VAX LISP/VMS Program Development Guide

Order Number: AA–MK69A–TE

July 1989

This guide contains information for a LISP language programmer to interpret, compile, and debug VAX LISP programs on VMS operating systems on DECwindows and terminal interfaces.

Revision/Update Information:	This is a new manual.
Operating System and Version:	VMS Version 5.1
Software Version:	VAX LISP Version 3.0

digital equipment corporation
maynard, massachusetts

First Printing, July 1989

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.

The Reader's Comments form at the end of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

AI VAXstation
DEC
DECnet
DECUS
MicroVAX
MicroVAX II
MicroVMS
PDP
ULTRIX
ULTRIX–11
ULTRIX–32
UNIBUS
VAX
VAX LISP
VAX LISP/ULTRIX
VAX LISP/VMS
VAXstation
VAXstation II
VMS
digital™

X Window System is a trademark of the Massachusetts Institute of Technology.

S828

This document was prepared using VAX DOCUMENT, Version 1.1.

Contents

Chapter 2 Using VAX LISP

Chapter 3 Using the VAX LISP Editor

Part II Using VAX LISP Facilities on the DECwindows Interface

Chapter 5 The DECwindows Interface to VAX LISP

Chapter 6 Starting LISP from DECwindows

Chapter 7 The Listener

Appendix A Using DECwindows

Examples

Figures

Tables

Preface

The *VAX LISP / VMS Program Development Guide* is for developing and debugging LISP programs and for compiling and executing LISP programs on the VMS operating system. It is intended for use on DECwindows and terminal interfaces. 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 the VMS operating system is helpful but not essential; familiarity with the *Introduction to 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

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

Part I: VAX LISP System Concepts and Facilities for VMS Systems

Part I consists of four chapters, which explain VAX LISP concepts and describe the VAX LISP facilities for the terminal interface. Most of the concepts described in Part I also apply to Part II.

- Chapter 1, Introduction to VAX LISP, provides an overview of VAX LISP, explains how to use the help facilities, describes 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.
- 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, Debugging Facilities, explains how to use the VAX LISP debugging facilities.

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

Part II: Using VAX LISP Facilities with the DECwindows Interface

Part II consists of six chapters, which explain how to use the VAX LISP facilities from the DECwindows interface.

- Chapter 5, The DECwindows Interface to VAX LISP, describes the LISP utilities that run on the DECwindows interface and how to use them. A sample LISP programming session is included in the chapter.
- Chapter 6, Starting LISP from DECwindows, describes the different ways you can invoke LISP to enter an interactive LISP session, compile a file, or resume a suspended session.
- Chapter 7, The Listener, describes how to use the Listener to enter LISP forms, select or edit text and objects, work with files, and get help.
- Chapter 8, Using the VAX LISP Editor in DECwindows, describes the different editing modes available from the DECwindows interface and how to access them.
- Chapter 9, Using the VAX LISP Inspector, describes how to invoke the Inspector and exit from it, inspect and modify objects, update the Inspector display, and return a value from the Inspector.
- Chapter 10, Using the Debugging Utilities from the DECwindows Interface, describes how to use the Break loop, the Debugger, the Stepper, and the Tracer from the DECwindows interface.

Associated Documents

The following documents are relevant to VAX LISP programming on VMS systems:

- *VAX LISP/VMS Installation Guide*
- *Common LISP: The Language*
- *VAX LISP/VMS Editor Programming Guide*
- *VAX LISP/VMS System Access Guide*
- *VAX LISP/VMS Interface to VWS Graphics*
- *VAX LISP/VMS System-Building Guide*
- *VMS DCL Dictionary*
- *VAX Architecture Handbook*
- VMS V 5.0 Documentation, Programming Subkit

For a complete list of VMS software documents, see the *Overview of VMS Documentation*.

Conventions

The following conventions are used in this manual:

Convention	Meaning
UPPERCASE	DCL commands and qualifiers and VMS file names are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The examples directory (SYS$SYSROOT:[VAXLISP.EXAMPLES] by default) contains sample LISP source files.
`UPPERCASE` `TYPEWRITER`	Defined LISP functions, macros, variables, constants, and other symbol names are printed in uppercase `TYPEWRITER` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The `CALL-OUT` macro calls a defined external routine....
`lowercase` `typewriter`	LISP forms are printed in the text in lowercase `typewriter` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: `(setf example-1 (make-space))`
SANS SERIF	Format specifications of LISP functions and macros are printed in a sans serif typeface. For example: CALL-OUT *external-routine* &REST *routine-arguments*
italics	Lowercase *italics* in format specifications and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters. For example: The *routine-arguments* must be compatible with the arguments defined in the call to the `DEFINE-EXTERNAL-ROUTINE` macro.
( )	Parentheses used in examples of LISP code and in format specifications indicate the beginning and end of a LISP form. For example: `(setq name lisp)`
[ ]	Square brackets in format specifications enclose optional elements. For example: [*doc-string*] Square brackets do not indicate optional elements when they are used in the syntax of a directory name in a VMS file specification. Here, the square bracket characters must be included in the syntax. For example: `(pathname "MIAMI::DBA1:[SMITH]LOGIN.COM;4")`
{ }	In function and macro format specifications, braces enclose elements that are considered one unit of code. For example: {*keyword value*}

Convention	Meaning
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered 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 that follow it are optional. For example: PPRINT *object* &OPTIONAL *stream* 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: CALL-OUT *external-routine* &REST *routine-arguments* Do not specify &REST when you invoke a 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 :LISTING :MACHINE-CODE :OPTIMIZE :OUTPUT-FILE :VERBOSE :WARNINGS Do not specify &KEY when you invoke a function or macro whose definition includes &KEY.
. . .	A horizontal ellipsis in a format specification means that the element preceding the ellipsis can be repeated. For example: *function-name* . . .
. . .	A vertical ellipsis in a code example indicates that all the information that the system would display in response to the function call is not shown; or that all the information a user is to enter is not shown.
Return	A word inside a box indicates that you press a key on the keyboard. For example: Return or Tab In code examples, carriage returns are implied at the end of each line. However, Return is used in some examples to emphasize carriage returns.
Ctrl/*x*	Two key names enclosed in a box indicate a control key sequence in which you hold down Ctrl while you press another key. For example: Ctrl/C or Ctrl/S
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key.

Convention	Meaning
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	By default, MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. You can rebind the mouse buttons.

Red print	In interactive examples, user input is shown in red. For example:
	`Lisp> (cdr '(a b c))`
	`(B C)`
	`Lisp>`

Part I
VAX LISP System Concepts and Facilities for VMS Systems

Chapter 1

Introduction to VAX LISP

LISP is a general-purpose programming language. The language is 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 explicitly invokes the LISP interpreter
- Automatic storage allocation and garbage collection

This manual describes VAX LISP on the VMS operating system, but refers to it as VAX LISP, where practicable.

This chapter provides an overview of the VAX LISP language. The overview parallels the structure of this manual and the remaining VAX LISP documentation. The chapter also explains how to get on-line help at the DCL and the LISP language levels of operation and describes:

- 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
- Inspector
- Debugging facilities
- Extensible pretty printer
- Facility for calling out to external routines

- Facility for calling back to LISP functions from external routines
- Facility for defining non-LISP data structures (alien structures)
- Facility for defining interrupt functions (that is, functions that execute asynchronously due to VMS AST's)
- Access to the VWS capabilities of the VAXstation workstation
- DECwindows Toolkit interface
- CLX interface to the X Window System
- Utility for creating custom LISP systems

These extensions are described in Section 1.1.1 through Section 1.1.14.

Some of the functions, macros, and facilities defined by Common LISP are modified for the VAX LISP implementation. *VAX LISP Implementation and Extensions to Common LISP* 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 *VAX LISP/VMS Object Reference Manual.*

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

1.1.1 DCL LISP Command

The DCL LISP command invokes VAX LISP from the VMS command level on the terminal interface. 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

Chapter 6 describes how to invoke LISP from the DECwindows interface.

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 command line fashion.

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 7 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, indent, and evaluate LISP code. Chapter 3 describes how to use the Editor to write and edit LISP code. Chapter 8 describes how to access the Editor from the DECwindows interface.

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 keyboard. In this way, you can build up alternative editing systems or complete applications based on the Editor. See the *VAX LISP/VMS 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. *VAX LISP Implementation and Extensions to Common LISP* describes the error handler and explains how to create your own error handler.

1.1.4 Inspector

VAX LISP provides an Inspector, which is an implementation of the Common LISP INSPECT function. The Inspector is only available on the DECwindows interface. You can use the Inspector to examine and modify data structures. Chapter 9 explains how to use the Inspector.

1.1.5 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 4 explains how to use the debugging facilities. Chapter 10 explains how to use the debugging facilities from the DECwindows interface.

1.1.6 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 read 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. *VAX LISP Implementation and Extensions to Common LISP* explains how to use the pretty printer in each way.

1.1.7 Call-Out Facility

VAX LISP includes a call-out facility, which lets you call routines written in other VAX/VMS programming languages as well as run-time library (RTL) routines and VMS and RMS system services. Chapter 4 of the *VAX LISP/VMS System Access Guide* describes the call-out process and explains how to use the call-out facility.

1.1.8 Call-Back Facility

VAX LISP includes a call-back facility, which lets you invoke a LISP function from an external routine that has been called out to by a LISP call-out. Chapter 4 of the *VAX LISP/VMS System Access Guide* describes how to use the call-back facility.

1.1.9 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 6 of the *VAX LISP/VMS System Access Guide* describes the alien structure facility and explains how to use it.

1.1.10 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 7 of the *VAX LISP/VMS System Access Guide* describes how to define and use interrupt functions.

1.1.11 VWS Graphics Interface

VAX LISP provides access to the VWS graphics capabilities of the VAXstation 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 Interface to VWS Graphics* describes this interface.

1.1.12 DECwindows Interface

VAX LISP provides access to the DECwindows Toolkit, which is a set of application development tools and run-time routines that help you implement DECwindows applications. *VAX LISP/VMS DECwindows Programming Guide* describes how to use VAX LISP to access the Toolkit.

1.1.13 CLX Interface

VAX LISP provides an implementation of the CLX (Common LISP X) interface to the X Window System protocol. CLX is a package of LISP routines that give you access to the lower level capabilities of the X Window System without having to explicitly call out to external routines or define non-LISP data structures. *VAX LISP/VMS DECwindows Programming Guide* describes the VAX LISP implementation of the CLX interface to the X Window System.

1.1.14 System-Building Utility

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

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

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

1.1.15 VAX LISP Function, Macro, and Variable Descriptions for VMS

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

- Implementation-dependent objects mentioned but not fully defined in *Common LISP: The Language*
- VAX LISP objects that implement the parts of VAX LISP that are described in this manual
- VAX LISP extensions to the Common LISP I/O system
- Editor-specific objects
- System access-specific objects (pertaining to the call-out, alien structure, interrupt function, and program synchronization facilities)
- Graphics-specific objects
- Objects that implement the VAX LISP System-Building Utility

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

The LISP objects in the first two categories listed above are described in *VAX LISP/VMS Object Reference Manual.* VAX LISP extensions to Common LISP I/O are described in *VAX LISP Implementation and Extensions to Common LISP* and *VAX LISP/VMS System Access Guide.* Editor-specific objects are described in Part III of the *VAX LISP/VMS Editor Programming Guide.* System access-specific objects are described in the *VAX LISP/VMS System Access Guide.* Graphics-specific objects are described in the *VAX LISP/VMS Interface to VWS Graphics* and the *VAX LISP/VMS DECwindows Programming Guide.* The VAX LISP System-Building Utility is described in the *VAX LISP/VMS System-Building Guide.*

1.2 Help Facilities

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

1.2.1 DCL Help

The 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 available information.

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

```
$ HELP LISP
```

1.2.2 DECwindows Help

The Help menu lets you obtain on-line information about any DECwindows application. Help provides brief information about screen objects, concepts, or tasks you can perform in applications.

- To get help on tasks in DECwindows applications, choose the Overview menu item from the application's Help menu.
- To get help on objects, such as menus, scroll bars, and dialog boxes, point to the screen object and press and hold the Help key on your keyboard while you click MB1. Note that help is not available on all objects in the VAX LISP development environment.
- To get help on a menu item, press the Help key while you press and hold MB1 on the menu item, then release MB1.

For more detailed information on DECwindows help, see Appendix A.

1.2.3 LISP Help

VAX LISP provides two functions 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 the *VAX LISP/VMS Object Reference Manual.*

1.3 VMS File Specifications

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

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

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

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

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

Table 1–1: File Specification Defaults

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

1.4 Logical Names

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

For more information on logical names, see the *VMS DCL Concepts Manual.*

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.
- 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 (-).
- 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 pressing Ctrl/U.
- You can interrupt command execution by pressing 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 pressing 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 procedures for the terminal interface and VWS workstation:

- Invoking LISP
- Exiting LISP
- Entering input
- Deleting and editing input
- Entering the Debugger
- Using control key characters
- Creating programs
- Loading files
- Compiling programs
- Using suspended systems
- Using subprocesses

See Chapter 5 for additional information and restrictions on how to perform these procedures from the DECwindows interface.

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

VAX LISP[TM], V3.0
Copyright © Digital Equipment Corporation. 1989.
All Rights Reserved

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 pressing 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 characters from the cursor position back to the beginning of the line.

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 or F14 †	Switches between overstrike and insert modes in the current line.
Ctrl/B or ↑	Recalls the last line entered.
Ctrl/D or ←	Moves the cursor one character to the left.
Ctrl/E	Moves the cursor to the end of the line.
Ctrl/F or →	Moves the cursor one character to the right.
Ctrl/H or F12 †	Moves the cursor to the beginning of the line.
⌫	Deletes the character to the left of the cursor.
Ctrl/J or Linefeed or 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 or later keyboards.

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 4. For information on how to use the VAX LISP Debugger from the DECwindows interface, see Chapter 10.

Pressing Ctrl/C is a quick way to exit from the VAX LISP Debugger. If you want to recover from an error by discarding the expression you typed and start over, press 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 calls the ABORT function. If you want to recover from an error by discarding the expression you typed and start over, press Ctrl/C. (See ABORT in *VAX LISP / VMS Object Reference Manual* for an example of changing the behavior of Ctrl/C.)
Ctrl/O	Discards output being sent to the terminal until you press another Ctrl/O.
Ctrl/Q	Resumes terminal output that was halted with Ctrl/S.
Ctrl/R	Redisplays what is on a line.
Ctrl/S	Stops output to the terminal until a Ctrl/Q is pressed.
Ctrl/T	Displays process information. This is useful during a computation to watch the resources used.
Ctrl/U	Deletes characters from the cursor position back to the beginning of the current input line. The prompt is not echoed in LISP.
Ctrl/X	Deletes all input not yet 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 to 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 only useful for experimenting with small functions and macros.

2.8 Loading Files

Before you can use the definitions stored in a file in interactive LISP, you must load the file into the LISP system. The file can be compiled or interpreted; compiled files tend to load more quickly. You can load a file into the LISP system in three ways from a terminal interface:

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

```
$ LISP/INITIALIZE=MYINIT.LSP

VAX LISP[TM], Version V3.0
Copyright © Digital Equipment Corporation. 1989.
All Rights Reserved.

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 objects loaded in an initialization file to be listed at the terminal. For more information on the /VERBOSE qualifier, see Section 2.10.13.

- 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 *VAX LISP/VMS Object Reference Manual*.

- 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 within LISP or the LISP verb with the /COMPILE qualifier from DCL.

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 *VAX LISP/VMS Object Reference Manual*, 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.

At a terminal interface, 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. For additional information on compilation from the DECwindows interface, see Chapter 6.

The /COMPILE qualifier is described in Section 2.10.2. The COMPILE-FILE function is described in *VAX LISP/VMS Object Reference Manual*. 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 contain code that runs more quickly than code in uncompiled files. Fast-loading files are automatically placed in the directory containing the source files from which they are compiled.

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 compile only 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 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 may 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 faster, but the compiled code executes much 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 interpreted 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 the error to be signaled may not be accessible. The stepper facility is also more informative with interpreted than with compiled functions. See Chapter 4 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=20000
```

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

Section 2.10.2 through Section 2.10.14 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). /COMPILE is not the default qualifier for the LISP command; /INTERACTIVE is the default qualifier.
/CSTACK=*number*	Specifies the size of the control stack in the LISP session in 512-byte pages. The default is 185 pages.
/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.
/[NO]INITIALIZE=(*file-spec*,...)	Causes the LISP system to load an initialization file(s). The default file type for an initialization file is LSP or FAS. NOINITIALIZE suppresses the loading of initialization files.
/INTERACTIVE	Starts an interactive LISP session. /INTERACTIVE is the default qualifier for the LISP command.
/[NO]LIST=[*file-spec*]	Specifies that a listing file be created during compilation. A listing consists of the file name, date of compilation, names of the LISP expressions compiled (if the /VERBOSE qualifier is specified), and warning and error messages. The default file type for a listing file is LIS. /NOLIST suppresses a listing file and is the default except in batch mode. In such jobs, /LIST is the default.
/[NO]MACHINE_CODE	Includes VAX LISP machine code in the listing file. /NOMACHINE_CODE suppresses a listing of the machine code and is the default.
/MEMORY=*number*	Specifies the initial amount of dynamic virtual memory LISP allocates in 512-byte pages. The default is 5000 pages.

(continued on next page)

Table 2–3 (Cont.): DCL LISP Command Qualifiers

Qualifier	Function
/[NO]OPTIMIZE=(SPEED:*n*,SPACE:*n*, SAFETY:*n*,COMPILATION_SPEED:*n*)	Tells the compiler that each quality has the corresponding value. SPEED is the speed at which the object code runs, SPACE is the space occupied or used by the code, SAFETY is the run-time error checking of the code, and COMPILATION_SPEED is the speed of the compilation process. *n* is an integer in the range 0 to 3. The value 0 is the lowest priority value; the value 3 is the highest. The default value for *n* is 1. See *VAX LISP Implementation and Extensions to Common LISP* 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 name is the name of the file being compiled. The default output file type is FAS. /NOOUTPUT_FILE prevents compiled code from being written to a file. /OUTPUT_FILE is the default.
/RESUME=file	Resumes a suspended LISP system. The default file type for a suspended LISP system is SUS. See Section 2.11 on Using Suspended Systems.
/[NO]VERBOSE	Lists on the output device and the listing file, if any, the names of functions and macros defined in a file. /NOVERBOSE suppresses a listing of function and macro names defined in a file. /NOVERBOSE is the default.
/[NO]WARNINGS	Specifies that the compiler is to produce warning messages. /NOWARNINGS suppresses warning messages. /WARNINGS is the default.

2.10.1 Three Ways to Use the DCL LISP Command

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

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

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

Table 2–4: DCL LISP Command Qualifier Modes

Qualifier	Mode
/COMPILE	COMPILE
/CSTACK	INTERACTIVE or COMPILE
/ERROR_ACTION	INTERACTIVE or COMPILE or RESUME
/[NO]INITIALIZE	INTERACTIVE or COMPILE

(continued on next page)

Table 2–4 (Cont.): DCL LISP Command Qualifier Modes

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

2.10.2 /COMPILE

The /COMPILE qualifier invokes the VAX LISP compiler to compile one or more source files. The compiler creates a fast-loading (FAS) file from each source file. Unlike other compilers, such as those for BASIC and COBOL, the LISP compiler does not generate VMS object modules. Consequently, files processed by the LISP compiler do not have the OBJ 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 2.10.13). If you want to compile files with the aid of initialization files, use the /INITIALIZE qualifier (see Section 2.10.5). 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 /CSTACK

The /CSTACK qualifier lets you specify the size of the default control stack in a LISP session in 512-byte pages. The system requires a minimum of 32 pages to function. If you specify fewer than 32 pages with the /CSTACK qualifier, the system disregards the requested number of pages and uses 32 pages. The default number of pages is 185.

You can determine the total space allocated to all stacks from within LISP with the ROOM or ROOM-ALLOCATION function. However, since you cannot use the /CSTACK qualifier with the /RESUME qualifier, if you want to specify the size of the control stack, you must do it when you first invoke LISP. For information on the ROOM and ROOM-ALLOCATION functions, see the *VAX LISP/VMS Object Reference Manual.*

Format

LISP/CSTACK=*number-of-pages*

or

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

Example

```
$  LISP/CSTACK=200

VAX LISP[TM], V3.0
Copyright © Digital Equipment Corporation. 1989
All Rights Reserved.
Lisp>
```

Mode

Interactive or Compile

2.10.4 /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 *VAX LISP Implementation and Extensions to Common LISP*). 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 4 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 *VAX LISP/VMS Object Reference Manual*).

Format

LISP/ERROR_ACTION=*value*

Example

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

Mode

Interactive, Compile, or Resume

2.10.5 /[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 may 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 2.10.4 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 or modified 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 2.10.13) 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

VAX LISP[TM], Version V3.0
Copyright © Digital Equipment Corporation. 1989
All Rights Reserved.

; 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 *VAX LISP/VMS Object Reference Manual.*

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

2.10.6 /INTERACTIVE

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

Mode

Interactive

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 Monday, 23 January 1989 by JONES
Lisp Version V3.0

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 5000 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 5000 pages. If you specify fewer than 5000 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 5000 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 need more memory, LISP allocates it for you automatically.

Format

LISP/MEMORY=*number-of-pages*

or

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

Example

```
$ LISP/MEMORY=15000

VAX LISP[TM], V3.0
Copyright © Digital Equipment Corporation. 1989.
All Rights Reserved

Lisp>
```

Mode

Interactive, 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 *VAX LISP Implementation and Extensions to Common LISP*, Appendix B of this manual, 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 /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 qualifier cannot be used with the /CSTACK or /INITIALIZE qualifier.

Format

LISP/RESUME=*file-spec*

Example

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

Mode

Resume

2.10.13 /[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
   VAX LISP[TM], V3.0
   Copyright © Digital Equipment Corporation. 1989.
   All Rights Reserved.

   ; 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

2.10.14 /[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 *VAX LISP Implementation and Extensions to Common LISP* 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 may want to do this if your work is incomplete. By resuming LISP from a suspended system, you can continue your work from the point at which you stopped.

NOTE

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

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

2.11.1 Creating a Suspended System

The VAX LISP SUSPEND function puts in a file the LISP memory in use during an interactive LISP session, enabling you to resume the same LISP session at a later time. The SUSPEND function does not stop the current LISP session; you can continue to use the LISP session after the SUSPEND function has put a copy of memory into a file. The SUSPEND function also automatically invokes a garbage collection of dynamic memory space. See *VAX LISP Implementation and Extensions to Common LISP* 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 *VAX LISP/VMS System Access Guide* and *Common LISP: The Language* for a description of pathnames.

```
Lisp> (suspend "filex.sus")
; Starting full GC . . .
;  . . . Full GC finished
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 *VAX LISP/VMS Object Reference Manual.*

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 2.10.12 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 *VAX LISP/VMS Object Reference Manual* for a description of the effects of suspending a system.

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

2.12 Using Subprocesses

A subprocess is a process that you create or go to from the LISP system for executing Command Language Interpreter (CLI) commands. The purpose of a subprocess is to permit you to interrupt execution of a LISP process and to optionally execute the specified CLI command. You may want to do this to read a mail message or manipulate files while saving the state of an interactive LISP session.

2.12.1 Creating a Subprocess

The VAX LISP SPAWN function creates a process for executing CLI commands. 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 VMS hibernation state until the subprocess completes its execution. A hibernation state is one in which a process is inactive, but can become active at a later time. For a description of the SPAWN function, see *VAX LISP/VMS Object Reference Manual.*

In the following example, the user creates a subprocess to read a mail message, exits from MAIL, and resumes LISP.

```
Lisp> (spawn :command-string "mail")
MAIL> read
#1   5-OCT-1988     14:38:10.02
From:  VLSP::JONES
To:  Smith
CC:
Subj: Tomorrow's committee meeting

We will hold tomorrow's meeting as scheduled.
```

```
P. Jones
MAIL> exit
Lisp>
```

The value of :COMMAND-STRING must be a DCL command. By default, the SPAWN function does not process a command. In this example, because the user did not specify the :PARALLEL keyword, the LISP process was put into a hibernation state until the subprocess completed its execution.

NOTE

> The VAX LISP SPAWN function can be used with the DECwindows interface only if you specify the :PARALLEL keyword with a value of T.

2.12.2 Connecting to a Subprocess

The ATTACH function connects your terminal to a process and puts the current LISP process into a VMS hibernation state. You can use this function to switch terminal control from one process to another. The ATTACH function is described in *VAX LISP/VMS Object Reference Manual*.

In the following example, the call to the SPAWN creates a subprocess, the DCL ATTACH command attaches the terminal back to the process SMITH, and the call to the VAX LISP ATTACH function returns control to the process SMITH_1.

```
Lisp> (spawn)
$ ATTACH SMITH
Lisp> (attach "smith_1")
%DCL-S-RETURNED, control returned to process SMITH_1
$
```

NOTE

> The ATTACH function can be used only if LISP is invoked from DCL on a terminal interface; it cannot be used if LISP is invoked from another command language interpreter or from DECwindows.

2.12.3 Exiting from a Subprocess

If you entered a subprocess with a value assigned to :COMMAND-STRING, exiting that process returns you to LISP. If you spawned a subprocess without assigning a value to :COMMAND-STRING, logging out returns you to LISP.

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 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 D.

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 DECWINDOWS USERS

> Chapter 8 describes how to use the Editor in the DECwindows environment.

NOTE TO VWS VAXSTATION USERS

> When you use the Editor on a VWS 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 VWS VAXstation.

3.1 Introduction to the Editor

The VAX LISP Editor is a general-purpose text editor. It includes some capabilities that make it 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 from the LISP environment. The Editor is a part of LISP and cannot be used outside 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 works 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 lets you have more than one LISP object or file available for editing at one time. Each object or file resides in its own buffer. Commands let you 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, 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:

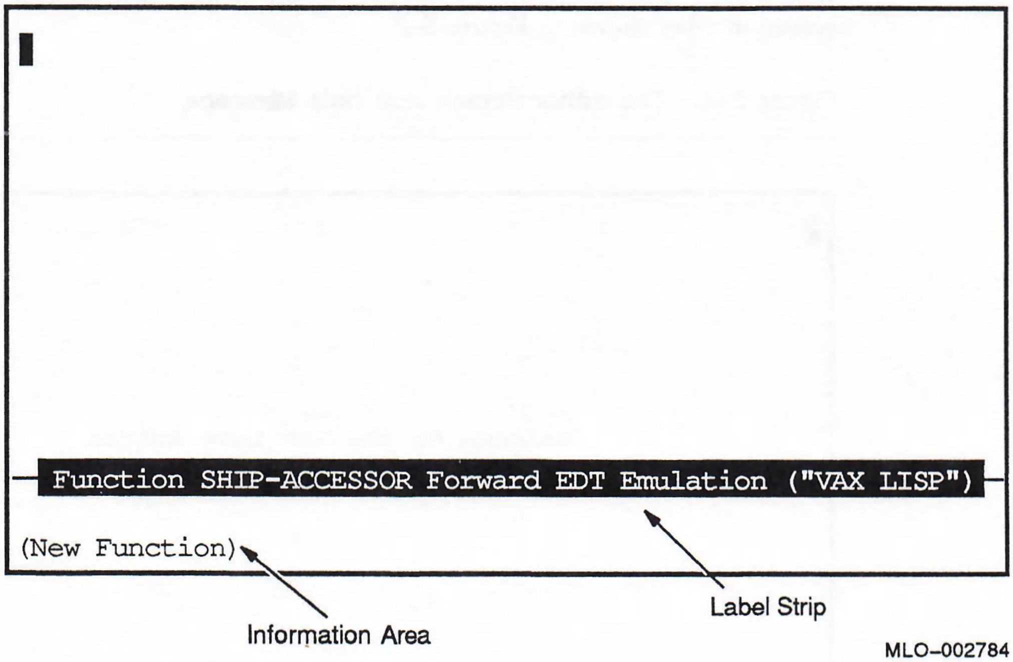
```
Lisp> (ed 'ship-list :type :value)
```

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 appears as shown in Figure 3–1.

Figure 3–1: Invoking the Editor



```
Function SHIP-ACCESSOR Forward EDT Emulation ("VAX LISP")
(New Function)
```

NOTE TO VWS VAXSTATION USERS

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, the 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.

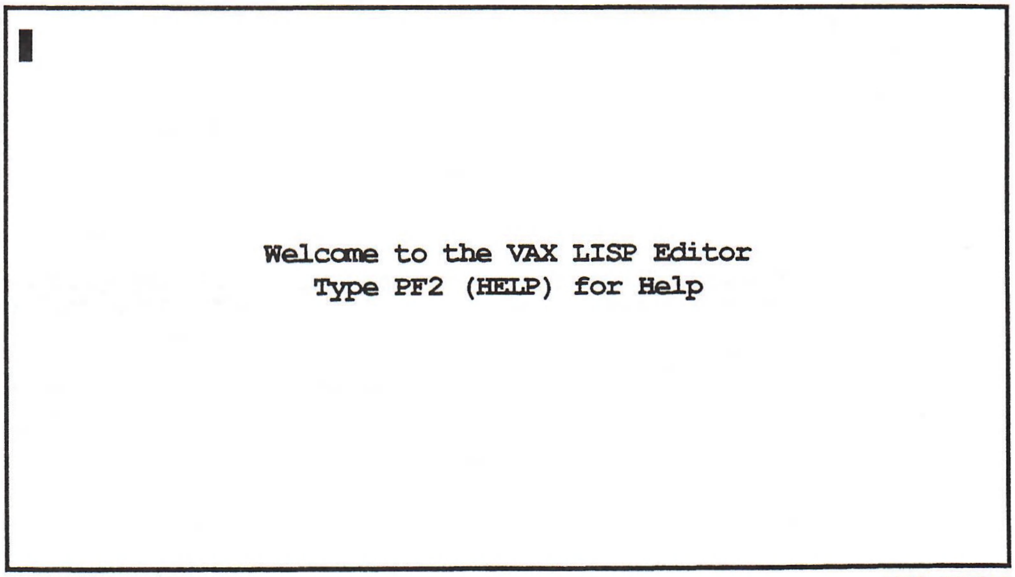
After you have used the Editor, you can pause it (see Section 3.1.5) and return to the LISP interpreter. Later, you may want to resume your editing. If you want to return to the Editor state you left, call the ED function without arguments:

```
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 screen display shown in Figure 3–2.

Figure 3–2: The Editor Screen with Help Message



```
Welcome to the VAX LISP Editor
   Type PF2 (HELP) for Help
```

MLO–002785

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

NOTE

In the interpreter, 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 pressing the control character. If you do this, do not specify a value greater than `1` with the `BIND-KEYBOARD-FUNCTION :LEVEL` keyword. Using a value greater than `1` may disrupt the Editor's operation.

3.1.3 Interacting with the Editor

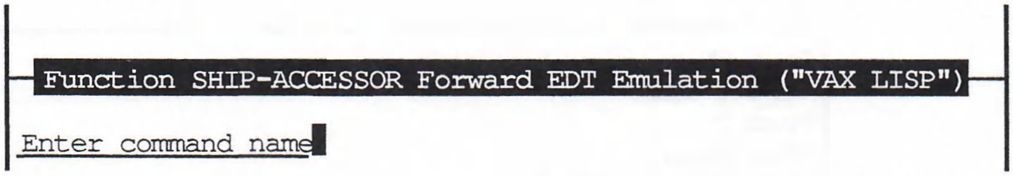
You interact with the Editor through commands. Commands do the following:

- 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 press 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 press Ctrl/Z. This causes a prompt to appear just below the label strip as shown in Figure 3–3.

Figure 3–3: Entering Commands at the Editor Prompt



```
Function SHIP-ACCESSOR Forward EDT Emulation ("VAX LISP")

Enter command name
```

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 pressing 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 press Ctrl/Z, type `"Pause Editor"` in response to the command prompt, and press Return. Or, you can press 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 press 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, press Ctrl/C. Ctrl/C stops the current command and makes the Editor ready to accept other commands.

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

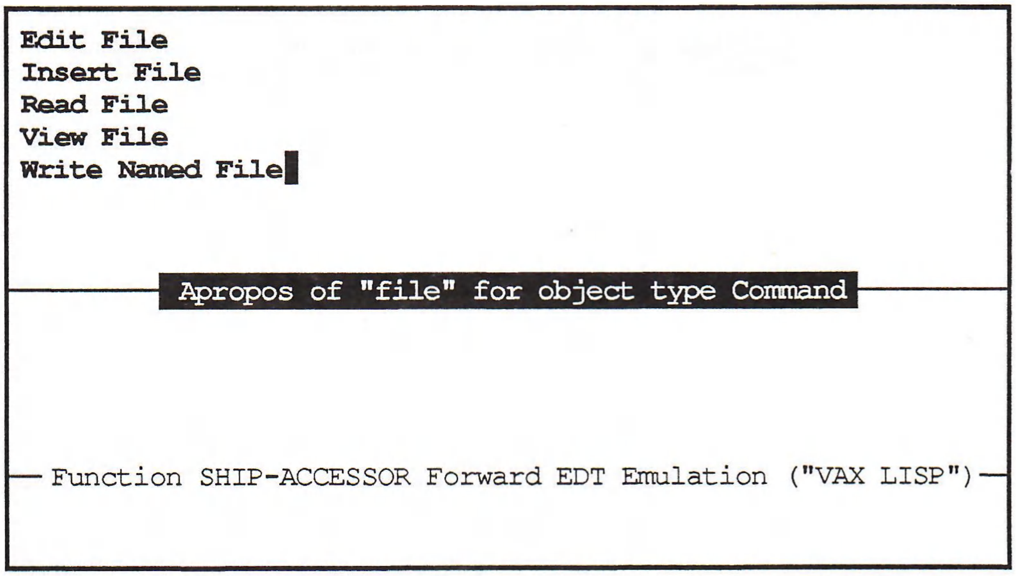
3.1.3.1 Getting Help

The Editor provides different kinds of help. You can press the Help key (either PF2 on the numeric keypad or the Help key on the LK201 or later keyboards) at any time to get help. 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, press Ctrl/X Ctrl/R.

If you press the Help key while the Editor is displaying a prompt—for example, after you have pressed 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 display shown in Figure 3–4.

Figure 3–4: Using APROPOS in the Editor



```
Edit File
Insert File
Read File
View File
Write Named File

-------- Apropos of "file" for object type Command --------




— Function SHIP-ACCESSOR Forward EDT Emulation ("VAX LISP") —
```

MLO–002787

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.

NOTE TO VWS VAXSTATION USERS

> 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 press 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".

The second way is by listing available alternatives. At any point when entering information to a prompt, you can obtain a list of the available alternatives by pressing 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 press 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 the 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 enough to let you correct the problem. If the short message is not enough, the Ctrl/X ? key sequence can 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 procedure is 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.)

NOTE TO VWS VAXSTATION USERS

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 lets you 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 E 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.
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 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.

† Keys marked with an asterisk (*) are available only on LK201 or later keyboards.

(continued on next page)

Table 3–1 (Cont.): General-Purpose Commands and Key Bindings

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.

† Keys marked with an asterisk (*) are available only on LK201 or later keyboards.

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 3–5 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 3–5. 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.

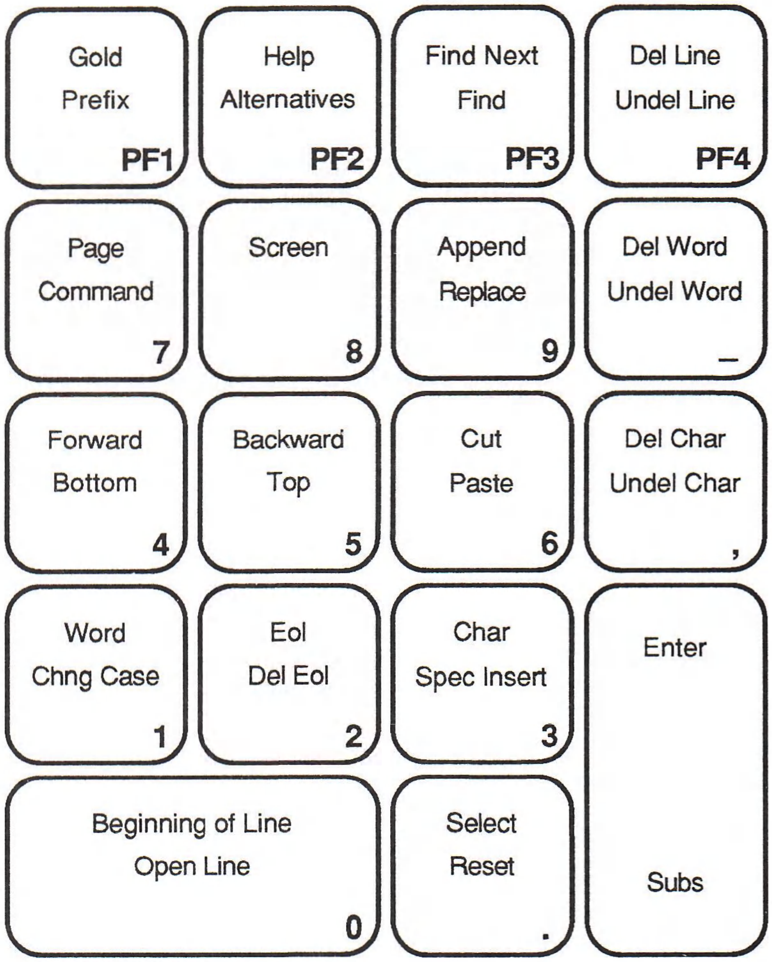
3.2.2 Inserting and Formatting Text

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 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.

