collection. If you set the *PRE-GC-MESSAGE* variable, the message you establish supersedes all system messages, regardless of cause.

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-LINES Variable

Specifies the number of lines to be printed by an outermost logical block. The default for this variable is NIL, which specifies no abbreviation. *PRINT-LINES* is effective only when pretty printing is enabled. When the system limits output to the number of lines specified by *PRINT-LINES*, it indicates abbreviation by replacing the last four characters on the last line printed with " ...".

The WRITE and WRITE-TO-STRING functions have been extended in VAX LISP to accept the :LINES keyword. If you specify this keyword, *PRINT-LINES* is bound to the value you supply with the keyword before any output is produced.

See Chapter 6 for more information on using the *PRINT-LINES* variable.

Example

```
Lisp> (SETF *PRINT-LINES* 4)
4
Lisp> (FORMAT T "Stars: ~:!~/LINEAR/~."
'(POLARIS DUBHE MIRA MIRFAK BELLATRIX CAPELLA ALGOL
MIRZAM POLLUX CANOPUS ALBIREO CASTOR ALPHECCA
ANTARES))
Stars: POLARIS
       DUBHE
       MIRA
       MI ...
```

- With *PRINT-LINES* set to 4, printing stops at the end of the fourth line.
- The last four characters of the last line are not printed. MIRFAK becomes MI.

PRINT-MISER-WIDTH Variable

Controls miser mode printing. If the available line width between the indentation of the current logical block and the end of the line is less than the value of this variable, the pretty printer enables miser mode. When output is printed in miser mode, all indentations are ignored. In addition, a new line is started for every conditional new line directive (~_, ~:_, ~@_). The default value for *PRINT-MISER-WIDTH* is 40.

You can prevent the use of miser mode by setting the *PRINT-MISER-WIDTH* variable to NIL.

The WRITE and WRITE-TO-STRING functions have been extended in VAX LISP to accept the :MISER-WIDTH keyword. If you specify this keyword, *PRINT-MISER-WIDTH* is bound to the value you supply with the keyword before any output is produced.

For more information about miser mode and the use of the *PRINT-MISER-WIDTH* variable, see Sections 6.5 and 6.8.

Example

```
Lisp> (SETF *PRINT-RIGHT-MARGIN* 60)
60
Lisp> (SETF *PRINT-MISER-WIDTH* 35)
35
Lisp> (FORMAT T "~!Stars with Arabic names: ~:@!~S ~:_~S ~
                 ~27I~:_~S ~:I~@_~S ~_~S ~1I~_~S~.~."
          '(BETELGEUSE (DENEB SIRIUS VEGA)
            ALDEBERAN ALGOL (CASTOR POLLUX) BELLATRIX)
Stars with Arabic names: BETELGEUSE
                         (DENEB SIRIUS VEGA)
                         ALDEBERAN
                         ALGOL
                         (CASTOR POLLUX)
                         BELLATRIX
```

- The text, "Stars with Arabic names:", in the outer logical block causes the inner logical block to begin at column 26. With *PRINT-MISER-WIDTH* set to 35, FORMAT enables miser mode when the logical block begins past column 25.

- FORMAT conserves space by starting a new line at every multiline mode new line directive (~_) and every if-needed new line directive (~:_).

- FORMAT starts a new line at the miser mode new line directive (~@_) and ignores the indentation directives (~*n*I).

PRINT-RIGHT-MARGIN Variable

Specifies the right margin for pretty printing. Output may exceed this margin if you print long symbol names or strings, or if your FORMAT control string specifies no new line directives of any type. If the value of *PRINT-RIGHT-MARGIN* is NIL, the print function uses a value appropriate to the output device.

The WRITE and WRITE-TO-STRING functions have been extended in VAX LISP to accept the :RIGHT-MARGIN keyword. If you specify this keyword, *PRINT-RIGHT-MARGIN* is bound to the value you supply with the keyword before any output is produced.

See Chapter 6 for more information about using the *PRINT-RIGHT-MARGIN* variable.

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 *PRINT-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.

PRINT-RIGHT-MARGIN Variable (cont.)

- The call to the SETF macro sets the value of the *PRINT-RIGHT-MARGIN* variable to 40.
- The call to the PPRINT function shows the effect the variable's value has on the pretty-printed output. PPRINT-DEFINITION stops printing each line before reaching column 40.

PRINT-SIGNALED-ERROR Function

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 Chapter 4.

You can include a call to this function in an error handler that you create (see Chapter 4).

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)
```

PRINT-SIGNALED-ERROR Function (cont.)

```
                (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 Variable

Determines how the slot names of a structure are formatted when they are displayed. The value can be 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.

REQUIRE Function

Examines the *MODULES* variable to determine if a specified module has been loaded. 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 the LISP environment. This search forces the function to locate a module you have created before the function 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 to which the logical name LISP$SYSTEM refers 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 earlier in Part II.

The REQUIRE function checks the *MODULES* variable to determine if a module has already been loaded. However, the REQUIRE function, when loading a module, does not update the *MODULES* variable to indicate that the module has been loaded. The PROVIDE function (described in *COMMON LISP: The Language*) does update the *MODULES* variable. Use the PROVIDE function in a file containing a module to be loaded to indicate to the LISP system that the file contains a module of this name.

If the loaded file does not contain a corresponding PROVIDE, a subsequent REQUIRE of the module will cause the file to be reloaded.

REQUIRE Function (cont.)

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 evaluation of 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 evaluation of the *MODULES* variable indicated that the module was not loaded; therefore, the function loaded the module RELATIVE.
- The second evaluation of the *MODULES* variable shows that the module RELATIVE was loaded.

ROOM Function

Displays information about LISP memory. Information is displayed for the following memory spaces:

- Read-only space
- Static space
- Dynamic space

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: 4352, Current Allocation: 4113, Free: 5%
```

All counts are in 512-byte pages.

Format

ROOM &OPTIONAL *value*

Argument

value

Optional argument whose value can be 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: 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
```

Table 6 lists the headings and VAX LISP data types the ROOM function displays for each type of storage space.

ROOM Function (cont.)

Return Value

No value.

Table 6: 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)
S Flo Vecs	Simple float vectors
D Flo Vecs	Simple double float vectors
L Flo Vecs	Simple long float vectors
L Wrd Vecs	Simple longword vectors
Bignums	Bignums
Sngl Flos	Single float numbers
Dbl Flos	Double float numbers
Long Flos	Long float numbers
Ratios	Ratios
Complexes	Complex numbers
Symbols	Symbols
Conses	Conses
Ctrl Stack	Control Stack
Bind Stack	Binding Stack

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.

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 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 :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*.

SET-TERMINAL-MODES Function (cont.)

:HALF-DUPLEX

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 Reference Manual: Part I*.

:PASS-ALL

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.

NOTE

:PASS-ALL has been kept for compatibility with with Version 1 of VAX LISP, but it is not recommended that you use :PASS-ALL.

:PASS-THROUGH

Specifies whether the terminal is in pass-through mode. The value can be either T or NIL. This mode is the same as the :PASS-ALL mode, except that "TTSYNC" protocol (CTRL/S, CTRL/Q) is still used.

:TYPE-AHEAD

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.

:WRAP

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.

SET-TERMINAL-MODES Function (cont.)

Return Value

Undefined.

Example

```
Lisp> (DEFVAR *OLD-TERMINAL-STATE*)
*OLD-TERMINAL-STATE*
Lisp> (DEFUN PASS-THROUGH-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-THROUGH-HANDLER
Lisp> (DEFUN UNUSUAL-INPUT NIL
        (LET ((*OLD-TERMINAL-STATE* (GET-TERMINAL-MODES))
              (*UNIVERSAL-ERROR-HANDLER* #'PASS-THROUGH-HANDLER))
          (UNWIND-PROTECT (PROGN
                            (SET-TERMINAL-MODES
                            :PASS-THROUGH
                            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-THROUGH-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-THROUGH-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.

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"
```

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.

SPAWN Function (cont.)

:LOGICAL-NAMES

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

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

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

Specifies the name of the subprocess to be created. If you omit this keyword, the system generates a unique name.

Return Value

Undefined.

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.

SPAWN Function (cont.)

3.
```
Lisp> (DEFUN SPAWN-IN-WINDOW
                (&OPTIONAL (PROCESS-NAME NIL))
          (LET ((DEVICE-STRING
                    (UIS:CREATE-TERMINAL
                            :BANNER-TITLE
                            (OR PROCESS-NAME "Subprocess"))))
            (SPAWN :INPUT-FILE DEVICE-STRING
                   :OUTPUT-FILE DEVICE-STRING
                   :PROCESS-NAME PROCESS-NAME
                   :PARALLEL T)))
SPAWN-IN-WINDOW
Lisp> (SPAWN-IN-WINDOW "Smith_1")
Lisp>
```

This example works only on a VAXstation. It defines a function named SPAWN-IN-WINDOW that creates a process in VAXstation terminal emulator window. The function UIS:CREATE-TERMINAL creates a terminal emulator window and returns the window's device name. By supplying this return value with the :INPUT-FILE and :OUTPUT-FILE keyword arguments, SPAWN-IN-WINDOW arranges for input to and output from the subprocess to be directed through the terminal emulator window. SPAWN-IN-WINDOW accepts an optional argument that becomes the name of the subprocess and the title of the window.

When the SPAWN-IN-WINDOW function is called, a subprocess and a terminal emulator window named "Smith_1" are created. The cursor switches to the terminal emulator window. However, the user can switch the cursor back to the LISP prompt and continue to use LISP without logging out of the subprocess.

See the *VAX LISP/VMS Graphics Programming Guide* for information about the UIS:CREATE-TERMINAL function.

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 1>
```

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*
(FUNCTION-X (- X 1))
Step> (EVALHOOK '(- x 1) NIL NIL *STEP-ENVIRONMENT*)
2
```

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: (+ FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))
Step> STEP
: : : : : : : : #38: (FUNCTION-X (- X 1))
Step> EVAL *STEP-FORM*
(FUNCTION-X (- 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 (FUNCTION-X (- 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. Any *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*

SUSPEND Function (cont.)

Argument

pathname

A pathname, namestring, or symbol that represents the file name to which the function writes the LISP-system state.

Return Value

T, when the LISP system is resumed at a later time and NIL, when execution continues after a suspend 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>
```

SUSPEND Function (cont.)

- 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 Function

Transfers control. This function exists only for compatibility with VAX LISP/VMS V1.x, in which it transferred control to a numbered command level. VAX LISP V2 does not have numbered command levels. In VAX LISP V2, THROW-TO-COMMAND-LEVEL either throws to the CANCEL-CHARACTER-TAG tag or does nothing.

Format

THROW-TO-COMMAND-LEVEL *level*

Argument

level

Either an integer or a keyword. Depending on the argument, THROW-TO-COMMAND-LEVEL takes the following action:

integer	No action
:CURRENT	Throw to CANCEL-CHARACTER-TAG
:PREVIOUS	No action
:TOP	Throw to CANCEL-CHARACTER-TAG

Return Value

Undefined.

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> (THROW-TO-COMMAND-LEVEL :TOP)
Lisp>
```

- The debugger is invoked, because an error was signaled when the FACTORIAL function was called.
- The call to the THROW-TO-COMMAND-LEVEL function returns control to the top-level loop.