Figure 3–5: Numeric Keypad



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.

MLO–002788

You can start a new line by pressing the Return key. If you press 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 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] or the command "Close Outermost Form" 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, "Indent LISP Line", indents the line that currently contains the cursor, relative to the preceding LISP code.
- The Ctrl/X Tab key sequence or the "Indent Outermost Form" command 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.

The commands "Indent LISP Region", "Indent Outermost Form", and "Indent LISP Line" (used in "VAX LISP" style), normally indent calls to user-defined macros as data. These commands will indent such calls properly if:

- You include &BODY in the argument list of the macro's definition. (Note that if the macro's argument list contains both &BODY and &WHOLE, the Editor does not properly indent a call to the macro.)
- You load the macro definition into VAX LISP before you indent the macro call in the Editor.

NOTE TO VWS VAXSTATION USERS

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 as the start of a command. 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 will handle the case of #\^X; for example, 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, but minus the leading #\. For example, Ctrl/X character in LISP prints as #\^X. The Editor displays this as <^X>.

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 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 let you 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 lets you 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 let you 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 `"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 let you 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 E 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 (VWS 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 ⌫ on the LK201 and later keyboards) 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 or later 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. 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 pressing Ctrl/X Ctrl/Space.

NOTE TO VWS VAXSTATION USERS

> 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, until you cut more text.

NOTE TO VWS VAXSTATION USERS

> 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 let you 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 press 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 lets you 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 lets you 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.
Quoted Insert	Ctrl/X \	Causes the next character typed to be inserted in the text without interpretation by the Editor.

† Keys marked with an asterisk (*) are available only on LK201 and later keyboards.

(continued on next page)

Table 3–2 (Cont.): Editing Commands and Key Bindings

Name	Binding †	Description
	Text Insertion Commands	
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	→	Moves the cursor to the next character.
Backward Character	←	Moves the cursor to the previous character.
Next Line	↓	Moves the cursor to current column in next line.
Previous Line	↑	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.
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.

† Keys marked with an asterisk (*) are available only on LK201 and later keyboards.

(continued on next page)

Table 3–2 (Cont.): Editing Commands and Key Bindings

Name	Binding †	Description
	Cursor Movement Commands	
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.
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 ⌫	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.
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.

† Keys marked with an asterisk (*) are available only on LK201 and later keyboards.

(continued on next page)

Table 3–2 (Cont.): Editing Commands and Key Bindings

Name	Binding †	Description
	Text Modification Commands	
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.
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.

† Keys marked with an asterisk (*) are available only on LK201 and later keyboards.

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.

NOTE TO VWS VAXSTATION USERS

> 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 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

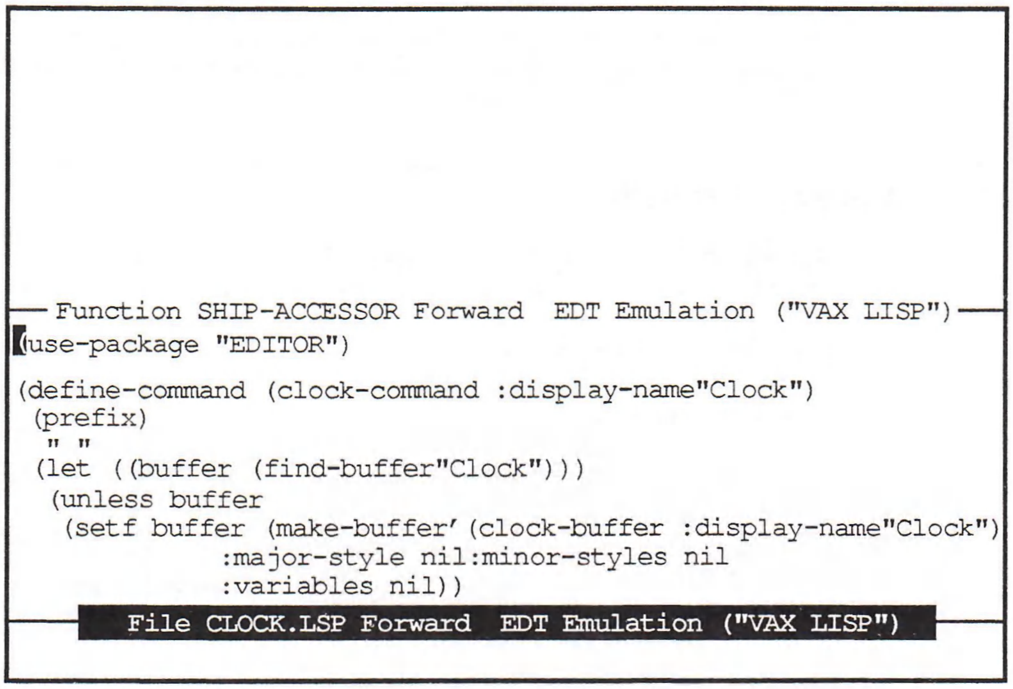
```
Lisp> (ed 'ship-accessor)
```

and then, after pausing the Editor, typed

```
Lisp> (ed "clock.lsp")
```

the screen might appear as shown in Figure 3–6.

Figure 3–6: The Editor with Two Buffers Open



```
— Function SHIP-ACCESSOR Forward  EDT Emulation ("VAX LISP") —
(use-package "EDITOR")

(define-command (clock-command :display-name"Clock")
 (prefix)
  " "
 (let ((buffer (find-buffer"Clock")))
  (unless buffer
   (setf buffer (make-buffer'(clock-buffer :display-name"Clock")
            :major-style nil:minor-styles nil
            :variables nil))
——  File CLOCK.LSP Forward  EDT Emulation ("VAX LISP")  ——
```

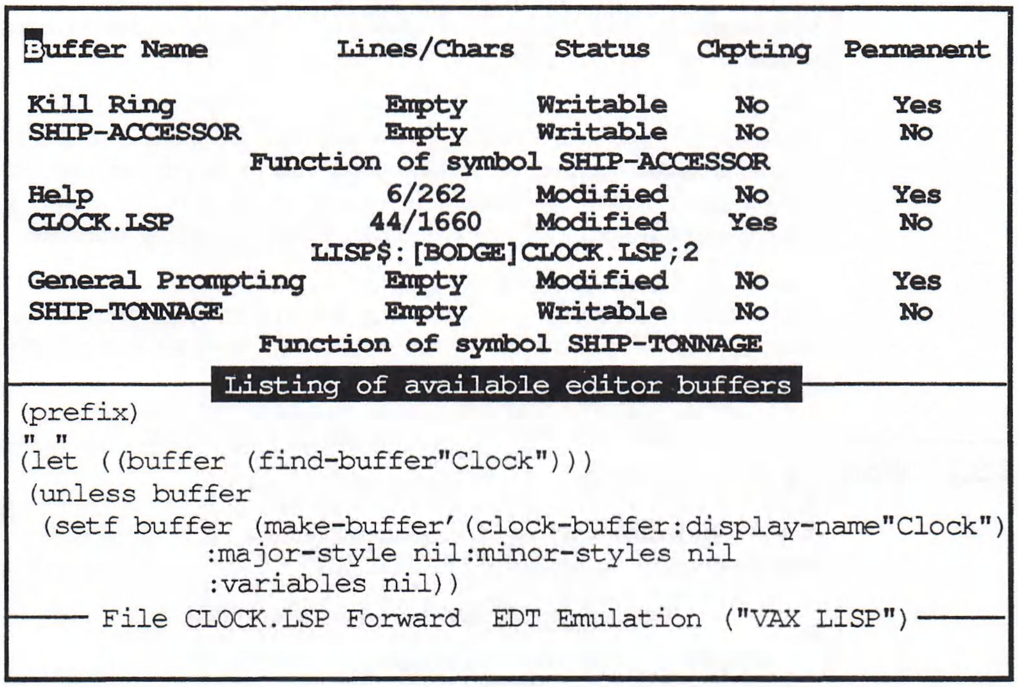
MLO–002789

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, the result of the command "List Buffers" might appear as shown in Figure 3–7.

Figure 3–7: Listing Buffers in the Editor



```
Buffer Name          Lines/Chars   Status     Ckpting   Permanent

Kill Ring                Empty     Writable      No        Yes
SHIP-ACCESSOR            Empty     Writable      No        No
               Function of symbol SHIP-ACCESSOR
Help                     6/262     Modified      No        Yes
CLOCK.LSP              44/1660     Modified      Yes       No
                   LISP$:[BODGE]CLOCK.LSP;2
General Prompting        Empty     Modified      No        Yes
SHIP-TONNAGE             Empty     Writable      No        No
               Function of symbol SHIP-TONNAGE
─────────── Listing of available editor buffers ───────────
(prefix)
" "
(let ((buffer (find-buffer"Clock")))
 (unless buffer
  (setf buffer (make-buffer'(clock-buffer:display-name"Clock")
          :major-style nil:minor-styles nil
          :variables nil))
──── File CLOCK.LSP Forward  EDT Emulation ("VAX LISP") ────
```

MLO–002790

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 let you 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 lets you 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 directly with buffers 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 press 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 let you 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 lets you 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.
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.

(continued on next page)

Table 3–3 (Cont.): Commands for Manipulating Buffers and Windows

Name	Binding	Description
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 Section 3.5.1.1 and Section 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. Section 3.5.1.1 and Section 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 lets you 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 echo:

```
<^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:

```
<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 lets you 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 and later keyboards 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

(continued on next page)

Table 3–4 (Cont.): Characters Generated by Keys

Key	Characters Generated
	Numeric Keypad Keys (LK201 and VT100)
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)
↑	`#\ESCAPE #\[ #\A`
↓	`#\ESCAPE #\[ #\B`
→	`#\ESCAPE #\[ #\C`
←	`#\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 #\~`

(continued on next page)

Table 3–4 (Cont.): Characters Generated by Keys

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 can 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, 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.)

NOTE TO VWS VAXSTATION USERS

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 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 pressing 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) completes 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 uppercase and lowercase. 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 keyboards that do not have an Escape key, Ctrl/[transmits the `#\ESCAPE` character.)

Finally, do not select a key or key sequence that is already bound to a useful command. Appendix E 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. See Section 3.1.2.

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 checks to see that the key is:

1. Bound in the context of the current buffer.
2. Bound in the context of one of the current minor styles, examining the most recently activated style first.
3. Bound in the context of the current major style.
4. 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 tc `"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 VWS VAXstation

The behavior and capabilities of the Editor when operating on a VWS 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 Before You Start

Before using the Editor on a VWS VAXstation, you must do the following:

- Activate specific lines in the VAX LISP startup file, LISPSTART.COM.
- Set the terminal emulator to NOFALLBACK mode.

These topics are discussed in detail in the following sections.

3.6.1.1 Activating Lines in LISPSTART.COM

Two lines in the installation file LISPSTART.COM affect your use of the pointer when you use the Editor on a VWS VAXstation. These lines must execute when the system starts up, before any process that runs the Editor is created. The lines are as follows:

```
$ DEFINE/SYSTEM/EXECUTIVE UIS$VT_ENABLE_OSC_STRINGS TRUE
$ DEFINE/SYSTEM/EXECUTIVE UIS$VT_ENABLE_LOCATOR TRUE
```

The first line defines a logical name that lets the Editor switch the input focus back to the terminal emulator when the Editor is paused. The second line allows the VAX LISP Editor to use a locator (mouse) on a VWS VAXstation display. Comment these lines out if LISP is not running on a VWS VAXstation.

3.6.1.2 Setting the Terminal Emulator to NOFALLBACK Mode

Before you start the Editor on a VWS VAXstation, check that the terminal emulator is in NOFALLBACK mode. When the terminal emulator is in FALLBACK mode, LISP hangs when switching input focus from the Editor to the window in which the LISP process is running.

Use the DCL command SHOW TERMINAL to show which mode is active. To set the terminal emulator to NOFALLBACK mode, enter the DCL command SET TERMINAL/NOFALLBACK.

3.6.2 Screen Appearance and Behavior

The most obvious difference in Editor behavior on a VAXstation is that the Editor occupies a separate 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.

NOTE

> Do not attempt to delete or change the size of the Editor window. Although these commands are enabled on the Window Options menu, their use is unsupported and results are unpredictable.
>
> You cannot adjust the width of the Editor window on a VWS VAXstation, using the `"Set Screen Width"` command.

3.6.3 Editing with the Pointer

The pointer cursor is the cursor that you move around the screen by moving the mouse or other pointing device. By contrast, the text insertion cursor is the blinking cursor that you move around the Editor windows by using conventional Editor commands.

You can use the 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.

The description of initial pointer bindings assumes that the pointing device is set for right-handed operation (the default). If you have set the pointing device for left-handed operation (in VAXstation setup mode), reverse the indications of right and left buttons in the following sections and in the Appendix E icon representations.

NOTE

> Pointer movement is signaled at the character-cell level instead of at the pixel level. As a result, the Editor does not know the pointer has moved unless it crosses a character-cell boundary. If you are trying to establish a select region with the pointer, and the select region fails to reach the current pointer cursor location, simply move the pointer cursor over a character-cell boundary. This will signal movement, and the Editor will update the select region.

3.6.3.1 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.

3.6.3.2 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.

3.6.3.3 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.

3.6.3.4 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.

3.6.3.5 Information About Pointer Effects

You can find out what action a pointer button invokes by moving the pointer cursor to the information area. 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.

3.6.4 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

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 states 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 4–1 lists the debugging functions and macros with a brief description of each. See the *VAX LISP/VMS Object Reference Manual* for more detailed descriptions.

Table 4–1: Debugging Functions and Macros

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

† The Inspector is available only on the DECwindows interface.

(continued on next page)

Table 4–1 (Cont.): Debugging Functions and Macros

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

This chapter provides the following:

- A list of the functions and macros 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—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 lets you edit programs and data structures.

4.1 Control Variables

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

4.2 Control Stack

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

1. Pushes the name of the function associated with the function, macro, or special form that is being called onto the stack frame.

2. Pushes the function's arguments onto the stack.
3. Creates a new stack frame.
4. Invokes the function.

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

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

Debugger commands show only significant stack frames unless you specify the ALL modifier with a Debugger command (see Section 4.4.3.1). Significant stack frames store one of the following calls:

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

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

4.3 Break Loop

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

4.3.1 Invoking the Break Loop

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

- 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 directly invoke the BREAK function while your program is being evaluated (see the *VAX LISP/VMS Object Reference Manual* 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:
your message.

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

```
Break>
```

4.3.2 Exiting the Break Loop

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

```
Break> (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 the *VAX LISP/VMS Object Reference Manual*.

4.3.3 Using the Break Loop

Once you are in the break loop, you can check what your program is doing by interacting with the LISP system as though you were in the top-level loop. For example, suppose you define a variable named *FIRST* and a function named COUNTER, which uses the variable *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) Return
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)
Continuing from Break loop . . .
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 4.4). For information on lexical environments, see *Common LISP: The Language*.

4.3.4 Break Loop Variables

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

4.4 Debugger

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

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

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

4.4.1 Invoking the Debugger

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

```
Lisp> (debug)
```

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

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

The letter *n* in the prompt represents an integer, which indicates the number of the nested command level you are in. The value of *n* increases by one each time the command level increases. For example, the top-level read-eval-print loop is level 0. If an error is invoked from the top-level loop, the Debugger displays the prompt Debug 1>. If you make another error while in 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 the *VAX LISP/VMS Object Reference Manual.*

4.4.2 Exiting the Debugger

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

```
Debug 2> quit
Debug 1>
```

If you specify the QUIT command when the Debugger command level is 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 leave this debug environment?
```

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

```
Do you really want to leave this debug environment? yes
Lisp>
```

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

```
Do you really want to leave this debug environment? no
Debug 1>
```

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

```
Debug 1> quit t
Lisp>
```

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

4.4.3 Using Debugger Commands

The Debugger commands 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 4.4.3.1.

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 Return. For example:

```
Debug 1> backtrace  Return
```

If you press only Return, the Debugger prompts you for another command.

Table 4–2 provides a summary of the Debugger commands. Detailed descriptions of the commands are in Section 4.4.3.2.

Table 4–2: Debugger Commands

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

4.4.3.1 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 Return
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 Return. For example:

```
Debug 1> down all Return
```

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

- Debugger command
- Symbol
- Form
- Function name
- Integer
- 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 in the current debugging session.

Table 4–3 provides a summary of the modifier arguments you can specify with Debugger commands.

Table 4–3: 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.

(continued on next page)

Table 4–3 (Cont.): Debugger Command Modifiers

Modifier	Command Modification
QUICK	Displays the function name in the control stack frames.
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, and some variable bindings in the control stack frames.

4.4.3.2 Debugger Commands

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

Help 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 evaluate a parenthesized form, you can specify the form and then press Return. If you want the system to evaluate a symbol, you must use the EVALUATE command. You can also evaluate expressions by entering the break loop. For information on the break loop, see Section 4.3.

EVALUATE

The EVALUATE command explicitly evaluates a specified form. You must specify the command with an argument 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 similar to the `CONTINUE` function. See *VAX LISP Implementation and Extensions to Common LISP* 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`. This command is similar to the `ABORT` function.

Error-Handling Commands

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, 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. If you omit the argument, the Debugger prompts you for it. The RETURN command can accept multiple arguments, each one an expression to be evaluated. Each evaluated form produces only one value to be returned.

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.

Movement Commands

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 from a compiled file.

HERE — A description of the current stack frame.

Backtrace Command

BACKTRACE The BACKTRACE command displays the argument list of each stack frame in the control stack, starting from the top of the stack. You can specify the command with modifiers to specify the style and extent of the backtrace. The modifiers you can specify are ALL, NORMAL, QUICK, HERE, TOP, or VERBOSE. By default, the command uses the NORMAL and the TOP modifiers. The following list describes the style or extent the BACKTRACE command uses when you specify each modifier:

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

4.4.4 Using the DEBUG-CALL Function

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

```
Lisp> (defvar adjustable-string
                (make-array 10 :element-type 'string-char
                               :initial-element #\space
                               :adjustable t))
ADJUSTABLE-STRING
Lisp> (schar adjustable-string 3)

Error in SCHAR:
Argument is not of type (SIMPLE-ARRAY CHARACTER 1): "          "

Control Stack Debugger
Apply #6: (SCHAR "          " 3)
Debug 1> (type-of (second (debug-call)))

(VECTOR CHARACTER 10)
Debug 1> ret #\space
#\SPACE
Lisp>
```

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

4.4.5 Sample Debugging Sessions

1.
```
Lisp> (defun first-element (x) (car x))
FIRST-ELEMENT
Lisp> (first-element 3)
Error in CAR: Argument must be a list: 3

Control Stack Debugger
Apply #8: (CAR 3)
Debug 1> down
Eval  #7: (CAR X)
Debug 1> down
Eval  #6: (BLOCK FIRST-ELEMENT (CAR X))
Debug 1> down
Apply #4: (FIRST-ELEMENT 3)
Debug 1> show here
It is an interpreted function
Lambda-list: FIRST-ELEMENT X
-- Arguments --
X : 3
Debug 1> set
Type of SET operation: argument
Argument Name: x
New Value: '(1 2 3)
Debug 1> where
Apply #4: (FIRST-ELEMENT (1 2 3))
Debug 1> redo
1
Lisp>
```

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

2.
```
Lisp> (defun plus-y (x) (+ x y))
PLUS-Y
Lisp> (plus-y 4)
Error in PLUS-Y: Symbol has no value: Y

Control Stack Debugger
Eval  #8:  Y
Debug 1> down
Eval  #7: (+ X Y)
Debug 1> down
Eval  #6: (BLOCK PLUS-Y (+ X Y))
Debug 1> (setf y 1)
1
Debug 1> where
Eval  #6: (BLOCK PLUS-Y (+ X Y))
Debug 1> evaluate
Evaluate: y
1
Debug 1> down
Apply #4: (PLUS-Y 4)
Debug 1> redo
5
Lisp>
```

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

3.
```
Lisp> (defun one-plus (x) (1+ x))
ONE-PLUS
Lisp> (one-plus '(1 2 3 4))
Error in 1+: Argument must be a number: (1 2 3 4)

Control Stack Debugger
Apply #8: (1+ (1 2 3 4))
Debug 1> set function
Function: 'car
Debug 1> where
Apply #8: (CAR.(1 2 3 4))
Debug 1> down
Eval #7: (1+ X)
Debug 1> up
Apply #8: (CAR (1 2 3 4))
Debug 1> redo
1
Lisp> (pprint-definition 'one-plus)
(DEFUN ONE-PLUS (X) (1+ X))
Lisp>
```

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

4.5 Stepper

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

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

The stepper prints its command interaction and output to the stream bound to the *DEBUG-IO* variable.

4.5.1 Invoking the Stepper

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

```
Lisp> (step (factorial 3))
```

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

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

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.

4.5.2 Exiting the Stepper

Usually, when you use the stepper, you press Return 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>
```

The QUIT command is described in Section 4.5.4.2.

4.5.3 Stepper Output

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

- 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))
: #9: (FACTORIAL 3)
Step> step
: : #15: (BLOCK FACTORIAL (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
Step> step
: : : #22: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step> step
: : : : #28: (<= N 1)
Step> step
: : : : : #33: N#=>#3
: : : : #28=>#NIL
: : : : #27: (* N (FACTORIAL (- N 1)))
Step> step
: : : : : #32: N#=>#3
: : : : : #31: (FACTORIAL (- N 1))
Step> step
: : : : : : #36: (- N 1)
```

```
Step> step
: : : : : : : #41: N#=>#3
: : : : : : #36=>#2
: : : : : : #37: (BLOCK FACTORIAL (IF (<= N 1) 1 (* N (FACTORIAL (- N 1)))))
Step> over
: : : : : : #37=>#2
: : : : : #31=>#2
: : : : #27=>#6
: : : #22=>#6
: : #15=>#6
: #9=>#6
6
```

Note that the FACTORIAL function is a recursive function and, in the preceding example, 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 by 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.

4.5.4 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 Return. For example:

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

If you press only Return, 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> Return
: : : #22: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step>
```

Table 4–4 provides a summary of the stepper commands. The stepper commands are described in Section 4.5.4.2.

Table 4–4: 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.
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.
QUIT	Exits the stepper.
RETURN	Forces the current stack frame to return a value.
SHOW	Displays the subform in the current stack frame.
STEP	Evaluates the subform in the current stack frame with the stepper enabled.
UP	Evaluates subforms with the stepper disabled until the stepper gets back to a subform that contains the subform in the current stack frame.

4.5.4.1 Arguments

Stepper command arguments modify the operations the stepper commands perform. 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 Return. For example:

```
Step> evaluate (<= n 1) Return
```

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

```
Step> evaluate Return
Evaluate: (<= N 1)
```

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

4.5.4.2 Stepper Commands

The stepper provides 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 Return. 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. If you do not specify the command with an argument, you are prompted for one. The argument must be the form you want the system to evaluate. The system evaluates the form in the lexical environment of the form currently being stepped.

Debugger Command

DEBUG

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

Display Command

SHOW

The SHOW command displays the subform 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. If you do not specify the command with an argument, you are prompted for one. The argument 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 Return.

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 one level deep. You cannot specify the command with an optional argument.

4.5.5 Using Stepper Variables

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

4.5.5.1 *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.

4.5.5.2 *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 evaluation of forms in the context of the stepped form.

4.5.5.3 Example Use of Stepper Variables

The following example shows how to use the *STEP-FORM* and *STEP-ENVIRONMENT* special variables.

```
Lisp> (setf x "top level value of x")
"top level value of x"
Lisp> (defun fibonacci (x)
        (if (< x 3) 1
            (+ (fibonacci (- x 1)) (fibonacci (- x 2)))))
FIBONACCI
Lisp> (step (fibonacci 5))
#4: (FIBONACCI 5)
Step> step
: #10: (BLOCK FIBONACCI (IF (< X 3) 1
                            (+ (FIBONACCI (- X 1))
                               (FIBONACCI (- X 2)))))
Step> step
: : #14: (IF (< X 3) 1 (+ (FIBONACCI (- X 1))
                          (FIBONACCI (- X 2))))
Step> step
: : : #18: (< X 3)
Step> step
: : : : #22: X => 5
: : : #18 => NIL
: : : #17: (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))
Step> step
: : : : #21: (FIBONACCI (- X 1))
Step> step
: : : : : #25: (- X 1)
Step> step
: : : : : : #29: X => 5
: : : : : #25 => 4
: : : : : #27: (BLOCK FIBONACCI (IF (< X 3) 1
                                     (+ (FIBONACCI (- X 1))
                                        (FIBONACCI (- X 2)))))
Step> step
: : : : : : #31: (IF (< X 3) 1
                      (+ (FIBONACCI (- X 1))
                         (FIBONACCI (- X 2))))
Step> step
: : : : : : : #35: (< X 3)
Step> step
: : : : : : : : #39: X => 4
: : : : : : : #35 => NIL
: : : : : : : #34: (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))
Step> step
: : : : : : : : #38: (FIBONACCI (- X 1))
Step> eval *step-form*
(FIBONACCI (- X 1))
```

```
Step> step
: : : : : : : : : #42: (- X 1)
Step> step
: : : : : : : : : : #46: X => 4
: : : : : : : : : #42 => 3
: : : : : : : : : #44: (BLOCK FIBONACCI
                          (IF (< X 3) 1
                            (+ (FIBONACCI (- X 1))
                               (FIBONACCI (- X 2)))))
Step> eval *step-form*
(BLOCK FIBONACCI
  (IF (< X 3) 1 (+ (FIBONACCI (- X 1)) (FIBONACCI (- X 2)))))
Step> step
: : : : : : : : : : #48: (IF (< X 3) 1
                             (+ (FIBONACCI (- X 1))
                                (FIBONACCI (- X 2))))
Step> step
: : : : : : : : : : : #52: (< X 3)
Step> step
: : : : : : : : : : : : #56: X => 3
: : : : : : : : : : : #52 => NIL
: : : : : : : : : : : #51: (+ (FIBONACCI (- X 1))
                              (FIBONACCI (- X 2)))
Step> step
: : : : : : : : : : : : #55: (FIBONACCI (- X 1))
Step> eval x
3
Step> (eval 'x)
"top level value of x"
Step> eval *step-form*
(FIBONACCI (- X 1))
Step> (evalhook 'x nil nil nil)
"top level value of x"
Step> (evalhook 'x nil nil *step-environment*)
3
Step> (evalhook (cadr *step-form*) nil nil *step-environment*)
2
Step> step
: : : : : : : : : : : : : #59: (- X 1)
Step> step
: : : : : : : : : : : : : : #63: X => 3
: : : : : : : : : : : : : #59 => 2
: : : : : : : : : : : : : #61: (FIBONACCI
                                  (IF (< X 3) 1
                                      (+ (FIBONACCI (- X 1))
                                         (FIBONACCI (- X 2)))))
Step> finish
5
```

This example shows that the *STEP-FORM* special variable is bound to the form being evaluated while stepping. The example also shows that the *STEP-ENVIRONMENT* special variable is bound to the lexical environment in which the currently stepped form is being evaluated.

The call to EVALHOOK evaluates the form (- X 1) in the lexical environment of the stepper; that is, with the local binding of *x*. A call to EVALHOOK with a null environment specified shows that *x*'s value in the null lexical environment differs from that in the stepper. The EVAL command uses the *STEP-ENVIRONMENT* environment; the EVAL function uses the null lexical environment.

4.5.6 Sample Stepper Sessions

1.
```
Lisp> (defun first-element (x) (car x))
FIRST-ELEMENT
Lisp> (setf my-list '(first second third))
(FIRST SECOND THIRD)
Lisp> (step (first-element my-list))
#10: (FIRST-ELEMENT MY-LIST)
Step> step
: #15: MY-LIST => (FIRST SECOND THIRD)
: #17: (BLOCK FIRST-ELEMENT (CAR X))
Step> step
: : #22: (CAR X)
Step> evaluate (car x)
FIRST
Step> finish
FIRST
Lisp>
```

2.
```
Lisp> (defun plus-y (x) (+ x y))
PLUS-Y
Lisp> (setf y 5)
5
Lisp> (step (plus-y 10))
#10: (PLUS-Y 10)
Step> step
: #17: (BLOCK PLUS-Y (+ X Y))
Step> evaluate
Evaluate: (+ x y)
15
Step> step
: : #22: (+ X Y)
Step> backtrace
(+ X Y)
(BLOCK PLUS-Y (+ X Y))
(PLUS-Y 10)
Step> show
(+ X Y)
Step> over
: : #22 => 15
: #17 => 15
#10 => 15
15
Lisp>
```

3.

```
Lisp> (defun addition (x) (+ x y))
ADDITION
Lisp> (setf y 5)
5
Lisp> (step (addition 4))
#10: (ADDITION 4)
Step> step
: #17: (BLOCK ADDITION (+ X Y))
Step> step
: : #22: (+ X Y)
Step> backtrace
(+ X Y)
(BLOCK ADDITION (+ X Y))
(ADDITION 4)
Step> evaluate
Evaluate: (+ x y)
9
Step> STEP
: : : #27: X=>#4
: : : #27: Y=>#5
: : #22=>#9
: #17=>#9
#10=>#9
9
Lisp>
```

4.6 Tracer

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

NOTE

You cannot trace special forms.

4.6.1 Enabling the Tracer

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

```
Lisp> (trace factorial addition counter)
(FACTORIAL ADDITION COUNTER)
```

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

If you call the TRACE macro without an argument, it returns 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 the *VAX LISP/VMS Object Reference Manual*.

4.6.2 Disabling the Tracer

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

```
Lisp> (untrace factorial addition counter)
(FACTORIAL ADDITION COUNTER)
```

You can disable tracing for all the functions for which tracing is enabled by calling the UNTRACE macro without an argument.

The UNTRACE macro is described in *Common LISP: The Language.*

4.6.3 Tracer Output

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

- 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.

4.6.4 Tracer Options

You can modify the output of the tracer by specifying options in the call to the TRACE macro. Each option consists of a keyword-value pair. The 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 :POST-DEBUG-IF	Invoke the Debugger
:PRINT :PRE-PRINT :POST-PRINT	Add information to tracer output
:STEP-IF	Invokes the stepper
:SUPPRESS-IF	Removes information from tracer output
:DURING	Determines when a function or macro is traced

4.6.4.1 Invoking the Debugger

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

4.6.4.2 Adding Information to Tracer Output

You can add information to tracer output by specifying the :PRINT, :PRE-PRINT, or :POST-PRINT keyword. You must specify these keywords with a list of forms. The LISP system evaluates each form in the list and the tracer displays their return values before, after, or before and after each call to the function or macro being traced. The tracer displays the values one per line and indents them to match other tracer output. If the forms to be evaluated cause an error, the Debugger is invoked.

4.6.4.3 Invoking the Stepper

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

4.6.4.4 Removing Information from Tracer Output

You can remove information from tracer output by specifying the :SUPPRESS-IF keyword. You must specify this keyword with a form. The LISP system evaluates the form before each call to the function or macro 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.

4.6.4.5 Defining When a Function or Macro Is Traced

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

4.6.5 Tracer Variables

You can use two special variables with the TRACE macro: *TRACE-CALL* and *TRACE-VALUES*. These are helpful debugging tools. 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.

4.6.5.1 *TRACE-CALL*

The *TRACE-CALL* variable is bound to the function or macro call being traced. The following example shows how to use the variable:

```
Lisp> (defun fibonacci (x)
        (if (< x 3) 1
            (+ (fibonacci (- x 1)) (fibonacci (- x 2)))))
FIBONACCI
Lisp> (trace (fibonacci
                :pre-debug-if (< (second *trace-call*) 2)
                :suppress-if t))
(FIBONACCI)
Lisp> (fibonacci 5)
Control Stack Debugger
Apply #30: (DEBUG)
Debug 1> down
Eval  #27: (FIBONACCI (- X 2))
Debug 1> down
Eval  #26: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
Debug 1> down
Eval #25: (IF (< X 3) 1
              (+ (FIBONACCI (- X 1))
                 (FIBONACCI (- X 2))))
Debug 1> down
Eval #24: (BLOCK FIBONACCI
            (IF (< X 3) 1
                (+ (FIBONACCI (- X 1))
                   (FIBONACCI (- X 2)))))
Debug 1> down
Apply #22: (FIBONACCI 3)
Debug 1> (cadr (debug-call))
3
Debug 1> continue
Control Stack Debugger
Apply #22: (DEBUG)
Debug 1> continue
5
```

- In this example, FIBONACCI is first defined.
- Then, the TRACE macro is called for FIBONACCI. TRACE is specified to invoke the Debugger if the first argument to FIBONACCI (the function call being traced) is less than 2. Since the PRE-DEBUG-IF option is specified, the Debugger is invoked before the call to FIBONACCI. As the :SUPPRESS-IF option has a value of T, calls to FIBONACCI do not cause any trace output.
- The DOWN command moves the pointer down the control stack.
- The DEBUG-CALL function returns a list representing the current debug frame function call. In this case, the CADR of the list is 3. This accesses the first argument to the function in the current stack frame.
- Finally, the CONTINUE command continues the evaluation of FIBONACCI.

4.6.5.2 *TRACE-VALUES*

The *TRACE-VALUES* variable is bound to the list of values returned by a traced function. Consequently, the variable can be used only with the :POST- options to the TRACE macro. Before being bound to the return values, the variable returns NIL. The following example shows how to use the variable:

```
Lisp> (trace (fibonacci
              :post-debug-if (> (first *trace-values*) 2)))
(FIBONACCI)
Lisp> (fibonacci 5)
#5: (FIBONACCI 5)
. #13: (FIBONACCI 4)
. . #21: (FIBONACCI 3)
. . . #29: (FIBONACCI 2)
. . . #29=> 1
. . . #29: (FIBONACCI 1)
. . . #29=> 1
. . #21=> 2
. . #21: (FIBONACCI 2)
. . #21=> 1
Control Stack Debugger
Apply #14: (DEBUG)
Debug 1> backtrace
-- Backtrace start --
Apply #14: (DEBUG)
Eval  #11: (FIBONACCI (- X 1))
Eval  #10: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
Eval  #9: (IF (< X 3) 1
              (+ (FIBONACCI (- X 1))
                 (FIBONACCI (- X 2))))
Eval  #8: (BLOCK FIBONACCI
             (IF (< X 3) 1
                 (+ (FIBONACCI (- X 1))
                    (FIBONACCI (- X 2)))))
Apply #6: (FIBONACCI 5)
Eval  #3: (FIBONACCI 5)
Apply #1: (EVAL (FIBONACCI 5))
-- Backtrace end --
Apply #14: (DEBUG)
Debug 1> continue
. #13=> 3
. #13: (FIBONACCI 3)
. . #21: (FIBONACCI 2)
. . #21=> 1
. . #21: (FIBONACCI 1)
. . #21=> 1
. #13=> 2
Control Stack Debugger
Apply #6: (DEBUG)
Debug 1> continue
#5=> 5
5
```

TRACE is called for FIBONACCI (the same function as in the previous example) to start the Debugger if the value returned exceeds 2. The value returned exceeds 2 twice—once when it returns 3 and at the end when it returns 5.

4.7 The Editor

The VAX LISP Editor is a powerful, extensible editor that lets you 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/VMS Editor Programming Guide*.

Part II
Using VAX LISP Facilities on the DECwindows Interface

Chapter 5

The DECwindows Interface to VAX LISP

You can run VAX LISP on a VAX workstation as a DECwindows application. With the DECwindows interface to LISP, you choose commonly used LISP operations and utilities from pull-down menus. Most LISP utilities run in separate windows, so that utility commands and messages are separate from interactions with LISP. All VAX LISP utilities that are provided with the terminal interface are also provided with the DECwindows interface. These utilities are:

- The Listener, which runs the READ-EVAL-PRINT loop, accepts LISP forms, evaluates them, and prints messages as a result of those forms
- The Editor, which lets you write LISP source code within the LISP environment, load that code into LISP, and run it
- The Debugger, which helps you determine where programming errors occur

In addition to these utilities, the DECwindows interface provides an Inspector, which lets you examine and modify static LISP objects. This utility is not available with the terminal interface to LISP.

This chapter takes you through a sample programming session with VAX LISP. It shows you how to do the following:

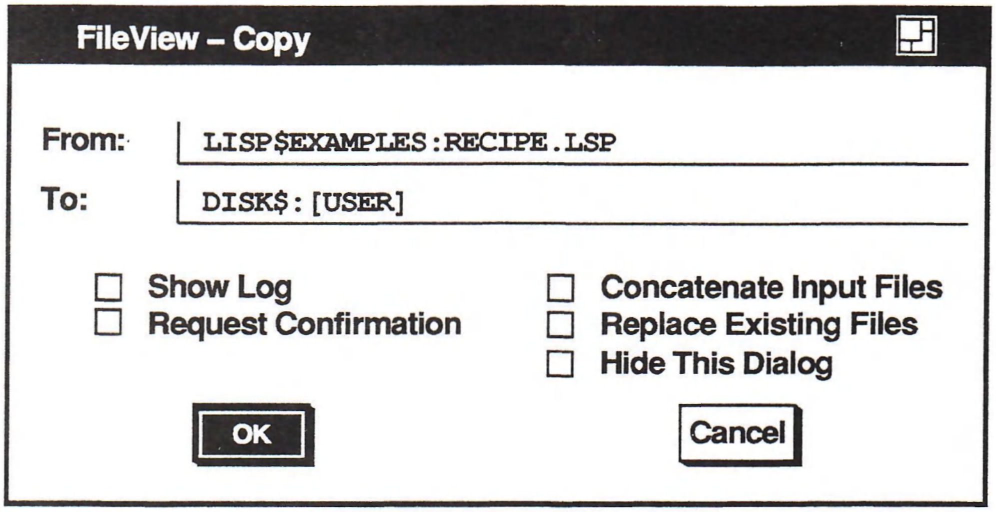
- Load source files and enter LISP forms with the Listener
- Examine LISP objects with the Inspector
- Define new functions with the Editor
- Locate programming errors with the Debugger

Subsequent chapters describe each of these utilities in detail.

Throughout the sample session, this chapter uses a simple program to show each stage of program development. The sample program defines LISP objects to represent ingredients used in recipes. A LISP function calculates the number of calories per serving for each ingredient or recipe.

When VAX LISP is installed, a number of sample programs are placed in the directory defined by the logical name LISP$EXAMPLES. Among these programs is a file called RECIPE.LSP, which contains the sample program. To try the examples in this chapter, copy the source file from the LISP$EXAMPLES directory to your working directory. Figure 5–1 shows the FileView-Copy dialog box as it would appear when using FileView to copy the RECIPE.LSP file from LISP$EXAMPLES to a working directory called DISK$:[USER].

Figure 5–1: Copying a Sample Source File



From: LISP$EXAMPLES:RECIPE.LSP

To: DISK$:[USER]

□ Show Log
□ Request Confirmation
□ Concatenate Input Files
□ Replace Existing Files
□ Hide This Dialog

MLO–002855

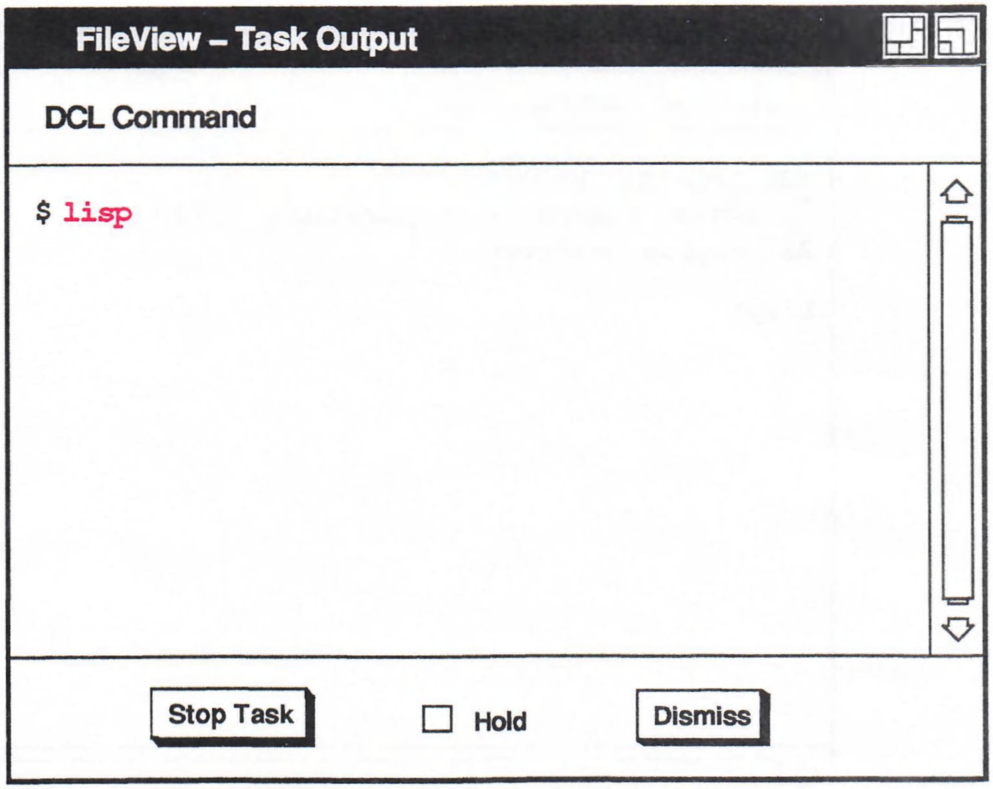
5.1 Invoking VAX LISP

There are a number of ways to invoke VAX LISP:

- From a terminal or DECterm window, use the same command line you would use on a regular terminal.
- From a FileView window, select DCL Command in the Utilities menu. You can then use the same command line you would use from an ordinary terminal or DECterm window.

Figure 5–2 shows a standard FileView window that you can use to invoke VAX LISP. See your *VMS DECwindows User's Guide* for information on using FileView windows. For more information on invoking the DECwindows interface to VAX LISP, see Chapter 6.

Figure 5–2: Invoking LISP



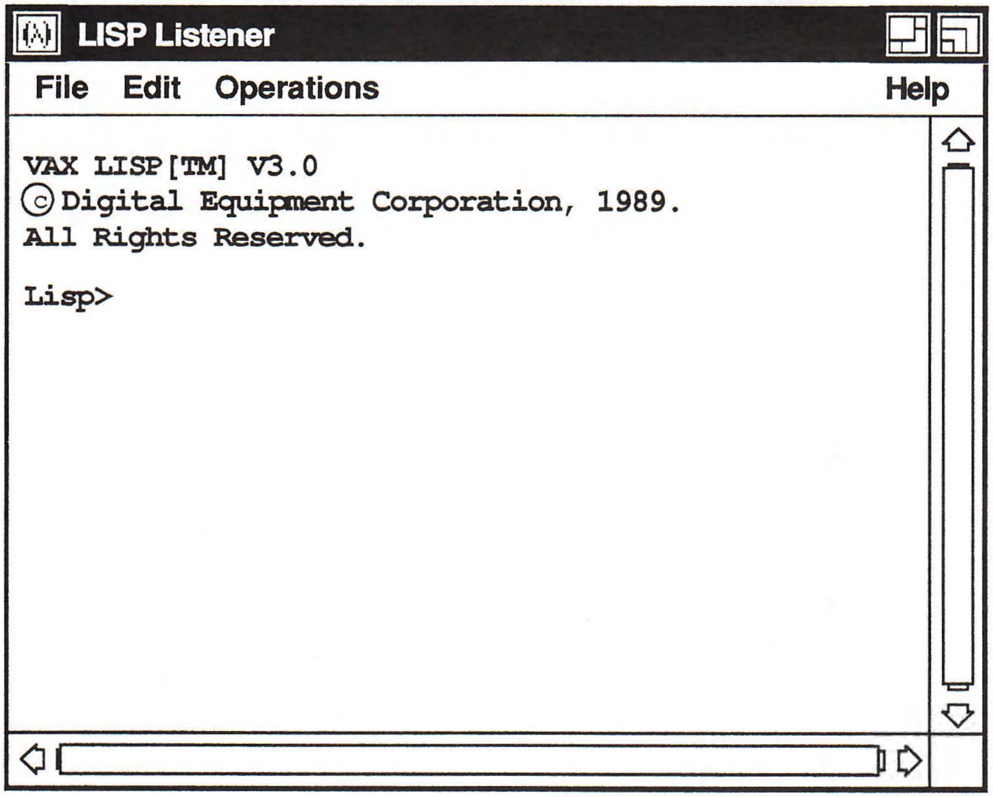
```
$ lisp
```

MLO–002856

5.2 Using the Listener

The Listener is the first utility that appears. It occupies its own window, shown in Figure 5–3.

Figure 5–3: Listener Window

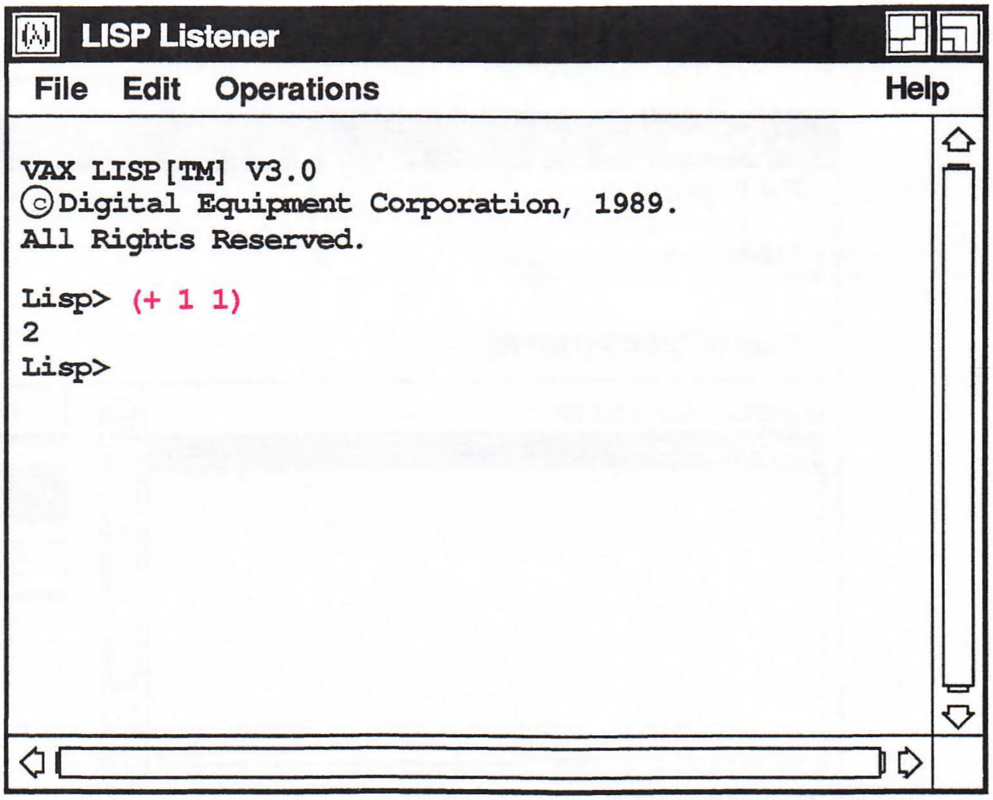


```
VAX LISP[TM] V3.0
©Digital Equipment Corporation, 1989.
All Rights Reserved.

Lisp>
```

MLO–002857

You can enter any LISP form at the `Lisp>` prompt. For example, LISP can evaluate an arithmetic expression such as the one shown in Figure 5–4.

Figure 5–4: Evaluating a LISP Form

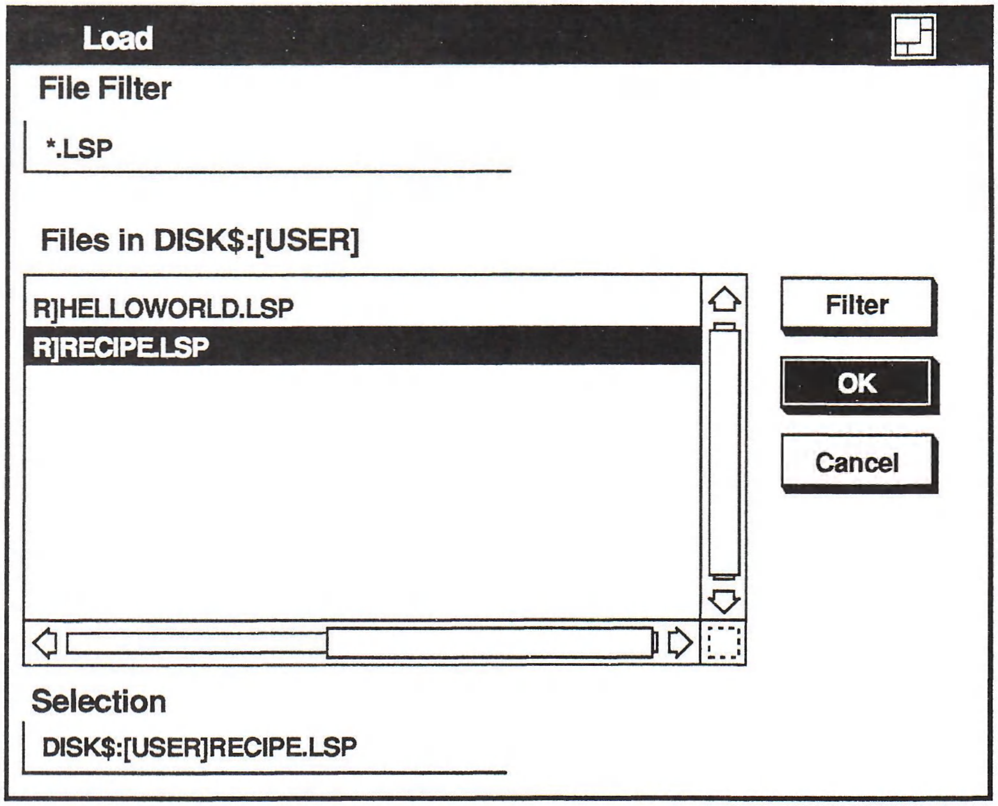


```
VAX LISP[TM] V3.0
©Digital Equipment Corporation, 1989.
All Rights Reserved.

Lisp> (+ 1 1)
2
Lisp>
```

MLO–002858

Commonly used LISP operations are available through pull-down menus at the top of the Listener window. For example, to load source files, pull down the File menu, then choose Load. The Load menu choice produces a list of LISP source files displayed in a window called a "file selection dialog box," shown in Figure 5–5.

Figure 5–5: Loading a LISP Source File



Load

File Filter

*.LSP

Files in DISK$:[USER]

R]HELLOWORLD.LSP

R]RECIPE.LSP

Selection

DISK$:[USER]RECIPE.LSP

MLO–002859

To select the source file for the sample program, scroll through the list of files until the pointer points to RECIPE.LSP. Click MB1 to highlight the file name. RECIPE.LSP appears as the selection at the bottom of the window. Next, you can either click on the file name again or position the pointer over the OK button and click MB1. The file selection dialog box disappears, LISP loads the source file, and the Listener window reappears.

For more information on the Listener, see Chapter 7.

5.3 Using the Inspector

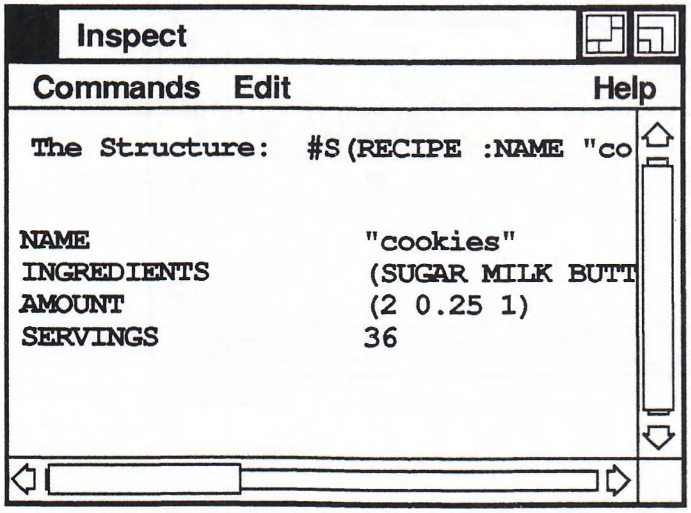
The sample program calculates the number of calories for certain foods. Foods like milk, bread, butter, and eggs are "ingredients" that can be combined into "recipes." The sample program uses symbols to represent ingredients and structures to represent recipes. You can examine this data with the Inspector.

For example, you can invoke the Inspector from the Listener window with the LISP INSPECT function and inspect the COOKIES structure defined in RECIPE.LSP:

```
Lisp> (inspect cookies)
```

When LISP evaluates this form, it opens an Inspector window and an Inspector History window. The Inspector window, in Figure 5–6, shows that the structure has the slots :NAME, :INGREDIENTS, :AMOUNT, and :SERVINGS and it shows the values assigned to those slots for the COOKIES recipe.

Figure 5–6: Inspecting a Structure

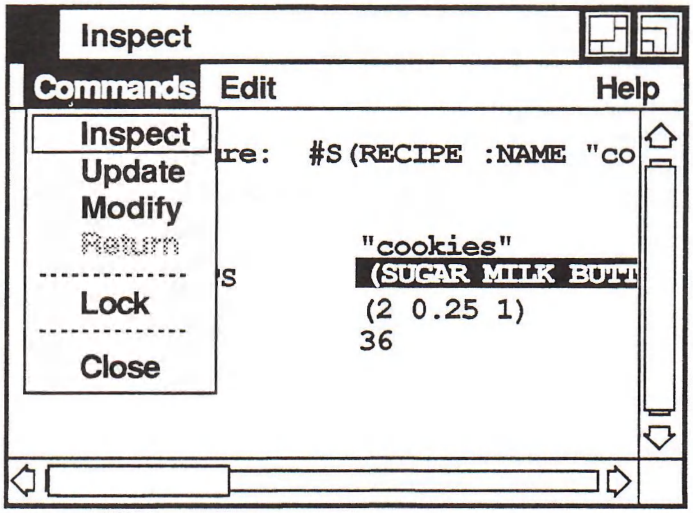


```
The Structure:   #S(RECIPE :NAME "co

NAME                     "cookies"
INGREDIENTS              (SUGAR MILK BUTT
AMOUNT                   (2 0.25 1)
SERVINGS                 36
```

MLO–002860

You can also use the mouse to select objects that you want to inspect. For example, if you click MB1 anywhere within the list (SUGAR MILK BUTTER), the Inspector highlights the list. You can inspect that list by choosing Inspect from the Commands menu at the top of the window, as shown in Figure 5–7.

Figure 5–7: Choosing the Inspect Menu Item

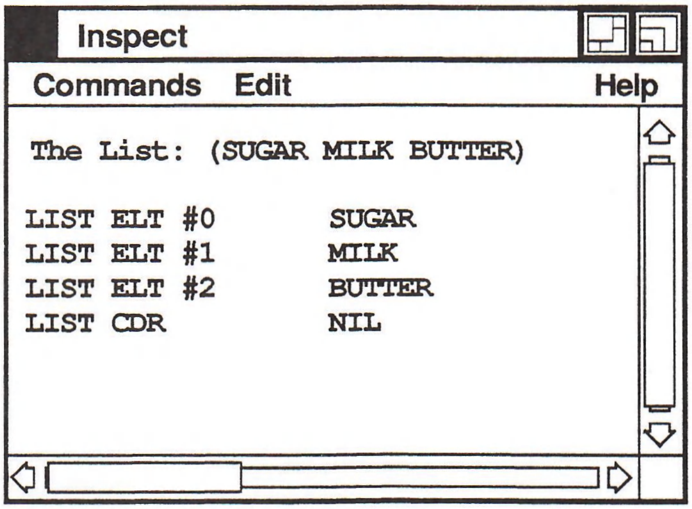


```
re:   #S(RECIPE :NAME "co
s              "cookies"
               (SUGAR MILK BUTT
               (2 0.25 1)
               36
```

MLO–002861

When you choose Inspect from the menu, the Inspector opens a second window and displays information about the list, as shown in Figure 5–8.

Figure 5–8: Inspecting a List



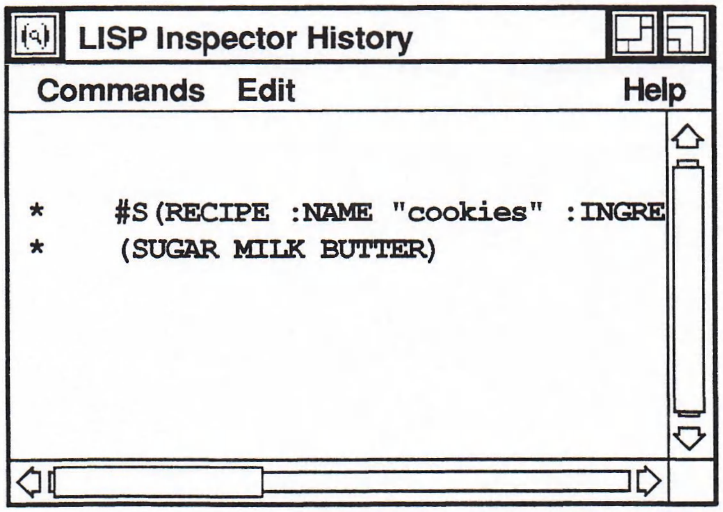
```
The List: (SUGAR MILK BUTTER)

LIST ELT #0       SUGAR
LIST ELT #1       MILK
LIST ELT #2       BUTTER
LIST CDR          NIL
```

MLO–002862

By default, the Inspector can open up to five Inspect windows at a time. When you inspect more than five objects, the Inspector reuses those windows. If you do not want to reuse a window, so that the object in it will remain visible, you can lock it. The Inspector will create more unlocked windows until it reaches its maximum of five. You may change the maximum number of unlocked windows that the Inspector can open. Appendix B describes how you can customize the Inspector and the other DECwindows utilities.

In addition to the Inspect windows, the Inspector keeps a running history of the objects that you have inspected. This information is displayed in the Inspector History window. Figure 5–9 shows the History window that the Inspector displays for the objects inspected so far. The asterisk before the object name indicates that the window is currently displayed and unlocked.

Figure 5–9: Inspector History Window



```
*     #S(RECIPE :NAME "cookies" :INGRE
*     (SUGAR MILK BUTTER)
```

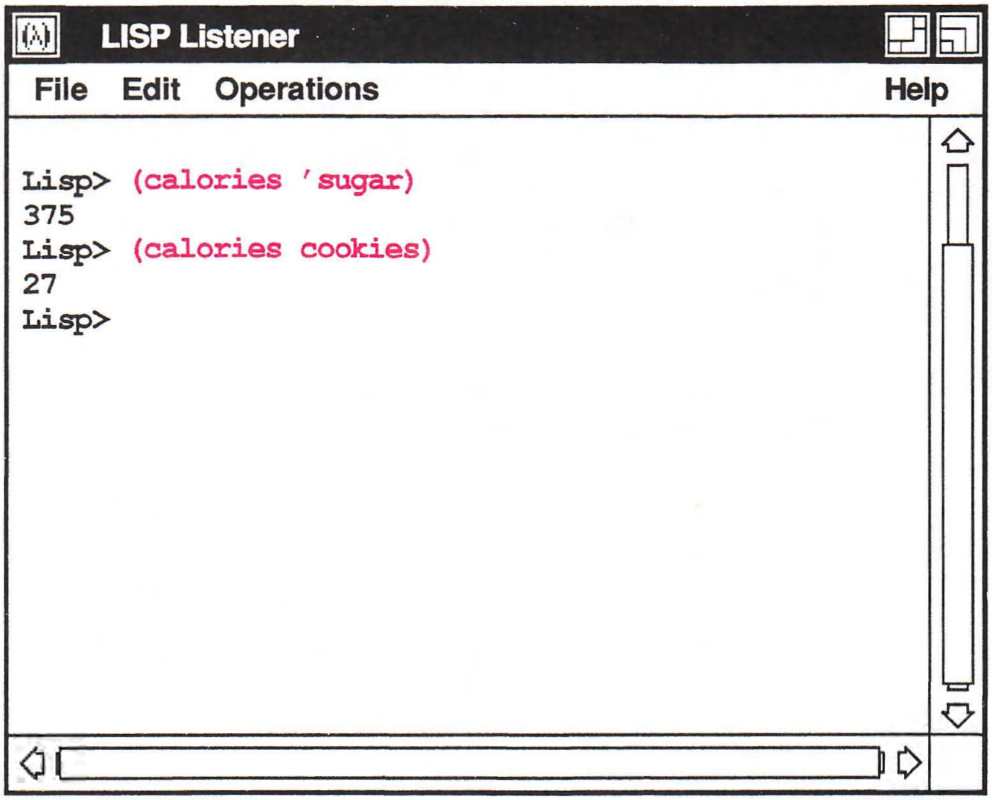
MLO–002863

For more information on the Inspector, see Chapter 9.

5.4 Running a Sample Function

The sample program contains a function called CALORIES, which calculates the number of calories per unit of measure for an ingredient or per serving for a recipe. Figure 5–10 shows what the CALORIES function returns for the symbol SUGAR and the recipe structure COOKIES.

Figure 5–10: Running a LISP Function



```
Lisp> (calories 'sugar)
375
Lisp> (calories cookies)
27
Lisp>
```

MLO–002864

The sample program is only the beginning of what could be a menu-planning application. It contains a small amount of information about ingredients and recipes and the formula for calculating calories. The sections that follow describe how to add a pretty-printing function to this program.

5.5 Using the Editor

VAX LISP has a built-in editor for creating source files. Although you can create LISP source files with any editor you like, the LISP Editor contains language-specific features that other editors do not have.

The Tab key indents lines appropriately, according to the cursor's position within a LISP form, as follows:

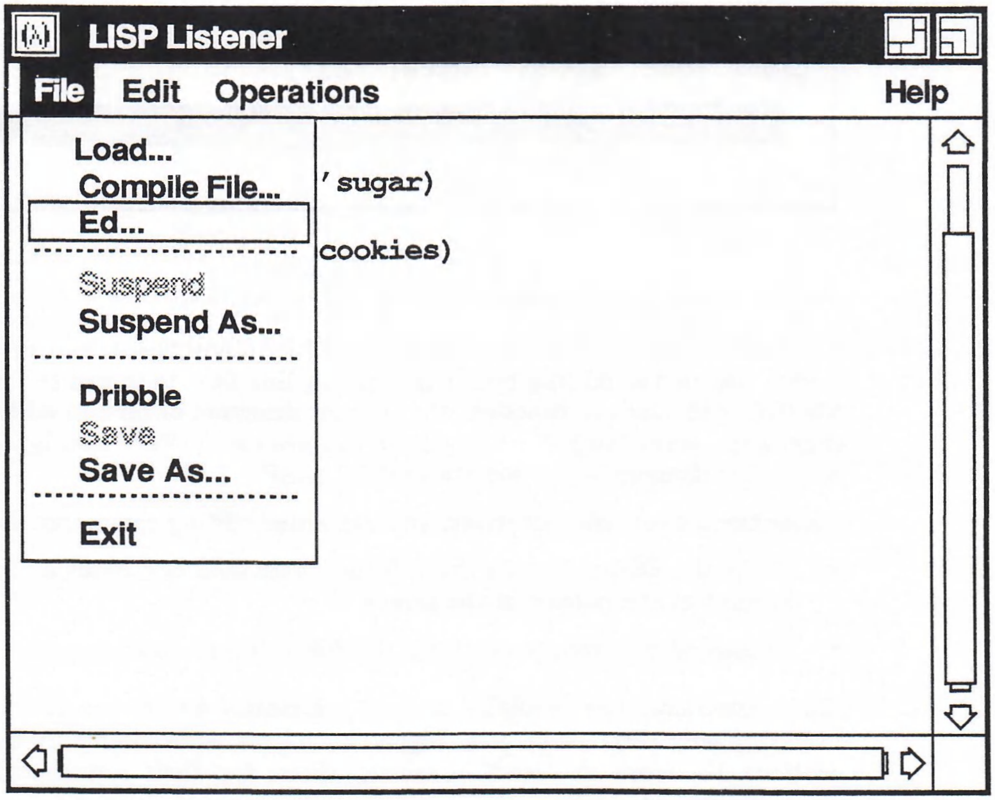
- Nested LISP forms are indented below their containing form.
- Forms at the same level of nesting are aligned.
- Function arguments on separate lines are aligned below each other and indented below their containing form.

The Editor is also sensitive to parentheses. Whenever you type a closing parenthesis, it highlights the corresponding opening parenthesis. If the opening parenthesis has scrolled off the window, the Editor displays it in the command area at the bottom of the screen.

You can invoke the Editor from the `Lisp>` prompt with the `ED` function, giving the name of the file or function you want to edit. This is the only way to invoke the Editor when using the terminal interface to VAX LISP.

The DECwindows interface also lets you invoke the Editor as a menu choice from either the File or the Operations menu. The File menu item, shown in Figure 5–11, is similar to Load. It brings up a file selection box that allows you to specify a file to edit. To add a function definition to RECIPE.LSP, choose RECIPE.LSP in the file selection dialog box.

Figure 5–11: Choosing the ED Function



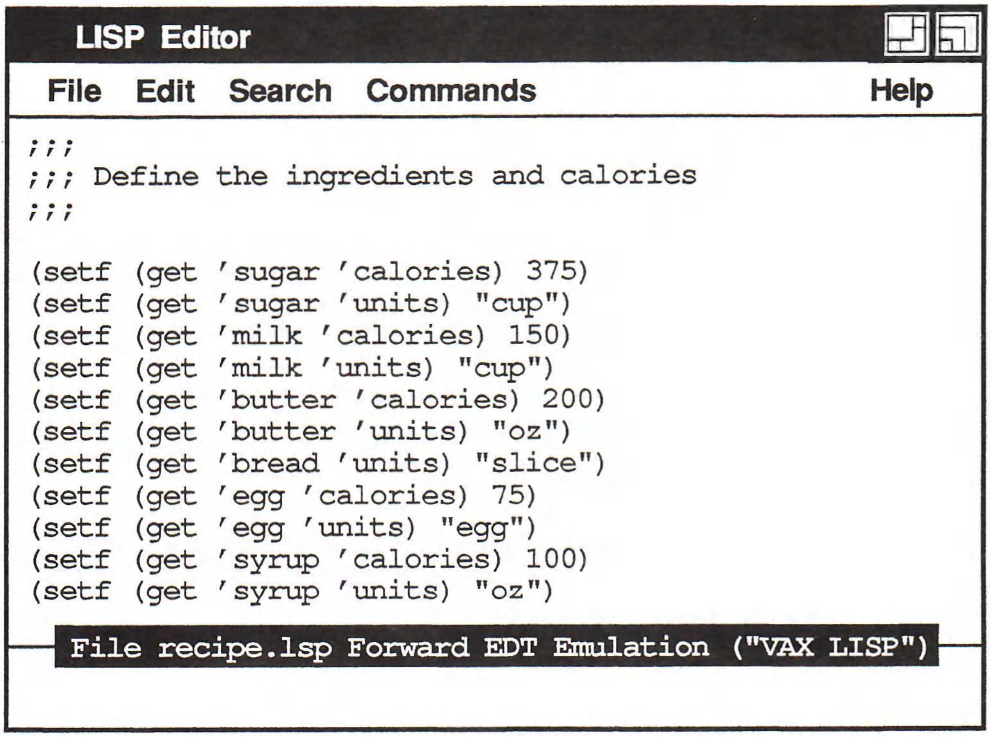
```
'sugar)
cookies)
```

MLO–002865

Choosing the ED item from the Operations menu is similar to running the ED function. Both let you edit the currently selected object.

After you invoke the Editor, it places the file or function in an editing buffer, as shown in Figure 5–12.

Figure 5–12: Editing a File



```
;;;
;;; Define the ingredients and calories
;;;

(setf (get 'sugar 'calories) 375)
(setf (get 'sugar 'units) "cup")
(setf (get 'milk 'calories) 150)
(setf (get 'milk 'units) "cup")
(setf (get 'butter 'calories) 200)
(setf (get 'butter 'units) "oz")
(setf (get 'bread 'units) "slice")
(setf (get 'egg 'calories) 75)
(setf (get 'egg 'units) "egg")
(setf (get 'syrup 'calories) 100)
(setf (get 'syrup 'units) "oz")
```

```
File recipe.lsp Forward EDT Emulation ("VAX LISP")
```

MLO–002866

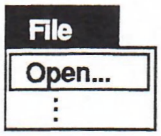
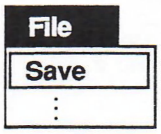
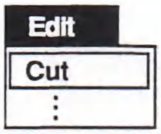
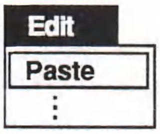
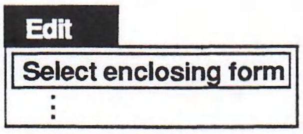
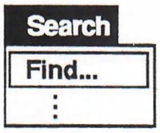
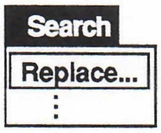
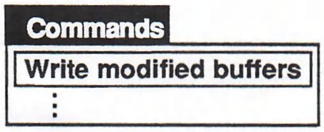
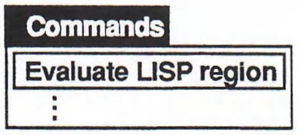
The Editor displays the file or function at the top of the screen in an editing buffer. Below the editing buffer is a status line that tells you the name of the file (RECIPE.LSP) or function, the current direction of certain editing commands (Forward), what "major" editing style you are using (EDT Emulation), and what "minor" editing style you are using (VAX LISP).

Using the default editing styles, you can enter editing commands in two ways:

- While the Editor has the input focus, press Ctrl/Z and enter a command at the prompt at the bottom of the screen.
- Choose editing commands from the Editor's pull-down menus.

Many commands are available as both commands and menu choices; others are available only through menus. Figure 5–13 lists some commonly used editing operations, the menu choices that execute them, and their corresponding command lines, if available.

Figure 5–13: Basic Editing Commands

Menu Choice	Command	Description
	Edit File	Opens a file for editing.
	Write Current Buffer	Writes the contents of the buffer to the corresponding LISP object or file.
	No command equivalent	Places the selected text in the clipboard and removes the selected text from its current location.
	No command equivalent	Places the current contents of the clipboard at the cursor location within the text.
	Select Enclosing Form	Selects the text of the innermost LISP form containing the cursor or selected region.
	EDT Query Search	Searches for a text string that you specify, either forward or backward in the file.
	Query Search Replace	Replaces all or single occurrences of a string with another string.
	Write Modified Buffers	Writes the contents of all modified buffers to the corresponding LISP objects or to new versions of their corresponding files.
	Evaluate LISP Region	Evaluates the selected text and makes the results available to the Listener.

MLO–002867

For more information on the Editor, see Chapter 8 of this manual.

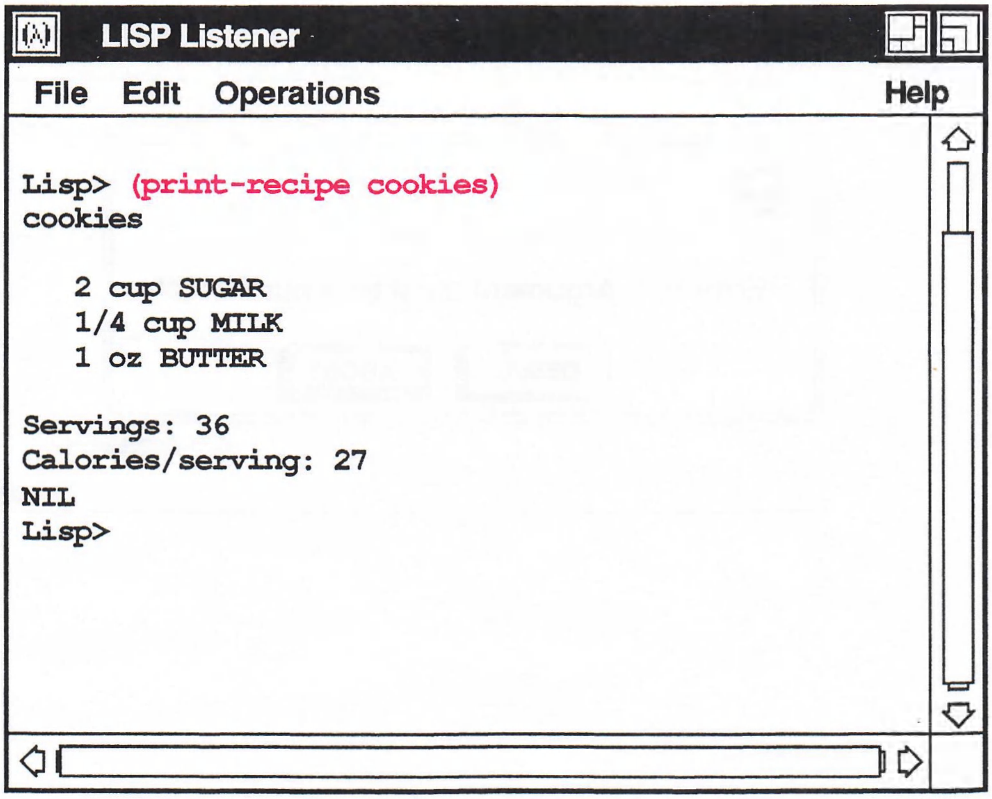
Press the Next Screen key until the cursor is at the end of the file. Then, try entering the PRINT-RECIPE function shown in Example 5–1. This function pretty-prints the information returned by CALORIES.

Example 5–1: Defining a LISP Function

```
(defun print-recipe (recipe)
"pretty-print a recipe, number of servings, and calories/serving"
  (unless (recipe-p recipe)
        (error "Not a recipe: ~S" recipe))
  (format t "~A~2%" (recipe-name recipe))
  (mapc #'(lambda (a u i) (format t "~4T ~A ~A ~A ~%"
                                  (rationalize a) u i))
        (recipe-amount recipe)
            (mapcar
                #'(lambda (i) (get i 'units))
                (recipe-ingredients recipe))
          (recipe-ingredients recipe))
  (format t "~%Servings: ~A ~%" (recipe-servings recipe))
  (format t "Calories/serving: ~A ~2%" (calories recipe)))
```

To test this function, select the definition with the Select Outermost Form command, and evaluate it with the Evaluate LISP Region command. When you use COOKIES as an argument to PRINT-RECIPE, the function prints the recipe as shown in Figure 5–14.

Figure 5–14: Running the PRINT-RECIPE Function

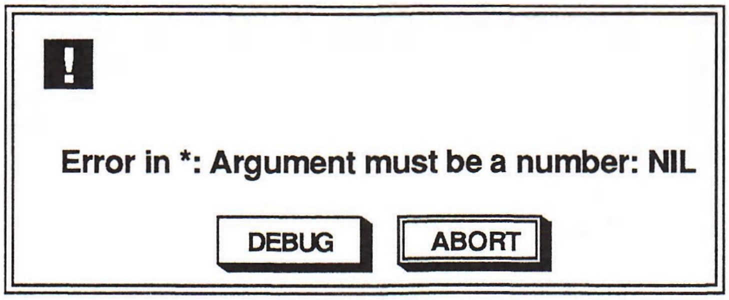