TIME Macro

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: 0.03 sec., Real Time: 0.23 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: 0.03 sec., Real Time: 0.23 sec.
6
```

TOP-LEVEL-PROMPT Variable

Lets you change the top-level prompt. The value of this variable can be:

- 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> ".

TRACE Macro

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 7.

NOTE

The arguments specified in a call to the TRACE macro are not evaluated when the call to TRACE is executed. Some forms are evaluated repeatedly, as described below.

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 which enables tracing for that function(s) and/or macro(s).

  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 function or macro 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.

TRACE Macro (cont.)

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

Table 7 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 and the current dynamic environment.

Table 7: 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 Chapter 5).

TRACE Macro (cont.)

Table 7 (cont.)

Keyword-Value Pair	Description
: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 Chapter 5).
: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 Chapter 5).
: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 Chapter 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 function or macro specified by the TRACE function is traced only when it is called

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
Frame #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
Frame #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
Frame #53: (DEBUG)
Debug 1> CONTINUE

. . . . #52=> 0
```

TRACE Macro (cont.)

```
Control Stack Debugger
Frame #41: (DEBUG)
Debug 1> CONTINUE

. . . #40=> 1
Control Stack Debugger
Frame #29: (DEBUG)
Debug 1> CONTINUE

. . #28=> 2
Control Stack Debugger
Frame #17: (DEBUG)
Debug 1> CONTINUE

. #16=> 3
Control Stack Debugger
Frame #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.)

```
NIL
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)
#5: (FACTORIAL 2)
#9: (BLOCK FACTORIAL (IF (> 2 N) 1 (* N (FACTORIAL (1- N)))))
Step>
: #16: (IF (> 2 N) 1 (* N (FACTORIAL (1- N))))
Step>
: : #22: (> 2 N)
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 LISP-LENGTH. The LIST-LENGTH function is traced during the call to the PRINT-LENGTH function.

12.
```
Lisp> (DEFUN FUNCTION-X (X)
        (IF (< X 3) 1
            (+ (FUNCTION-X (- X 1)) (FUNCTION-X (- X 2)))))
FUNCTION-X

Lisp> (TRACE (FUNCTION-X
                :PRE-DEBUG-IF (< (SECOND *TRACE-CALL*) 2)
                :SUPPRESS-IF T))
(FUNCTION-X)
Lisp> (FUNCTION-X 5)
```

TRACE Macro (cont.)

```
Control Stack Debugger
Frame #26: (DEBUG)
Debug 1> DOWN
Frame #21: (BLOCK FUNCTION-X
              (IF (< X 3) 1
                  (+ (FUNCTION-X (- X 1))
                     (FUNCTION-X (- X 2)))))
Debug 1> DOWN
Frame #19: (FUNCTION-X 3)
Debug 1> (CADR (DEBUG-CALL))
3
Debug 1> CONTINUE
Control Stack Debugger
Frame #19: (DEBUG)
Debug 1> CONTINUE
5
```

- In this example, FUNCTION-X is first defined.
- Then the TRACE macro is called for FUNCTION-X. TRACE is specified to invoke the debugger if the first argument to FUNCTION-X (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 FUNCTION-X. As the :SUPPRESS-IF option has a value of T, calls to FUNCTION-X 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 FUNCTION-X.

13.
```
Lisp> (TRACE (FUNCTION-X
                :POST-DEBUG-IF (> (FIRST *TRACE-VALUES*) 2)))
(FUNCTION-X)
Lisp> (FUNCTION-X 5)
#4: (FUNCTION-X 5)
. #11: (FUNCTION-X 4)
. . #18: (FUNCTION-X 3)
. . . #25: (FUNCTION-X 2)
. . . #25=> 1
. . . #25: (FUNCTION-X 1)
. . . #25=> 1
. . #18=> 2
```

TRACE Macro (cont.)

```
. . #18: (FUNCTION-X 2)
. . #18=> 1
Control Stack Debugger
Frame #12: (DEBUG)
Debug 1> BACKTRACE
-- Backtrace start --
Frame #12: (DEBUG)
Frame #7: (BLOCK FUNCTION-X
             (IF (< X 3) 1
                 (+ (FUNCTION-X (- X 1))
                    (FUNCTION-X (- X 2)))))
Frame #5: (FUNCTION-X 5)
Frame #1: (EVAL (FUNCTION-X 5))
-- Backtrace ends --
Frame #12: (DEBUG)
Debug 1> CONTINUE
. #11=> 3
. #11: (FUNCTION-X 3)
. . #18: (FUNCTION-X 2)
. . #18=> 1
. . #18: (FUNCTION-X 1)
. . #18=> 1
. #11=> 2
Control Stack Debugger
Frame #5: (DEBUG)
Debug 1> CONTINUE
#4=> 5
```

TRACE is called for FUNCTION-X (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 (FUNCTION-X
                :SUPPRESS-IF (> (SECOND *TRACE-CALL*) 1)))
```

   This causes FUNCTION-X to be traced only if its first argument is 1 or less

2.
```
Lisp> (TRACE (FUNCTION-X
                :SUPPRESS-IF (<= (LENGTH *TRACE-CALL*) 2)))
```

   This causes FUNCTION-X to be traced if it is called with more than 1 argument.

3.
```
Lisp> (TRACE (FUNCTION-X
                :PREDEBUG-IF (< (SECOND *TRACE-CALL*) 2)
                :SUPPRESS-IF (< (SECOND *TRACE-CALL*) 2)))
FUNCTION-X
```

   In this case, the TRACE macro is enabled for FUNCTION-X. The debugger will be invoked and tracing suppressed if the first argument to FUNCTION-X (the SECOND of the value of the *TRACE-CALL* variable) is less than 2. So for example, if FUNCTION-X is called with the arguments 3 and 5, *TRACE-CALL* is bound to the form (FUNCTION-X 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.

TRANSLATE-LOGICAL-NAME Function

Searches a logical name table for a logical name, translates it, and returns it as a list of strings.

The TRANSLATE-LOGICAL-NAME function performs only one level of logical-name translation.

This function is equivalent to the DCL SHOW LOGICAL command. For additional information about the SHOW LOGICAL command or about using logical names, see the *VAX/VMS DCL Dictionary*.

Format

TRANSLATE-LOGICAL-NAME *string* &KEY :TABLE :CASE-SENSITIVE

Arguments

string

The logical name for which the function is to search.

:TABLE

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 in that order. The values you can specify with the :TABLE keyword are the following:

:PROCESS	Process name table (LNM$PROCESS_TABLE)
:GROUP	Group name table (LNM$GROUP)
:SYSTEM	System name table (LNM$SYSTEM_TABLE)
:ALL	Search all three tables (LNM$DCL_LOGICAL) (the default)

:CASE-SENSITIVE

Used to restrict the search to a case-sensitive search. Valid values are T (for case-sensitive search) or NIL (for case-insensitive search). The default is NIL. Use a value of T if you have multiple logical names that differ only in case.

Return Value

If the logical name has any translations, they are returned as a list of strings. If is match found, NIL is returned.

TRANSLATE-LOGICAL-NAME Function (cont.)

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 the function displays.

UNBIND-KEYBOARD-FUNCTION Function

Removes the binding of a function from a control character.

Format

UNBIND-KEYBOARD-FUNCTION *control-character*

Argument

control-character

The control character from which a function's binding is to be removed.

Return Value

T, if a binding is removed. NIL, if the control character is not bound to a function.

Example

```
Lisp> (BIND-KEYBOARD-FUNCTION #\^B #'BREAK)
T
Lisp> (UNBIND-KEYBOARD-FUNCTION #\^B)
T
```

- The call to the BIND-KEYBOARD-FUNCTION function binds CTRL/B to the BREAK function.
- The call to the UNBIND-KEYBOARD-FUNCTION function removes the binding of the function that is bound to CTRL/B.

UNCOMPILE Function

Restores the interpreted function definition of a symbol, if the symbol's definition was compiled with a call to the COMPILE function.

The UNCOMPILE function is useful for editing function definitions and debugging. For example, if you are not satisfied with the results of a function compilation, you can uncompile the function, edit it, and then recompile it.

NOTE

You cannot uncompile system functions and macros or functions and macros that were loaded from files that were compiled by the COMPILE-FILE function or the DCL /COMPILE qualifier of the LISP command.

Format

UNCOMPILE *symbol*

Argument

symbol

The symbol that represents the function that is to be uncompiled.

Return Value

The name of the function, if the specifed symbol represents an existing compiled lambda expression and has an interpreted definition; NIL, if it does not.

Example

```
Lisp> (DEFUN ADD2 (FIRST SECOND) (+ FIRST SECOND))
ADD2
Lisp> (COMPILE 'ADD2)
ADD2 compiled.
ADD2
Lisp> (UNCOMPILE 'ADD2)
ADD2
```

- The call to the DEFUN macro defines the function ADD2.
- The call to the COMPILE function compiles the function ADD2.
- The call to the UNCOMPILE function successfully restores the interpreted definition of the function ADD2, because the function is defined and was compiled with the COMPILE function.

UNDEFINE-LIST-PRINT-FUNCTION Macro

Disables the list-print function defined for a symbol. If another list-print function was superseded by the list-print function undefined, the older function is reenabled. Otherwise, no other list-print function exists for the given symbol.

See Chapter 6 for more information about list-print functions.

Format

UNDEFINE-LIST-PRINT-FUNCTION *symbol*

Argument

symbol

The name of the list-print function to be disabled.

Return Value

The name of the list-print function that has been disabled.

Example

```
Lisp> (UNDEFINE-LIST-PRINT-FUNCTION MY-SETQ)
MY-SETQ
```

Undefines the list-print function named MY-SETQ.

UNIVERSAL-ERROR-HANDLER Function

The function to which the VAX LISP system sends all errors that are signaled during program execution. By default, the VAX LISP *UNIVERSAL-ERROR-HANDLER* variable is bound to this function.

The VAX LISP error handler is described in Chapter 4.

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 Chapter 4.

UNIVERSAL-ERROR-HANDLER Variable

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 special form LET binds the *UNIVERSAL-ERROR-HANDLER* variable to the error handler named CRITICAL-ERROR-HANDLER, while the PERFORM-CRITICAL-OPERATION function is evaluated.

WARN Function

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 WARN functions returns 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 Chapter 4.

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.

Example

```
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.

WITH-GENERALIZED-PRINT-FUNCTION Macro

Locally enables a generalized print function when it evaluates the specified forms. See Chapter 6 for more information about using generalized print functions.

Format

WITH-GENERALIZED-PRINT-FUNCTION *name* &BODY *forms*

Arguments

name

A symbol identifying the generalized print function to be enabled. The enabled generalized print function supersedes any previously enabled generalized print function for *name*.

forms

A call or calls to print functions.

Return Value

Output generated by the call or calls to print functions.

Example

```
Lisp> (DEFINE-GENERALIZED-PRINT-FUNCTION PRINT-NIL-AS-LIST
                (OBJECT STREAM)
                (NULL OBJECT)
          (PRINC "( )" STREAM))
PRINT-NIL-AS-LIST
Lisp> (WITH-GENERALIZED-PRINT-FUNCTION 'PRINT-NIL-AS-LIST
        (PPRINT NIL))
( )
```

The PPRINT call prints ( ), because the generalized print function is enabled locally and pretty printing is enabled.

APPENDIXES

APPENDIX A

PERFORMANCE HINTS

LISP code normally does much type checking at runtime. You can reduce execution time and amount of memory required by using data structures more efficiently and by using certain programming and debugging techniques.

This appendix lists what you can do to optimize the speed of execution of your LISP code and the amount of memory required. The sections also give the following information:

- Number of instructions executed by certain functions
- Relative speed of certain functions compared with others that can be used to achieve the same result
- Explanations of why certain functions and operations require so much time and memory
- Data structure representation

This information can help you choose the most efficient way to code a program.

Some VAX instructions are mentioned in this appendix. Refer to the *VAX Architecture Handbook* for more information on the VAX instruction set.

A.1 DATA STRUCTURES

This section describes how to optimize the use of data structures in your code.