```
Lisp> (print-recipe cookies)
cookies

   2 cup SUGAR
   1/4 cup MILK
   1 oz BUTTER

Servings: 36
Calories/serving: 27
NIL
Lisp>
```

MLO–002868

When you use FRENCH-TOAST as an argument, the function generates the error shown in Figure 5–15. Whenever an error occurs, LISP gives you the opportunity to use the Debugger. A message box pops up that contains a brief message and two buttons. When the error is fatal, the buttons are labeled DEBUG and ABORT. ABORT is the default; you can avoid entering the Debugger by pressing Return. When it is a continuable error, the buttons are labeled DEBUG and CONTINUE. Clicking on the CONTINUE button or pressing Return is the same as entering the Debugger and immediately typing CONTINUE.

Figure 5–15: Debugging Dialog Box



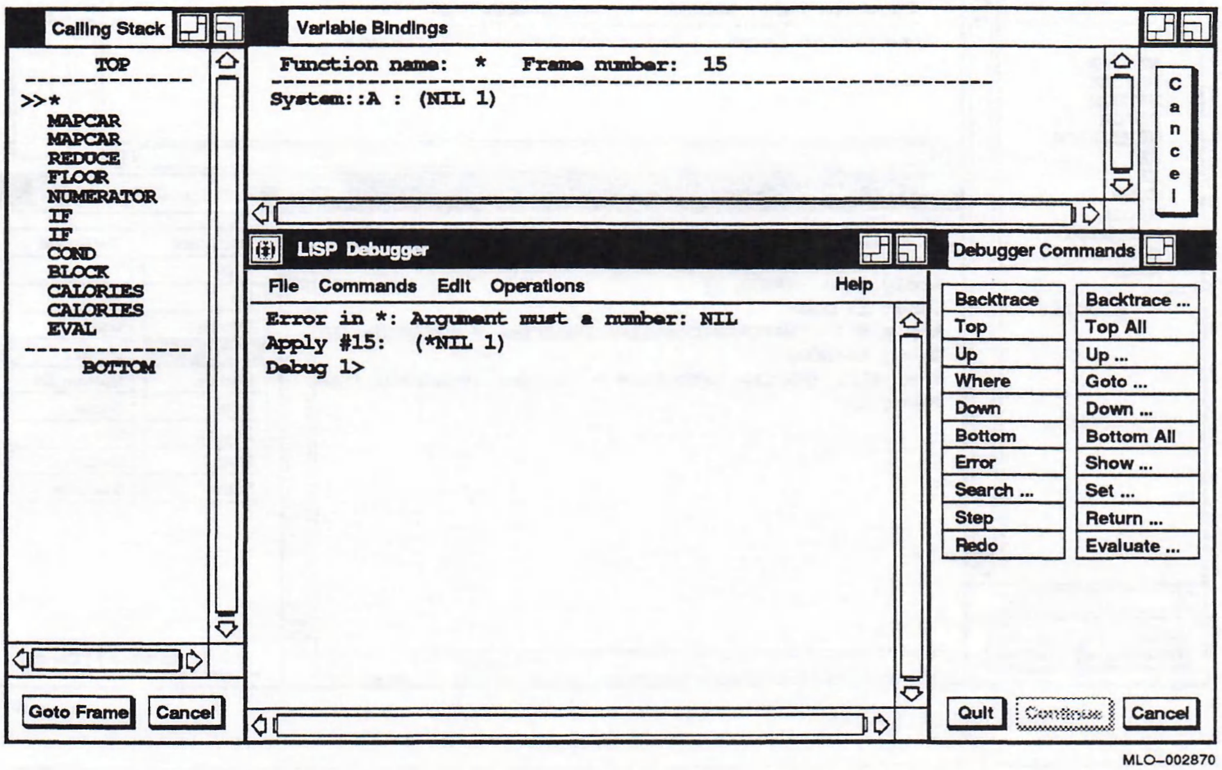
Error in *: Argument must be a number: NIL

MLO-002869

5.6 Using the Debugger

When you click on the DEBUG button, VAX LISP opens a set of debugging windows, as shown in Figure 5–16.

Figure 5–16: Debugging Windows



```
Function name:  *   Frame number:  15
-------------------------------------------------------
System::A : (NIL 1)
```

```
Error in *: Argument must a number: NIL
Apply #15:   (*NIL 1)
Debug 1>
```

```
     TOP
---------------
>>*
   MAPCAR
   MAPCAR
   REDUCE
   FLOOR
   NUMERATOR
   IF
   IF
   COND
   BLOCK
   CALORIES
   CALORIES
   EVAL
---------------
     BOTTOM
```

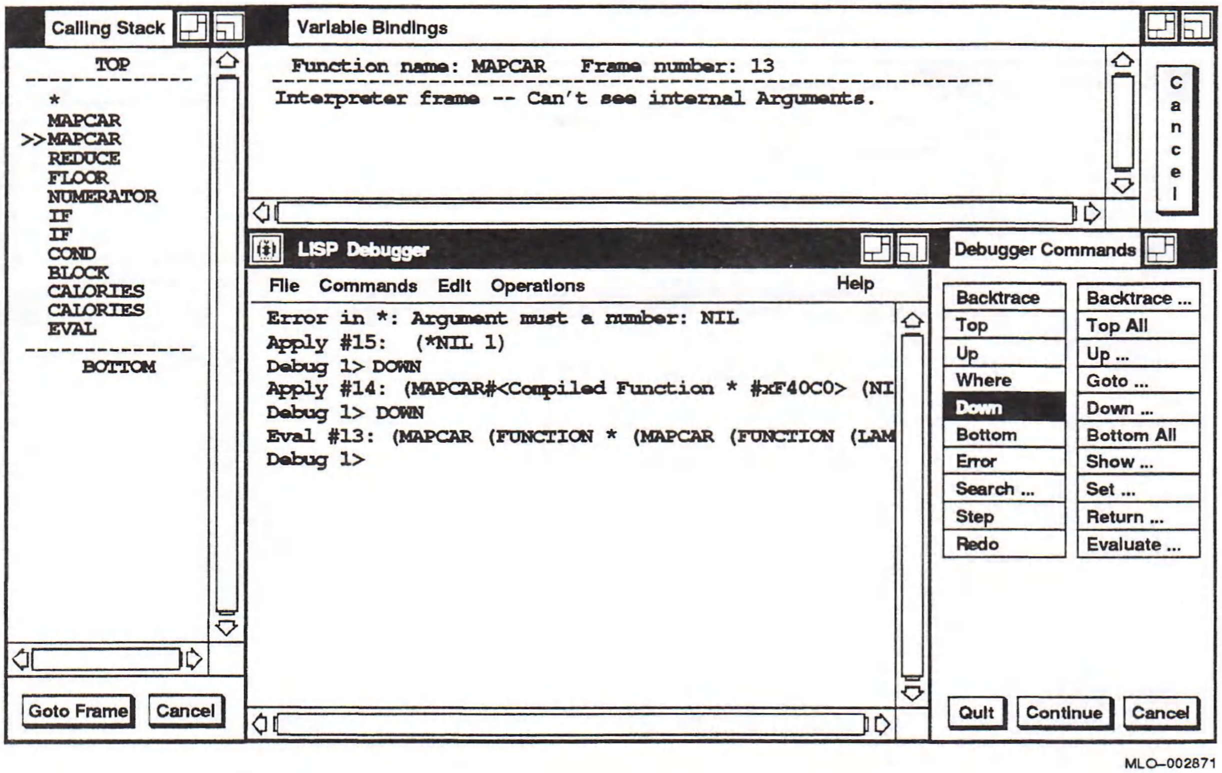
The main debugging window, LISP Debugger, contains messages from the Debugger and lets you enter debugging commands at the prompt. Above the Debugger window is a window that displays Variable Bindings. It shows the values of the variables in the current frame.

The Debugger Commands window, to the right of the LISP Debugger window, contains a set of command buttons. You can enter debugging commands by clicking on these buttons.

To the left of the Debugger window is the Calling Stack window. This window shows a backtrace of the frames on the control stack. A double arrow (>>) points to the current frame. Debugging commands can move the pointer up and down the Calling Stack. You can change the current frame by selecting a symbol in this window and clicking on Goto Frame or by double-clicking on a symbol.

For example, you can click the DOWN button twice in the Commands window to move the pointer to the second invocation of MAPCAR in the Calling Stack window. Figure 5–17 shows the messages that the Debugger displays when you do this.

Figure 5–17: Using the DOWN Command



```
TOP
--------------
 *
 MAPCAR
>>MAPCAR
 REDUCE
 FLOOR
 NUMERATOR
 IF
 IF
 COND
 BLOCK
 CALORIES
 CALORIES
 EVAL
--------------
BOTTOM
```

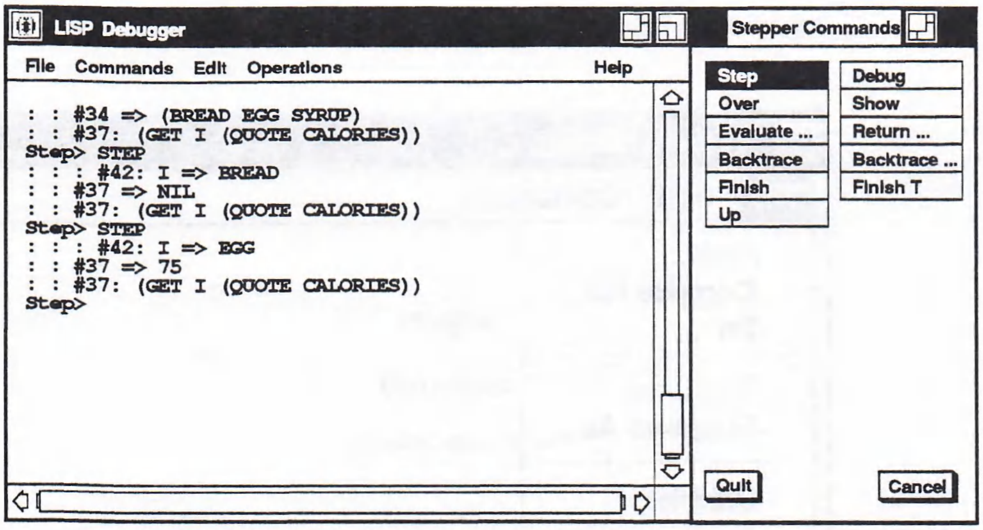
```
 Function name: MAPCAR   Frame number: 13
-----------------------------------------------------------
Interpreter frame -- Can't see internal Arguments.
```

```
Error in *: Argument must a number: NIL
Apply #15:  (*NIL 1)
Debug 1> DOWN
Apply #14: (MAPCAR#<Compiled Function * #xF40C0> (NI
Debug 1> DOWN
Eval #13: (MAPCAR (FUNCTION * (MAPCAR (FUNCTION (LAM
Debug 1>
```

Each time you issue the DOWN command, the Debugger displays the function that LISP would evaluate at that point in the Calling Stack. You can then issue the STEP command to evaluate the function, one step at a time. For example, when the Calling Stack points to the outer occurrence of MAPCAR, you can click on the STEP command to begin reevaluating the function. When you invoke the Stepper, it presents a different set of commands in a Stepper Commands window, the Calling Stack and Variable Bindings windows disappear, and the Debugging window displays the Step prompt rather than the Debugger prompt.

Figure 5–18 shows what the Stepper displays when you step through the second MAPCAR function invocation. Note how the Stepper returns the result of the form (GET I (QUOTE CALORIES)) for each of the elements in the list (BREAD EGG SYRUP). The value returned for BREAD is NIL. This causes the program error—the CALORIES function expects this value to be a number.

Figure 5–18: Stepping Through a Function



```
: : #34 => (BREAD EGG SYRUP)
: : #37: (GET I (QUOTE CALORIES))
Step> STEP
: : : #42: I => BREAD
: : #37 => NIL
: : #37: (GET I (QUOTE CALORIES))
Step> STEP
: : : #42: I => EGG
: : #37 => 75
: : #37: (GET I (QUOTE CALORIES))
Step>
```

MLO–002872

To exit from the Stepper, choose Quit from the Stepper menu. This closes all Debugger and Stepper windows and returns you to the Listener. At the Lisp> prompt, you can fix the program by entering the following form to define the number of calories per slice of bread:

```
Lisp> (setf (get 'bread 'calories) 100)
100
```

You can check that the program now works correctly by entering the form:

```
Lisp> (calories french-toast)
275
```

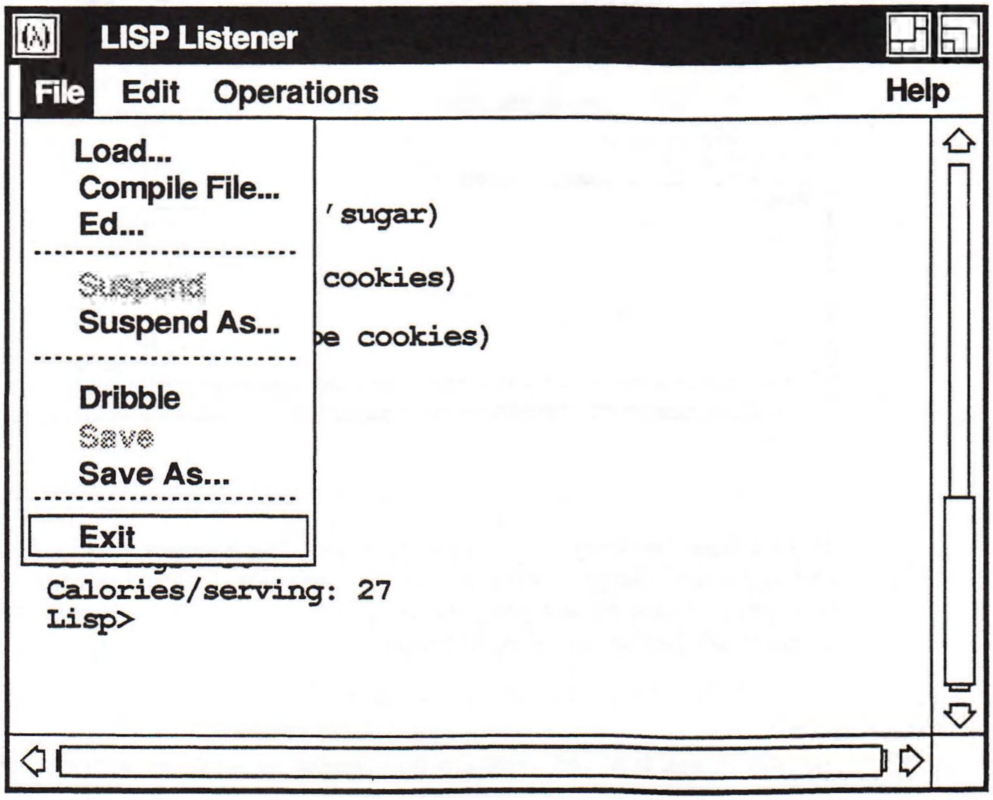
For more information on the Debugger, see Chapter 10.

5.7 Exiting from VAX LISP

You have now seen how to perform some basic operations with the DECwindows interface to VAX LISP. The chapters that follow describe each utility in detail.

You can exit from LISP by typing `(EXIT)` at the `Lisp>` prompt or by choosing Exit from the File menu, as shown in Figure 5–19. If you choose the menu item, a caution box appears to confirm that you want to exit. Click on the Yes button to exit from LISP.

Figure 5–19: Exiting from VAX LISP



```
'sugar)
cookies)
pe cookies)
Calories/serving: 27
Lisp>
```

MLO–002873

Chapter 6

Starting LISP from DECwindows

This chapter describes how to:

- Define the display system (DECwindows or terminal) that LISP uses from the DCL command line
- Invoke an interactive LISP session from DECwindows
- Resume a suspended LISP session from DECwindows
- Compile a LISP file from DECwindows

6.1 Defining the Display System

You can define the interface on which VAX LISP runs on a workstation with the VMS SET DISPLAY command. You can have VAX LISP run with the terminal or DECwindows interface on your workstation. You define the interface you want from your workstation or from a terminal that is a node in the same network.

6.1.1 Displaying LISP on DECwindows

To display VAX LISP on a workstation, define that workstation with the SET DISPLAY command. For further information about the SET DISPLAY command, see *VMS Version 5.1 New Features Manual.*

6.1.2 Displaying LISP on a Terminal

To display VAX LISP on a terminal, use the DCL LISP verb as described in Chapter 2. If the logical LISP$DISPLAY has the value "TERMINAL", or if the logical DECW$DISPLAY has no value, VAX LISP will run with its terminal interface.

6.2 Invoking LISP from DECwindows FileView

To invoke VAX LISP from DECwindows FileView:

- Use the LISP.EXE file.
- Use the DCL Command item in the Utilities menu.

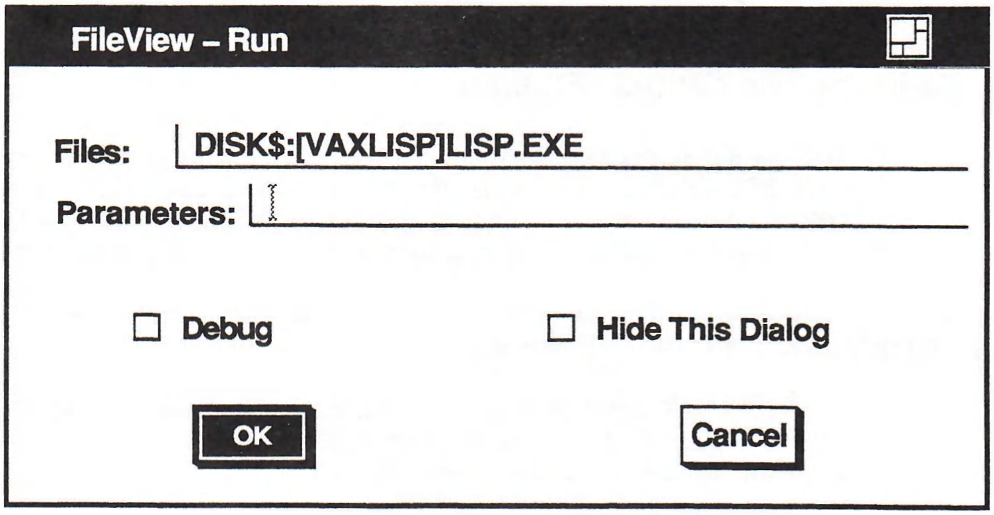
6.2.1 Invoking LISP with the LISP.EXE File

To invoke an interactive session of VAX LISP from FileView with the LISP.EXE file:

- Select LISP.EXE and choose the RUN item from the Files Menu.
- Press MB2 on LISP.EXE and choose the RUN item from the pop-up menu.
- Double-click on LISP.EXE.

If you invoke LISP with one of the menu items, the DECwindows FileView-Run dialog box is displayed on your screen (see Figure 6–1). You can enter standard LISP parameters in the Parameters field of this dialog box. Double-clicking on LISP.EXE invokes LISP without any qualifiers. The next time you select LISP.EXE and choose the RUN item, the parameters that you previously used appear, and you can edit them for this invocation. Subsequently, double-clicking on LISP.EXE invokes LISP with the most recently supplied parameters.

Figure 6–1: DECwindows FileView-Run Dialog Box



FileView – Run

Files: DISK$:[VAXLISP]LISP.EXE

Parameters:

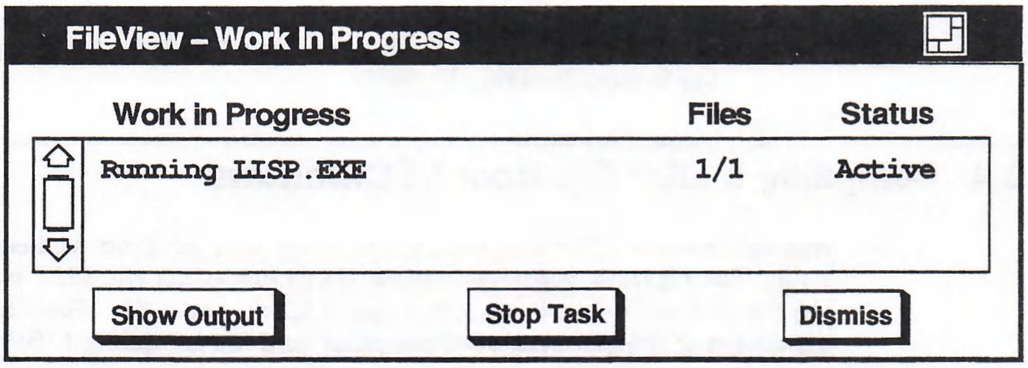
□ Debug

□ Hide This Dialog

MLO–002985

To continue the invocation, click on the OK button, and the FileView-Work in Progress dialog box (see Figure 6–2) appears on your screen followed by the LISP Listener with the `Lisp>` prompt. Chapter 7 describes the Listener in more detail.

Figure 6–2: DECwindows FileView-Work in Progress Dialog Box



```
Running LISP.EXE                    1/1      Active
```

MLO–002986

6.2.2 Invoking LISP with the DCL Command Item

You can invoke an interactive session of VAX LISP from the FileView by choosing the DCL Command item from the Utilities menu. This brings up a FileView Task Output dialog box into which you can type the LISP DCL command with qualifiers and arguments. By default, this causes LISP to display on your workstation. To have LISP display on some other node, use the methods described in Section 6.1. Enter the commands to define the node before entering the LISP command.

6.3 Resuming LISP from DECwindows

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 state of the windows that were displayed on your workstation screen is not included in this file. 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.

To resume a suspended system from DECwindows, type the LISP command with the /RESUME qualifier and the name of the file containing the suspended system at any of the previously described places where you could use the LISP command, such as the $ prompt, in the DCL Command box. Section 6.2.2 describes how to get to the DCL Command box. The LISP Listener is displayed on your workstation screen; you must rebuild the other windows if you want to re-create the display that was on your screen when you suspended the LISP session.

See Section 2.11.1 for detailed information on the SUSPEND function.

NOTE

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

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

6.4 Compiling a LISP File from DECwindows

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. See Chapter 2 for a discussion of the advantages of compiling and not compiling LISP expressions.

To compile files from DECwindows:

1. Select any number of .LSP files.
2. Choose the COMPILE item on the FileView Programming menu. The Programming menu is not available by default in the FileView menu bar. You add it with the MENU BAR... item on the FileView Control menu.

For each .LSP file you select, a FileView prompting window is displayed on your workstation screen. You type the options you want for the compilation of that file in the prompting window. These options are the qualifiers that are allowable with the /COMPILE qualifier. Chapter 2 provides a table and a detailed description of each of these options. The options that you type in the prompting box are used only for the compilation of that file. Each file is compiled in a separate invocation of LISP.

Chapter 7

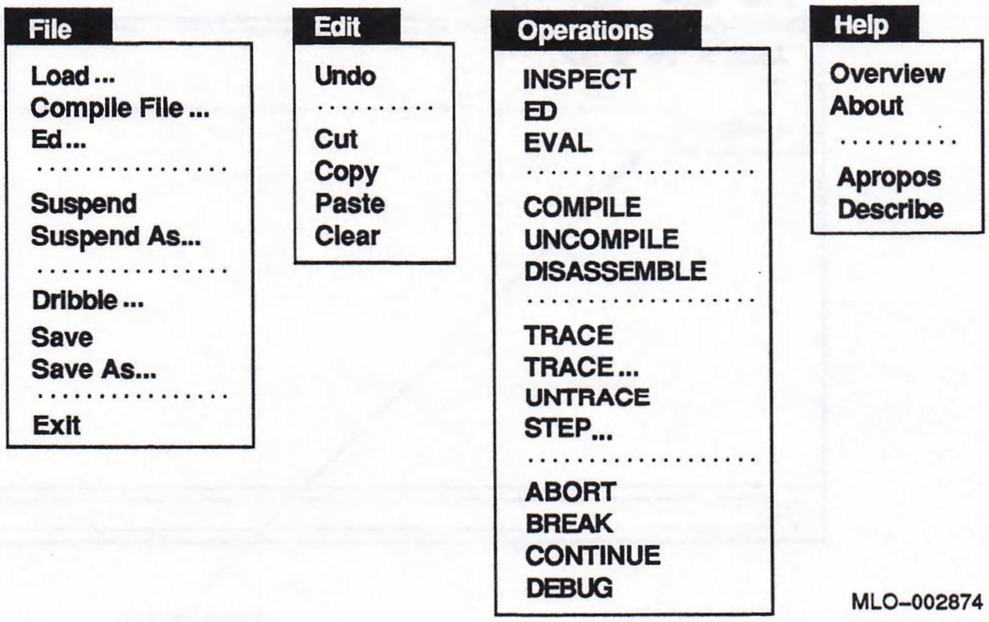
The Listener

The Listener utility appears on the screen when you invoke LISP in a DECwindows environment. The Listener corresponds to the top level in a terminal-based LISP system. When you type forms in the Listener, LISP reads and evaluates them, and prints the results in the same window.

You can access other LISP utilities by using the menus the Listener provides. You can also use the menus to load, compile, and save LISP files.

Figure 7–1 shows the Listener menus.

Figure 7–1: Listener Menus



This chapter describes how to:

- Enter LISP forms
- Exit from the Listener
- Edit text and objects
- Work with files

Appendix C provides information on how to customize the Listener.

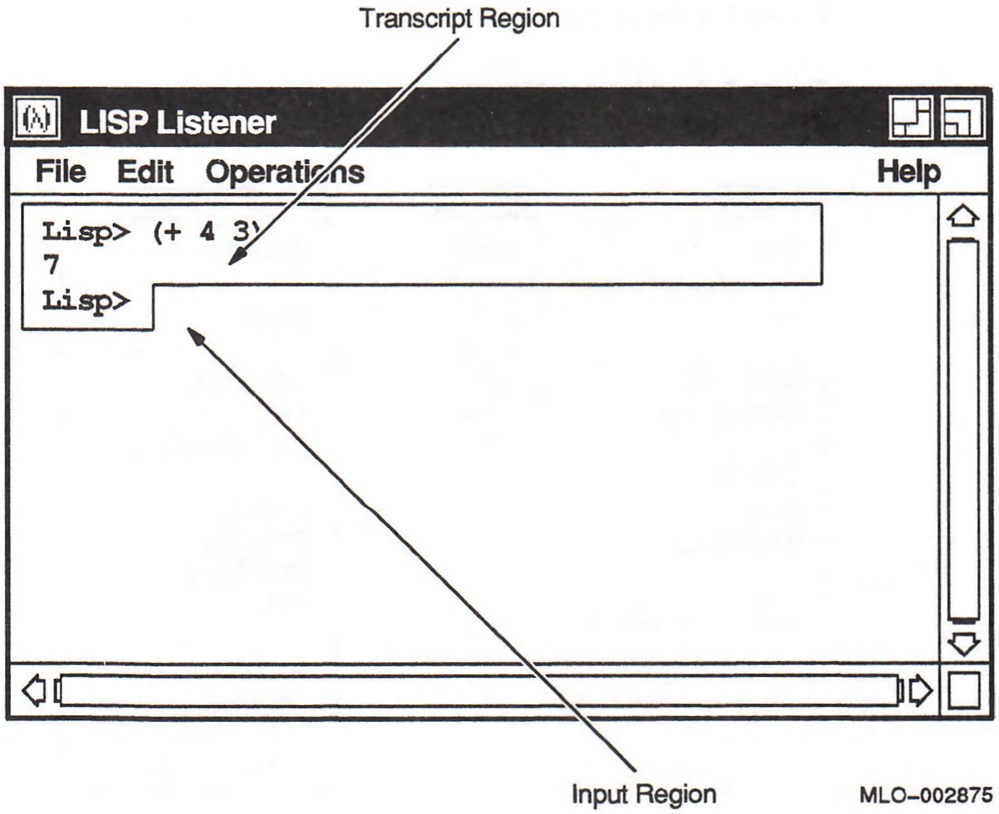
7.1 Entering LISP Forms

When you invoke LISP with the DECwindows interface, a Listener window appears on the screen. After displaying the copyright notice, the `Lisp>` prompt and the text cursor appear at the top left of the work area.

The work area is divided into two logical portions: the transcript region and the input region. The **transcript region** consists of the text preceding the current `Lisp>` prompt including the current `Lisp>` prompt. The **input region** is the area following the current `Lisp>` prompt, where the text insertion point is located.

As you enter text, it appears in the input region. When you complete a form and press Return, the Listener reads and evaluates the form. The text of the input form and the value returned by LISP become part of the transcript. Figure 7–2 shows the transcript region and the input region of a LISP window after you enter the form `(+ 4 3)` and press Return.

Figure 7–2: Listener Window