A.1.1 Integers

Fixnum arithmetic is much faster than bignum arithmetic. Therefore, if possible use numbers in the range -2**29 to 2**29-1. (The range of integers represented as fixnums in future versions is likely to be cut in half: -2**28 to 2**28-1. Keep this in mind when placing fixnum declarations in your programs.) You must use fixnum declarations for each argument to an arithmetic function and for the result as well to generate fixnum-only in-line VAX instructions. The result must be declared to be type fixnum, and even though all input values for an arithmetic function may be fixnums, the result may not be. (That is, fixnums are not closed under arithmetic operations).

When fixnum declarations are used, fixnum arithmetic takes two instructions for each addition or subtraction operation (except incrementing and decrementing, which require one instruction each) and four instructions for each multiplication and division operation. Fixnum comparisons consist of a CMPL instruction and the appropriate branch; the result's type need not be declared.

Fixnums are never allocated (they are immediate: they are always manipulated directly, rather than through pointers). Therefore, fixnum arithmetic requires less memory and less time for garbage collection than arithmetic with bignums.

Bignums require two longwords for a header and enough space to represent the number in two's complement format. Therefore, working with bignums consumes much more time than working with fixnums. For example, to print 1000 factorial takes much longer than to compute it. Much more garbage is produced while calculating the print representation than in calculating the result.

A.1.2 Floating-Point Numbers

When using floating-point arithmetic, the system allocates new space for the results. In-line code is generated only when both arguments to an arithmetic function are declared to be of the same floating-point type. In-line conversions (CVTxx) are not done. The VMS math library routines are used for complicated functions, such as trigonometric functions.

Floating-point numbers always have a 1-longword header.

A.1.3 Ratios

When working with ratios, the system calls the GCD function after each ratio is created, and stores the ratio in canonical form. Use the TRUNCATE or REM function when you do not need exact answers or when

you want a remainder. The TRUNCATE function executes faster if you can declare the result to be a fixnum. The TRUNCATE and REM functions are faster than the FLOOR and MOD functions. These in turn are faster than the ROUND function.

Ratios occupy two longwords; they do not have headers.

A.1.4 Characters

When representing characters, it is usually not necessary to specify bit and font attributes. String characters utilize an 8-bit code that is compatible with the ASCII and DIGITAL multinational standards, and with the VAX architecture.

The CHAR= function used without type checking is the same as the EQ function. The CHAR<, CHAR<=, CHAR>, and CHAR>= functions generate the same code as the fixnum comparisons when no type checking is required because declarations were used. This code consists of a CMPL instruction followed by the appropriate branch. Like fixnums, characters are never allocated (they are immediate), thereby requiring less memory and less time for garbage collection.

A.1.5 Symbols

Symbols let you easily associate data with a name. Symbols are interned when read by the READ function, and remain interned until they are uninterned from all packages using them. So, when you create anonymous variables and functions, use uninterned symbols (created using the MAKE-SYMBOL or GENSYM function).

For VAX LISP, accessing a dynamic variable may require several instructions, depending on the declarations and optimizations used. Normally, accessing a dynamic variable is slower than accessing local variables but faster than accessing closed-over lexical variables. A local variable can be accessed quickly because it is stored on the stack. A closed-over variable is stored in a vector and passed to other functions that use them. Therefore, to access a closed-over variable may require several instructions. To reduce the overhead of dynamic variable access to one instruction, set the optimization declaration SPEED to 3 and SAFETY to 0, eliminating unbound variable checking, and thus reducing execution time.

When a special variable is bound to a new value, LISP saves the symbol and its old value on the binding stack and stores the new value in the value cell of the symbol. This requires either four or five instructions. Unbinding a special variable requires one instruction. Accessing the parts of a symbol, such as its name, property list, package, and value, requires only one instruction each, if you have

used the appropriate declarations to declare the variable as a symbol. However, setting a symbol's function cell is very slow.

Symbols occupy five longwords each.

A.1.6 Lists and Vectors

Use lists when the number of elements changes often. Typically, you push elements onto and pop elements off the front of the list to simulate a stack. Conses are convenient for creating tree structures, especially when you need values only at the leaves. If you must access many values at each internal node of a tree, use structures rather than lists. Conses require two longwords.

Use vectors when you must access elements often at any position. Vectors use half as much space as lists, and can cause less paging when accessed because vector elements are stored in adjacent memory locations. A simple-vector has a single-longword header.

Use the noncopying (or destructive) versions of the sequence and list functions whenever possible. For example, the NCONC function is faster than the APPEND function and the NSTRING-UPCASE function is faster than the STRING-UPCASE function. You can use the form (NREVERSE (THE LIST x)) rather than the copying version (the REVERSE function) to get elements back to their original order if you are just gathering the results in a list. To copy input lists or strings once and then do destructive operations is more efficient than to always use copying versions of functions.

Copying vectors by using the COERCE or SUBSEQ function results in simple vectors (of the type SIMPLE-VECTOR, SIMPLE-STRING, SIMPLE-BIT-VECTOR, or SIMPLE-ARRAY) which can be manipulated by simpler, faster operations. Therefore, you can copy a vector to manipulate it quickly thereafter. However, to avoid numerous garbage collections, do not use copying versions of functions unless you must.

NOTE

Use destructive versions of functions with care, as shared data may be modified.

CAR, CDR, and the other list-manipulating functions by default always check their arguments to make sure they are lists and not atoms. To increase the speed of list-intensive applications, properly declare all lists and use the optimization declaration SPEED = 2 or use SPEED = 3 and SAFETY = 0. The CAR, CDR, RPLACA, and RPLACD functions each require one instruction when used with these declarations.

If you frequently splice or concatenate lists, use a pointer to the middle or end of the list. This is faster than using the NTHCDR, MEMBER, APPEND, and NCONC functions on the entire list, as they always process from the beginning of the list. The fastest (and default) tests for the MEMBER, ASSOC, and RASSOC functions are EQ and EQL.

Use property lists when you want values for keys to be global in scope. Do not use property lists if the number of keys is fairly constant and known in advance. Instead, use structures and include a slot in the structure for a list to be used like a property list for the keys that change.

Use association lists when you want values for keys to be dynamic in scope, since pushing entries onto the front of an association list shadows later entries. You can use dynamic variables as pointers into association lists to help you recall additions to the lists.

A.1.7 Strings, General Vectors, and Bit Vectors

Simple-vectors are processed faster than nonsimple vectors (vectors with fill pointers, adjustable vectors, or displaced vectors). Simple-vectors take less space since they do not have separate array headers and they are created faster.

Avoid using lists of characters when manipulating symbol names (that is, never implement EXPLODE or IMPLODE). Strings are fully supported in this language, unlike in older versions of LISP. Some common operations on simple strings use the VAX character instructions.

Many data structures that used to be implemented with lists can be more efficiently implemented with simple-vectors (the default DEFSTRUCT representation). If the domain of a set is fixed and set operations are frequent, using simple bit vectors is much faster than using lists. Accessing or updating slots of a declared structure takes only one instruction given the appropriate declarations. Accessing or updating characters in a simple string or bits in a simple bit vector is slower than accessing or updating elements of a simple-vector; when accessing or updating characters in a simple string or bits in a simple bit vector, data must be converted between the internal representation and the LISP representation. For both characters and fixnums, this involves at least an ASHL instruction. However, there are specialized routines for handling simple strings and simple bit vectors (for example, the STRING-UPCASE and BIT-AND functions with the proper declarations).

These representations take less space than simple vectors that hold characters or bits.

A.1.8 Hash Tables

Hash tables provide a good way of storing and accessing arbitrary objects. Although some overhead is required for each access or store, the total time required is usually reasonable even for large numbers of objects. VAX LISP hash tables use chains to resolve collisions.

You can access hash tables that use the EQ and EQL functions faster than hash tables that use the EQUAL function, because the comparisons are faster. However, hash tables that use the EQ and EQL functions must be completely rehashed after each garbage collection. Hash tables are preferable to lists and bit vectors for representing sets, when the number of objects may be large and extremely variable.

A.1.9 Functions

Compiled code is faster than interpreted code; when interpreted code is evaluated, much consing occurs.

Closures are slower than regular functions.

You can compile single functions at any time without using files. For example, to compile a function you have just defined, you can use (COMPILE 'FUNCTION-NAME) or (COMPILE NIL `(LAMBDA () ,...) if you want to create anonymous code to be stored and executed later. You can use the FUNCTION or FTYPE type specifier in a declaration or proclamation to inform the compiler about the types of the arguments and the return type of a function.

A.2 DECLARATIONS

This section describes how to use declarations to optimize LISP code.

By default, most standard VAX LISP functions check their arguments for type and other attributes. The compiler can generate much faster code for many simple operations by assuming the arguments are of the correct type. Therefore, use declarations to supply this information.

Whether the compiler takes advantage of declarations, and to what extent it does, is controlled by the OPTIMIZE declaration. Depending on the values of the optimization qualities, different code may be generated, given the presence of type declarations or the assumption of such type declarations.

NOTE

Currently, the COMPILATION-SPEED quality is ignored.

The format for using the OPTIMIZE declaration and its qualities with the PROCLAIM and DECLARE functions is as follows:

```
(PROCLAIM '(OPTIMIZE (SPEED x) (SAFETY y) (SPACE z)))
```

or

```
(DECLARE (OPTIMIZE (SPEED x) (SAFETY y) (SPACE z)))
```

The possible switch values are:

x=1,y=1,z=1 (the default)

No particular optimizations done. Generally, type checking will be done on all arguments to LISP functions.

x=2,y<2

Observes user supplied declarations. Useful when some variables are guaranteed to be of the declared type and speed is desired, but when not all variables (such as function arguments) can be guaranteed to be correct. Some macros (such as DOTIMES and DOLIST) expand into code with these declarations already supplied.

x>1,y=0

Skips bounds checking for vector and array references.

x=3,y=0

Assumes correct argument types to many functions, such as CAR, SYMBOL-NAME, and SCHAR. Useful for guaranteed correct and debugged functions. Special variable references do not check for unbound values.

x>y

Does tail recursion removal, if it can.

y=3

The THE function generates tests for objects being the specified type. Useful for fixnum declarations to detect overflows into bignums.

x>z

Tries to open-code some sequence functions. Observes in-line declarations.

Explicit type checking code, such as (IF (CONSP X) ...), is always executed regardless of a type declaration for X and the optimization settings. Therefore, you can retain type checking and still increase the speed of execution by using declarations. In the following example, faster code is generated for incrementing X by using the appropriate optimization settings without having to rebind X. Meanwhile, type checking is retained at the start of the function by using the explicit type checking code (IF (FIXNUMP X).

```
(DEFUN FOO (X)
  (DECLARE (FIXNUM X))
   (IF (FIXNUMP X)
       (LET ... (INCF X) ...)
       (ERROR ...)))
```

Another function that always executes is COERCE, since it is assumed that a type check will be executed, even if no coercion needs to be done.

Use fixnum and floating-point declarations for fast arithmetic. The compiler needs to know the types of all the arguments (and for fixnums, the result type, too) before it can generate the fast, type-specific code available on a VAX. Floating-point operations with operands (and therefore results) of the same type can also generate fast code.

Use simple-vector and similar array declarations for fast sequence and array operations. Declaring structures is equally helpful.

The PROCLAIM and DECLARE functions are used to declare a function's arguments and results whenever the function is called. For example, when the proclamation (PROCLAIM '(FTYPE (FUNCTION (FIXNUM) SINGLE-FLOAT) MYFUNCTION)) is used, each time MYFUNCTION is called the arguments are automatically declared to be fixnums and its result is automatically declared to be a single-float. An FTYPE declaration does not automatically provide declaration of the LAMBDA-LIST variable in the function definition.

It is important to provide type declarations, especially for the SIMPLE-VECTOR, SIMPLE-STRING, and SIMPLE-BIT-VECTOR types, for the arguments to sequence functions. The compiler can generate fast code for many common cases such as calls without any keyword arguments.