```
Lisp> (+ 4 3)
7
Lisp>
```

The transcript region is read only. You can select and copy text and objects from this region but you cannot modify it.

The text you enter in the input region remains text when it becomes part of the transcript. It is not converted to LISP objects. However, if you paste an object into the input region, you can still select it as an object both when it is in the input region and when it becomes part of the transcript region.

7.2 Exiting LISP from the Listener

To exit from the Listener, choose the Exit menu item from the File menu. A confirmation box appears on the screen. To exit, click on the OK button. When you exit from the Listener, the Listener window and any other LISP windows on the screen disappear. The LISP image is now stopped.

7.3 Editing Text and Objects

When you use LISP in a DECwindows environment, you can move or copy text or objects:

- From one location in a window to another.
- Between windows—for example, from an Inspect window to the Listener window.

In addition, LISP defines specific keys to let you perform basic text editing operations. These keys let you move the cursor and delete text.

7.3.1 Selecting Text and Objects

Before you can move text and objects, you must select them with the mouse. You select text as you do in other DECwindows applications, by pressing MB1 and dragging over the region of text. Section A.16.1 describes the basic ways to select text.

LISP also lets you select larger blocks of text as follows:

User Action	Result
2 MB1 clicks	Selects the word indicated by the pointer.
3 MB1 clicks	Selects the line indicated by the pointer.
4 MB1 clicks	Selects the input or output text of the READ-EVAL-PRINT transaction.
5 MB1 clicks	Selects all text in the transcript and input regions.

When you move the pointer over a portion of text that is also an object, the object is underlined. When you move the pointer off the object, the underlining disappears.

To select an object, move the pointer to the underlined object and click MB1. The object is highlighted to show it is selected. To deselect the object, click MB1 again anywhere in the window.

To select the text instead of the underlined object, move the pointer to the first character of the text you want to select and press and drag MB1. To deselect the text, click MB1 once anywhere in the text.

7.3.2 Moving Text and Objects to Another Location

You can select text and objects and move them to other locations in the LISP development environment, using the Edit menu.

You move the selected range by cutting it and then pasting it in a new location. To cut and paste the selected range:

1. Select the text or objects you want, using mouse clicks.
2. Choose Cut from the Edit menu.
3. Place the cursor where you want the information and choose Paste from the Edit menu of the utility to which you are moving the text.

To copy the selected range instead of cutting it, choose the Copy item from the Edit menu. Then place the cursor where you want the information and choose Paste.

For further information on copying text in the DECwindows environment, see Sections A.16.2 and A.16.3.

7.3.3 VAX LISP Default Key Bindings

Table 7–1 lists the default key bindings for VAX LISP. You can use these key bindings to move the cursor or delete text while in the Listener, Inspector, and Debugger. For information on using key bindings in the Editor, see Section 8.1.3.

Table 7–1: VAX LISP Default Key Bindings

User Action	Result
Return or Enter	Inserts a new line. If you have completed the input, it will be processed.
Tab	Inserts leading whitespace to correctly indent current line.
←	Moves text cursor one character to the left.
Shift/←	Moves text cursor one word to the left.
Ctrl/←	Moves text cursor to beginning of current line.
↑	Replaces input buffer with previous input buffer from the transcript region.
→	Moves text cursor one character to the right.
Shift/→	Moves text cursor one word to the right.
Ctrl/→	Moves text cursor to the end of current line.
↓	Replaces input buffer with next input buffer from the transcript region.
F12	Moves text cursor to beginning of input buffer.
Shift/F12	Moves text cursor to the end of input buffer.
Delete	Deletes character to the left of text cursor.
Shift/Delete	Deletes character to the right of text cursor.
Ctrl/Delete	Deletes input buffer. Contents of buffer cannot be retrieved.
Ctrl/A	Moves text cursor to beginning of input line.

(continued on next page)

Table 7–1 (Cont.): VAX LISP Default Key Bindings

User Action	Result
Ctrl/B	Moves text cursor one character to the left.
Ctrl/D	Deletes the character to the right of text cursor.
Ctrl/E	Moves text cursor to the end of current line.
Ctrl/F	Moves text cursor one character to the right.
Ctrl/G	Clears input buffer. Contents of buffer cannot be retrieved.
Ctrl/H	Moves text cursor to the beginning of input buffer.
Ctrl/I	Inserts leading whitespace to correctly indent current line.
Ctrl/J	Inserts linefeed and correct amount of whitespace to indent next line. Places text cursor at the end of new whitespace.
Ctrl/K	Deletes all input to the right of text cursor.
Ctrl/N	Moves text cursor to next line in input buffer.
Ctrl/O	Inserts leading whitespace to correctly indent next line. Cursor remains at the end of current line.
Ctrl/P	Moves text cursor to previous line in input buffer.
Ctrl/U	Deletes all input to the left of text cursor.
Ctrl/W	Clears input buffer. Contents of input buffer are stored for retrieval by Ctrl/Y.
Ctrl/Y	Replaces input buffer with input buffer stored by Ctrl/W.

7.4 Working with Files

You can perform operations on files by clicking on items from the File menu. From the Listener, you can:

- Load a LISP file
- Compile a LISP file
- Invoke the Editor on a LISP file
- Save the text of the transcript region to a file
- Create a LISP suspended image file
- Record your interactive LISP session to a file

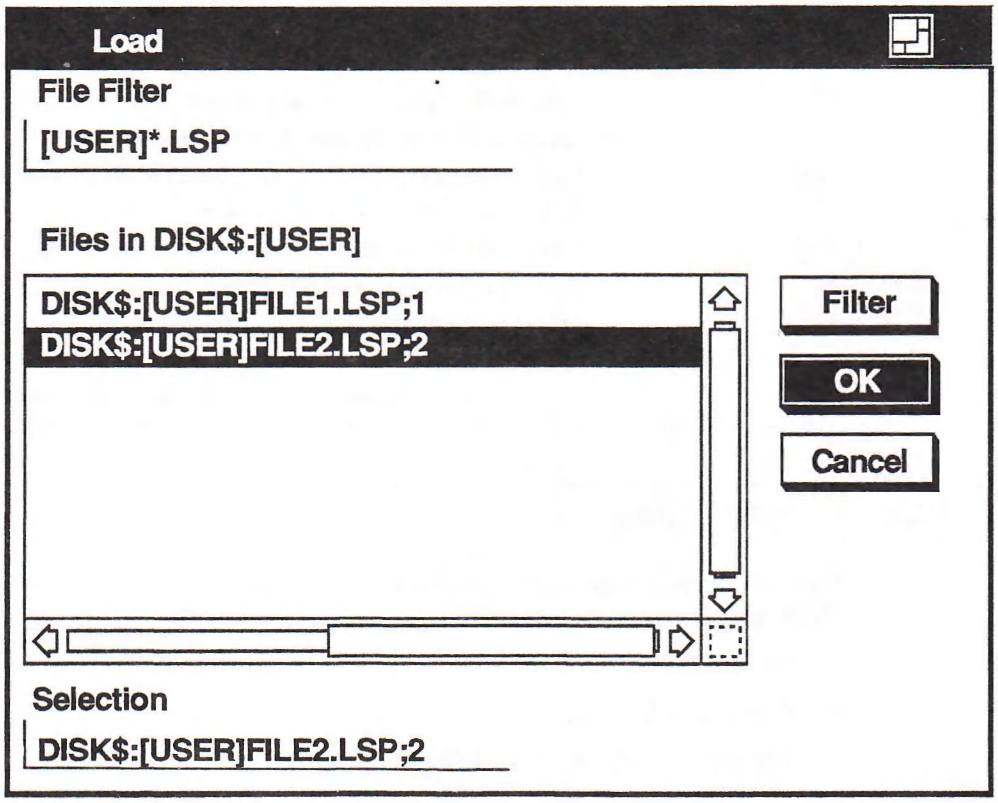
7.4.1 Loading a LISP File

Load a LISP file from the Listener as follows:

1. Choose the Load menu item from the File menu.

   The Listener displays a File Selection box with the names of any LISP source files in the current directory. Figure 7–3 shows a File Selection box.

Figure 7–3: File Selection Box



File Filter

[USER]*.LSP

Files in DISK$:[USER]

DISK$:[USER]FILE1.LSP;1
DISK$:[USER]FILE2.LSP;2

Selection

DISK$:[USER]FILE2.LSP;2

MLO–002876

2. Select the name of the file you want to load.

   To see a listing of files in another directory, enter the directory name in the File Filter field and click on the Filter button or press Return. The list box displays the files in that directory. To display specific files in the list box, enter a directory name and file specification in the File Filter field.

3. Click on the OK button, press Return, or double-click on the file.

4. The Listener clears the input region of any partial forms and fills it with a call to the LOAD function. As the file is loaded, LISP displays the names of the functions contained in the file on the screen. When the command finishes running, the `Lisp>` prompt returns and any partial form entered before loading the file is restored.

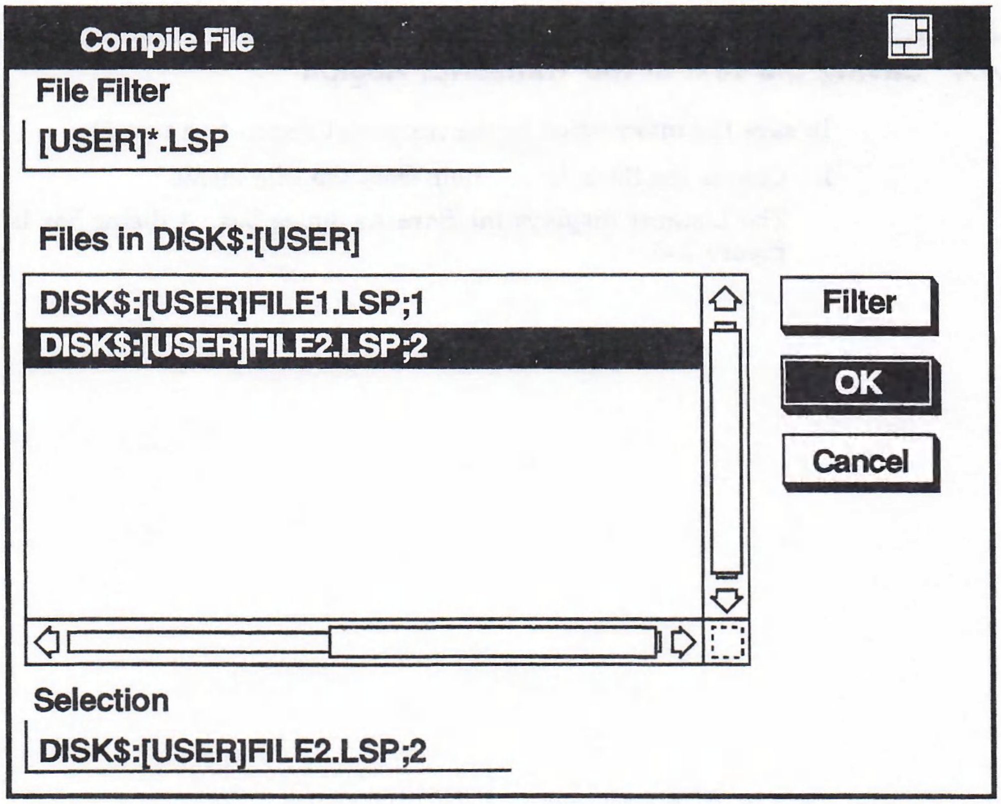
7.4.2 Compiling a LISP File

You can use the COMPILE-FILE function to compile a LISP file without leaving the LISP environment. To compile a file:

1. Choose the Compile File item from the File menu.

   The Listener displays a File Selection box with the names of all LISP source files in the current directory. Figure 7–4 shows the Compile-File File Select box.

Figure 7–4: Compile-File File Select Box



Compile File

File Filter

[USER]*.LSP

Files in DISK$:[USER]

DISK$:[USER]FILE1.LSP;1

DISK$:[USER]FILE2.LSP;2

Selection

DISK$:[USER]FILE2.LSP;2

MLO–002877

2. Select the name of the file you want to compile.
3. Click on the OK button, press Return, or double-click on the file.
4. The Listener clears the input region of any partial forms and fills it with a call to the COMPILE-FILE function. As the file compiles, the names of the functions in the file appear in the Listener window. After the file is compiled, LISP returns the Lisp> prompt and restores any partial form that was entered before compiling the file.

7.4.3 Invoking the Editor on a LISP File

To invoke the Editor on a LISP file:

1. Choose the Ed item from the File menu. A File Selection box appears on the screen.
2. Select a file name from the list box or enter a new file name on the Selection line.
3. Click on the OK button.
4. The Editor window appears. If you selected the name of an existing file, the file is loaded into an Editor buffer and appears in the Editor window. If you entered the name of a new file, the Editor window appears as shown in Figure 8–2.

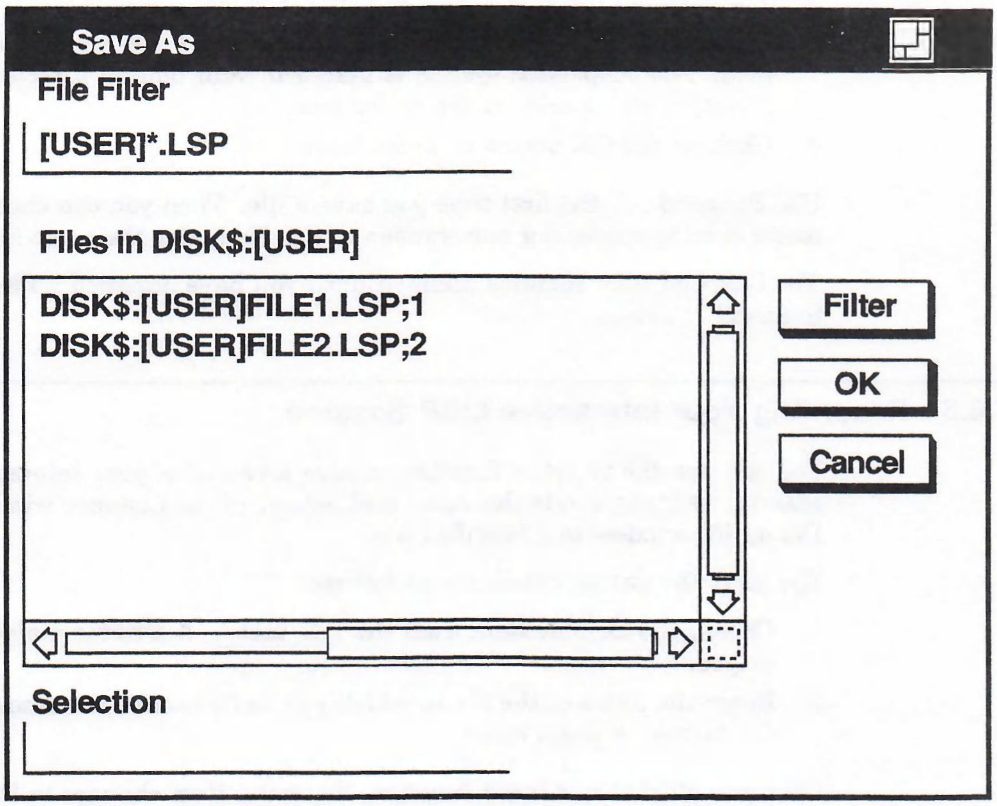
7.4.4 Saving the Text of the Transcript Region

To save the information in the transcript region to a text file:

1. Choose the Save As . . . item from the File menu.

   The Listener displays the Save As dialog box. A dialog box is shown in Figure 7–5.

Figure 7–5: Save As Dialog Box



Save As

File Filter

[USER]*.LSP

Files in DISK$:[USER]

DISK$:[USER]FILE1.LSP;1
DISK$:[USER]FILE2.LSP;2

Selection

MLO-002878

2. To save the text to a new file, enter the file name in the Selection field and click on OK or press Return.

   To save the text to a new version of a file appearing in the list box, click on the file name and then click on the OK button or press Return.

   To see a listing of files in another directory, type the directory name in the File Filter field and click on the Filter button. The list box displays the files in that directory. To display specific files in the list box, enter a directory name and file specification in the File Filter field.

After you have saved a file with Save As, you can save a new version of the file under the same file name by choosing the Save menu item. A message indicates that the file is being saved.

If you choose Save before using Save As in this session, the Save dialog box appears to prompt you for a file name.

7.4.5 Creating a LISP Suspended Image File

You can use the SUSPEND function to save the state of the LISP system, making it possible to resume the LISP system at a later time. After you create the suspended file, the system returns you to the Lisp> prompt. You can continue the current LISP session or exit from the session. For further information on creating and resuming suspended systems, see Section 2.11.

To create a suspended system:

1. Choose the Suspend As . . . item from the File menu.
2. The Listener displays a File Selection box and prompts you for a file name.
3. Choose a file from the list or type the file name you want in the Selection field. The suspended system is placed in your default directory or in the directory you specify in the dialog box.
4. Click on the OK button or press Return.

Use Suspend . . . the first time you save a file. Then you can choose the Suspend menu item to suspend a new version of the file under the same file name.

The Suspend item remains dimmed until you have assigned a file name with the Suspend . . . item.

7.4.6 Recording Your Interactive LISP Session

You can use the `DRIBBLE` function to save a record of your interactive LISP session. `DRIBBLE` sends the input and output of the Listener window and the Debug I/O window to a specified file.

You start the `DRIBBLE` function as follows:

1. Choose the Dribble item from the File menu. A Dribble dialog box is displayed.
2. Enter the name of the file in which you want to save the session. Click on the OK button or press Return.

Once you start the `DRIBBLE` function, the menu item changes to Stop Dribble. To stop the `DRIBBLE` function, choose this item.

The system continues to copy all input and output to the dribble file until you choose Stop Dribble or enter the form `(dribble)` at the `Lisp>` prompt.

7.5 Compiling a Function

You can use the `COMPILE` function to compile a function or a macro from within a currently running LISP session.

You normally call the LISP function first in interpreted form to see if it works. Once it works as interpreted, you can test it in compiled form without writing it to a file.

When you compile a function or macro that is not in a file, the compiled definition exists only in the current LISP session. To save the compiled definition once you leave LISP, you must write the interpreted function to a file and compile the file.

You can use the `UNCOMPILE` function to restore the interpreted function definition of a symbol, if the symbol's definition was compiled with the `COMPILE` function.

To compile a function or macro:

1. Select a symbol that names the function or macro you want to compile.
2. Choose the COMPILE item on the Operations menu.

   The COMPILE item is dimmed unless there is a selection.

3. The Listener clears the input region of any partial form appearing at the Lisp> prompt and fills it with a call to the COMPILE function. When the compilation finishes, the Lisp> prompt returns and any partial input entered before the compiling operation is restored.

To uncompile a function or macro that was compiled with the COMPILE function:

1. Select the function or macro you want to uncompile.
2. Choose the UNCOMPILE item on the Operations menu. The UNCOMPILE item is dimmed unless there is a selection.

7.6 Disassembling a Function

The DISASSEMBLE function takes compiled LISP code and "reverse-assembles" it. This lets you view the compiler output and check the efficiency of your code. To disassemble a function or macro:

1. Select a symbol that names the function or macro you want to disassemble.
2. Choose the DISASSEMBLE item on the Operations menu. The DISASSEMBLE item is dimmed unless there is a selection.
3. The Listener clears the region of any partial form appearing at the Lisp> prompt and fills it with a call to the DISASSEMBLE function. After the compiled code has been reverse assembled, the Lisp> prompt returns, and any partial input entered before the disassembling operation is restored.

7.7 Using the EVAL Function

The EVAL function takes a form as input and evaluates it. You can evaluate a form by selecting it and choosing the EVAL item as follows:

1. Select the form you want to evaluate. You can do this even if you have a partially entered form at the current Lisp> prompt.
2. Choose the EVAL item from the Operations menu. The EVAL item is dimmed unless there is a selection.
3. The Listener suspends any partial form and fills the input region with the appropriate calls to the EVAL function. The evaluation occurs and return value appears on the screen. The Listener displays a new Lisp> prompt and restores any partial form to the input region.

7.8 Invoking Other LISP Utilities

You can invoke the Inspector, tracer and Editor utilities from the Listener by choosing the appropriate menu item from the Operations menu. To invoke these utilities:

1. Select a string or symbol for the Editor or tracer or any object for the Inspector.
2. Choose the appropriate menu item on the Operations menu. The menu items representing the utilities are dimmed unless there is selection.

   The utility window appears on the screen and has input focus.

7.9 Interrupting a Program

You can interrupt your program by choosing the ABORT, BREAK, or DEBUG functions from the Operations menu. These items call asynchronous functions on the control stack of your program.

The ABORT function halts the execution of your program. To halt program execution:

- Choose the ABORT item from the Operations menu while your program is executing. This calls the ABORT function on your program's stack, as if your program had called ABORT directly. See the description of the ABORT function in the *VAX LISP/VMS Object Reference Manual*.
- Press Ctrl/C in the Listener window while your program is executing.

You can temporarily interrupt the execution of your program with the BREAK or DEBUG item on the Operations menu. The BREAK item causes a break loop to be run in the Listener (see Section 10.3). The DEBUG item invokes the Debugger on your program, displaying the frame where execution was interrupted (see Section 10.4). To interrupt your program with a break loop or the Debugger, choose the appropriate item from the Operations menu.

7.10 Getting Help

You can get help in VAX LISP by using the Help menu or the Help key with the pointer. Help provides brief information on the following:

- VAX LISP concepts
- VAX LISP product and version number
- Screen objects

You can also access the VAX LISP functions APROPOS and DESCRIBE from the Help menu.

7.10.1 Invoking Help on VAX LISP

To get help on VAX LISP, choose the Overview item from the Help menu. A Help window opens to display an overview of VAX LISP.

To get information about the product and version number, choose the About item. You can access the overview information from this screen by clicking on the Overview item under Additional Topics.

7.10.2 Invoking Help on Screen Objects

To get help on screen objects such as menu names and menu items:

1. Point to the screen object.
2. Press and hold the Help key while you click MB1.
3. Release the Help key.

   A Help window opens with information on the screen object.

7.10.3 Using the APROPOS and DESCRIBE Functions

The APROPOS and DESCRIBE functions provide information about VAX LISP objects. The APROPOS function searches through a package for a symbol whose print name contains a specified string. The DESCRIBE function displays information about a specified object. For more information about APROPOS and DESCRIBE, see *VAX LISP/VMS Object Reference Manual.*

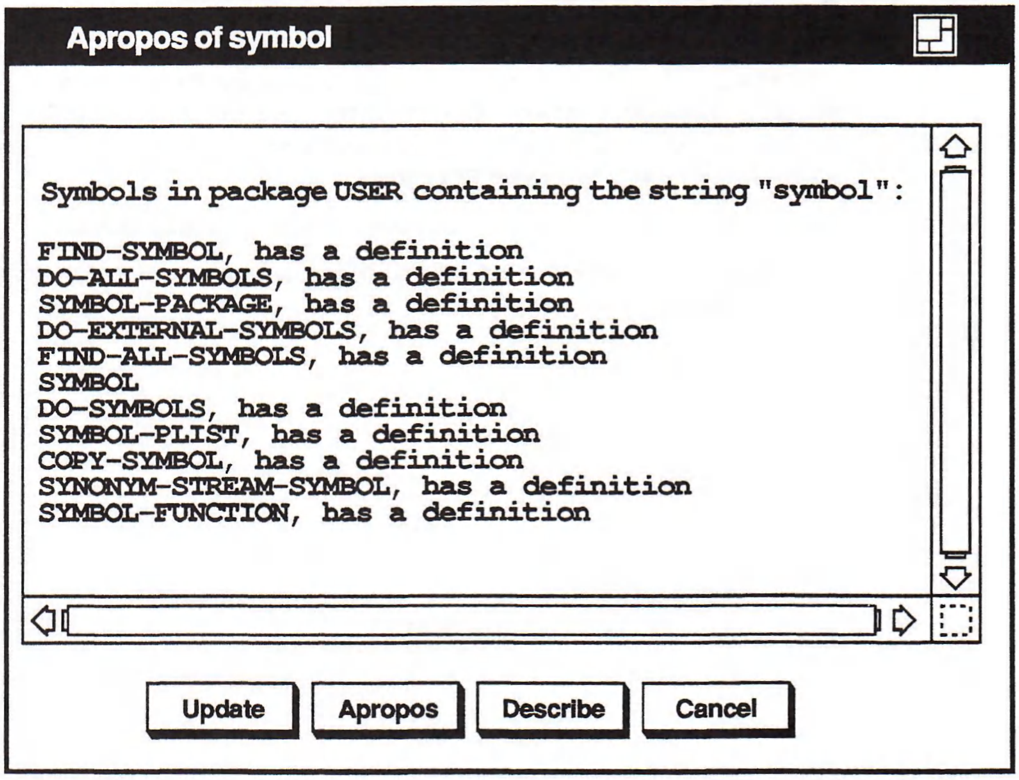
To use the APROPOS or DESCRIBE function from the Help menu:

1. Select a text string or an object. If you select an object for APROPOS, it must be a string or a symbol.

2. Choose the Apropos or Describe item. If there is no selection, these items are dimmed.

   If you click on Apropos, the Apropos dialog box appears on the screen. It contains a list of all the symbols accessible from the current package that contain the selected text string or object. You can use the scroll bar to scroll through the information. Figure 7–6 shows an Apropos dialog box.

Figure 7–6: Apropos Dialog Box

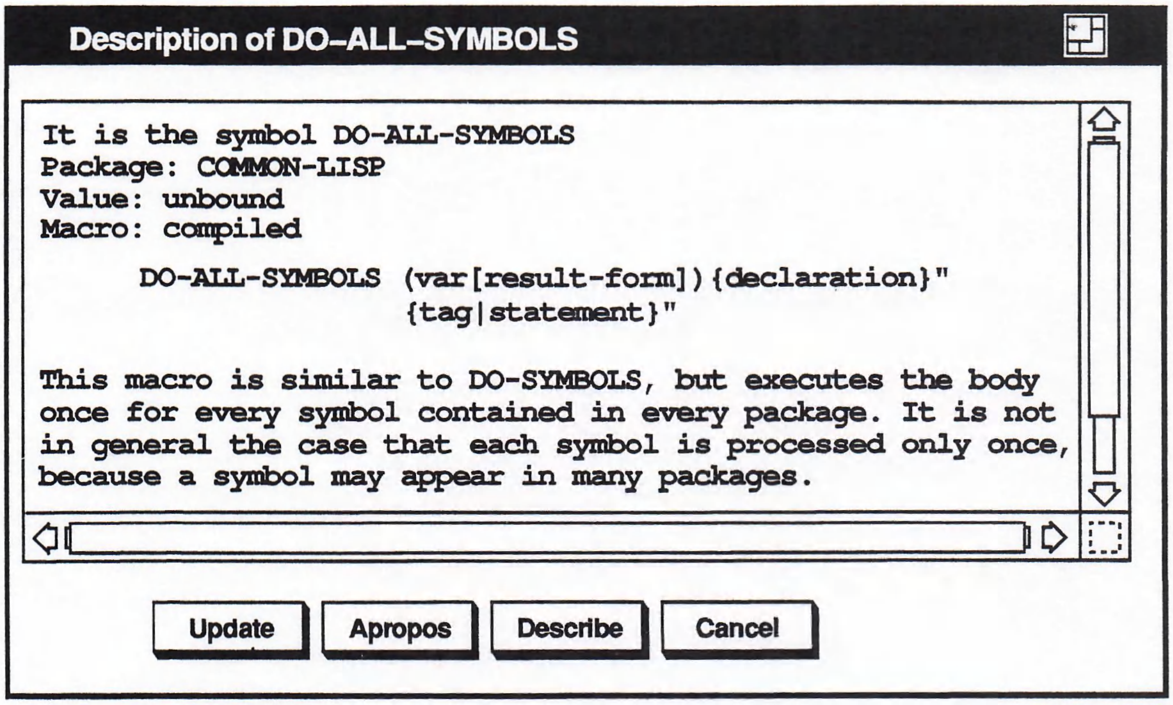


```
Symbols in package USER containing the string "symbol":

FIND-SYMBOL, has a definition
DO-ALL-SYMBOLS, has a definition
SYMBOL-PACKAGE, has a definition
DO-EXTERNAL-SYMBOLS, has a definition
FIND-ALL-SYMBOLS, has a definition
SYMBOL
DO-SYMBOLS, has a definition
SYMBOL-PLIST, has a definition
COPY-SYMBOL, has a definition
SYNONYM-STREAM-SYMBOL, has a definition
SYMBOL-FUNCTION, has a definition
```

MLO–002879

If you click on Describe, the Describe dialog box appears on the screen. It contains a description of the selected text string or object. You can use the scroll bar to scroll through the information. A Describe dialog box is shown in Figure 7–7.

Figure 7–7: Describe Dialog Box



```
It is the symbol DO-ALL-SYMBOLS
Package: COMMON-LISP
Value: unbound
Macro: compiled

      DO-ALL-SYMBOLS (var[result-form]){declaration}"
                     {tag|statement}"

This macro is similar to DO-SYMBOLS, but executes the body
once for every symbol contained in every package. It is not
in general the case that each symbol is processed only once,
because a symbol may appear in many packages.
```

MLO–002880

You can click on the Cancel button to dismiss to the dialog box or leave the dialog box on the screen. The next time you select a text string or object and click on Apropos or Describe, another dialog box appears.

You can select objects or text in the Apropos or Describe dialog box and then click on the Apropos or Describe buttons at the bottom of the dialog boxes. The appropriate dialog box appears with information on that object.

The Apropos and Describe dialog boxes provide snapshot information about the selected text string or object. The information in the dialog box may become obsolete if you continue to work in LISP while the dialog box is present on the screen. To obtain the most current information about the topic, click on the Update button on the dialog box.

Chapter 8

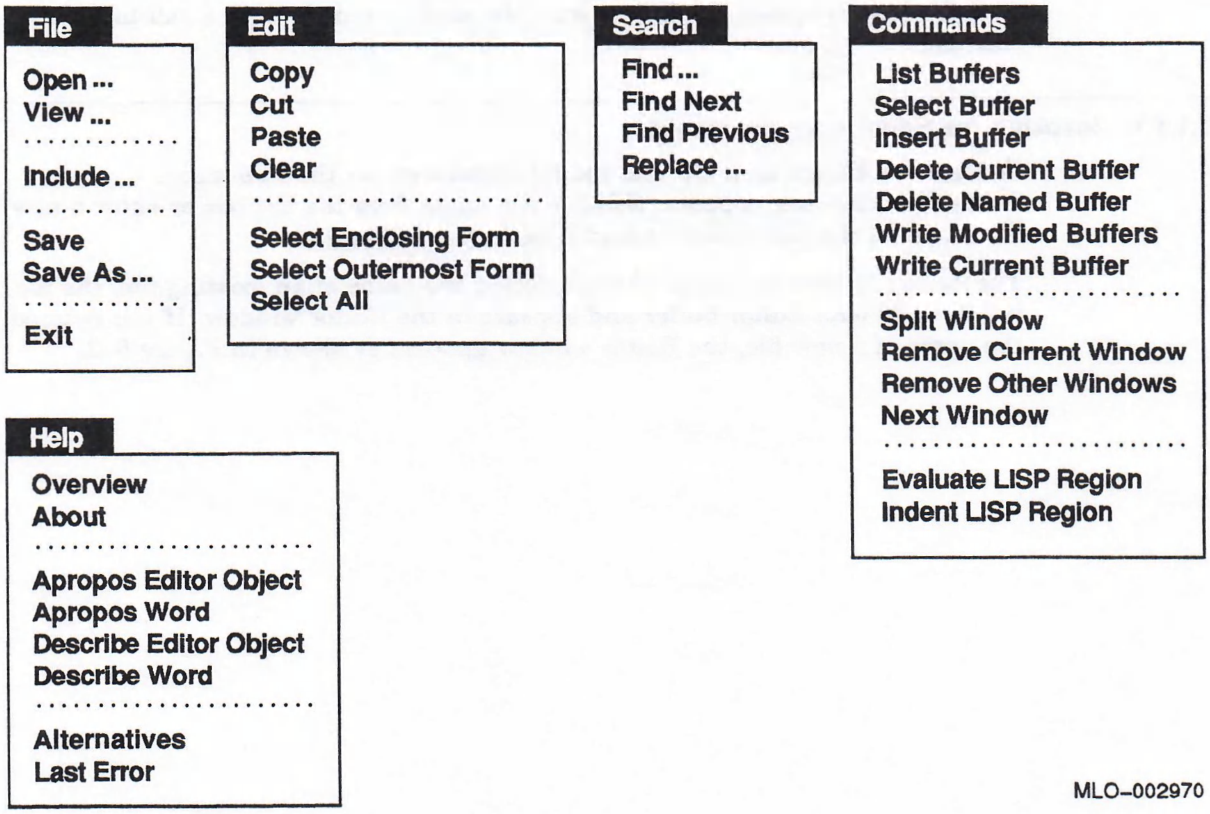
Using the VAX LISP Editor in DECwindows

VAX LISP provides a built-in editor for creating and editing LISP symbols and files. The Editor has special features that aid you in writing LISP programs, such as matching parentheses and indenting lines of code.

When you use the Editor in a DECwindows environment, you can move back and forth from the Editor to other LISP utilities without having to enter commands. You can also execute many of the Editor commands by clicking on menu items instead of typing command names or key bindings.

Figure 8–1 shows the Editor menus.

Figure 8–1: Editor Menus



This chapter explains how to use the Editor in default mode. 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 D.

Many of the Editor's capabilities can be modified or extended by writing new LISP code. For more information on how to customize the Editor, see the *VAX LISP/VMS Editor Programming Guide*.

8.1 Introduction to the Editor

This section describes the following basic steps in the editing cycle:

- Invoking the Editor
- Entering text
- Evaluating work in the Editor
- Saving work in the Editor
- Returning to LISP to evaluate code
- Returning to the Editor to make changes
- Exiting the Editor

8.1.1 Invoking the Editor

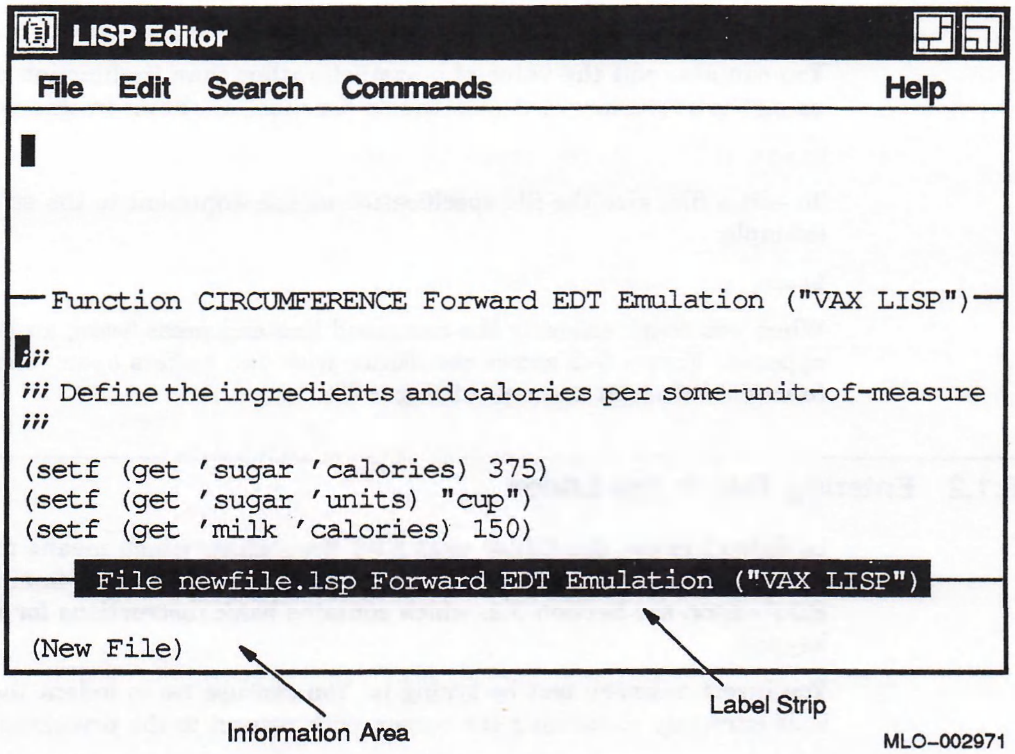
You normally invoke the Editor from the Listener utility. The following sections describe how to invoke the Editor from the menus and through a call to the `ED` function.

8.1.1.1 Invoking the Editor from the Menus

To start the Editor on a file, use the Ed menu item on the File menu.
A File Selection box appears. Select a file name from the list box or enter a new file name on the Selection line and click on the OK button.

The Editor window appears. If you selected the name of an existing file, the file is loaded into an Editor buffer and appears in the Editor window. If you entered the name of a new file, the Editor window appears as shown in Figure 8–2.

Figure 8–2: Editor Window with Two Buffers



```
Function CIRCUMFERENCE Forward EDT Emulation ("VAX LISP")
```

```
;;;
;;; Define the ingredients and calories per some-unit-of-measure
;;;

(setf (get 'sugar 'calories) 375)
(setf (get 'sugar 'units) "cup")
(setf (get 'milk 'calories) 150)
```

```
File newfile.lsp Forward EDT Emulation ("VAX LISP")
```

```
(New File)
```

The cursor appears at the top left of the Editor window. The label strip near the bottom of the window displays the following information:

- The name of the file currently being edited.
- The direction of movement, either forward or backward, for EDT keypad commands that require this information.
- The major and minor style currently in use. Figure 8–2 reflects the default values for these items, EDT Emulation and VAX LISP, respectively. For more information on the major and minor styles, see Section 8.6.1.4.

The information area at the bottom of the screen provides messages about the Editor operations and errors. For example, the message "New File" appears in this area when you begin to edit a new file.

To invoke the Editor on a symbol's function definition instead of a file, use the ED item on the Operations menu. Select the symbol whose function definition you want to edit using the mouse and then click on the ED item. An Editor window appears. See Figure 8–2.

8.1.1.2 Invoking the Editor Using a Command Line

You can use the ED function to invoke the Editor. The first time you invoke the Editor during a LISP session, supply an argument to the ED function. The argument identifies the object or file you want to edit.

To edit a function definition, give a symbol as the argument. For example, you can enter the following form to edit the function definition of the symbol CALORIES:

```
Lisp> (ed 'calories)
```

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

```
Lisp> (ed 'calories :type :value)
```

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

```
Lisp> (ed "newfile.lsp")
```

When you finish entering the command line and press Return, an Editor window appears. Figure 8–2 shows the Editor with two buffers open: one editing a function definition and one editing a file.

8.1.2 Entering Text in the Editor

In default mode, the Editor uses EDT Emulation, which means the keypad keys function the same as in the EDT editor. If you are not familiar with the EDT editor, see Section 3.2, which contains basic instructions for using the EDT keypad.

You insert ordinary text by typing it. You can use Tab to indent the line of LISP code currently containing the cursor, with respect to the preceding lines of code.

If you type a right parenthesis, the Editor highlights the left parenthesis momentarily. If the left parenthesis has scrolled off the screen, the Editor displays the line that contains it in the information area with the left parenthesis highlighted.

8.1.3 Using Editor Commands

You can use Editor commands to format text or select text for further editing operations. You can execute commands in three ways:

- Enter the command by name
- Click on the appropriate menu item
- Enter the appropriate key bindings

To enter a command by name, press Ctrl/Z. The prompt "Enter command name" appears in the information area. Type the command name and press Return. To cancel a command and clear the information area, press Ctrl/C.

Many commonly used Editor commands appear on Editor menus. When you execute an Editor command by choosing a menu item, the result is the same as if you had entered the command by name.

Many commands are bound by default to keys or key sequences. If the command is bound, you can execute it by entering the key binding.

For example, the "Select Outermost Form" command selects the text at the outermost LISP form containing the cursor. You can execute this command by clicking on the Select Outermost Form item on the Operations menu. You can also run this command by entering Ctrl/Z and the command name and pressing Return. A third way to execute the command is to press Ctrl/X Ctrl/Space to which the command is bound. In each case, the result is the same.

Appendix E contains a complete listing of the Editor commands and their key bindings.

8.1.4 Evaluating Work in the Editor

Before trying out or examining your code in another utility, you must evaluate it in the Editor. The "Evaluate LISP Region" command evaluates a region of LISP code and transports it to the LISP environment. To execute this command, select the region you want to evaluate and choose the Evaluate LISP Region item on the Commands menu. As the Editor evaluates the code, it displays messages informing you of the status of the evaluation process. After the code has been evaluated, it is available immediately in another LISP utility.

You can also use the "Write Current Buffer" and "Write Modified Buffers" commands to replace an existing function definition or value with a new function definition or value. "Write Current Buffer" affects only the current buffer; that is, the buffer whose window contained the cursor when you enter the command. "Write Modified Buffers" affects any buffer you have worked on since the last time the buffer was written. Both these commands are available as menu items on the Editor's Commands menu.

Evaluating code in the Editor provides a way to try out functions easily in other utilities. However, when you exit the LISP environment, all function definitions are lost. Therefore, before leaving LISP, be sure to save all function definitions to LISP files.

8.1.5 Saving Work in the Editor

To save your work in a file, choose the Save As item on the File menu. Type the file name and press Return. Saving a file this way is equivalent to typing the "Write Named File" command.

If you have already saved your work in a file, the "Write Current Buffer" and "Write Modified Buffers" commands provide another way to write a new version of the file. "Write Current Buffer" affects only the current buffer; that is, the buffer whose window contained the cursor when you enter the command. "Write Modified Buffers" affects any buffers you have worked on since the last time the buffer was written. Both these commands are available as menu items on the Editor's Commands menu.

Note that if you use these commands when you are editing a function definition or value, they update the existing function definition or value but do not write the LISP code to a file.

8.1.6 Returning to LISP

You can leave the Editor to examine or run your source code in another utility by clicking on the desired utility window. If you evaluated code in the Editor, it is immediately available. If you saved your work in a file, you must load the file before you can use it.

When you go to another utility, the Editor window remains on the screen. If you forgot to save your work, you can return to the Editor window and your file or function will be still available. However, if you exit the Editor without saving your files and then exit LISP, the work you did will be lost.

To end your editing session, use the Exit item on the Editor's File menu. See Section 8.1.8 for more information on exiting the Editor.

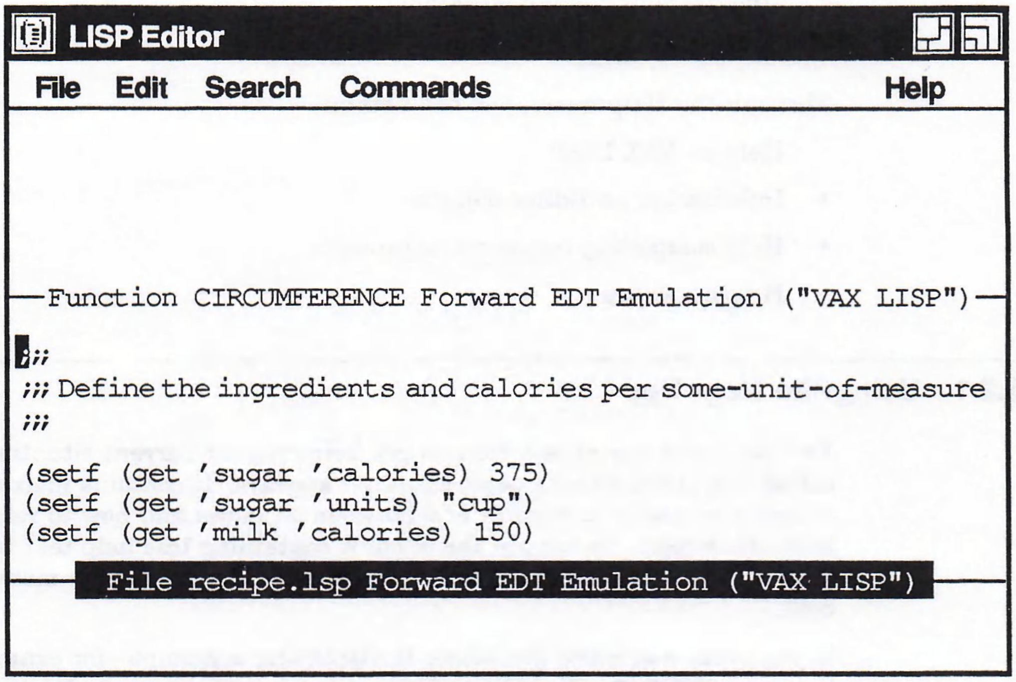
8.1.7 Returning to the Editor

When you leave the Editor without exiting, the Editor keeps open the buffer containing your LISP object or file. You can resume your editing session in the state you left it and make further modifications to the code, by simply going to the Editor window.

To start editing a new file in the Editor window, proceed as if you were invoking the Editor for the first time. Choose the Ed item on the File menu in the Listener or the Open item on the File menu in the Editor to edit a new file or the ED item on the Operations menu to edit a new LISP object.

The Editor opens another buffer and places the cursor in that buffer. Both the new buffer and the previously opened buffer appear in the Editor window in Figure 8–3.

Figure 8–3: Editor Window with Two Buffers



```
—Function CIRCUMFERENCE Forward EDT Emulation ("VAX LISP")—
```

```
;;;
;;; Define the ingredients and calories per some-unit-of-measure
;;;

(setf (get 'sugar 'calories) 375)
(setf (get 'sugar 'units) "cup")
(setf (get 'milk 'calories) 150)
```