Multidimensional array operations also need declarations. Unlike the vector operations, multidimensional arrays need the actual (fixnum) bounds for each dimension at compile-time, to generate efficient array indexing code. In these cases it is helpful to use the DEFTYPE macro or a macro that expands into a call to the DECLARE function.

The functions defined in the following examples will be compiled with either (1) type-checking code if SPEED is less than 2, or (2) non-type-checking code if SPEED equals 3 and SAFETY equals 0. However, the second example produces code that does not check the type of X but does check the type of (CDR X), when SPEED equals 2 and SAFETY is less than 2. This is because there is a declaration allowing the optimization of the CDR operation, but no declaration for the CAR operation.

```
(DEFUN EXAMPLE1 (X)
  (CADR X))

(DEFUN EXAMPLE2 (X)
  (DECLARE (LIST X))
  (CADR X))
```

In the following examples, a call to EXAMPLE3 always produces generic code, since it is not known that the result of the addition will necessarily be a fixnum. The declaration in EXAMPLE4 provides that information, and all the arithmetic operations are fixnum-specific.

```
(DEFUN EXAMPLE3 (X Y)
  (DECLARE (FIXNUM X Y))
  (+ X Y))

(DEFUN EXAMPLE4 (X Y)
  (DECLARE (FIXNUM X Y))
  (THE FIXNUM (+ X Y)))
```

The next example returns a list of the first, indexed, and last characters. With SPEED greater than or equal to 2 and SAFETY equal to 0, all the character fetching from the STRING argument will be very fast. The LENGTH operation will also be very fast, since it need not check for the type of the argument like the generic sequence function normally would. (This also means executing the form (LENGTH (THE LIST X)) is faster than executing the form (LENGTH X).) If SAFETY is greater than 0, bounds checking is still done, but type checking (of the string, for example) may not be, depending on what optimizations are used.

```
(DEFUN EXAMPLE5 (STRING INDEX)
  (DECLARE
    (SIMPLE-STRING STRING)
    (FIXNUM INDEX))
  (LIST (AREF STRING 0)
        (CHAR STRING INDEX)
        (SCHAR STRING (1- (LENGTH STRING)))))
```

Array access is fast in the following code:

```
(EVAL-WHEN (COMPILE LOAD EVAL)
 (DEFCONSTANT I-SIZE 3)
 (DEFCONSTANT J-SIZE 4)
 (DEFCONSTANT K-SIZE 5)
 (DEFTYPE FOOARRAY (&OPTIONAL ELEMENT-TYPE)
   `(SIMPLE-ARRAY ,ELEMENT-TYPE (,I-SIZE ,J-SIZE ,K-SIZE))))
 .
 .
 .
(DEFUN FOO ()
 (DECLARE (TYPE (FOOARRAY T) X)
          (TYPE (FOOARRAY STRING-CHAR) Y))
  .
  .
  .
 (DOTIMES (I I-SIZE)
  (DOTIMES (J J-SIZE)
   (DOTIMES (K K-SIZE)
    (SETF (AREF X I J K)
    (FOO (AREF Y I J K)))))))
```

A.3 PROGRAM STRUCTURE

Avoid using closed-over variables (that is, lexical variables used in functions created within their scope). References to closed-over variables are slower than references to true local variables (which are stack allocated), because closed-over variables must be found in simple vectors that represent the lexical environment that may take several instructions.

In tight inner loops, use macros or in-line functions rather than called functions. Always compile macros, functions declared in-line, and calls to the DEFSTRUCT macro before compiling code that uses them. Normally, you proclaim a function in-line just before defining it. Any calls to that function will then have the body expanded in-line at the calling site, unless you use the NOTINLINE declaration. If you declare or proclaim a function using the INLINE declaration without later providing a definition, a compiler error will result because no definition was provided for an in-line function.

The FUNCALL and APPLY functions are slower than calls to functions whose names are known at compile time. This is because the LISP system must check the following:

- Whether the object is a function
- What kind of function (by symbol or function object, interpreted or compiled)

- The number of arguments the function takes

The FUNCALL and APPLY functions are usually two to three times slower than a compiled call to a fixed function with a fixed number of arguments.

The CATCH special form and operations that use the catch-throw mechanism are slower than calling a function, using the APPLY function.

No more penalty is inflicted for using the lambda-list keyword &OPTIONAL than for using required arguments. However, an &REST variable causes a list to be created for those arguments passed after the required and &OPTIONAL arguments. &KEY arguments are the slowest; they have the consing overhead of &REST keyword, plus the run-time code to parse that list and assign the proper values for the given keywords.

Using multiple values requires less time and space than consing a list or vector of results. Both methods are slower than just returning single values. (Consing requires garbage collections later.)

The READ function is slower than the READ-LINE or READ-CHAR function, since READ has to parse the input according to the current LISP reader syntax, create numbers, and intern symbols. The READ-CHAR function is slower than the READ-LINE function, due to the general overhead of streams and RMS.

The WRITE, FORMAT, and PPRINT functions are slower than explicit calls to the PRINC and PRIN1 functions.

Using the xxx-TO-STRING functions for getting a string representation of a LISP object is faster than using the WITH-OUTPUT-TO-STRING function. The WITH-OUTPUT-TO-STRING function must create a stream and use the usual stream functions. The READ-FROM-STRING and PARSE-INTEGER functions are faster than the WITH-INPUT-FROM-STRING function for the same reason.

The compiler compiles each top-level form in a file when it compiles a file by surrounding arbitrary forms in the following manner:

```
(PROGN (DEFUN #:TOP-LEVEL-FUNCTION () arbitrary-top-level-form)
       (#:TOP-LEVEL-FUNCTION))
```

An *arbitrary-top-level-form* is any LISP form other than a call to the EVAL-WHEN or PROGN special form, the DEFUN or DEFMACRO macro, the PROCLAIM function, or a package function. Creating, compiling, dumping, and loading these temporary functions takes time, so it is wise to gather many arbitrary forms into functions of reasonable size. Typically, such forms can be calls to data initialization functions (such as (SETF (GET ...) ...)). To have these function calls inside a function definition anyway is desirable so that you can do selective initialization from the program without having to reload the file.

A.4 COMPILER REQUIREMENTS

The PROCLAIM, PROVIDE, REQUIRE, and package functions like USE-PACKAGE and IN-PACKAGE must be used at "top level" for the compiler to recognize them. A top-level form is defined as a form without surrounding parentheses, or a form at top level within a call to either the EVAL-WHEN or PROGN special form. Uses of the DEFUN macro and anonymous lambdas that would get evaluated in code get compiled as separate functions (closures if they use closed-over variables). This is true in the following call to the DEFUN macro and to the anonymous lambda that follows.

```
(LET ((COUNTER 0)) (DEFUN NEXT () (INCF COUNTER)))

(TRY #'(LAMBDA (X) (PRINT X)))
```

If you want functions as data objects (that is, in data structures where they would not be processed during normal evaluation), you must compile them explicitly. This is exemplified by the difference between the following:

```
(LIST #'(LAMBDA () (FOO))
      #'(LAMBDA () (BAR)) )
```

and

```
'( #'(LAMBDA () (FOO))
   #'(LAMBDA () (BAR)) )
```

In the first case, the compiler recognizes the functions and creates compiled-function objects for them. In the second case, the compiler does not notice the fucntions since the entire form is quoted.

If you leave the code in the list at run time, the explicit calls to the FUNCALL function on each element of the list would run the code interpretively. So, to have compiled code in the list, you must fill it with compiled functions. You can do this at run time by using the COMPILE function with NIL as the first argument, or you can fill the list with compiled functions once, when loading. Or, you can compile a file, using macros that expand into definitions of functions with names created using the GENSYM function. Then, have an initialization function fill up the list with those compiled functions at load time.

APPENDIX B

USING THE "EMACS" EDITOR STYLE

This appendix provides information on the "EMACS" Editor style. The "EMACS" style consists of a collection of key bindings that cause the Editor to behave like the EMACS editor. This appendix lists these bindings and explains how to activate the "EMACS" style in the Editor, but does not provide any tutorial information on using EMACS.

This appendix is organized as follows:

- Section B.1 explains to a new user how to learn about the Editor.
- Section B.2 describes how to activate the "EMACS" style as a minor or major style, thus making the "EMACS" key bindings available to you.
- Section B.3 lists the key bindings in the "EMACS" style.

B.1 INTRODUCTION TO THE EDITOR

To learn about the Editor, read Chapter 3 of this manual. Most of the information in Chapter 3 is also true when you are using the "EMACS" style. The chief difference when you are using the "EMACS" style lies in the key bindings. In many instances, keys or key sequences that invoke one command when you are not using the "EMACS" style invoke a different command when the "EMACS" style is active. Table B-1 compares default Editor key bindings with EMACS key bindings, showing where differences exist. When reading in Chapter 3, keep these key binding differences in mind. Table B-1 is arranged in the approximate order that the key bindings and commands are presented in Chapter 3. (Table B-1 lists only those commands listed in Chapter 3. The full set of "EMACS" style key bindings is presented in Section B.3.)

Section 3.2, which concerns editing operations, contains information on editing using (among other things) the numeric keypad. Keys and key sequences on the numeric keypad are set up to emulate the EDT

editor. If you are using the "EMACS" style, you still can use the keypad keys to do editing (as long as the "EDT Emulation" style is active). However, the operations performed by these keys, while similar to EMACS editing operations, may be different enough to produce confusion in a seasoned EMACS user.

Table B-1: Differences Between "EMACS" Key Bindings and Default Bindings

Default Binding	"EMACS" Binding	Command
	General-Purpose Commands	
CTRL/Z	ESCAPE x	Execute Named Command
CTRL/X CTRL/Z	CTRL/G	Pause Editor
None	CTRL/X s	Write Current Buffer
None	CTRL/X CTRL/M	Write Modified Buffers
None	CTRL/X CTRL/W	Write Named File
CTRL/X CTRL/N	CTRL/X p	Next Window
CTRL/X CTRL/R	CTRL/X d	Remove Current Window
None	CTRL/X 1	Remove Other Windows
CTRL/W	CTRL/L	Redisplay Screen
	Editing Commands	
None	CTRL/X CTRL/I	Insert File
None	ESCAPE q	Query Search Replace
None	ESCAPE CTRL/G	Exit Recursive Edit
None	ESCAPE u	Upcase Word
None	ESCAPE l	Downcase Word
None	ESCAPE c	Capitalize Word
	Buffer and Window Commands	
None	CTRL/X b	Select Buffer
None	CTRL/X CTRL/B	List Buffers
None	CTRL/X CTRL/D	Delete Current Buffer
None	CTRL/X CTRL/E	Ed
None	CTRL/X CTRL/V	Edit File
None	CTRL/X z	Grow Window
None	CTRL/X CTRL/Z	Shrink Window
None	CTRL/X 2	Split Window
	Customizing Commands	
CTRL/X CTRL/E	CTRL/X e	Execute Keyboard Macro

B.2 ACTIVATING THE "EMACS" STYLE

By default, the Editor has "EDT Emulation" as its major style and "VAX LISP" as its only minor style. (If you are not editing LISP code, the "VAX LISP" style will not be active.) Section 3.5.1.4 contains information about styles. To summarize: Whenever you press a key, the Editor looks in various places to see if that key is bound to a command. The Editor first checks the current buffer; then checks the minor styles, looking at the most recently activated minor style first; then checks the major style; and finally checks to see if the key is bound globally. This means that key bindings in minor styles take precedence over, or "shadow," key bindings in the major style or global key bindings.

You can activate the "EMACS" style as either a minor or the major style:

- If you leave "EDT Emulation" as the major style and activate "EMACS" as a minor style, key binding conflicts between "EDT Emulation" and "EMACS" (such as CTRL/U and CTRL/W) will be settled in favor of "EMACS".

- If you make "EMACS" the major style and activate "EDT Emulation" as a minor style, key binding conflicts will be settled in favor of "EDT Emulation".

- If you make "EMACS" the major style and do not activate "EDT Emulation" as a minor style, you will not have access to the keypad editing capabilities of "EDT Emulation". (However, you can bind the keypad keys to any commands you like in the "EMACS" style; see Section 3.5.1.)

B.2.1 Activating "EMACS" as a Minor Style

You can activate "EMACS" as a minor style from within the Editor by using the "Activate Minor Style" command. This command activates a minor style for the current buffer only. However, use of this command may cause problems if you are editing LISP code, because "EMACS" will become the most recently activated style; thus, "EMACS" key bindings will take precedence over conflicting "VAX LISP" key bindings.

A better approach is to make "EMACS" a default minor style, which will cause "EMACS" to be activated before the "VAX LISP" style when you start editing LISP code. To make "EMACS" a default minor style, call the following function from the LISP interpreter or in your LISP initialization file:

```
(PUSH "EMACS" (EDITOR:VARIABLE-VALUE "Default Minor Styles"))
```

B.2.2 Making "EMACS" the Major Style

To make "EMACS" the Editor's major style, call the following function from the LISP interpreter or in your LISP initialization file:

```
(SETF (EDITOR:VARIABLE-VALUE "Default Major Style") "EMACS")
```

This call causes "EMACS" to replace "EDT Emulation" as the Editor's major style. If you wish to reinstate "EDT Emulation" as one of the minor styles, call the following:

```
(PUSH "EDT Emulation"
      (EDITOR:VARIABLE-VALUE "Default Minor Styles"))
```

B.3 "EMACS" STYLE KEY BINDINGS

Table B-2 lists the key bindings supplied in the "EMACS" style. Appendix C contains short descriptions of the available commands, and a list of the key bindings supplied with the Editor. The table of key bindings in Appendix C is especially useful for finding key binding conflicts; that is, where the same key or key sequence is bound to two or more different commands in different contexts.

Key sequences containing alphabetic characters are case-sensitive; you must enter the alphabetic character in the case shown.

Use CTRL/[to generate an #\ESCAPE character from keyboards not possessing an ESCAPE key.

Table B-2: "EMACS" Style Key Bindings

Key(s)	Command
Cursor Movement	
CTRL/F	Forward Character
CTRL/B	Backward Character
ESCAPE f	Forward Word
ESCAPE b	Backward Word
CTRL/A	Beginning of Line
CTRL/E	End of Line
CTRL/P	Previous Line
CTRL/N	Next Line
ESCAPE a	Beginning of Paragraph
ESCAPE e	End of Paragraph
ESCAPE p	Previous Paragraph
ESCAPE n	Next Paragraph
ESCAPE v	Previous Screen
CTRL/V	Next Screen

Table B-2 (cont.)

Key(s)	Command
ESCAPE <	Beginning of Buffer
ESCAPE >	End of Buffer
ESCAPE ,	Beginning of Window
ESCAPE .	End of Window
CTRL/Z	Scroll Window Down
ESCAPE z	Scroll Window Up
ESCAPE !	Line to Top of Window
Searching	
CTRL/^	EMACS Forward Search
CTRL/R	EMACS Backward Search
Deleting	
DELETE	Delete Previous Character
CTRL/D	Delete Next Character
ESCAPE DELETE	Delete Previous Word
ESCAPE d	Delete Next Word
ESCAPE CTRL/D	Delete Whitespace
Killing, Yanking, and Regions	
CTRL/K	Kill Line
ESCAPE k	Kill Paragraph
CTRL/W	Kill Region
CTRL/Y	Yank
ESCAPE y	Yank Previous
ESCAPE CTRL/Y	Yank Replace Previous
ESCAPE CTRL/W	Undo Previous Yank
CTRL/SPACE	Set Select Mark
ESCAPE CTRL/SPACE	Unset Select Mark
CTRL/X CTRL/X	Exchange Point and Select Mark
Text Insertion and Modification	
CTRL/O	Open Line
CTRL/X q	Quoted Insert
CTRL/X CTRL/I	Insert File
ESCAPE c	Capitalize Word
ESCAPE l	Downcase Word
ESCAPE u	Upcase Word
CTRL/T	Transpose Previous Characters
ESCAPE t	Transpose Previous Words
ESCAPE q	Query Search Replace

Table B-2 (cont.)

Key(s)	Command
Multiple Windows and Buffers	
CTRL/X n	Previous Window
CTRL/X p	Next Window
CTRL/X d	Remove Current Window
CTRL/X 1	Remove Other Windows
CTRL/X z	Grow Window
CTRL/X CTRL/Z	Shrink Window
ESCAPE CTRL/V	Page Next Window
CTRL/X 2	Split Window
CTRL/X b	Select Buffer
CTRL/X CTRL/B	List Buffers
CTRL/X CTRL/D	Delete Current Buffer
Starting and Saving Work	
CTRL/X CTRL/E	Ed
CTRL/X CTRL/V	Edit File
CTRL/X CTRL/R	Read File
CTRL/X CTRL/F	View File
CTRL/X s	Write Current Buffer
CTRL/X CTRL/M	Write Modified Buffers
CTRL/X CTRL/W	Write Named File
Editor Control	
ESCAPE x	Execute Named Command
CTRL/G	Pause Editor
ESCAPE CTRL/G	Exit Recursive Edit
CTRL/L	Redisplay Screen
ESCAPE CTRL/U	Supply Prefix Argument
CTRL/U	Supply EMACS Prefix
CTRL/X e	Execute Keyboard Macro
CTRL/X CTRL/T	Show Time
CTRL/X =	What Cursor Position

APPENDIX C

EDITOR COMMANDS AND KEY BINDINGS

This appendix briefly describes the Editor commands and lists the key bindings that are supplied with the Editor. The appendix is organized as follows:

- Section C.1 lists the Editor commands, along with each command's key bindings and a brief description of the command.
- Section C.2 lists the keys and key sequences that are bound to commands and explains how to determine to which command a key or key sequence is bound in a given context.

C.1 EDITOR COMMAND DESCRIPTIONS

Table C-1 alphabetically lists the Editor commands. The second column of the table lists the keys or key sequences that are bound to that command (if any) and the context in which they are bound. The third column contains a brief description of the command. For a full description of each command, refer to the *VAX LISP/VMS Editor Programming Guide*.

Table C-1: Editor Commands And Key Bindings

Name	Binding(s)[1]	Description
Activate Minor Style	None	Prompts for the name of a minor style and then activates that style as a minor style in the current buffer
Apropos	None	Prompts for a string, then displays the names of objects of a specified type containing that string
Apropos Word	(:STYLE "VAX LISP") [ESCAPE] ?	Displays the result of evaluating the APROPOS function with the word at the cursor location as the argument
Backward Character	:GLOBAL [←] (:STYLE "EMACS") [CTRL/B]	Moves the cursor backward one character, or by the number of characters specified by the prefix argument
Backward Kill Ring	None	Rotates the kill ring backward by one element, or by the number of elements specified by the prefix argument
Backward Page	None	Moves the cursor to the previous page break, or to the preceding page break specified by the prefix argument
Backward Search	None	Prompts for a search string, then moves the cursor to the beginning of the first preceding occurrence of that string, or to the preceding occurrence specified by the prefix argument
Backward Word	(:STYLE "EMACS") [ESCAPE] b	Moves the cursor to the end of the previous word, or to the end of the preceding word specified by the prefix argument
Beginning of Buffer	(:STYLE "EDT Emulation") [PF1] [5] (:STYLE "EMACS") [ESCAPE] <	Moves the cursor to the beginning of the buffer
Beginning of Line	(:STYLE "EMACS") [CTRL/A]	Moves the cursor to the beginning of the current line, or to the beginning of the following line specified by the prefix argument
Beginning of Outermost Form	(:STYLE "VAX LISP") [CTRL/X] <	Moves the cursor to the beginning of the outermost form currently containing it, or, if the cursor is not currently contained in a form, to the beginning of the preceding outermost form
Beginning of Paragraph	(:STYLE "EMACS") [ESCAPE] A	Moves the cursor to the beginning of the current paragraph
Beginning of Window	(:STYLE "EMACS") [ESCAPE] ,	Moves the cursor to the top of the current window
Bind Command	None	Prompts for a command name, a key sequence to bind to the command, and a context in which to bind the key sequence, then binds the key sequence to the command
Capitalize Region	None	Capitalizes the first letter of each word in the current select region

[1] [] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [o↓•] Pointer button transition: o button up; • button held down; ↓ button pressed; ↑ button released; [o•o]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Capitalize Word	(:STYLE "EMACS") [ESCAPE] c	Capitalizes the first letter of the word at the cursor location
Close Outermost Form	(:STYLE "VAX LISP") [ESCAPE]]	Completes the outermost LISP form by inserting close-parentheses characters at the cursor position
Deactivate Minor Style	None	Prompts for the name of a minor style, then deactivates that minor style in the current buffer
Delete Current Buffer	(:STYLE "EMACS") [CTRL/X] [CTRL/D]	Deletes the current buffer; for modified buffers, asks if the contents of the buffer should first be saved
Delete Line	None	Deletes everything between the cursor and the end of the current line, or to the end of the following line specified by the prefix argument
Delete Named Buffer	None	Prompts for the name of a buffer, then deletes that buffer; if the buffer is modified, asks if the contents of the buffer should first be saved
Delete Next Character	(:STYLE "EMACS") [CTRL/D]	Deletes the character following the cursor, or the number of following characters specified by the prefix argument
Delete Next Word	(:STYLE "EMACS") [ESCAPE] d	Deletes everything from the cursor position to the end of the current word, or the number of following words specified by the prefix argument
Delete Previous Character	:GLOBAL [DELETE] (:STYLE "EMACS") [DELETE]	Deletes the character preceding the cursor position, or the number of preceding characters specified by the prefix argument
Delete Previous Word	(:STYLE "EMACS") [ESCAPE] [DELETE]	Deletes everything from the cursor position to the beginning of the current word, or the number of preceding words specified by the prefix argument
Delete Whitespace	(:STYLE "EMACS") [ESCAPE] [CTRL/D]	Deletes whitespace characters following the cursor location up to the next nonwhitespace character
Delete Word	None	Deletes everything from the cursor position to the beginning of the next word, including whitespace, or deletes the number of following words specified by the prefix argument
Describe	None	Prompts for the name and type of an object, then displays a description of that object
Describe Word	(:STYLE "VAX LISP") [CTRL/?]	Calls the DESCRIBE function with the word at the cursor position as the argument
Describe Word at Pointer[2]	(:STYLE "VAX LISP") [○○↓]	Calls the DESCRIBE function with the word at the pointer position as the argument
Downcase Region	None	Makes all alphabetic characters in the current select region lower case
Downcase Word	(:STYLE "EMACS") [ESCAPE] l	Makes all alphabetic characters in the word at the cursor position lower case