```
 File recipe.lsp Forward EDT Emulation ("VAX LISP")
```

MLO–002972

Each buffer has its own window and label strip. The buffer containing the cursor and the reverse-videoed label strip is currently active. For information on how to move around in buffers and windows, see Section 8.4.

8.1.8 Exiting the Editor

When you exit the Editor, the Editor window disappears from the screen and you are returned to the window that had input focus before the Editor.

To exit the Editor, click on the Exit item on the File menu or press Ctrl/Z and enter the `"Exit"` command. If you have modified buffers, the Editor informs you that all buffers will be lost and asks if you want to continue. If you type Y, the Editor displays the name of each modified buffer one at a time and asks if you want to save it.

If a buffer contains a file, the Editor saves a new version of the file. If a buffer contains a function, the Editor updates the function definition. The function definitions will eventually be lost when you exit LISP. To avoid this happening, write any function definitions you want to save to files before exiting the Editor as described in Section 8.1.5.

8.2 Getting Help

The Editor provides help through the Help key and the Help menu. By using the Help key, you can obtain:

- General help on the Editor
- Help on your current situation
- Help on prompts

Through the Help menu, you can obtain:

- Help on VAX LISP
- Information on Editor objects
- Help completing responses to prompts
- Help on errors

8.2.1 Using the Help Key

You can press Help at any time to get help on your current situation. A buffer 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, click on the Remove Current Window item on the Commands menu or press Ctrl/X Ctrl/R.

If you press Help while the Editor is displaying a prompt—for example, after you have pressed Ctrl/Z—the Editor displays help on the prompt. Typically, the help text 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.

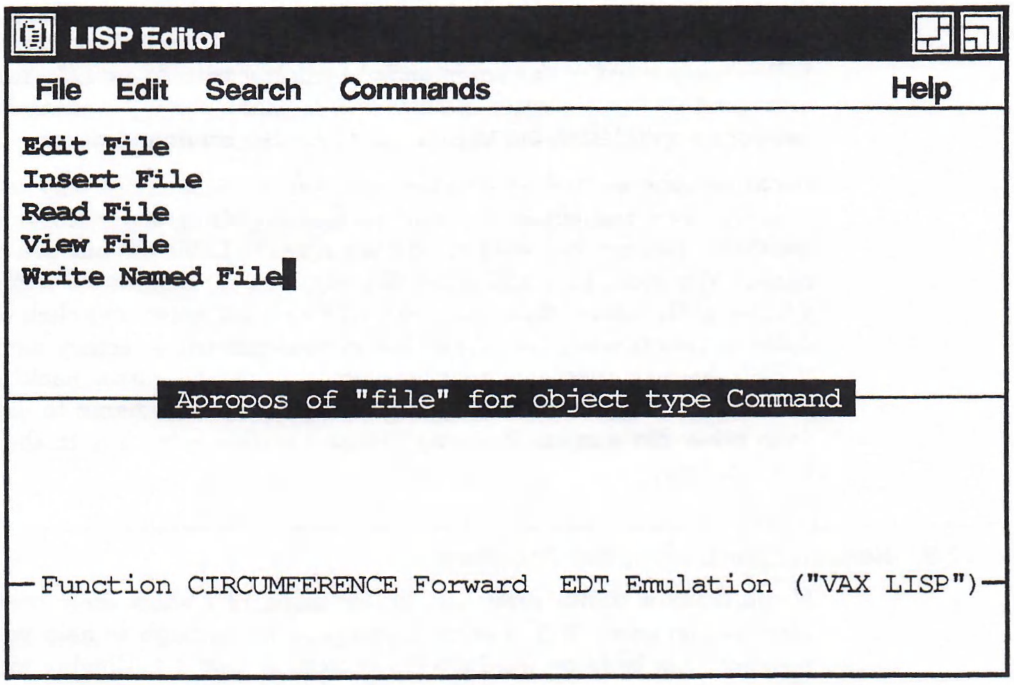
8.2.2 Using the Help Menu

You can use the Overview and About items on the Editor's Help menu to obtain general help and product information on VAX LISP, just as you can from the Listener. The Editor's Help menu also provides the specialized help on the Editor as described in the following sections.

8.2.2.1 Help on Editor Objects

You can use the Describe Editor Object and Apropos Editor Object items to obtain information on Editor objects. These items are equivalent to the `"Describe"` and `"Apropos"` commands and 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 display shown in Figure 8–4.

Figure 8–4: Apropos Display



```
Edit File
Insert File
Read File
View File
Write Named File

Apropos of "file" for object type Command

Function CIRCUMFERENCE Forward  EDT Emulation ("VAX LISP")
```

MLO–002973

You can also obtain descriptions of LISP symbols from the Editor when you are editing LISP code. Use the Describe Word item to invoke the LISP DESCRIBE function on the word at the current cursor position. Use the Apropos Word item to invoke the APROPOS function on the word at the current cursor position.

You can move the cursor around in the window containing help text. When you are done, use the Remove Current Window item on the Commands menu or Ctrl/X Ctrl/R to remove this window and return to editing.

8.2.2.2 Help on Responses to Prompts

The Editor can help you enter responses to prompts in two ways. The first way is input completion. If you press Ctrl/Space 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 "shr" followed by Ctrl/Space, the Editor will complete the command name "Shrink Window" 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 inform you that the input is ambiguous, because more than one command starts with the string "New Li".

The second way is by listing alternatives. At any point when entering information to a prompt, you can obtain a list of the available alternatives by choosing the Alternatives item on the Help menu. This is equivalent to pressing PF1 PF2. 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 click on the Alternatives item and 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 provide an alternative way to to fill out file specifications and 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 press Ctrl/Z and enter the `"Edit File"` command, which then prompts you for a file name. You can type ".LSP" at this point and click on Prompt Show Alternatives 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. Pressing Ctrl/Space at this point fills in the rest of the file specification.

8.2.2.3 Help on Errors and Other Problems

If you make a minor error, the Editor displays a short error message in the information area. If this error message is not enough to help you correct the problem, use Help on the Last Error item or Ctrl/X ? to 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 8.5 contains more information on how to recover from these problems.

8.3 Editing LISP Code

This section describes the Editor's default pointer bindings under DECwindows and some useful ways to edit LISP code, including:

- Cutting and pasting text
- Finding and replacing text
- Moving the cursor by LISP forms
- Inserting nongraphic characters
- Changing case
- Inserting a file or buffer

8.3.1 Using the Pointer

Under DECwindows, the Editor's default pointer bindings have been changed from previous versions of VAX LISP to be compatible with the DECwindows style. Table 8–1 shows the Editor's default pointer bindings under DECwindows.

Table 8–1: Editor Default Pointer Bindings

Action	Binding
Click MB1	Positions the text cursor.
Press and drag MB1	Selects a region of text.
Click MB3	Copies the current primary selection to the click point and repositions the text cursor.
Press and drag MB3	Selects the secondary select region.
Release MB3	Copies the secondary select region to the active text insertion point.

MB2 is not bound by default as it is reserved for pop-up menus. You are still able to bind Editor commands to MB2. For further information, see the *VAX LISP/VMS Editor Programming Guide*. Pressing MB3, dragging the pointer, and releasing causes the secondary selection region (denoted by underlining) to be copied to the active text insertion point in any other DECwindows application (including other LISP windows) that accept secondary selection input. Note that this does not include the Editor window. Formal descriptions of the Editor pointer binding commands are in Part III of the *VAX LISP/VMS Editor Programming Guide*.

If you want to use the UIS pointer binding syntax, you can change to it with the `Set UIS Pointer Syntax` command. To change from the UIS pointer binding syntax to the DECwindows pointer binding syntax, use the `Set DECwindows Pointer Syntax` command.

8.3.2 Cutting and Pasting Text

You can select text and move it to another location within the Editor window or another LISP utility. You select and move the text by using MB1 or MB3 (see Section A.16) or by selecting one of the following on the Edit menu:

Select All	Selects all text in the buffer.
Select Enclosing Form	Selects the text of the innermost LISP form containing the cursor. Equivalent to using the `"Select Enclosing Form"` command.
Select Outermost Form	Selects the text of the outermost LISP form containing the cursor. Equivalent to using the `"Select Outermost Form"` command.

Next, select the Cut or Copy item from the Edit menu. Position the cursor where you want to insert the text. Click on Paste in the window, utility, or application to which you are moving the text. For example, if you are moving text from the Editor to the Listener, click on Paste on the Listener's Edit menu.

8.3.3 Finding and Replacing Text

You can use the Search menu to search through a text region for an occurrence of text or to replace a specified text string with another.

To search for text, choose the Find item from the Search menu and type the text string you want when the Editor prompts you.

The Editor searches in the direction specified. If the text is found, the cursor appears at the beginning of the text string and the text string is highlighted. If the text is not found, you receive a message. To find the next occurrence of the same string, choose Find Next or Find Previous from the Search Menu. If there is no current search string, the Find Next and Find Previous items are dimmed.

To replace text, select the Replace . . . item. Type the name of the old string and the new string when the Editor prompts you. At each occurrence of the search string, the Editor prompts you for a search string and a replacement string. You can answer as follows:

Space — Replace this occurrence and move to the next one.

S — Replace this occurrence and stay here. This option lets you 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 lets you clean up a replacement site without losing your place in the search cycle.

? — Display help on the possible responses to the query.

8.3.4 Moving in LISP Code

You can use Editor commands to move around by LISP forms. The command "`Next LISP Form`" and the command "`Previous LISP Form`" move the cursor from form to form within the current parentheses nesting level. The command "`End of Outermost Form`" moves the cursor to the end of the current or next outermost LISP form. The command "`Beginning of Outermost Form`" moves the cursor to the beginning of the current or previous outermost LISP form.

Four other commands let you 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

For information on how to bind these key sequences to commands, see Section 8.6.1.

8.3.5 Inserting Nongraphic Characters

You cannot insert some characters directly into your text. For example, you cannot insert a `#\^X` character by pressing Ctrl/X because the Editor interprets that character. The Editor provides two ways around this problem. In most cases, you can use Ctrl/X \. Then the Editor takes the next character you type and inserts it without interpretation. This procedure handles the case of `#\^X`, for example; you 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 without the leading `#\`. For example, the Editor graphic display for `#\^X` is `<^X>`.

8.3.6 Changing Case

You can use Editor commands to change the case of alphabetic characters in your text. Nonalphabetic characters are not affected.

Four commands that let you 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, define the select region, press Ctrl/Z, and enter the command. To use the commands that affect a word, position the cursor anywhere in the word, press 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, press Ctrl/Z, and enter the command.

8.3.7 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.

To insert the contents of a file, choose the Include item on the File menu. The Editor prompts you for a file name. Type the name of the file and press Return. The Editor inserts the contents of this file at the cursor location. You can use the Alternatives item or request input completion with Ctrl/Space while responding to this prompt. Using the Include item is equivalent to typing the `"Insert File"` command.

To insert a buffer, click on the Insert Buffer item on the Commands menu. The Editor prompts for the name of a buffer and then inserts the contents of the buffer at the cursor location. You can use the Alternatives item or request input completion with Ctrl/Space while responding to this prompt. Section 8.4 contains more information about buffers.

8.4 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 Editor window.

8.4.1 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 inside the Editor window. The Editor can keep track of many buffers at once, but normally displays the contents of no more than two buffers at a time.

Each time you invoke the Editor with an argument, the Editor creates a buffer having the same name as the file or LISP object you specify.

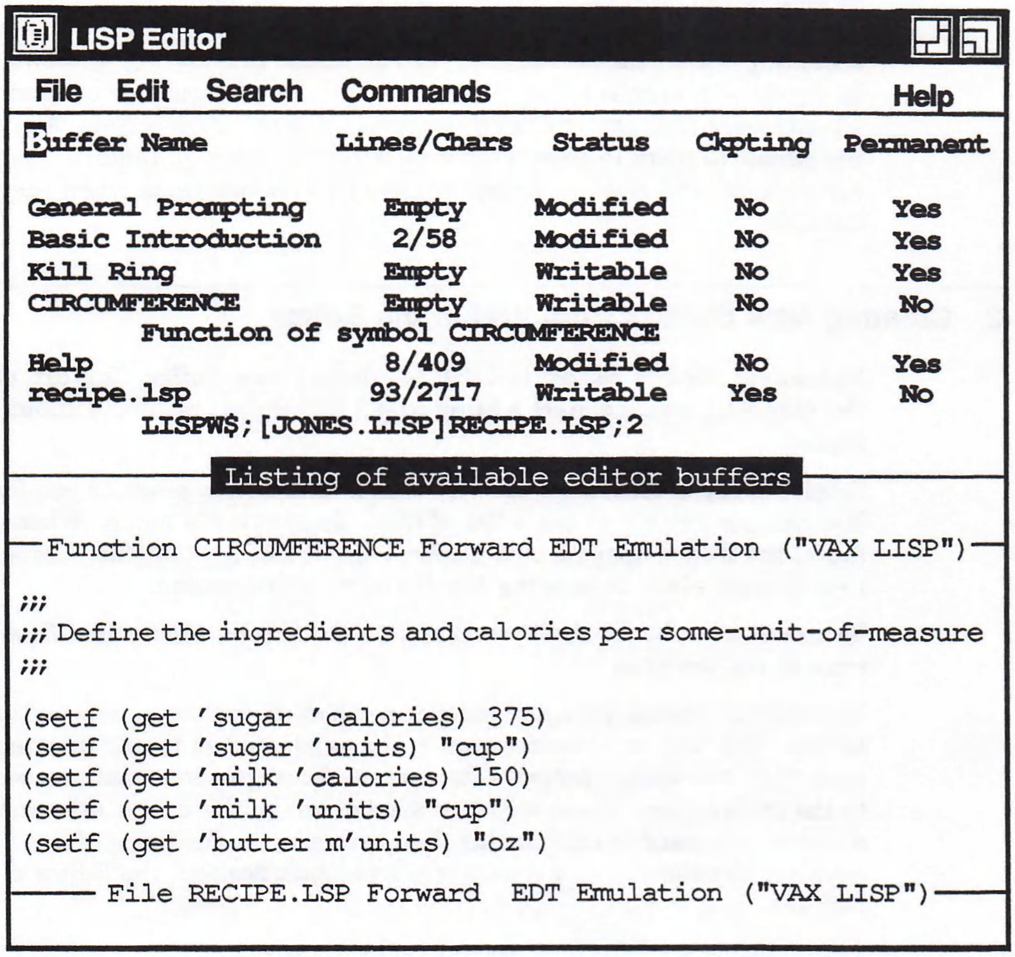
The buffer most recently created is the current window. If you type characters or enter Editor commands, the buffer viewed through the current window will be affected.

To make another window current, click on it. The cursor moves to the position occupied by the mouse. You can also change the current window by using the Next Window item on the Commands menu (or Ctrl/X Ctrl/N), which makes each window on the screen current in turn. When you change from one window to another this way, 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 inside the Editor window. Unless you use the Split Window item or 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. To obtain a listing of all the buffers in the Editor, use the List Buffers item on the Commands menu or the `"List Buffers"` command. For example, if you had invoked the Editor on a function and later a file, `"List Buffers"` might result in the display shown in Figure 8–5.

Figure 8–5: List Buffers Display



```
Buffer Name            Lines/Chars   Status     Ckpting   Permanent

General Prompting        Empty      Modified     No         Yes
Basic Introduction       2/58       Modified     No         Yes
Kill Ring                Empty      Writable     No         Yes
CIRCUMFERENCE            Empty      Writable     No         No
      Function of symbol CIRCUMFERENCE
Help                     8/409      Modified     No         Yes
recipe.lsp               93/2717    Writable     Yes        No
      LISPW$;[JONES.LISP]RECIPE.LSP;2
```

Listing of available editor buffers

Function CIRCUMFERENCE Forward EDT Emulation ("VAX LISP")

```
;;;
;;; Define the ingredients and calories per some-unit-of-measure
;;;

(setf (get 'sugar 'calories) 375)
(setf (get 'sugar 'units) "cup")
(setf (get 'milk 'calories) 150)
(setf (get 'milk 'units) "cup")
(setf (get 'butter m'units) "oz")
```

File RECIPE.LSP Forward EDT Emulation ("VAX LISP")

MLO–002974

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 CIRCUMFERENCE contains the line "Function of symbol CIRCUMFERENCE". The other buffers listed contain Editor information.

You can select a buffer for editing that is not currently on the screen by using the Select Buffer item on the Commands menu or the "Select Buffer" command. The Editor 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, choose 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 older one and replaces it with one displaying the contents of the buffer just selected.

Removing a window from the Editor 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 Remove Current Window item or Ctrl/X Ctrl/R to remove the current window from the screen and the Remove Other Windows item or 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 in the Editor window, 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. You can choose Next Window to move the cursor to them in turn, Remove Current Window to remove them when they are current, and Remove Other Windows to remove them when they are not current.

8.4.2 Creating New Buffers from Within the Editor

You do not need to return to LISP to create a new buffer. You can use one of the following ways to start editing new LISP objects or files without leaving the Editor.

Select the Open item from the File menu. The Editor prompts you for a file name. You can use PF1 PF2 to see a list of files. Type in a file name. When you press Return, the Editor opens a new buffer and window for that file. Using the Open item is equivalent to entering the "Edit File" command.

To create a read-only buffer from within the Editor, choose the View item and answer the prompts.

The "Ed" command provides another method of creating a new buffer within the Editor. The "Ed" command works in the same way as the ED function. When you type "Ed", the Editor 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.

8.4.3 Working with Buffers

Buffers generally take care of themselves. The only three common situations in which you need to deal directly with buffers 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 8.6.1.4)
- The major and minor styles that are active in that buffer (see Section 8.6.1.3)

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.

8.4.3.1 Saving Buffer Contents

You can use three commands to save buffer contents. The "Write Current Buffer" and "Write Modified Buffers" commands (discussed in Section 8.1) 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 item or "Exit" command and request that modified buffers be saved.

When you leave the Editor window to work in another utility, your buffers are not written, but they are available to you when you resume the Editor. If, however, you should leave the Editor and then exit LISP, the contents of your buffers will be lost. Section 8.5 explains how you can partially recover from this situation.

8.4.3.2 Deleting Buffers

You can use two commands to delete buffers. The "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 tne 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 RECIPE.LSP;1 and you delete the buffer "RECIPE.LSP", the file RECIPE.LSP;1 is not deleted. However, the file RECIPE.LSP;2, which would have been created if you had saved the buffer contents, is not created.

8.4.3.3 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 press Return, the Editor deletes the current contents of the buffer, replacing them with whatever you are trying to edit.

8.4.4 Manipulating Editor Windows

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

- "Next Window" to make the next window the current window
- "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 let you 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 or the Split Window item on the Commands menu let you 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 in the Editor window.

8.4.5 Moving Text Between Buffers

You can move or copy text from one buffer to another. For example, if you have worked on the definition of a function, you can move it to a buffer in which you are editing a file. Two ways of doing this are:

- Delete or cut the text from the source buffer, change to the destination buffer, and undelete or paste the text in the destination buffer
- Use the Insert Buffer item or the "Insert Buffer" command to insert an entire buffer in another.

8.5 Recovering from Problems

The Editor provides facilities that let you recover from problems with all or most of your work intact. Section 8.2.2.3 describes 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 RECIPE.LSP;2, the associated checkpoint file will be named RECIPE.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.)

8.6 Customizing the Editor

You can customize the Editor by binding keys or key sequences to commands. You can bind keys to commands that have no default bindings or change default bindings that exist. Section 8.6.1 describes how to bind keys to commands.

You can define keyboard macros to execute a series of keystrokes such as inserting key bindings that invoke commands. Section 8.6.2 describes how to define keyboard macros.

8.6.1 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 Section 8.6.1.1 and Section 8.6.1.2, 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. Section 8.6.1.1 and Section 8.6.1.2 describe how to specify the key or key sequence. Section 8.6.1 describes how to select a key or key sequence to bind.
- You can optionally supply the context in which the binding is effective. Section 8.6.1.4 explains the key binding context.

8.6.1.1 Binding Within the Editor

The "Bind Command" command lets you 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 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:

```
<^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:

```
<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 8.6.1.4 for more information on binding context.)

8.6.1.2 Binding from the LISP Interpreter

The BIND-COMMAND function lets you 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 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 8–2 lists each key and the character sequence it generates.

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 8–2: 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)
[↑]	`#\ESCAPE #\[ #\A`
[↓]	`#\ESCAPE #\[ #\B`
[→]	`#\ESCAPE #\[ #\C`
[←]	`#\ESCAPE #\[ #\D`

(continued on next page)

Table 8–2 (Cont.): Characters Generated by Keys

Key	Characters Generated
	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 #\~`
	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 8.6.1.4 describes binding context in more detail.

The following example binds 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 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.)

8.6.1.3 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) completes 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 uppercase and lowercase. 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 keyboards 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 E contains a list of all the key bindings supplied with the Editor. Section 8.6.1.4 explains how a single key or key sequence can be bound to two different commands in different contexts.

8.6.1.4 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 causes 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. See Section 8.1.1.1.

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 first checks to see if that key is:

1. Bound in the context of the current buffer.
2. Bound in the context of one of the current minor styles, examining the most recently activated style first.
3. Bound in the context of the current major style.
4. 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"`.

8.6.2 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.

Chapter 9

Using the VAX LISP Inspector

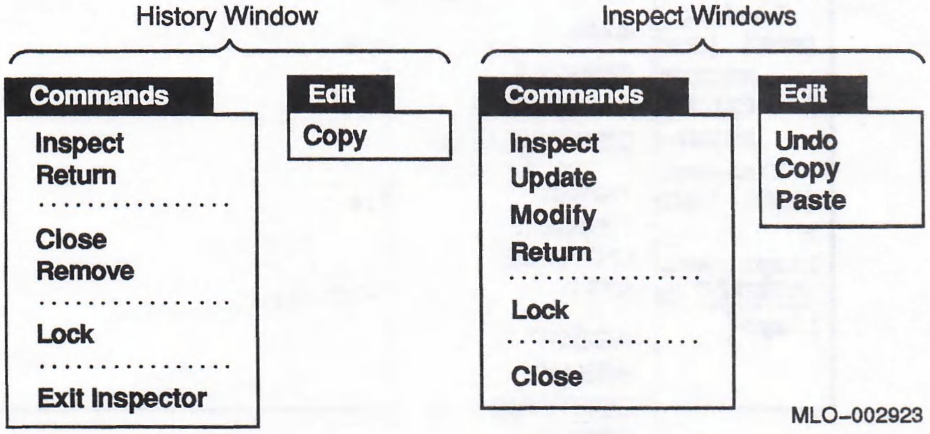
The VAX LISP Inspector, a utility running under DECwindows, is used to examine and modify static data structures. The Inspector displays an object's type and components. You can then select additional objects from this display, inspect them in turn, modify the modifiable components, and return the modified values to the form in which you called the INSPECT function.

This chapter describes the following operations:

- Invoking the Inspector
- Exiting the Inspector
- Inspecting objects
- Modifying objects
- Updating the Inspector display
- Returning a value from the Inspector

Figure 9–1 shows the Inspector's pull-down menus.

Figure 9–1: Inspector Menus



The following section explains the difference between History and Inspect windows. The examples in this chapter are based on the RECIPE program from Chapter 5. To run the examples, load the LISP$EXAMPLES:RECIPE.LSP file as described in Chapter 5.

9.1 Invoking the Inspector

You can invoke the Inspector from any VAX LISP utility: the Listener, Editor, or Debugger. The two methods for invoking the Inspector are:

- Choose the INSPECT item on the Operations menu.
- Call the `INSPECT` function in a `READ-EVAL-PRINT` loop.

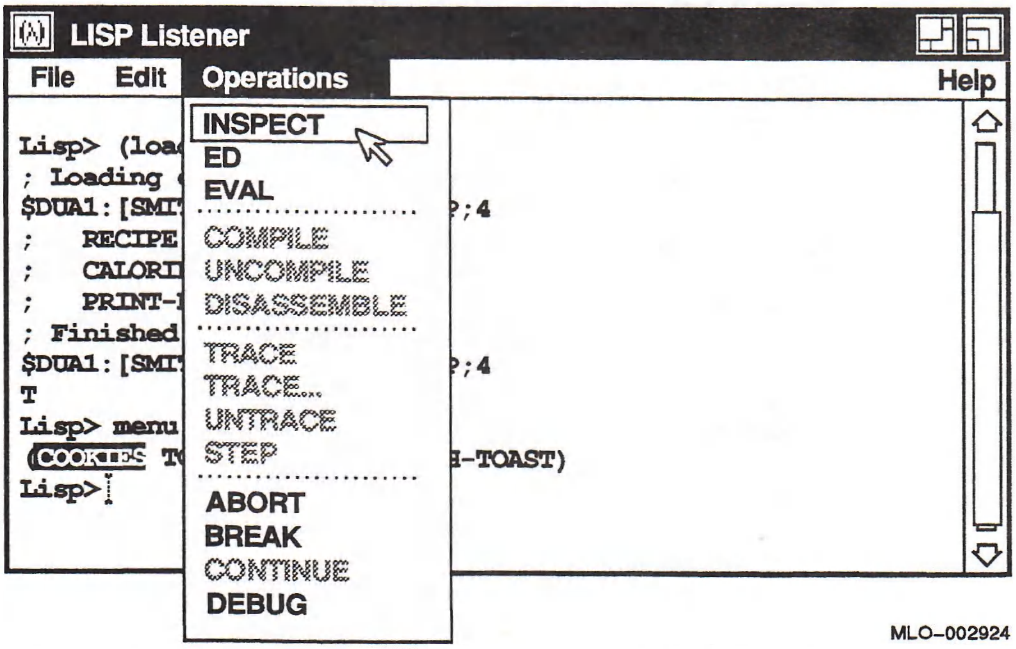
The INSPECT item is not available on the Editor's Commands menu, but you can evaluate a form containing the `INSPECT` function with the `"Evaluate LISP Region"` Editor command (see Chapter 8).

To use the INSPECT menu item, you must first select the LISP object to be examined. Then, pull down the Operations menu and choose INSPECT. When you invoke the Inspector this way, it runs asynchronously. See Section 9.1.1 for details on asynchronous mode. See Section 9.1.2 for details on synchronous mode.

Figure 9–2 illustrates using the Listener's Operations menu to invoke the Inspector on the symbol `COOKIES`, as follows:

1. Return a list of valid `RECIPE` structures by typing `MENU` at the Listener prompt.
2. Move the pointer to the `COOKIES` entry in the list and click MB1. This selects the object that you want to inspect.
3. Pull down the Operations menu and choose INSPECT.

Figure 9–2: Invoking the Inspector



```
Lisp> (loa
; Loading
$DUA1:[SMI                    ?;4
;    RECIPE
;    CALORI
;    PRINT-I
; Finished
$DUA1:[SMI                    ?;4
T
Lisp> menu
(COOKIES T                    I-TOAST)
Lisp>
```

To use the INSPECT function, type the object to be examined as its argument, as in:

```
Lisp> (inspect 'cookies)
```

If you do not specify an object when you invoke the Inspector for the first time, nothing is displayed in the History and Inspect windows. If the Inspector is already running, its windows are brought to the front.

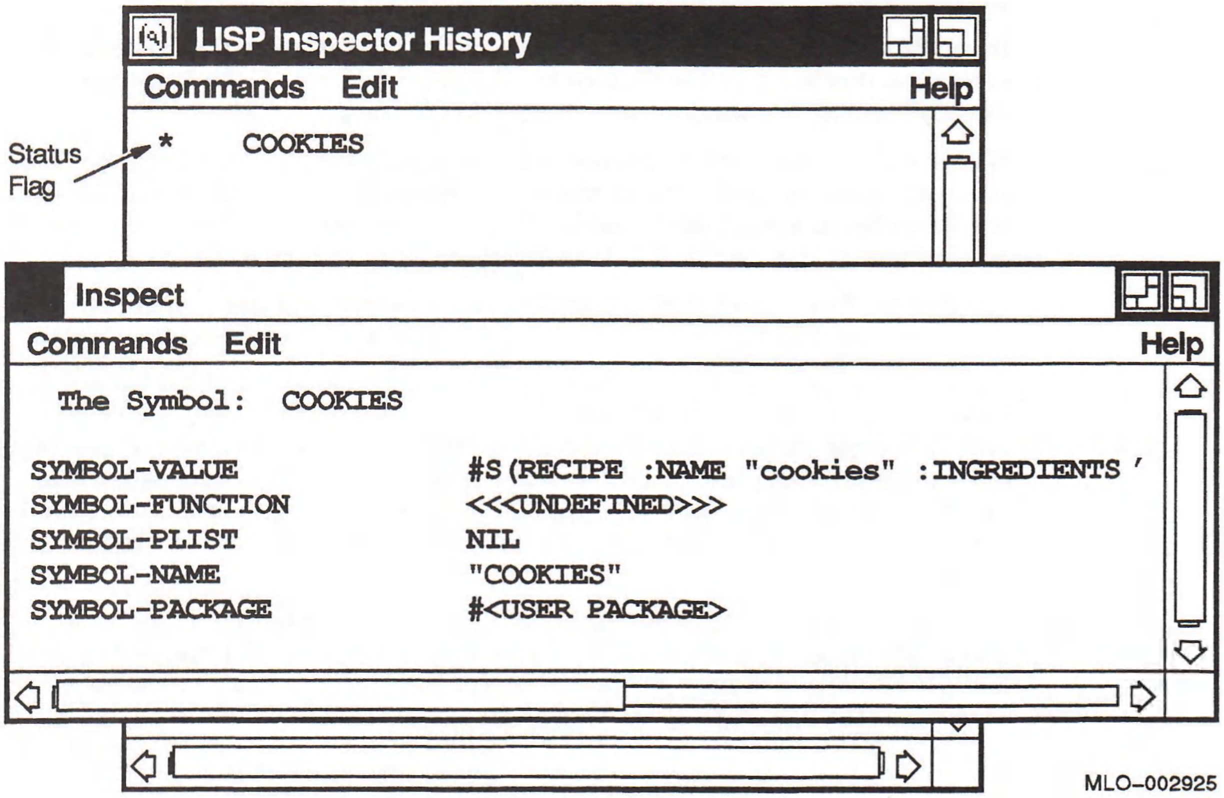
When you call the INSPECT function, you can specify whether the Inspector runs in asynchronous or synchronous mode. The default is :PARALLEL T, which runs the Inspector in asynchronous mode. Set :PARALLEL to NIL to run the Inspector synchronously. See Section 9.1.2 for details on synchronous mode.

Regardless of invocation method, the Inspector keeps a pointer to each object in LISP memory. The Inspector does not make its own copy of objects you inspect.

Figure 9–3 shows the windows that appear after invoking the Inspector on the symbol COOKIES. The Inspector uses two kinds of windows: LISP Inspect and LISP Inspector History. Each Inspect window displays the components and value bindings of an individual object. At this point, COOKIES is the only object that has been inspected, so there is only one Inspect window. (The Inspect window in Figure 9–3 has been resized and moved to show more of its contents and to fit on the page.)

The History window shows you a sequential list of all the objects you have examined (again, COOKIES is the only object at this point). The status flag at the left side of the History window indicates that COOKIES has an open Inspect window. Section 9.3 explains in detail how to use these windows.

Figure 9–3: Inspect and History Windows



```
*    COOKIES
```

```
  The Symbol:  COOKIES

SYMBOL-VALUE             #S(RECIPE :NAME "cookies" :INGREDIENTS '
SYMBOL-FUNCTION          <<<UNDEFINED>>>
SYMBOL-PLIST             NIL
SYMBOL-NAME              "COOKIES"
SYMBOL-PACKAGE           #<USER PACKAGE>
```

MLO–002925

You can run the Inspector in either asynchronous or synchronous mode. The following two sections explain each mode.

9.1.1 Asynchronous Mode

By default, the Inspector runs asynchronously, or concurrently, with your other VAX LISP utilities. The window from which you invoked the Inspector (in this case, the Listener) retains input focus, and any program running in that window continues to run. The Inspector immediately returns the object to which it was applied; the History and Inspect windows are a side effect.

CAUTION

> In asynchronous mode, the VAX LISP Inspector lets you modify data structures while you are running a program that may be using those structures. Be aware of the effects of such changes on your program and on the LISP image. (See Section 9.4 for information on modifying objects.)

The information displayed by the Inspector is static. Inspect windows are not automatically refreshed if another program in your LISP image changes the data structure being examined. You can make sure an Inspect window reflects the current state of your LISP image by choosing the Update item on the Commands menu (see Section 9.5 for more information on updating windows).

9.1.2 Synchronous Mode

When you invoke the Inspector in synchronous mode, your program is suspended until the Inspector returns. You run the Inspector in synchronous mode by calling the INSPECT function with the :PARALLEL keyword set to NIL. The format of the INSPECT function is:

INSPECT &OPTIONAL *object* &KEY :PARALLEL

The *object* may be any LISP object. (The default value for :PARALLEL is T.)

When the Inspector is running in synchronous mode, you can specify the value it returns by choosing the Return item on the Commands menu (see Section 9.6). If you do not specify a return value, the Inspector returns the object on which it was invoked when you exit the Inspector.

Synchronous mode lets you use the Inspector as a debugging tool. For example, you could use the following form with the RECIPE program:

```
Lisp> (calories (inspect 'toast :parallel nil))
```

The CALORIES function would suspend, while you used the Inspector to examine and perhaps modify TOAST and other data, and resume when you returned a value or exited the Inspector.

After you return a value, the Inspector continues to run, but in asynchronous mode.

9.2 Exiting the Inspector

Exit the Inspector by choosing the Exit Inspector item on the Commands menu. A caution box appears on your screen to confirm that you want to exit the Inspector. Click on the Yes button to exit the Inspector.

Exiting destroys the Inspector's pointers to objects examined during the session you are exiting. Thus, objects that only the Inspector points at are freed for garbage collection after you exit the Inspector.

If you are running the Inspector in synchronous mode (:PARALLEL NIL), the Inspector returns the object on which it was invoked when you exit the Inspector.

9.3 Inspecting Objects

Once you have invoked the Inspector by one of the methods described in Section 9.1, you can examine the components and values of any object in your LISP image. Table 9–1 lists each component that the Inspector displays for each LISP data type, and whether that component can be modified. Any component that is displayed can itself be inspected.

Table 9–1: Components of Inspectable Data Types

Data Type	Displayed Components	Modifiable Components
Integer	Integers	None
Floating-point number	Floating-point numbers	None
Ratio	Numerator Denominator	None
Complex number	Real part Imaginary part	None
Symbol	Value Function Plist Print name Package	Value Function Plist
Simple String	All characters	All characters
Array	All elements	All elements
Structure	All slot values	All slot values
Alien structure	All slot values	All slot values
Hash table	Rehash-size Rehash-threshold Associated array Size Count	Rehash-size Rehash-threshold Associated array
List	All elements	All elements
Function	Lambda-list	None

When LISP prints a sequence, such as a list or array, it displays the first 75 elements of the sequence then prompts if you want to see more. You can change the default number of elements with the CUSTOMIZATION function described in Appendix C.

The following sections explain how to:

- Specify objects to be inspected
- Manage Inspect windows
- Use the History window

9.3.1 Specifying Objects to Inspect

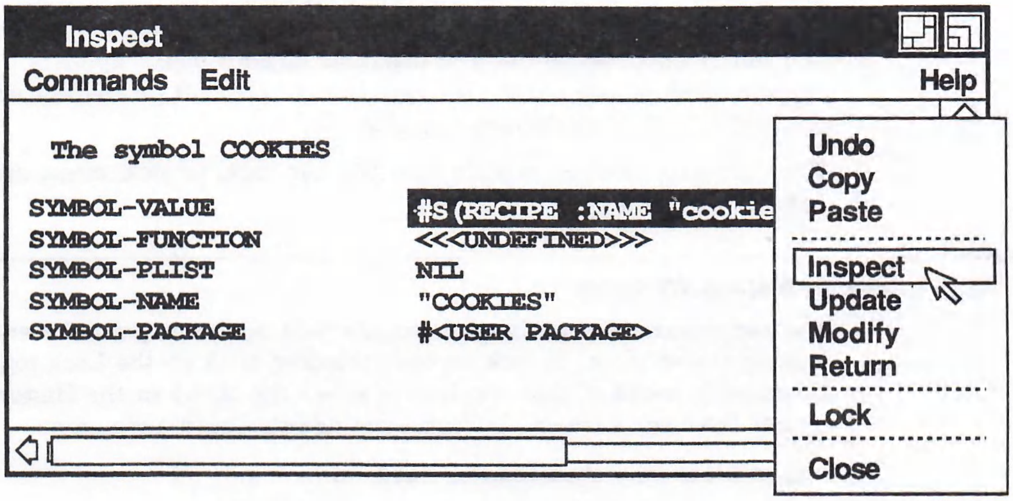
You can inspect objects in the History or Inspect windows in one of three ways:

- Select the object in either window and choose the Inspect item on any Commands menu.
- Double click MB1 on the object.
- Press MB2 on the object and choose the Inspect item on the pop-up menu.

Inspectable objects are underlined when you move the pointer cursor over them.

Figure 9–4 shows the Inspect window containing the COOKIES symbol. MB2 has been pressed on the symbol-value component to produce a pop-up menu. The availability of items on the pop-up menu depends on the status of the object under the pointer cursor when you press MB2. Menu choices are dimmed when they are disabled.

Figure 9–4: Inspecting a Component of an Inspected Object

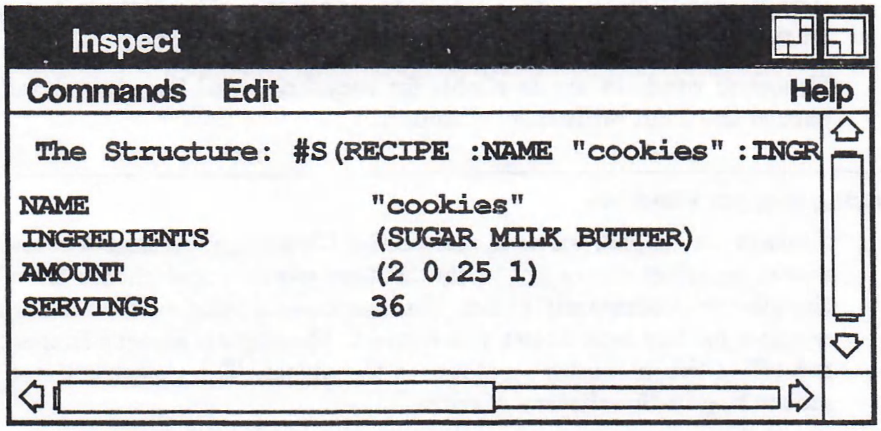