[1] [ ] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [○↓●] Pointer button transition: ○button up; ●button held down; ↓button pressed; ↑button released; [○●○]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Ed	(:STYLE "EMACS") CTRL/X CTRL/E	Prompts for a LISP object to edit and, if the object is a symbol, whether to edit its function definition or its value
Edit File	(:STYLE "EMACS") CTRL/X CTRL/V	Prompts for the specification of a file to edit; completion and alternatives are available during your response to the prompt
EDT Append	(:STYLE "EDT Emulation") 9	Appends the current select region to the contents of the paste buffer
EDT Back to Start of Line	(:STYLE "EDT Emulation") CTRL/H and BACKSPACE[4] and F12[3]	Moves the cursor to the beginning of the current line, or to the beginning of the previous line if the cursor is already at the beginning of a line; or moves back the number of lines specified by the prefix argument
EDT Beginning of Line	(:STYLE "EDT Emulation") 0	Moves the cursor to the beginning of the next line, if the current direction is forward, or to the beginning of the current or previous line, if the current direction is backward; moves by the number of lines specified by the prefix argument
EDT Change Case	(:STYLE "EDT Emulation") PF1 1	Changes the case (lower to upper and vice versa) of all characters in the select region, or, if no select region is defined, of the character at the cursor position
EDT Cut	(:STYLE "EDT Emulation") 6 and REMOVE[3] and ○↓○	Deletes the current select region and replaces the contents of the paste buffer with it
EDT Delete Character	(:STYLE "EDT Emulation") .	Deletes the character at the cursor position and stores it in the deleted character area; deletes the number of characters specified by the prefix argument
EDT Delete Line	(:STYLE "EDT Emulation") PF4	Deletes from the cursor position to the beginning of the next line and stores the deleted line in the deleted line area; deletes the number of lines specified by the prefix argument
EDT Delete Previous Character	(:STYLE "EDT Emulation") DELETE	Deletes the character preceding the cursor and stores it in the deleted character area; deletes the number of characters specified by the prefix argument
EDT Delete Previous Line	(:STYLE "EDT Emulation") CTRL/U	Deletes from the cursor position to the beginning of the current line or, if the cursor is at the beginning of a line, to the beginning of the previous line; stores the result in the deleted line area; deletes the number of lines specified by the prefix argument
EDT Delete Previous Word	(:STYLE "EDT Emulation") CTRL/J and LINEFEED[4] and F13[3]	Deletes from the cursor position to the beginning of the current word or, if the cursor is between words, to the beginning of the previous word; stores the result in the deleted word area; deletes the number of lines specified by the prefix argument

[1] [ ] indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 . Numeric keypad keys. → ↑ Arrow keys. ○↓● Pointer button transition: ○button up; ●button held down; ↓button pressed; ↑button released; ○●○→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

[4] Key available only on VT100 terminal.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
EDT Delete to End of Line	(:STYLE "EDT Emulation") [PF1] [2]	Deletes from the cursor position to the end of the current line or, if the cursor is at the end of a line, to the end of the next line; stores the result in the deleted line area; deletes the number of lines specified by the prefix argument
EDT Delete Word	(:STYLE "EDT Emulation") [-]	Deletes from the cursor position to the beginning of the next word; stores the result in the deleted word area; deletes the number of words specified by the prefix argument
EDT Deselect	(:STYLE "EDT Emulation") [PF1] [.]	Cancels the current select region; equivalent to "Unset Select Mark"
EDT End of Line	(:STYLE "EDT Emulation") [2]	Moves the cursor to the end of the current, next, or previous line, depending on starting cursor position and current direction; moves by the number of lines specified by the prefix argument
EDT Move Character	(:STYLE "EDT Emulation") [3]	Moves the cursor forward or backward by one character, according to the current direction; moves the number of characters specified by the prefix argument
EDT Move Page	(:STYLE "EDT Emulation") [7]	Moves the cursor to the preceding or following page break, depending on the current direction; moves the number of pages specified by the prefix argument
EDT Move Word	(:STYLE "EDT Emulation") [1]	Moves the cursor to the beginning of the next, current, or preceding word, depending on current direction and cursor starting position; moves the number of words specified by the prefix argument
EDT Paste	(:STYLE "EDT Emulation") [PF1] [6] and [INSERT HERE][3]	Inserts the contents of the paste buffer at the cursor location
EDT Paste at Pointer[2]	(:STYLE "EDT Emulation") [•↓○]	Inserts the contents of the paste buffer at the pointer cursor location
EDT Query Search	(:STYLE "EDT Emulation") [PF1] [PF3] and [FIND][3]	Prompts for a search string and moves the cursor to the following or preceding occurrence of the string, depending on the current direction; moves to the occurrence specified by the prefix argument
EDT Replace	(:STYLE "EDT Emulation") [PF1] [9]	Replaces the current select region with the contents of the paste buffer
EDT Scroll Window	(:STYLE "EDT Emulation") [8]	Scrolls the window in the direction indicated by the current direction
EDT Search Again	(:STYLE "EDT Emulation") [PF3]	Searches for the next or previous occurrence of the search string that was last entered, according to the current direction

[1] [] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [○↓•] Pointer button transition: ○button up; •button held down; ↓button pressed; ↑button released; [○•○]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

[3] Key available only on LK201 keyboard.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
EDT Select	(:STYLE "EDT Emulation") [.] and [SELECT][3]	Places a mark at the cursor position to indicate one end of a select region; equivalent to "Set Select Mark"
EDT Set Direction Backward	(:STYLE "EDT Emulation") [5]	Sets the current direction to backward
EDT Set Direction Forward	(:STYLE "EDT Emulation") [4]	Sets the current direction to forward
EDT Special Insert	(:STYLE "EDT Emulation") [PF1] [3]	Inserts the character whose ASCII code is specified by the prefix argument at the cursor position
EDT Substitute	(:STYLE "EDT Emulation") [PF1] [ENTER]	If the cursor is located at the beginning of the current search string, replaces the search string with the contents of the paste buffer, then finds the next occurrence of the search string
EDT Undelete Character	(:STYLE "EDT Emulation") [PF1] [,]	Inserts the contents of the deleted character area at the cursor location
EDT Undelete Line	(:STYLE "EDT Emulation") [PF1] [PF4]	Inserts the contents of the deleted line area at the cursor location
EDT Undelete Word	(:STYLE "EDT Emulation") [PF1] [-]	Inserts the contents of the deleted word area at the cursor location
EMACS Backward Search	(:STYLE "EMACS") [CTRL/R]	Searches backward for the first occurrence of the search string specified in the previous command; prompts for a search string if the previous command was not a searching command; searches for the occurrence of the search string specified by the prefix argument
EMACS Forward Search	(:STYLE "EMACS") [CTRL/^]	Searches forward for the first occurrence of the search string specified in the previous command; prompts for a search string if the previous command was not a searching command; searches for the occurrence of the search string specified by the prefix argument
End Keyboard Macro	:GLOBAL [CTRL/X])	Ends the collection of keystrokes for a keyboard macro
End of Buffer	(:STYLE "EDT Emulation") [PF1] [4] (:STYLE "EMACS") [ESCAPE] >	Moves the cursor to the end of the buffer
End of Line	(:STYLE "EMACS") [CTRL/E]	Moves the cursor to the end of the current line, or forward by the number of lines specified by the prefix argument and then to the end of the line
End of Outermost Form	(:STYLE "VAX LISP") [CTRL/X] >	Moves the cursor to the outermost form currently surrounding the cursor, or, if the cursor is between outermost forms, to the end of the following outermost form

[1] [ ] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [o↓•] Pointer button transition: obutton up; •button held down; ↓button pressed; ↑button released; [ooo]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
End of Paragraph	(:STYLE "EMACS") [ESCAPE] e	Moves the cursor to the end of the current paragraph
End of Window	(:STYLE "EMACS") [ESCAPE] .	Moves the cursor to the end of the text in the current window
Evaluate LISP Region	(:STYLE "VAX LISP") [CTRL/X] [CTRL/A]	Evaluates the select region as LISP code; displays the result of the evaluation in the information area
Exchange Point and Select Mark	(:STYLE "EMACS") [CTRL/X] [CTRL/X]	Moves the cursor to the other end of the current select region, and the mark delimiting the select region to the old cursor position; in other words, preserves the select region but places the cursor at the other end of it
Execute Keyboard Macro	:GLOBAL [CTRL/X] [CTRL/E] (:STYLE "EMACS") [CTRL/X] e	Executes the current keyboard macro once, or the number of times specified by the prefix argument
Execute Named Command	:GLOBAL [CTRL/Z] and [DO][3] (:STYLE "EDT Emulation") [PF1] [7] (:STYLE "EMACS") [ESCAPE] x	Prompts for the name of a command to execute; input completion and alternatives are available during your response to the prompt
Exit Editor	None	Returns control to the LISP interpreter, discarding the current Editor state; asks if modified buffers should first be saved
Exit Recursive Edit	(:STYLE "EMACS") [ESCAPE] [CTRL/G]	Terminates a recursive edit, or pauses the Editor if not doing a recursive edit
Forward Character	:GLOBAL [→] (:STYLE "EMACS") [CTRL/F]	Moves the cursor forward one character
Forward Kill Ring	None	Rotates the kill ring forward by one element, or by the number of elements specified by the prefix argument
Forward Page	None	Moves the cursor to the next page break, or to the following page break specified by the prefix argument
Forward Search	None	Prompts for a search string, then moves the cursor forward to the end of the first occurrence of the string; moves the cursor to the occurrence of the string specified by the prefix argument
Forward Word	(:STYLE "EMACS") [ESCAPE] f	Moves the cursor to the beginning of the next word, or the beginning of the word specified by the prefix argument
Grow Window	(:STYLE "EMACS") [CTRL/X] z	Increases the height of the current window by one row, or by the number of rows specified by the prefix argument
Help	:GLOBAL [PF2] and [HELP][3]	Displays help on your current situation

[1] [ ] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [o↓•] Pointer button transition: o button up; • button held down; ↓ button pressed; ↑ button released; [o•o]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Help on Editor Error	:GLOBAL CTRL/X ?	Displays information on the last Editor error that occurred
Illegal Operation	None	Signals an Editor error; use to disable a key binding locally within a style or buffer
Indent LISP Line	(:STYLE "VAX LISP") TAB	Adjusts the current LISP line so that it is indented properly in the context of the program
Indent LISP Region	None	Adjusts the indentation of the LISP code in the current select region
Indent Outermost Form	(:STYLE "VAX LISP") CTRL/X TAB	Indents each line in the outermost LISP form containing the cursor
Insert Buffer	None	Prompts for a buffer name, then inserts the contents of the specified buffer at the cursor location
Insert Close Paren and Match	(:STYLE "VAX LISP"))	Inserts a close-parenthesis character at the cursor location and highlights the corresponding open-parenthesis character
Insert File	(:STYLE "EMACS") CTRL/X CTRL/I	Prompts for a file specification, then inserts the contents of the file at the cursor location; input completion and alternatives are available during your response to the prompt
Kill Enclosing List	None	Deletes the LISP list that encloses the cursor and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted list; deletes the number of enclosing lists specified by the prefix argument
Kill Line	(:STYLE "EMACS") CTRL/K	Deletes the rest of the current line and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted line; deletes the number of lines specified by the prefix argument
Kill Next Form	None	Deletes the LISP form immediately following the cursor and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted form; deletes the number of following forms specified by the prefix argument within the current parentheses nesting level
Kill Paragraph	(:STYLE "EMACS") ESCAPE k	Deletes the rest of the current paragraph and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted paragraph; deletes the number of paragraphs specified by the prefix argument

[1] [ ] indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 . Numeric keypad keys. → ↑ Arrow keys. o↓• Pointer button transition: o button up; • button held down; ↓ button pressed; ↑ button released; o•o→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Kill Previous Form	None	Deletes the LISP form immediately preceding the cursor and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted form; deletes the number of preceding forms specified by the prefix argument within the current parentheses nesting level
Kill Region	(:STYLE "EMACS") CTRL/W and o↓o	Deletes the current select region and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted region
Kill Rest of List	None	Deletes the rest of the enclosing list and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted list fragment
Line to Top of Window	(:STYLE "EMACS") ESCAPE !	Moves the line containing the cursor to the top of the window
List Buffers	(:STYLE "EMACS") CTRL/X CTRL/B	Displays a list of all buffers
List Key Bindings	None	Displays a list of all visible key bindings or of all keys bound in a specified context
Maybe Reset Select at Pointer[2]	:GLOBAL ↑oo	If the pointer cursor has not moved, cancels the select region that was started with ↓oo; if the pointer cursor has moved since ↓oo, does nothing
Move Point and Select Region[2]	:GLOBAL •oo→	Moves the text cursor with the pointer cursor, marking a select region
Move Point to Pointer[2]	:GLOBAL •oo	Moves the text cursor to the pointer cursor
Move to LISP Comment	(:STYLE "VAX LISP") CTRL/X ;	If there is no comment on the current line, moves the cursor to the comment column and inserts a semicolon and space; if there is a comment, moves the cursor to the comment
New Line	:GLOBAL RETURN (:BUFFER "General Prompting") LINEFEED (:STYLE "EMACS") RETURN	Breaks a line at the cursor position, leaving the cursor at the start of the new line
New LISP Line	(:STYLE "VAX LISP") LINEFEED	Breaks a line at the cursor position, and indents the new line by the appropriate amount in the context of the program
Next Form	(:STYLE "VAX LISP") CTRL/X .	Moves the cursor to the end of the next form, or to the end of the following form specified by the prefix argument; does not move outside the current parentheses nesting level