```
 The symbol COOKIES

SYMBOL-VALUE               #S(RECIPE :NAME "cookie
SYMBOL-FUNCTION            <<<UNDEFINED>>>
SYMBOL-PLIST               NIL
SYMBOL-NAME                "COOKIES"
SYMBOL-PACKAGE             #<USER PACKAGE>
```

MLO–002926

Figure 9–5 shows the result of inspecting this structure.

Figure 9–5: Inspecting a Structure



```
 The Structure: #S(RECIPE :NAME "cookies" :INGR

NAME                     "cookies"
INGREDIENTS              (SUGAR MILK BUTTER)
AMOUNT                   (2 0.25 1)
SERVINGS                 36
```

MLO–002927

If you specify text to be inspected, the Inspector attempts to coerce the selection into a symbol and inspects the result, on the assumption that inspecting a string is not as interesting as inspecting the symbol whose print name is the string.

9.3.2 Managing Inspect Windows

Each Inspect window displays the information for one inspected object. As you inspect additional objects, new Inspect windows are created. The default limit on the number of Inspect windows you can create in a single session is five. When you exceed this limit, the windows are recycled, with the newest object replacing the first object. That is, when you inspect a sixth object, it appears in the window previously allocated to the first object, a seventh object appears in the second object's window, and so on. You can change the limit on Inspect windows with the CUSTOMIZATION function (see Appendix C).

The following sections explain how you can lock, unlock, close, and remove Inspect windows.

9.3.2.1 Locking Inspect Windows

You can ensure that an Inspect window will not be recycled for another object by locking the window. To lock an open window, click on the Lock toggle item on the Commands menu of that window, or select the object in the History window and choose the Lock item on the History window's Commands menu.

You can lock only open Inspect windows.

Locked windows do not count toward the maximum number of Inspect windows allowed. For example, suppose you have five open Inspect windows and your limit is five (the default). If you lock one of them, you can inspect a sixth object without recycling any of the four open but unlocked Inspect windows.

9.3.2.2 Unlocking Inspect Windows

You can unlock a window by turning off the Lock toggle item on its Commands menu, or by selecting the object in the History window and choosing the Unlock item on the History window's Commands menu. The Unlock item is also available on the pop-up menu if the pointer is in a locked window.

Unlocked windows are available for recycling until they are closed, even if they exceed the limit on Inspect windows.

9.3.2.3 Closing Inspect Windows

To close an Inspect window, choose the Close item on that window's Commands menu, or select the object in the History window and choose the Close item on the History Commands menu. You can close a window that is locked, freeing a window for the next object you inspect. Closing an object's Inspect window does not affect the Inspector's pointer to the object. The object is listed with a blank status flag in the History window.

The Close menu item is available on the Inspector's pop-up menu if the pointer cursor is over an object that has an open Inspect window.

9.3.2.4 Removing Inspected Objects

Removing an object from the Inspector History frees the Inspector's pointer to it and closes its Inspect window (if any). You can remove any object in the History window, regardless of the status of its Inspect window.

You can remove an object by selecting the object in the History window and choosing the Remove item on the History window's Commands menu. The Remove menu item is also available on the History window's pop-up menu if the pointer cursor is over an object that has been inspected.

NOTE

> When you remove an object and its Inspect window, the object is deleted from the History window. If the object was pointed to only by the Inspector, it is now available for garbage collection.

9.3.3 Using the History Window

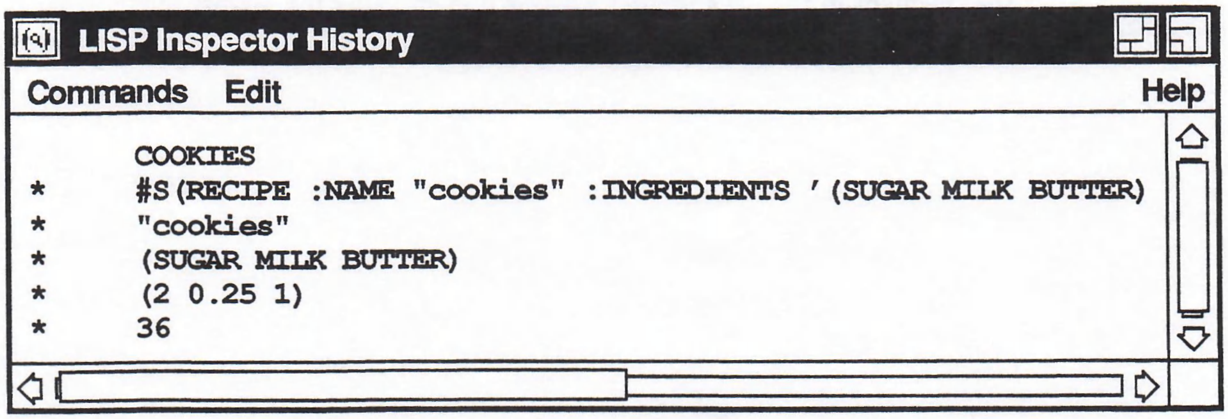
The Inspector History window shows a sequential list of all the objects you have examined in an Inspector session. It also shows the status of each object, by means of a status flag to the left of the object. The three possible flags are described in Table 9–2.

Table 9–2: Inspector History Status Flags

Flag	Meaning
space	The object does not have an open Inspect window.
*	The object has an open Inspect window.
L	The object has a locked Inspect window.

There is a default limit of five on the number of Inspect windows that can be open at one time. When you have exceeded that limit, the windows are recycled to accommodate the new objects. The status flag of an object whose Inspect window has been recycled changes from an asterisk to a blank space. Figure 9–6 shows the History window after inspecting COOKIES, the value component of COOKIES, and all four components of that value. Note that the status flag for COOKIES is a blank but the flags for the five most recently inspected objects are asterisks.

Figure 9–6: Inspector History Window



```
        COOKIES
*       #S(RECIPE :NAME "cookies" :INGREDIENTS '(SUGAR MILK BUTTER)
*       "cookies"
*       (SUGAR MILK BUTTER)
*       (2 0.25 1)
*       36
```

MLO–002628

You can reinspect a previously inspected object by double-clicking MB1 on the object in the History window. If you want to prevent an object's Inspect window from being recycled, you can lock it. (See Section 9.3.2.1 for details.)

The History window is the only Inspector window that has a shrink-to-icon button. Clicking MB1 on this button removes all Inspector windows from your screen but does not affect the Inspector's pointers to data structures. The three ways to expand the existing Inspector windows from the icon are:

- Click MB1 on the LISP Inspector icon in your Icon Box.
- Choose the Inspect menu item without first selecting an object.
- Call the `INSPECT` function without passing it an argument.

You can also reinvoke the Inspector on a new object. This expands the previous History and Inspect windows but also creates an Inspect window for the new object. Reinvoking the Inspector does not create a new History window.

9.4 Modifying Objects

You can change the modifiable components of an object by using the Modify menu item, or by using selection and clipboard operations in its Inspect window. You can modify only an entire component, not its subelements. See Table 9–1 for a list of the modifiable components of each LISP data type.

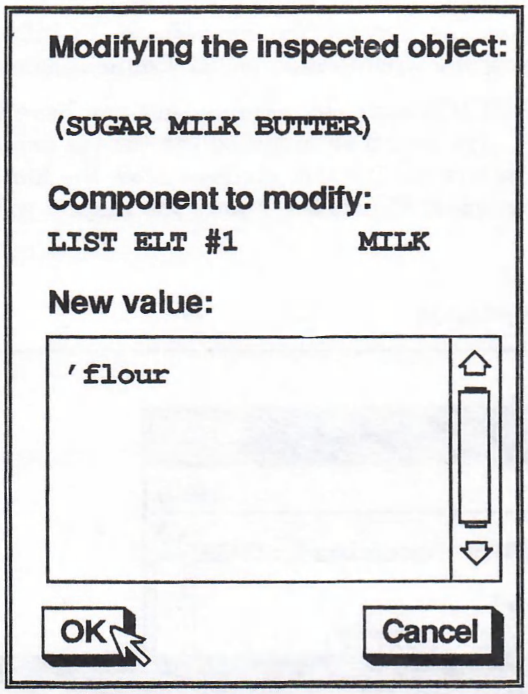
9.4.1 Using the Modify Menu Item

To change the modifiable components of an object, first select the component and then choose the Modify item on the Commands menu. A dialog box prompts you for the new value to be assigned to the component you have selected.

For example, suppose you have inspected the structure that is the value bound to `COOKIES` (see Figure 9–5) and want to change the `:INGREDIENTS` slot from `'(SUGAR MILK BUTTER)` to `'(SUGAR FLOUR BUTTER)`. If you select the `:INGREDIENTS` component in the structure's Inspect window, you must modify the entire list. (You cannot modify a subelement of a slot.) Therefore, inspect the list in the `:INGREDIENTS` slot. Then, you can change just one element of the list by selecting that element in the list's Inspect window and choosing the Modify menu item.

Figure 9–7 shows the Modify dialog box that appears when you change just one element in the list's Inspect window.

Figure 9–7: Modify Dialog Box



Modifying the inspected object:

(SUGAR MILK BUTTER)

Component to modify:

LIST ELT #1   MILK

New value:

'flour

MLO–002929

When you click on the OK button, the Inspector first evaluates your new value and then tries to set the value. If the evaluation or the modification results in an error, a message box describes the error and no modification occurs. Acknowledging the error returns you to the Modify dialog box.

Figure 9–8 shows the Inspect windows of the COOKIES structure and the :INGREDIENTS list after a successful modification.

9.4.2 Using the Clipboard

You can use the Paste item of the Inspect window's Edit menu to change the value of a component, with the following steps:

1. Transfer the object you want to be the new value to the clipboard (use the Cut or Copy item on any Edit menu).
2. Select the object in the slot you want to change.
3. Choose the Paste item on the Edit menu.

The clipboard contents replace the old slot value.

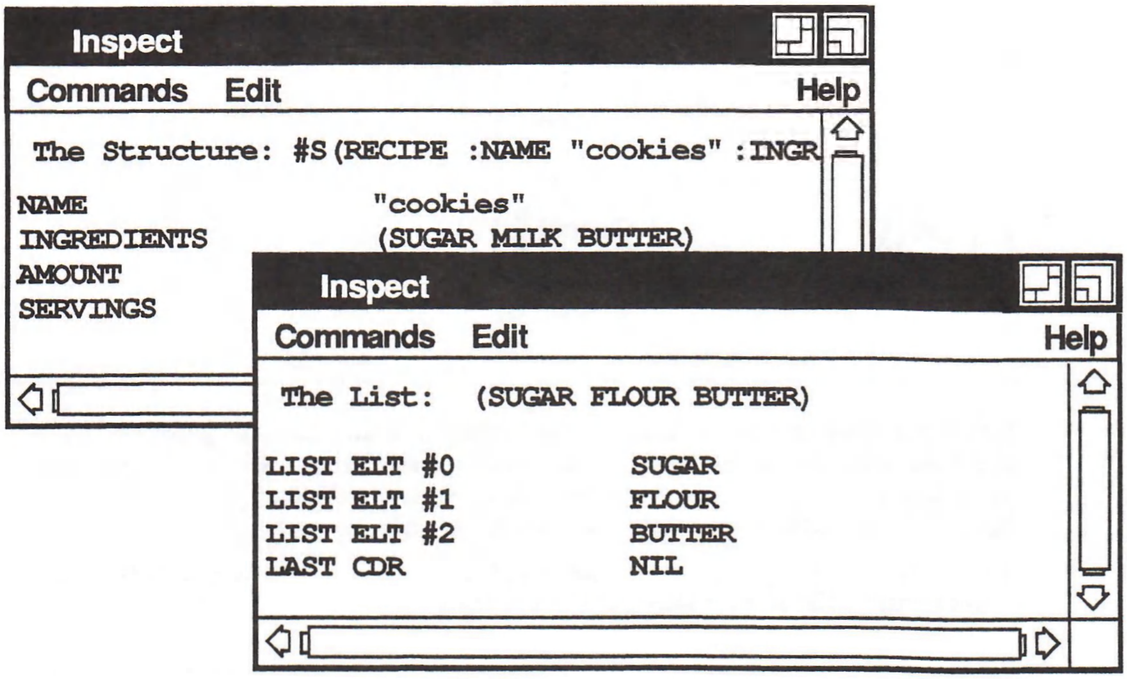
Immediately after a Paste operation, the Undo item on the corresponding Inspect window's Edit menu is active. Choosing the Undo item causes the slot to revert to its state before the Paste operation.

9.5 Updating the Inspector Display

When the Inspector is running asynchronously, the information in your Inspect windows may be obsolete. For example, a LISP program running in your Listener window could modify a data structure you have inspected. Similarly, using the Modify command in one Inspect window could make the print representation of an object in another window inaccurate. You can refresh the information in an Inspect window by choosing the Update item on the Commands menu.

For example, using the RECIPE example, assume that you have inspected `COOKIES` and its `:INGREDIENTS` list, then modified the list (as explained in Section 9.4). Figure 9–8 shows the Inspect windows after the Modify command but before the Update command; Figure 9–9 shows the Inspect windows after the Update command.

Figure 9–8: Inspect Windows Before Updating

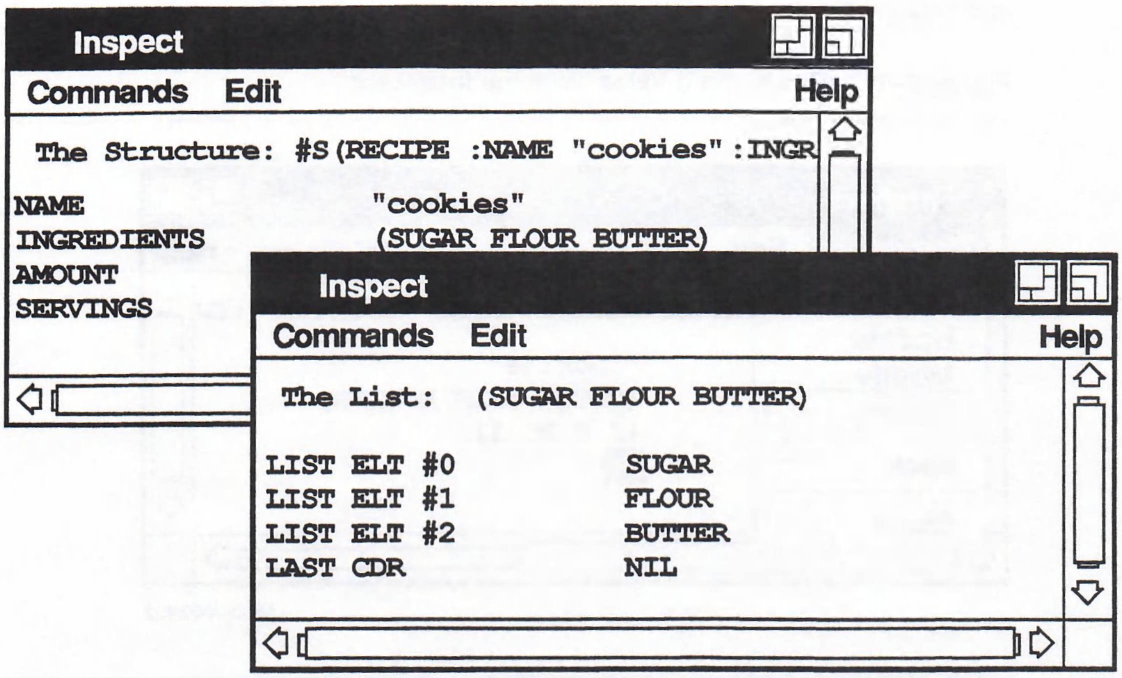


```
The Structure: #S(RECIPE :NAME "cookies" :INGR
NAME                    "cookies"
INGREDIENTS             (SUGAR MILK BUTTER)
AMOUNT
SERVINGS
```

```
The List:  (SUGAR FLOUR BUTTER)

LIST ELT #0          SUGAR
LIST ELT #1          FLOUR
LIST ELT #2          BUTTER
LAST CDR             NIL
```

MLO–002930

Figure 9–9: Inspect Windows After Updating



```
The Structure: #S(RECIPE :NAME "cookies" :INGR
NAME                    "cookies"
INGREDIENTS             (SUGAR FLOUR BUTTER)
AMOUNT
SERVINGS
```

```
The List:  (SUGAR FLOUR BUTTER)

LIST ELT #0          SUGAR
LIST ELT #1          FLOUR
LIST ELT #2          BUTTER
LAST CDR             NIL
```

The Update command is also available on the Inspector's pop-up menu. Press MB2 while in the Inspect window you want to refresh and choose the Update item.

9.6 Returning Values

When you invoke the Inspector in asynchronous mode, it immediately returns the object to which it was applied. However, when you invoke the Inspector in synchronous mode (`:PARALLEL NIL`), you can specify which object you want the Inspector to return. The object is returned to the form in which you invoked the Inspector.

To return the value of an object, select the object and then choose the Return item on the Commands menu. If you choose the Return menu item without having selected an object, the Inspector returns the object on which it was invoked.

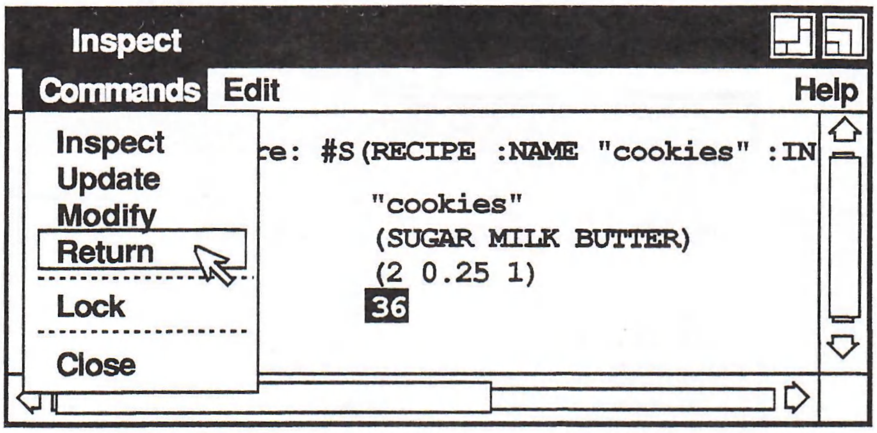
For example, suppose you want to perform some arithmetic on the number of servings for the `COOKIES` recipe. You invoke the Inspector in synchronous mode as follows:

```
Lisp> (+ 5 (inspect cookies :parallel nil))
```

This form invokes the Inspector and suspends the Listener until the Inspector returns a value. Ordinarily, the Inspector would return the object `COOKIES`. In this example, however, you want to return the value of the `SERVINGS` component.

To do this, select the value of the SERVINGS component, then pull down the Commands menu and choose Return, as shown in Figure 9–10. The INSPECT function returns the number 36 to the Listener, which performs the arithmetic and returns the number 41.

Figure 9–10: Returning a Value from the Inspector



```
e: #S(RECIPE :NAME "cookies" :IN
          "cookies"
          (SUGAR MILK BUTTER)
          (2 0.25 1)
          36
```

MLO–002932

After you return an object, the Inspector continues to run, but in asynchronous mode. The History and Inspect windows remain visible, and all data structures are intact, but you cannot return another value because nothing is waiting for the Inspector.

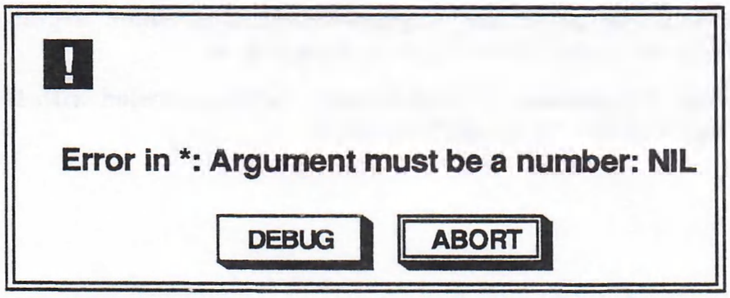
Chapter 10

Using the Debugging Utilities from the DECwindows Interface

Debugging is the process of locating and correcting programming errors. When you are in the Listener and an error is signaled, an error message is displayed and a caution box appears on the screen. This error message provides you with your initial debugging information: the error type, the name of the function that caused the error, and a description of the error.

The caution box shows you whether the error is fatal or continuable and lets you choose whether to return to the top-level prompt in the Listener or enter the Debugger. To enter the Debugger, click on the Debug button. To return to top-level from a fatal error, press Return or click on Abort. To continue from a continuable error, press Return or click on Continue. Figure 10–1 shows a caution box for a fatal error. Figure 10–2 shows a caution box for a continuable error.

Figure 10–1: Fatal Error Caution Box

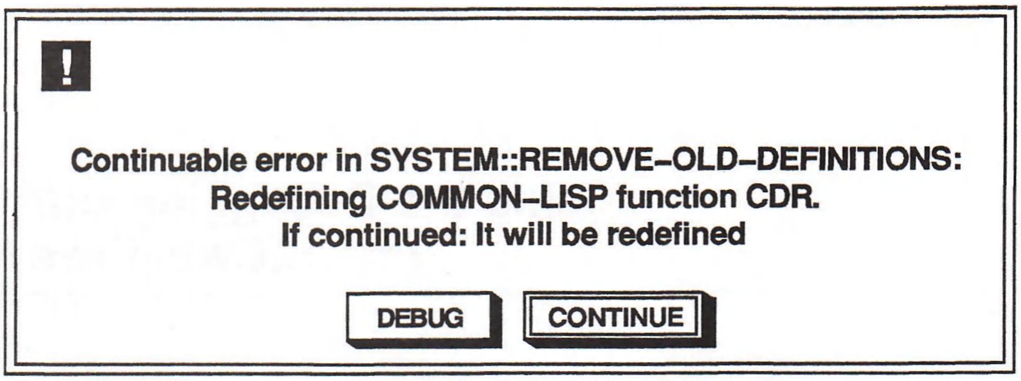


Error in *: Argument must be a number: NIL

MLO–002869

When you enter the Debugger, the following four windows are displayed on your workstation screen, unless you have closed one or more of them in a previous debugging session or with the VAX LISP CUSTOMIZATION function (see Appendix C).

Figure 10–2: Continuable Error Caution Box



Continuable error in SYSTEM::REMOVE-OLD-DEFINITIONS:
Redefining COMMON-LISP function CDR.
If continued: It will be redefined

MLO–002975

- Debug I/O
- Calling Stack
- Variable Bindings
- Debugger Commands

Figure 10–6 shows the four windows.

Input focus transfers to the Debug I/O window where the error message is displayed.

The Calling Stack window displays a quick backtrace of the control stack. A quick backtrace is a display of the function name in each frame of the control stack.

The Variable Bindings window displays the function name, argument list, and local variable bindings for the current stack frame.

The Debugger Commands window displays buttons labeled with the commands that you use with the VAX LISP Debugger.

Once you know the name of the function that caused the error, you can use the VAX LISP debugging functions and macros to locate and correct the programming error. Table 10–1 lists the debugging functions and macros with a brief description of each. See *VAX LISP/VMS Object Reference Manual* for more detailed descriptions.

Table 10–1: Debugging Functions and Macros

Name	Type	Menu	Description
APROPOS	Function	Help	Locates symbols whose print name contains a specified string argument or a substring and displays information about each symbol it locates.
APROPOS-LIST	Function	None	Locates symbols whose print names contain a specified string argument as a substring and returns a list of the symbols it locates.
BREAK	Function	Operations	Invokes the break loop.
DEBUG	Function	Operations	Invokes the VAX LISP Debugger.
DEBUG-CALL	Function	None	Returns a list representing the call at the current debug stack frame.
DESCRIBE	Function	Help	Displays detailed information about a specified object.
DISASSEMBLE	Function	Operations	Compiled code is reverse assembled and printed out in symbolic format.
DRIBBLE	Function	File	Copies the input and the output of an interactive LISP session to a specified file.
ED	Function	Operations	Invokes or resumes the VAX LISP Editor, potentially on the selection (only if the argument is a string will the ED function edit a file—most often it will edit a symbol's function definition).
INSPECT	Function	Operations	Invokes or resumes the VAX LISP Inspector, potentially on the selection.
ROOM	Function	None	Displays information about the state of internal storage and its management.
STEP	Macro	Operations	Invokes the stepper.
TIME	Macro	None	Displays timing information about the evaluation of a specified form.
TRACE	Macro	Operations	Enables tracing functions and macros.
UNTRACE	Macro	Operations	Disables tracing for functions and macros.

This chapter provides the following:

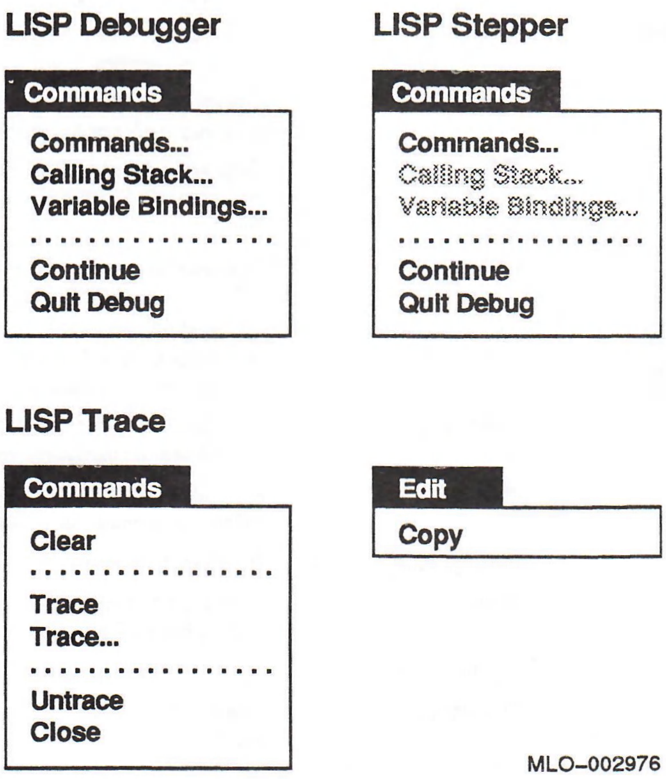
- A table of the functions and macros that give you debugging information.
- Illustrations of the debugging utility menus.
- Descriptions of two variables that control the output of the Debugger and the stepper and how to use them.
- Description of the control stack that stores calls to functions, macros, and special forms.

- Explanations of how to use the following debugging facilities from the DECwindows interface:
  - 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

The VAX LISP Editor and VAX LISP Inspector are two other tools that you can use for debugging. For an explanation of how to use the Editor, see Chapter 8. For an explanation of how to use the Inspector, see Chapter 9.

Figure 10–3 shows those debugging utilities pull-down menus that differ from Listener pull-down menus.

Figure 10–3: Debugging Utilities Pull-Down Menus



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

These variables can be changed with SETF (see Section 4.1).

10.2 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. Pushes the name of the function associated with the function, macro, or special form that is being called onto the stack frame.
2. Pushes the function's arguments onto the stack.
3. Creates a new stack frame.
4. Invokes the function.

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

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

Debugger commands show only significant stack frames, unless you specify the ALL modifier with a Debugger command.

10.3 Break Loop

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

10.3.1 Invoking the Break Loop

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

- Choose the BREAK item from the Operations menu while your program is being evaluated.
- Put the BREAK function in specific places in your program.

- 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 directly invoke the BREAK function while your program is being evaluated (see *VAX LISP/VMS Object Reference Manual* for a description of the BIND-KEYBOARD-FUNCTION function).

When invoked, 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:
your message
```

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

```
Break>
```

10.3.2 Exiting the Break Loop

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

- Choose the CONTINUE item from the Operations menu. The CONTINUE item on the Operations menu is available only when you are in a break loop and only has effect when the break loop is waiting for input. The CONTINUE item does not interrupt evaluation of your code. If you wish to stop evaluation of code in the break loop, choose the ABORT item first.
- Type Continue at the prompt.

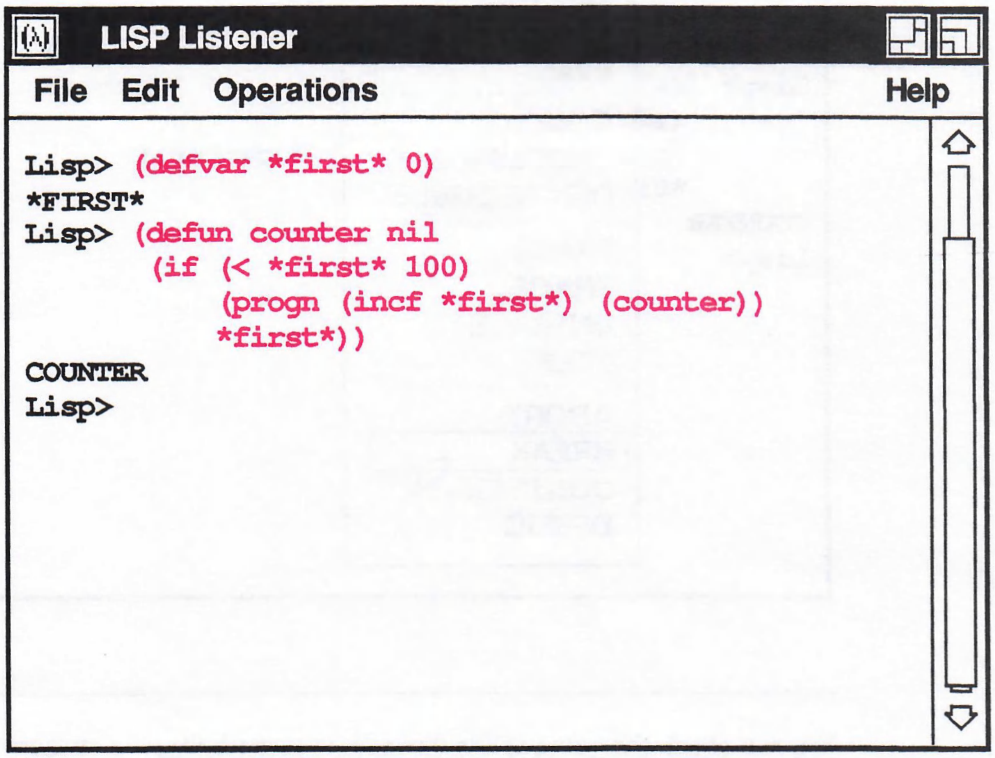
  ```
  Break> (continue)
  ```

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

10.3.3 Using the Break Loop

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

Figure 10–4: Defining a Variable

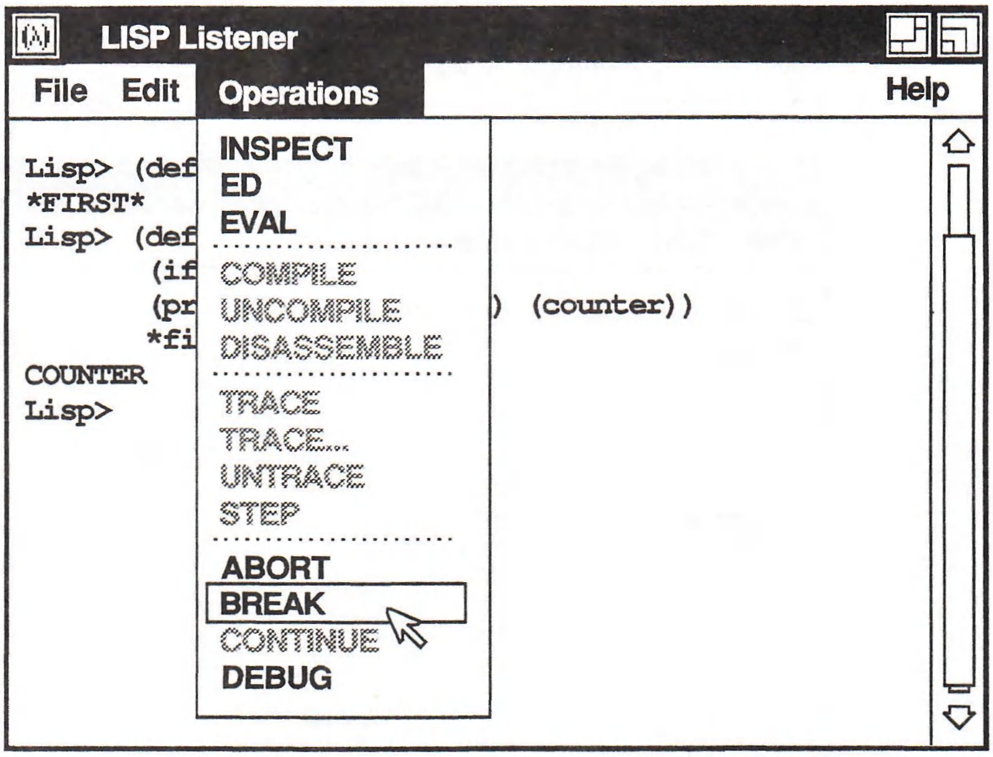


```
Lisp> (defvar *first* 0)
*FIRST*
Lisp> (defun counter nil
        (if (< *first* 100)
            (progn (incf *first*) (counter))
            *first*))
COUNTER
Lisp>
```

MLO–002977

You can interrupt the function's evaluation by invoking the break loop from the Operations menu during evaluation. For example:

Figure 10–5: Invoking the Break Loop



```
Lisp> (def
*FIRST*
Lisp> (def
        (if
        (pr                          ) (counter))
       *fi
COUNTER
Lisp>
```

MLO–002978

You can check the value of the variable `*FIRST*` by typing `*FIRST*` at the Break> prompt. The read-eval-print loop evaluates the variable at the point where the BREAK function interrupted evaluation of COUNTER.

If you call the CONTINUE function by choosing the Continue item from the Operations menu or typing Continue at the Break> prompt, the evaluation of the function COUNTER continues.

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*`.

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

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 enter with the mouse or by typing 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 10.4). For information on lexical environments, see *Common LISP: The Language*.

10.4 Debugger

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

- Display help.
- Evaluate a form or reevaluate the function call a stack frame stores.
- Handle errors.
- Change which stack frame is considered current.
- Inspect or modify the function call in a stack frame.
- Display a summary of the control stack.

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

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

10.4.1 Invoking the Debugger from DECwindows

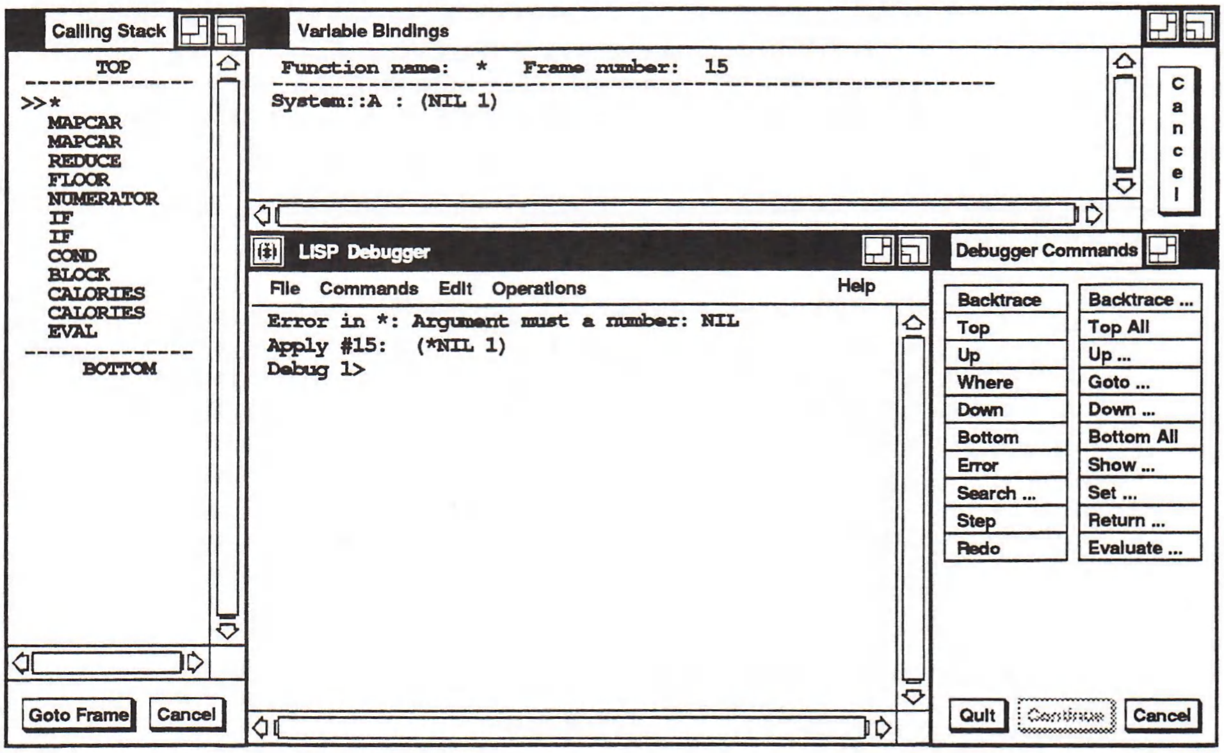
You can invoke the Debugger by:

- Clicking on the Debug button in one of the caution boxes
- Evaluating the `DEBUG` function
- Choosing the DEBUG item from the Operations menu

The DEBUG item ignores any objects that you have selected before invoking the debugger.