[1] indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 . Numeric keypad keys. → ↑ Arrow keys. o↓•Pointer button transition: obutton up; •button held down; ↓button pressed; ↑button released; o•o→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2]Available only on VAXstation.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Next Line	:GLOBAL [↓] (:STYLE "EMACS") [CTRL/N]	Moves the cursor to the next line or down by the number of lines specified by the prefix argument, keeping the cursor in the same column if possible
Next Paragraph	(:STYLE "EMACS") [ESCAPE] n	Moves the cursor to the beginning of the next paragraph, or to the following paragraph specified by the prefix argument
Next Screen	:GLOBAL [NEXT SCREEN][3]	Moves the window down in the buffer by one screenful, or by as many screenfuls as are specified by the prefix argument
Next Window	:GLOBAL [CTRL/X] [CTRL/N] (:STYLE "EMACS") [CTRL/X] p	Selects another window on the screen to be the current window; eventually circulates through all the windows on the screen
Open Line	(:STYLE "EDT Emulation") [PF1] [0] (:STYLE "EMACS") [CTRL/O]	Breaks a line at the cursor location, leaving the cursor at the end of the old line
Page Next Window	(:STYLE "EMACS") [ESCAPE] [CTRL/V]	Scrolls the next window on the screen down one page; or, if a prefix argument is supplied, scrolls the next window by that many rows
Pause Editor	:GLOBAL [CTRL/X] [CTRL/Z] (:STYLE "EMACS") [CTRL/G]	Saves the Editor state and returns control to the LISP interpreter
Previous Form	(:STYLE "VAX LISP") [CTRL/X] ,	Moves the cursor to the beginning of the previous form, or to the beginning of the preceding form specified by the prefix argument; does not move outside the current parentheses nesting level
Previous Line	:GLOBAL [↑] (:STYLE "EMACS") [CTRL/P]	Moves the cursor to the previous line, or up by the number of lines specified by the prefix argument; keeps the cursor in the same column if possible
Previous Paragraph	(:STYLE "EMACS") [ESCAPE] p	Moves the cursor to the end of the previous paragraph, or to the end of the preceding paragraph specified by the prefix argument
Previous Screen	:GLOBAL [PREV SCREEN][3] (:STYLE "EMACS") [ESCAPE] v	Moves the cursor up in the buffer by one screenful, or by as many screenfuls as are specified by the prefix argument
Previous Window	(:STYLE "EMACS") [CTRL/X] n	Makes another window on the screen into the current window; eventually circulates through all windows on the screen
Prompt Complete String	(:BUFFER "General Prompting") [CTRL/]	Attempts to complete your response to a prompt, based on what you have typed already and the choices available in the situation
Prompt Help	(:BUFFER "General Prompting") [PF2]	Displays information on whatever is being prompted for

[1] [] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [○↓●] Pointer button transition: ○button up; ●button held down; ↓button pressed; ↑button released; [○●○]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Prompt Read and Validate	(:BUFFER "General Prompting") RETURN and ENTER	Used to terminate prompt input
Prompt Scroll Help Window	(:BUFFER "General Prompting") CTRL/V	Scrolls the Help window down while another buffer is current; supplied to allow you to scroll the Help window while responding to a prompt
Prompt Show Alternatives	(:BUFFER "General Prompting") PF1 PF2	Displays a list of alternatives that can be entered in response to the current prompt, based on what you have typed already
Query Search Replace	(:STYLE "EMACS") ESCAPE q	Prompts for a search string and a replacement; offers a number of options at each replacement opportunity
Quoted Insert	:GLOBAL CTRL/X \	Inserts the next character typed at the cursor location without Editor interpretation
	(:STYLE "EMACS") CTRL/X q	
Read File	(:STYLE "EMACS") CTRL/X CTRL/R	Prompts for a file specification, then replaces the contents of the current buffer with the file; if the current buffer is modified, prompts for confirmation
Redisplay Screen	(:STYLE "EDT Emulation") CTRL/W	Erases and redisplays everything on the screen
	(:STYLE "EMACS") CTRL/L	
Remove Current Window	:GLOBAL CTRL/X CTRL/R	Removes the current window from the screen; does not delete the associated buffer
	(:STYLE "EMACS") CTRL/X d	
Remove Other Windows	(:STYLE "EMACS") CTRL/X 1	Removes all windows but the current window from the screen
Scroll Window Down	(:STYLE "EMACS") CTRL/Z	Scrolls the current window down in the buffer by one row, or by the number of rows specified by the prefix argument
Scroll Window Up	(:STYLE "EMACS") ESCAPE z	Scrolls the current window up in the buffer by one row, or by the number of rows specified by the prefix argument
Select Buffer	(:STYLE "EMACS") CTRL/X b	Prompts for a buffer name, then makes that buffer the current buffer; creates a new buffer if necessary
Select Enclosing Form at Pointer	(:STYLE "VAX LISP") ↓oo	Places the form enclosing the cursor in a select region; if the cursor is already in a select region, expands the region to the next outermost form
Select Outermost Form	(:STYLE "VAX LISP") CTRL/X CTRL/	Makes the outermost LISP form containing the cursor into a select region
Self Insert	:GLOBAL All graphic characters	Inserts the last character typed at the cursor location
Set Screen Height	None	Sets the screen height to the number of rows specified by the prefix argument; prompts for height if no prefix argument is defined

[1] indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 . Numeric keypad keys. → ↑ Arrow keys. o↓• Pointer button transition: o button up; • button held down; ↓ button pressed; ↑ button released; o•o→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
Set Screen Width	None	Sets the screen width to the number of columns specified by the prefix argument; prompts for the width if no prefix argument is defined
Set Select Mark	(:STYLE "EDT Emulation") [.] (:STYLE "EMACS") [CTRL/]	Places a mark at the cursor position to indicate one end of a select region
Show Time	(:STYLE "EMACS") [CTRL/X] [CTRL/T]	Displays the time and date in the information area
Shrink Window	(:STYLE "EMACS") [CTRL/X] [CTRL/Z]	Shrinks the current window by one row, or by the number of rows specified by the prefix argument
Split Window	(:STYLE "EMACS") [CTRL/X] 2	Splits the current window into two identical windows
Start Keyboard Macro	:GLOBAL [CTRL/X] (	Starts collecting keystrokes for a keyboard macro, replacing any unnamed keyboard macro that already exists
Start Named Keyboard Macro	None	Prompts for a name, then starts collecting keystrokes for a keyboard macro; the resulting keyboard macro is catalogued under the name you give and can be treated as a command
Supply EMACS Prefix	(:STYLE "EMACS") [CTRL/U]	Sets the prefix argument to four, if no prefix argument was defined, or to four times its former value, if a prefix argument was defined
Supply Prefix Argument	(:STYLE "EDT Emulation") [PF1] [PF1] (:STYLE "EMACS") [ESCAPE] [CTRL/U]	Prompts for a prefix argument; if a prefix argument is already defined, multiplies it by the number you enter
Transpose Previous Characters	(:STYLE "EMACS") [CTRL/T]	Transposes the two characters preceding the cursor
Transpose Previous Words	(:STYLE "EMACS") [ESCAPE] t	Transposes the words at and preceding the cursor
Undo Previous Yank	(:STYLE "EMACS") [ESCAPE] [CTRL/W]	Deletes the previously yanked region without pushing it onto the kill ring; more generally, deletes the select region without pushing it onto the kill ring
Unset Select Mark	(:STYLE "EDT Emulation") [PF1] [.] (:STYLE "EMACS") [ESCAPE] [CTRL/]	Cancels the current select region
Upcase Region	None	Changes all alphabetic characters in the current select region to upper case
Upcase Word	(:STYLE "EMACS") [ESCAPE] u	Changes all alphabetic characters in the word at the cursor location to upper case
View File	(:STYLE "EMACS") [CTRL/X] [CTRL/F]	Prompts for a file specification, then reads the specified file into a read-only buffer

[1] [ ] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [o↓•] Pointer button transition: obutton up; •button held down; ↓button pressed; ↑button released; [o••]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

Table C-1 (cont.)

Name	Binding(s)[1]	Description
What Cursor Position	(:STYLE "EMACS") CTRL/X =	Displays information about the cursor location
Write Current Buffer	(:STYLE "EMACS") CTRL/X s	Writes out the current buffer; creates a new file version, or updates the LISP symbol whose function or value slot is being edited
Write Modified Buffers	(:STYLE "EMACS") CTRL/X CTRL/M	Performs the "Write Current Buffer" operation for each buffer that has been modified
Write Named File	(:STYLE "EMACS") CTRL/X CTRL/W	Prompts for a file specification, then creates a file having that specification from the contents of the current buffer
Yank	(:STYLE "EMACS") CTRL/Y	Inserts the current kill-ring region at the cursor location; inserts as many copies as are specified by the prefix argument
Yank at Pointer[2]	(:STYLE "EMACS") •↓○	Inserts the current kill-ring region at the pointer cursor location
Yank Previous	(:STYLE "EMACS") ESCAPE y	Rotates the kill ring forward, then inserts the new current kill-ring region at the cursor location; inserts as many copies as are specified by the prefix argument
Yank Replace Previous	(:STYLE "EMACS") ESCAPE CTRL/Y	Deletes the previously yanked region, rotates the kill ring forward, and inserts the new current kill-ring region at the cursor location; inserts as many copies as are specified by the prefix argument

[1] [ ] indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 . Numeric keypad keys. → ↑ Arrow keys. ○↓• Pointer button transition: ○button up; •button held down; ↓button pressed; ↑button released; ○•○→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

C.2 EDITOR KEY BINDINGS

Table C-2 lists the keys and key sequences that are bound to Editor commands, and the context in which they are bound. Some keys or key sequences are bound to more than one command. To find out which command a key or key sequence will invoke, use the following procedure:

1. If your current buffer is "General Prompting" (that is, you are typing in response to a prompt) and the key or key sequence is bound to a command in the context (:BUFFER "General Prompting"), then the key or key sequence invokes that command.

2. Otherwise, if the key or key sequence is bound to a command in one or more minor styles, then the key or key sequence invokes the command to which it is bound in the most recently activated minor style. You can tell which minor style was activated most recently by examining the window's label strip. The label strip contains a list of the active minor styles, with the most recently activated style at the front of the list.

3. Otherwise, if the key or key sequence is bound to a command in the current major style, then the key or key sequence invokes that command. You can identify the major style by looking at the label strip; it precedes the list of minor styles. (If the list of minor styles is too long, the major style is omitted.)

4. Otherwise, if the key or key sequence is bound to a command in the :GLOBAL context, then the key or key sequence invokes that command.

5. Otherwise, the key or key sequence is unbound; typing it results in an error.

Table C-2: Editor Key Bindings

Key(s)	Context and Command
	Single Keys
CTRL/SPACE	(:BUFFER "General Prompting") Prompt Complete String (:STYLE "EMACS") Set Select Mark
CTRL/A	(:STYLE "EMACS") Beginning of Line
CTRL/B	(:STYLE "EMACS") Backward Character

Table C-2 (cont.)

Key(s)	Context and Command
CTRL/D	(:STYLE "EMACS") Delete Next Character
CTRL/E	(:STYLE "EMACS") End of Line
CTRL/F	(:STYLE "EMACS") Forward Character
CTRL/G	(:STYLE "EMACS") Pause Editor
CTRL/H or BACKSPACE	(:STYLE "EDT Emulation") EDT Back to Start of Line
TAB or CTRL/I	(:STYLE "VAX LISP") Indent LISP Line
CTRL/J or LINEFEED	(:BUFFER "General Prompting") New Line (:STYLE "VAX LISP") New LISP Line (:STYLE "EDT Emulation") EDT Delete Previous Word
CTRL/K	(:STYLE "EMACS") Kill Line
CTRL/L	(:STYLE "EMACS") Redisplay Screen
RETURN or CTRL/M	(:BUFFER "General Prompting") Prompt Read and Validate (:STYLE "EMACS") New Line :GLOBAL New Line
CTRL/N	(:STYLE "EMACS") Next Line
CTRL/O	(:STYLE "EMACS") Open Line
CTRL/P	(:STYLE "EMACS") Previous Line
CTRL/R	(:STYLE "EMACS") Backward Search
CTRL/T	(:STYLE "EMACS") Transpose Previous Characters
CTRL/U	(:STYLE "EMACS") Supply EMACS Prefix (:STYLE "EDT Emulation") EDT Delete Previous Line
CTRL/V	(:BUFFER "General Prompting") Prompt Scroll Help Window (:STYLE "EMACS") Next Screen
CTRL/W	(:STYLE "EMACS") Kill Region (:STYLE "EDT Emulation") Redisplay Screen
CTRL/Y	(:STYLE "EMACS") Yank

Table C-2 (cont.)

Key(s)	Context and Command
CTRL/Z	(:STYLE "EMACS") Scroll Window Down :GLOBAL Execute Named Command
CTRL/^	(:STYLE "EMACS") EMACS Forward Search
CTRL/?	(:STYLE "VAX LISP") Describe Word
DELETE or <X]	(:STYLE "EMACS") Delete Previous Character (:STYLE "EDT Emulation") Delete Previous Character :GLOBAL Delete Previous Character
)	(:STYLE "VAX LISP") Insert Close Paren and Match
keypad 0	(:STYLE "EDT Emulation") EDT Beginning of Line
keypad 1	(:STYLE "EDT Emulation") EDT Move Word
keypad 2	(:STYLE "EDT Emulation") EDT End of Line
keypad 3	(:STYLE "EDT Emulation") EDT Move Character
keypad 4	(:STYLE "EDT Emulation") EDT Set Direction Forward
keypad 5	(:STYLE "EDT Emulation") EDT Set Direction Backward
keypad 6	(:STYLE "EDT Emulation") EDT Cut
keypad 7	(:STYLE "EDT Emulation") EDT Move Page
keypad 8	(:STYLE "EDT Emulation") EDT Scroll Window
keypad 9	(:STYLE "EDT Emulation") EDT Append
keypad .	(:STYLE "EDT Emulation") Set Select Mark
keypad ENTER	(:BUFFER "General Prompting") Prompt Read and Validate
keypad ,	(:STYLE "EDT Emulation") EDT Delete Character
keypad -	(:STYLE "EDT Emulation") EDT Delete Word
keypad PF2	(:BUFFER "General Prompting") Prompt Help (:STYLE "EDT Emulation") Help :GLOBAL Help
keypad PF3	(:STYLE "EDT Emulation") EDT Search Again

Table C-2 (cont.)

Key(s)	Context and Command
keypad PF4	(:STYLE "EDT Emulation") EDT Delete Line
Up arrow	:GLOBAL Previous Line
Down arrow	:GLOBAL Next Line
Right arrow	:GLOBAL Forward Character
Left arrow	:GLOBAL Backward Character
All graphics characters	:GLOBAL Self Insert
Single Keys -- LK201 Keyboard Only	
F12	(:STYLE "EDT Emulation") EDT Back to Start of Line
F13	(:STYLE "EDT Emulation") EDT Delete Previous Word
HELP	:GLOBAL Help
DO	:GLOBAL Execute Named Command
FIND	(:STYLE "EDT Emulation") EDT Query Search
INSERT HERE	(:STYLE "EDT Emulation") EDT Paste
REMOVE	(:STYLE "EDT Emulation") EDT Cut
SELECT	(:STYLE "EDT Emulation") EDT Select
PREV SCREEN	:GLOBAL Previous Screen
NEXT SCREEN	:GLOBAL Next Screen
Two-Key Sequences Starting with CTRL/X	
CTRL/X CTRL/SPACE	(:STYLE "VAX LISP") Select Outermost Form
CTRL/X CTRL/A	(:STYLE "VAX LISP") Evaluate LISP Region
CTRL/X CTRL/B	(:STYLE "EMACS") List Buffers
CTRL/X CTRL/D	(:STYLE "EMACS") Delete Current Buffer
CTRL/X CTRL/E	(:STYLE "EMACS") Ed :GLOBAL Execute Keyboard Macro

Table C-2 (cont.)

Key(s)	Context and Command
CTRL/X CTRL/F	(:STYLE "EMACS") View File
CTRL/X TAB or CTRL/X CTRL/I	(:STYLE "EMACS") Insert File (:STYLE "VAX LISP") Indent Outermost Form
CTRL/X RETURN or CTRL/X CTRL/M	(:STYLE "EMACS") Write Modified Buffers
CTRL/X CTRL/N	:GLOBAL Next Window
CTRL/X CTRL/R	(:STYLE "EMACS") Read File :GLOBAL Remove Current Window
CTRL/X CTRL/T	(:STYLE "EMACS") Show Time
CTRL/X CTRL/V	(:STYLE "EMACS") Edit File
CTRL/X CTRL/W	(:STYLE "EMACS") Write Named File
CTRL/X CTRL/X	(:STYLE "EMACS") Exchange Point and Select Mark
CTRL/X CTRL/Z	(:STYLE "EMACS") Shrink Window :GLOBAL Pause Editor
CTRL/X (	:GLOBAL Start Keyboard Macro
CTRL/X)	:GLOBAL End Keyboard Macro
CTRL/X ,	(:STYLE "VAX LISP") Previous Form
CTRL/X .	(:STYLE "VAX LISP") Next Form
CTRL/X 1	(:STYLE "EMACS") Remove Other Windows
CTRL/X 2	(:STYLE "EMACS") Split Window
CTRL/X ;	(:STYLE "VAX LISP") Move to LISP Comment
CTRL/X <	(:STYLE "VAX LISP") Beginning of Outermost Form
CTRL/X >	(:STYLE "VAX LISP") End of Outermost Form
CTRL/X =	(:STYLE "EMACS") What Cursor Position
CTRL/X ?	:GLOBAL Help on Editor Error

Table C-2 (cont.)

Key(s)	Context and Command
CTRL/X \	:GLOBAL Quoted Insert
CTRL/X b	(:STYLE "EMACS") Select Buffer
CTRL/X d	(:STYLE "EMACS") Remove Current Window
CTRL/X e	:GLOBAL Execute Keyboard Macro
CTRL/X n	(:STYLE "EMACS") Previous Window
CTRL/X p	(:STYLE "EMACS") Next Window
CTRL/X q	(:STYLE "EMACS") Quoted Insert
CTRL/X s	(:STYLE "EMACS") Write Current Buffer
CTRL/X z	(:STYLE "EMACS") Grow Window
Two-Key Sequences Starting with ESCAPE	
ESCAPE CTRL/SPACE	(:STYLE "EMACS") Unset Select Mark
ESCAPE CTRL/D	(:STYLE "EMACS") Delete Whitespace
ESCAPE CTRL/G	(:STYLE "EMACS") Exit Recursive Edit
ESCAPE CTRL/U	(:STYLE "EMACS") Supply Prefix Argument
ESCAPE CTRL/V	(:STYLE "EMACS") Page Next Window
ESCAPE CTRL/W	(:STYLE "EMACS") Undo Previous Yank
ESCAPE CTRL/Y	(:STYLE "EMACS") Yank Previous Replace
ESCAPE !	(:STYLE "EMACS") Line to Top of Window
ESCAPE ,	(:STYLE "EMACS") Beginning of Window
ESCAPE .	(:STYLE "EMACS") End of Window
ESCAPE <	(:STYLE "EMACS") Beginning of Buffer
ESCAPE >	(:STYLE "EMACS") End of Buffer
ESCAPE ?	(:STYLE "VAX LISP") Apropos Word
ESCAPE]	(:STYLE "VAX LISP") Close Outermost Form

Table C-2 (cont.)

Key(s)	Context and Command
ESCAPE a	(:STYLE "EMACS") Beginning of Paragraph
ESCAPE b	(:STYLE "EMACS") Backward Word
ESCAPE c	(:STYLE "EMACS") Capitalize Word
ESCAPE d	(:STYLE "EMACS") Delete Next Word
ESCAPE e	(:STYLE "EMACS") End of Paragraph
ESCAPE f	(:STYLE "EMACS") Forward Word
ESCAPE k	(:STYLE "EMACS") Kill Paragraph
ESCAPE l	(:STYLE "EMACS") Downcase Word
ESCAPE n	(:STYLE "EMACS") Next Paragraph
ESCAPE p	(:STYLE "EMACS") Previous Paragraph
ESCAPE q	(:STYLE "EMACS") Query Search Replace
ESCAPE t	(:STYLE "EMACS") Transpose Previous Words
ESCAPE u	(:STYLE "EMACS") Upcase Word
ESCAPE v	(:STYLE "EMACS") Previous Screen
ESCAPE x	(:STYLE "EMACS") Execute Named Command
ESCAPE y	(:STYLE "EMACS") Yank Previous
ESCAPE z	(:STYLE "EMACS") Scroll Window Up
ESCAPE DELETE or ESCAPE <X]	(:STYLE "EMACS") Delete Previous Word
Two-Key Sequences Starting with Keypad PF1	
keypad PF1 0	(:STYLE "EDT Emulation") Open Line
keypad PF1 1	(:STYLE "EDT Emulation") EDT Change Case
keypad PF1 2	(:STYLE "EDT Emulation") EDT Delete to End of Line
keypad PF1 3	(:STYLE "EDT Emulation") EDT Special Insert

Table C-2 (cont.)

Key(s)	Context and Command
keypad PF1 4	(:STYLE "EDT Emulation") End of Buffer
keypad PF1 5	(:STYLE "EDT Emulation") Beginning of Buffer
keypad PF1 6	(:STYLE "EDT Emulation") EDT Paste
keypad PF1 7	(:STYLE "EDT Emulation") Execute Named Command
keypad PF1 9	(:STYLE "EDT Emulation") EDT Replace
keypad PF1 .	(:STYLE "EDT Emulation") Unset Select Mark
keypad PF1 ENTER	(:STYLE "EDT Emulation") EDT Substitute
keypad PF1 ,	(:STYLE "EDT Emulation") EDT Undelete Character
keypad PF1 -	(:STYLE "EDT Emulation") EDT Undelete Word
keypad PF1 PF1	(:STYLE "EDT Emulation") Supply Prefix Argument
keypad PF1 PF3	(:BUFFER "General Prompting") Prompt Show Alternatives
keypad PF1 PF3	(:STYLE "EDT Emulation") EDT Query Search
keypad PF1 PF4	(:STYLE "EDT Emulation") EDT Undelete Line

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.

-D-

-F-

-I-

-M-

-N-

-Q-

-S-

-T-

-W-

-Y-