When you invoke the Debugger, four windows are displayed on your screen: the Debug I/O window, the Calling Stack window, the Variable Bindings window, and the Debugger Commands window. If you have closed the Calling Stack, Variable Bindings, or Debugger Commands window in a previous debugging session or with the VAX LISP `CUSTOMIZATION` function, that window is not displayed on your screen (see Appendix C). Figure 10–6 shows these four windows.

Figure 10–6: Debug I/O, Calling Stack, Variable Bindings, and Debugger Commands Windows



```
     TOP
--------------
>>*
   MAPCAR
   MAPCAR
   REDUCE
   FLOOR
   NUMERATOR
   IF
   IF
   COND
   BLOCK
   CALORIES
   CALORIES
   EVAL
--------------
     BOTTOM
```

```
 Function name:  *   Frame number:  15
-------------------------------------------------------------
System::A : (NIL 1)
```

```
Error in *: Argument must a number: NIL
Apply #15:   (*NIL 1)
Debug 1>
```

MLO–002870

10.4.1.1 The Debug I/O Window

The Debug I/O window (titled LISP Debugger in Figure 10–6) displays the same information that appears on the terminal screen. When you invoke the Debugger, a message that identifies the Debugger, a message that identifies the current stack frame preceded by Apply or Eval, and the command prompt are displayed in the Debug I/O window in the format displayed in Figure 10–6.

The integer in the prompt indicates the number of the nested command level you are in. Its value 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 in the Debugger, that error causes the Debugger to display the prompt Debug 2>.

10.4.1.2 The Calling Stack Window

The Calling Stack window displays the function name in each frame of the control stack. The current frame is indicated by two angle brackets at the left margin of the Calling Stack window. As you move the mouse pointer up or down in the stack, the function name at which you are pointing is underlined. To select a particular frame, double-click on that frame with MB1, or select that frame and then click on the Goto Frame button.

You can dismiss this window from your screen with the Cancel button. To return the Calling Stack window to your screen, choose the Calling Stack item from the pull-down Commands menu of the Debug I/O window (see Figure 10–3). The state of the Calling Stack window (displayed or not displayed) stays the same across invocations of the Debugger. For example, if you remove the Calling Stack window from the screen, you must explicitly bring it back, or it will not reappear.

10.4.1.3 The Variable Bindings Window

The Variable Bindings window displays the function name, argument list, and local variable bindings for the current stack frame. As you change the current stack frame in the Calling Stack window, the display changes in the Variable Bindings window. A new window is not created. The Variable Bindings window is shown in Figure 10–6.

You can dismiss this window from your screen with the Cancel button. To return the Variable Bindings window to your screen, choose the Variable Bindings item from the pull-down Commands menu of the Debug I/O window. The state of the Variable Bindings window (displayed or not displayed) stays the same across invocations of the Debugger. For example, if you remove the Variable Bindings window from the screen, you must explicitly bring it back, or it will not reappear.

10.4.1.4 The Debugger Commands Window

The Debugger Commands window (see Figure 10–6) displays the commands you can use with the VAX LISP Debugger. Some of the commands, for example, Backtrace, take optional arguments. When a command takes an optional argument, it is followed by an ellipsis. Thus, if you want to invoke the Backtrace command with an argument, choose the Backtrace... item. The Backtrace... item brings up a dialog box for you to enter the optional argument.

You can dismiss this window from your screen with the Cancel button. To return the Debugger Commands window to your screen, choose the Commands item from the pull-down Commands menu of the Debug I/O window. The state of the Debugger Commands window (displayed or not displayed) stays the same across invocations of the Debugger. For example, if you remove the Debugger Commands window from the screen, you must explicitly bring it back, or it will not reappear.

10.4.2 Exiting the Debugger

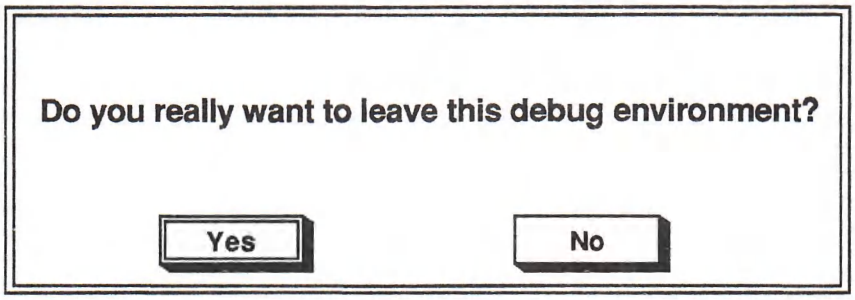
To exit from the Debugger, do one of the following:

- Choose the Quit item from the
  - Debugger Commands window
  - Pull-down Commands menu of the Debug I/O window
- Type Quit at the Debug *n*> prompt

These cause the Debugger to return control to the previous command level.

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 Listener. Specifying the Quit command at a higher Debugger command level returns the Debugger to the next lower command level. If a continuable error caused the Debugger to be invoked, the Quit command, by default, displays a confirmation box on the screen before the Debugger exits. Figure 10–7 shows the confirmation box.

Figure 10–7: The Quit Command Confirmation Box



Do you really want to leave this debug environment?

MLO–002980

If you click on Yes, the Debugger windows disappear from the screen, and control returns to the Listener.

If you click on No, the Debugger prompts you for another command.

You can prevent the Debugger from displaying the confirmation box on the screen with the VAX LISP CUSTOMIZATION function (see Appendix C). To prevent the Debugger from requesting confirmation when you type the Quit command at the prompt, specify the Quit command with a value other than NIL. For example:

```
Debug 1> quit t
Lisp>
```

10.4.3 Using Debugger Commands

The Debugger commands let you inspect and modify the current control stack frame and move to other stack frames. You invoke the Debugger commands from the Debugger Commands window. Some Debugger commands take arguments that qualify command operations. Certain debugging utility commands require arguments, others take optional arguments, and some take no arguments. An argument whose value is an integer is usually optional; an argument whose value is a symbol or form is required. In the Debugger Commands window, commands taking no arguments appear only as the command name with no ellipsis. Commands that require an argument appear only as the command name with an ellipsis. Commands taking optional arguments appear both as the command name with no ellipsis and as the command name with an ellipsis. (For detailed information about the arguments to Debugger commands, see Section 10.4.3.1.)

When you choose a command item with no ellipsis, the command is executed. When you choose a command item with an ellipsis following the command name, a dialog box appears on the screen for you to enter arguments to the command. Table 10–2 provides a summary of the commands in the Debugger Commands window. Table 10–4 describes the stepper commands in similar detail. Detailed descriptions of the commands are provided in Section 4.4.3.2.

Table 10–2: Debugger Commands

Command Button Label	Description
BACKTRACE BACKTRACE...	Displays a backtrace of the control stack.
BOTTOM BOTTOM...	Moves the current frame pointer to the first stack frame on the control stack.
CONTINUE	Continues execution by returning from the continuable error that invoked the Debugger; causes the Debugger to return NIL; is only available if the Debugger was invoked by a continuable error.
DOWN DOWN...	Moves the current frame pointer down the control stack.
ERROR	Redisplays the error message that was displayed when the Debugger was invoked.
EVALUATE...	Evaluates a specified form.
GOTO...	Moves the pointer to a specified control stack frame.
REDO	Reinvokes the function in the current stack frame.
RETURN...	Evaluates its arguments and causes the current stack frame to return the same values the evaluation returns; removes the Debugger windows from the screen.
SEARCH...	Searches the control stack for a frame containing a specified function.
SET...	Sets the values of the components in the current stack frame.
SHOW...	Displays information stored in the current stack frame.
STEP	Resumes execution, single-stepping at the current frame.
TOP TOP...	Moves the pointer to the last stack frame in the control stack.
UP UP...	Moves the pointer up the control stack.
WHERE	Redisplays the argument list and the function name in the current stack frame.

10.4.3.1 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.

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

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

NOTE

Only parenthesized expressions and arguments to evaluate (that is, arguments specified or selected 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 in the Debug I/O window 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 in the current Debugger session.

Table 10–3 provides a summary of the modifier arguments you can specify with Debugger commands.

Table 10–3: Debugger Command Modifiers

Modifier	Effect
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.
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.

10.5 Stepper

The stepper is a facility you can use to step interactively through the evaluation of a form. You can control the stepper with commands from the Stepper Commands dialog box as the stepper 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 stack frame for which the stepper displayed information.

The stepper prints its command interaction and output to the stream bound to the *DEBUG-IO* variable (the Debug I/O window).

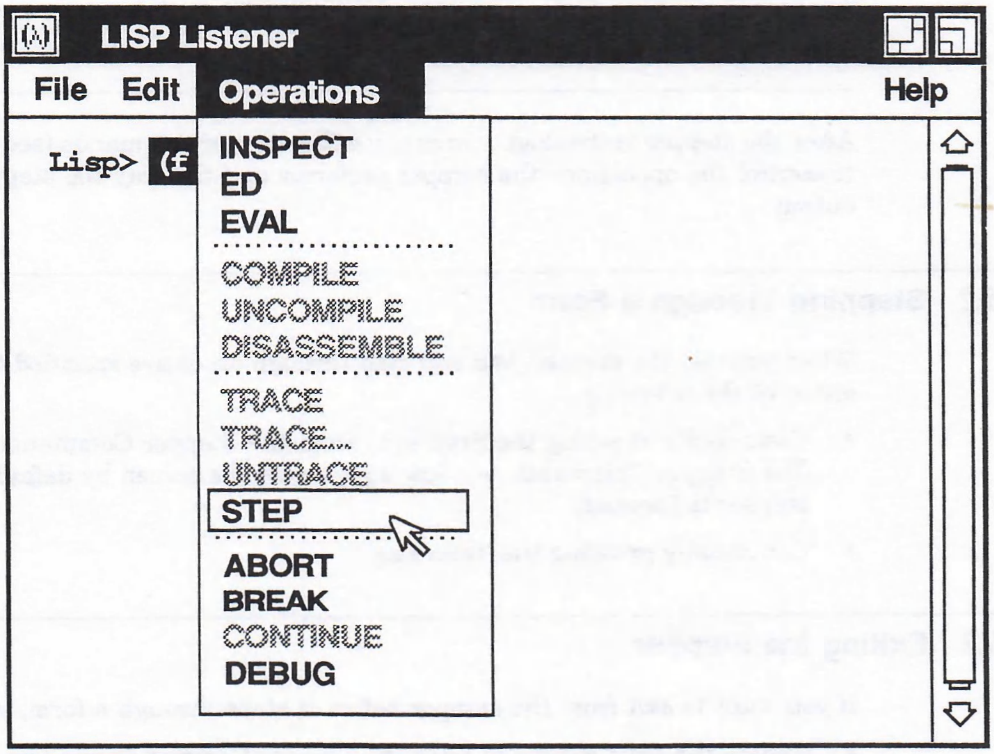
10.5.1 Invoking the Stepper

You can invoke the stepper macro with a form as an argument. The three ways of invoking the stepper are as follows:

- Select a form and choose the Step item from the Operations menu. If you do not select the form before choosing the Step item, the Step item is not available on the Operations menu.
- Type Step at a prompt with a form as an argument. If you do not include the form, the system displays an error message and invokes the Debugger.
- Choose the Step item from the Debugger Commands dialog box.

The example in Figures 10–8 and 10–9 invokes the stepper from the Operations menu with a call to a function named FACTORIAL.

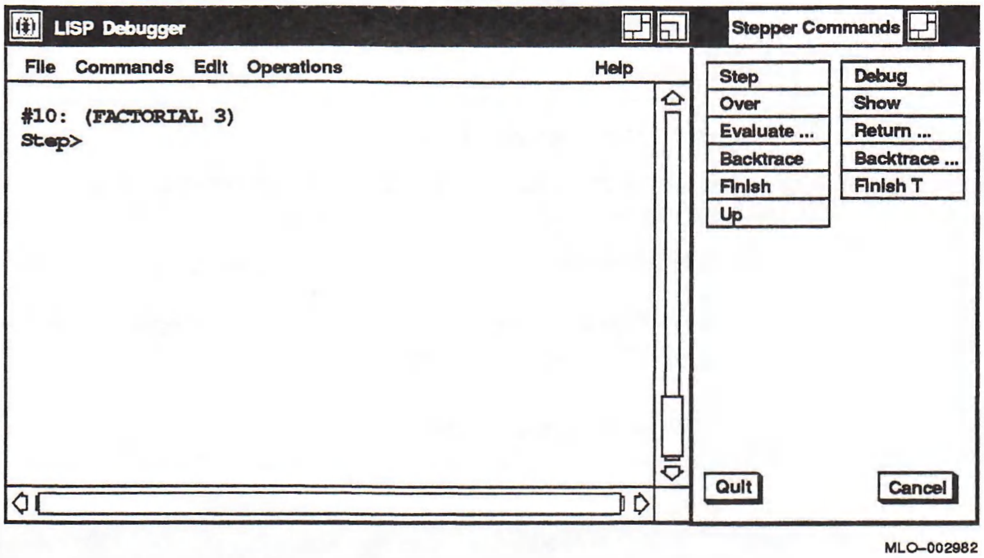
Figure 10–8: Invoking the Stepper



MLO–002981

When the stepper is invoked, the Debug I/O window and the Stepper Commands window are displayed on your screen. The stepper displays a line of text in the Debug I/O window that includes the first subform of the specified form and the stepper prompt. The output is displayed in the following format:

Figure 10–9: Stepper Window Display



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

After the stepper is invoked, you can use the stepper commands (see Section 10.5.5) to control the operations the stepper performs and the way the stepper displays output.

10.5.2 Stepping Through a Form

When you use the stepper, you can step through an entire specified form by doing either of the following:

- Continually choosing the Step item from the Stepper Commands window. The Stepper Commands window appears on the screen by default when the stepper is invoked.
- Continually pressing the Return key.

10.5.3 Exiting the Stepper

If you want to exit from the stepper before it steps through a form, you can:

- Choose the Quit command from the Stepper Commands window.
- Type Quit at the prompt.

The Quit command causes the stepper to return control to the previous command level and window that was active when the stepper was invoked. By default, the Quit command requires confirmation before the stepper exits. If you invoked the stepper from the Listener window, the Debug I/O window disappears from the screen.

You can prevent the stepper from displaying the confirmation box on the screen with the VAX LISP `CUSTOMIZATION` function (see Appendix C). To prevent the stepper from displaying the dialog box when you type the Quit command at the prompt, specify the Quit command with a value other than NIL. For example:

```
Step> quit t
Lisp>
```

10.5.4 Stepper Output

Stepper output in the Debug I/O window is the same as stepper output on a terminal screen. For a description of stepper output and a sample session, see Section 4.5.3.

10.5.5 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.

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. You enter stepper commands from the Stepper Commands window. Figure 10–9 shows the Stepper Commands window.

Certain stepper commands require arguments, others take optional arguments, and some take no arguments. In the Stepper Commands window, the commands that require arguments have an ellipsis following the command name, and the commands that take no arguments appear only as the command name with no ellipsis. Commands that take optional arguments appear both as the command name and as the command name with an ellipsis. Table 10–4 provides a summary of the stepper commands. Descriptions of the stepper commands are provided in Chapter 4.

Table 10–4: Stepper Commands

Command	Description
BACKTRACE BACKTRACE . . .	Displays a backtrace of the current form's evaluation.
DEBUG	Invokes the Debugger.
EVALUATE . . .	Evaluates a specified form with the stepper disabled.
FINISH FINISH T	Completes evaluation of the form that was specified or selected in the call to the STEP macro with the stepper disabled.
OVER	Evaluates the subform in the current stack frame with the stepper disabled.
QUIT	Exits the stepper.
RETURN . . .	Returns the specified values and removes the Debug I/O window and Stepper Commands window from the screen.
SHOW	Displays the subform in the current stack frame without abbreviation.
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.

10.6 Tracer

The VAX LISP tracer is a macro that you can use to follow 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. By default, this output is displayed in the Trace window. To use the tracer, you must enable it for each function and macro you want traced.

NOTE

> You cannot trace special forms.

10.6.1 Enabling the Tracer

You can enable the tracer from a window for one or more functions and/or macros as follows:

- Select one or more functions and/or macros and choose the TRACE or TRACE... items from the Operations menu.
- Type the TRACE macro at the prompt, specifying one or more function and/or macro names as arguments.

NOTE

> If you redefine a function or macro after tracing it (for example, by loading it from a file or by evaluating it from the Editor), you must call TRACE on it again. Otherwise, the tracer only traces the former definition.

10.6.1.1 Enabling the Tracer from the Operations Menu

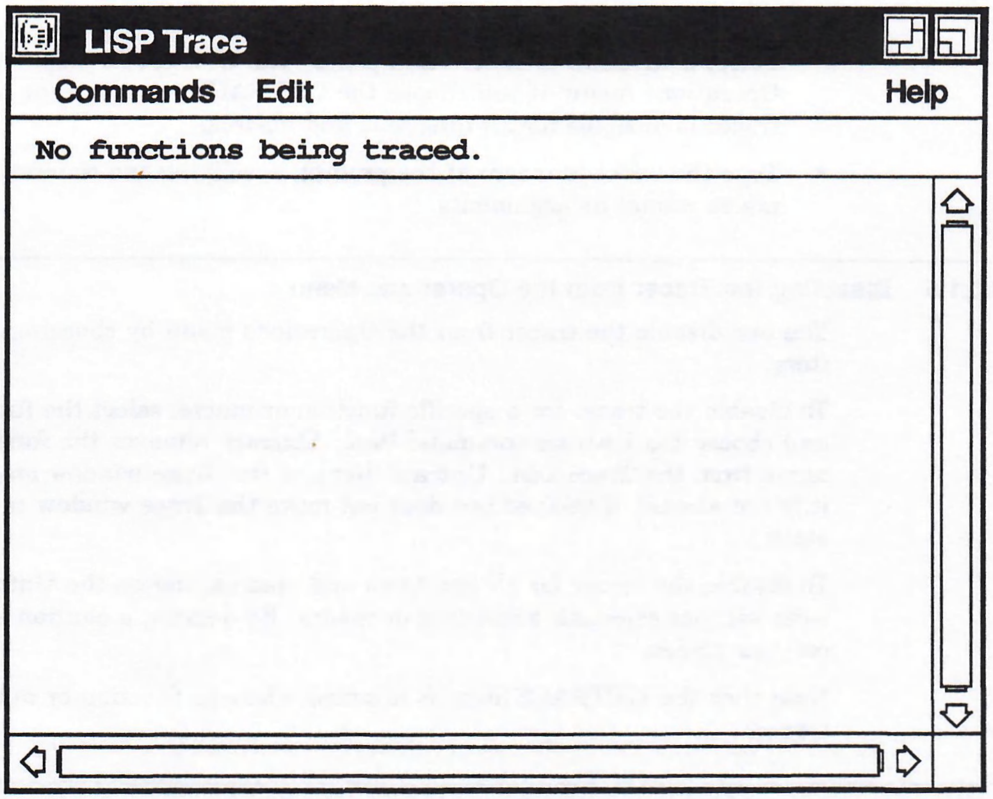
You can enable the tracer without setting or modifying options by choosing the TRACE item. Figure 10–10 shows a Trace window. Choosing the TRACE item after selecting a function or macro creates the Trace window if it did not already exist and adds the function or macro name to the Trace List if it did not already appear there. The Trace List is a list of the symbols being traced; it is located immediately below the menu bar in the Trace window.

You can enable the tracer from the Operations menu and set or modify tracer options by choosing the TRACE... item. Choosing the TRACE... item with no function or macro selected displays the Trace window at the front of your screen. Choosing the TRACE... item after selecting a function or macro displays the Trace Options dialog box, which lets you set options and invoke the trace. The Trace Options dialog box is shown in Figure 10–11. You invoke the trace from the dialog box by clicking on the OK button after setting options.

Note that if you invoke the tracer on the same function or macro with the TRACE item after you invoked it with the TRACE... item and set options for the tracer, tracer output is no longer modified by those options.

A description of the TRACE macro is provided in *VAX LISP/VMS Object Reference Manual*.

Figure 10–10: Trace Window



No functions being traced.

MLO–002983

10.6.1.2 Enabling the Tracer from a Prompt

You can enable the tracer from the Listener or Debug I/O windows for one or more functions or macros by specifying the function and macro names as arguments in a call to the TRACE macro. The TRACE macro returns a list of traced functions and macros. It adds the function or macro names you specified to the Trace List if they are not already there, and if the Trace window already exists.

If you type TRACE at the prompt without specifying a function or macro, TRACE returns a list of traced functions. If there are no functions currently being traced, TRACE returns an empty list.

Note that TRACE typed at a prompt ignores any objects you selected with the mouse; it is only affected by the arguments you type at the prompt.

10.6.1.3 Clearing the Tracer

You should periodically clear the stored and visible information from the Trace Output area of the Trace window. The Trace Output area is located immediately below the Trace List. You clear the information from the Trace Output area by choosing the Clear item from the Commands menu. Clearing the stored and visible information from Trace Output allows the space used by Trace Output to be freed and allows the objects to which pointers have been kept to be garbage collected. If object recording is turned on and you do not perform the Clear operation occasionally, the objects recorded will not be collected, and the LISP process could run out of virtual memory.

10.6.1.4 Disabling the Tracer

You can disable the tracer from a window for one or more functions and/or macros as follows:

- Select a function or macro and choose the UNTRACE item from the Operations menu; if you choose the UNTRACE item with no selection, the tracer is disabled for all functions and macros.
- Type the UNTRACE macro at the prompt, specifying one or more function and/or macro names as arguments.

10.6.1.5 Disabling the Tracer from the Operations Menu

You can disable the tracer from the Operations menu by choosing the UNTRACE item.

To disable the tracer for a specific function or macro, select the function or macro and choose the Untrace command item. Untrace removes the function or macro name from the Trace List. Untrace displays the Trace window on your screen if it is not already displayed but does not move the Trace window to the top of the stack.

To disable the tracer for all functions and macros, choose the Untrace command item without selecting a function or macro. By default, a caution box is displayed on your screen.

Note that the UNTRACE item is dimmed when no function or macro is being traced.

10.6.1.6 Disabling the Tracer from a Prompt

To disable the tracer from a prompt for specific functions or macros, specify the names of the functions or macros in a call to the UNTRACE macro. Tracing stops for these functions and macros, and their names are removed from the Trace List.

You can disable tracing for all the functions for which tracing is enabled by calling the UNTRACE macro without any arguments. Untrace removes all traced items from the Trace List.

Note that UNTRACE typed at a prompt ignores any objects you selected with the mouse; it is only affected by the arguments you type at the prompt.

The UNTRACE macro is described in *Common LISP: The Language.*

10.6.2 Tracer Output

Tracer output appears in the Trace window below the Trace List—the list of all items being traced.

Once you enable the tracer for a function or macro, the tracer displays in the Trace window 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 that 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
```

Chapter 4 has examples illustrating the format of tracer output.

10.6.3 Modifying Tracer Options from the Dialog Box

You can modify the output of the tracer when you call it with the TRACE... menu item by specifying options in the Trace Options dialog box. The Trace Options dialog box is shown in Figure 10–11. When you use the Trace Options dialog box, you can specify options for only one function or macro at a time.

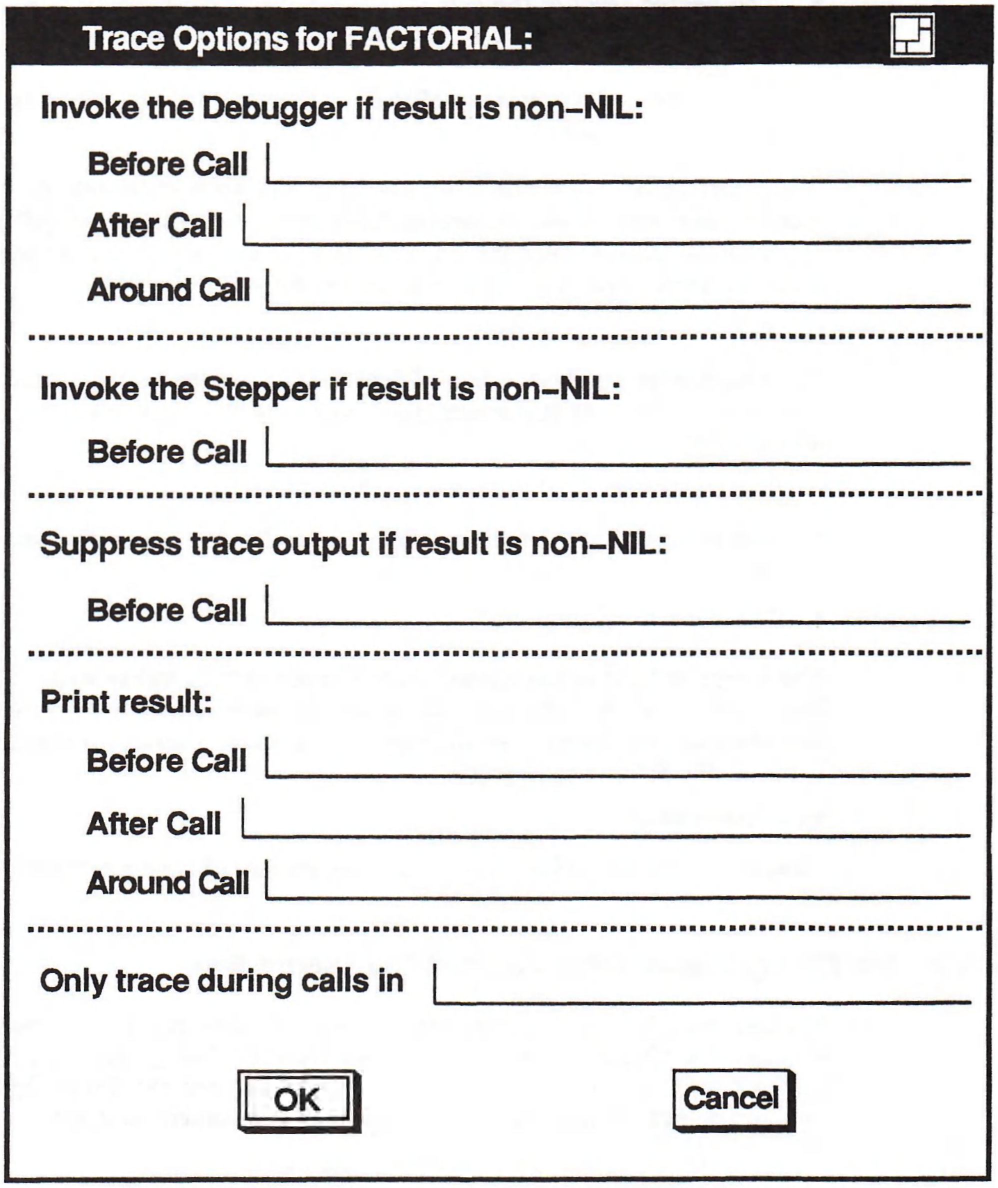
NOTE

> 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.*

10.6.3.1 Invoking the Debugger from the Trace Options Dialog Box

Typing a LISP form in one of the three text fields under Invoke the Debugger if result is non-NIL invokes the debugger if the form returns a value other than NIL. The LISP system evaluates the form before, after, or before and after each call to the function or macro being traced.

Figure 10–11: Trace Options Dialog Box



Trace Options for FACTORIAL:

Invoke the Debugger if result is non–NIL:

Before Call

After Call

Around Call

Invoke the Stepper if result is non–NIL:

Before Call

Suppress trace output if result is non–NIL:

Before Call

Print result:

Before Call

After Call

Around Call

Only trace during calls in

MLO–002984

10.6.3.2 Invoking the Stepper from the Trace Options Dialog Box

Typing a LISP form in the Before call text field under Invoke the Stepper if result is non-NIL invokes the stepper if the form returns a value other than NIL. The LISP system evaluates the form before each call to the function or macro being traced.

10.6.3.3 Removing Information from Tracer Output from the Trace Options Dialog Box

Typing a LISP form in the Before call text field under Suppress trace output if result is non-NIL suppresses trace output if the form returns a value other than result NIL. 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.

10.6.3.4 Adding Information to Tracer Output from the Trace Options Dialog Box

Typing a list of LISP forms in one of the three text fields under Print result adds information to tracer output. The LISP system evaluates each form in the list and the tracer displays the return value, 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.

10.6.3.5 Defining When a Function or Macro Is Traced from the Trace Options Dialog Box

Typing a function or macro name or a list of function and/or macro names in the text field next to Only trace during calls in defines when a function or macro, for which tracing is enabled, is to be traced. 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 name is specified in the text field.

10.6.4 Modifying Tracer Options in the Call to the TRACE Macro

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
          ...))
```

In the call to the TRACE macro, 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
                           ...))
```

Specifying options at the prompt in a window is the same as specifying options at the prompt on a terminal interface. For a detailed explanation of how to do this, see Section 4.6.4.

Appendix A

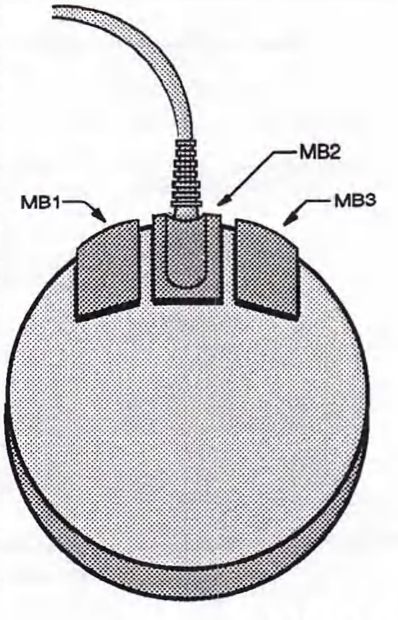
Using DECwindows

You work with most DECwindows applications by using the same handful of mouse and windowing techniques. This appendix summarizes some basic techniques you'll use in DECwindows—from clicking and dragging to editing text.

A.1 Using the Mouse

The mouse—the hand-held pointing device attached to your workstation monitor—makes using DECwindows as easy as pointing to an object on your screen and clicking a button. You use the mouse to choose commands from a menu, to expand and shrink windows, and to rearrange windows on your screen.

The mouse has three buttons. Unless you specify otherwise, MB1 (for "mouse button 1") is on the left, MB2 is in the middle, and MB3 is on the right. This button arrangement naturally suits right-handed users. If you are left handed, you can easily rearrange this configuration by changing the button arrangement setting in the Session Manager's Customize Pointer dialog box.



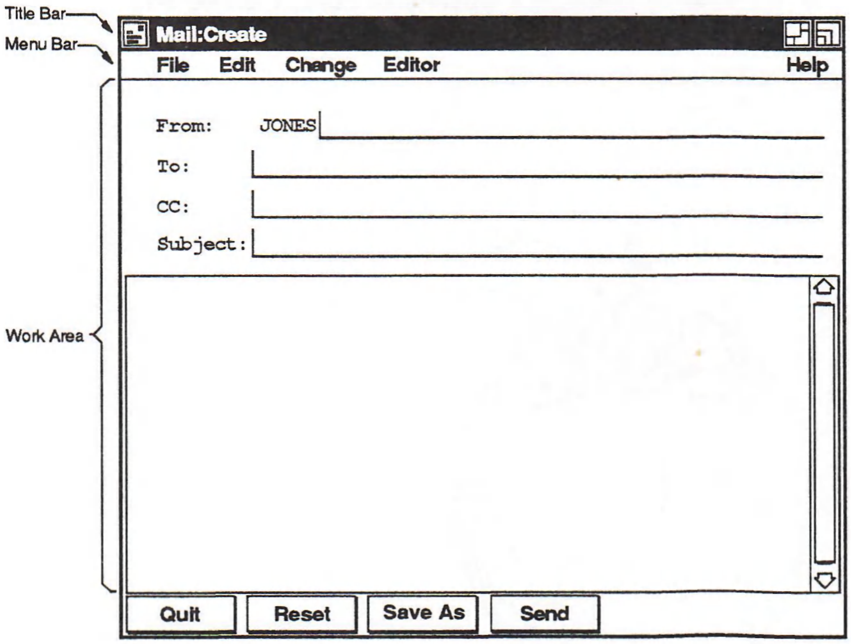
ZK-0250A-GE

You can do all your work with DECwindows by mastering the following mouse techniques:

- Point: Using the mouse, move the cursor to where you want the next action to occur.
- Click: Quickly press and release MB1. You should hear and feel a faint click.
- Press: Point to the menu name, stepping arrow, or wherever you want the action to occur. Without moving the mouse, press and hold MB1 or MB2. If you are pointing at a menu name, pressing MB1 pulls down a menu and keeps it down until you release MB1.
- Drag: Press and hold MB1, move the pointer, and release MB1. For example, you move a window to another location on the screen by dragging its outline. To cancel a drag in progress, click MB3 before releasing MB1. If you are displaying a pull-down menu, cancel the drag by moving the pointer outside the menu and releasing MB1.
- Double click: Point to the object and click MB1 twice in quick succession.
- Shift click: Point to the object. Press and hold the Shift key and click MB1. Release the Shift key.

A.2 What Are Windows?

A window is an area on your workstation screen that represents all or part of an application. For example, Mail's Create window is just one of several windows available in the Mail application.



From: JONES
To:
CC:
Subject:

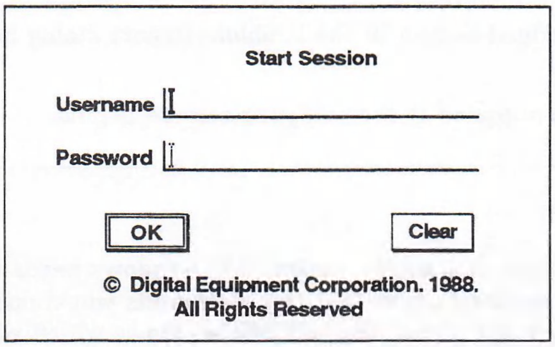
ZK-0808A-GE

- The title bar at the top identifies the window.
- The menu bar directly beneath the title bar lets you access the application's commands.

- The work area in the space remaining displays the application's text and graphics.

A.3 Starting a Session

If the system startup procedure has been successful, your screen looks like this:



© Digital Equipment Corporation. 1988.
All Rights Reserved

ZK-0245A-GE

The Start Session dialog box prompts you for your user name and password. DECwindows displays a dialog box whenever it needs information from you.

You type your user name and password in the appropriate text entry fields. A text insertion cursor is visible in each field. The text cursor in the Username field blinks to indicate this field has input focus. When a text field or window has input focus, you see your keystrokes echoed there. The text cursor in the Password field is dimmed, indicating that you cannot currently enter text in it.

To start a session:

1. Type your user name.

   If you make a typing mistake, press the Delete key (⌫) to erase the character to the left of the text cursor. To insert a character in the middle of text you already typed, point where you want the text inserted and click MB1. Or, use the right and left arrow keys to move the text cursor right or left. The new characters you type push existing ones to the right.

2. Select the Password field by pointing to the Password field and clicking MB1.

   You always select the object or information you want to work on next. You can also move to the Password field by simply pressing Tab. The text cursor in the Password field blinks to indicate this field now has input focus.

3. Type your password.

   To preserve the secrecy of your password, the letters you type are not displayed on the screen.

   If you make a typing mistake, click on the Clear button. This erases all text in both the Username and Password fields so you can retype your information correctly. Clicking on buttons in dialog boxes lets you tell DECwindows what to do with the information you supplied.

4. Click on the OK button or press Return.

The double outline around the OK button in the Start Session dialog box indicates it is the default option. Default options are those you will choose most frequently. DECwindows provides you with a shortcut to choose default options: Whenever you see a button with a double outline, pressing the Return key achieves the same results as clicking on that button.

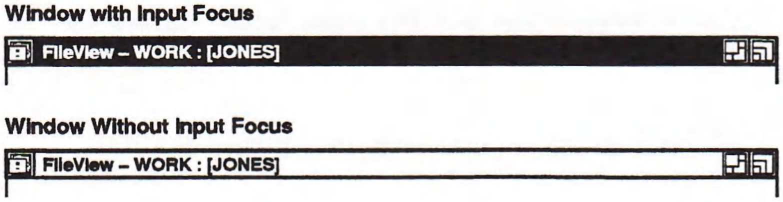
If you provide wrong information or make a typing mistake and do not correct it, DECwindows does not let you start a session. Instead, it displays a Problem Report dialog box as a warning that some information is incorrect.

Click on the Acknowledged button in the Problem Report dialog box or press the Return key to try again.

If the information you supplied is correct, your session begins.

A.4 Selecting Windows

When you have more than one window open, DECwindows needs to know which one you are currently working on so that the commands you choose and the text you type end up in the right place. You tell DECwindows which window you want to work with by selecting it. When you select a window, it moves to the front of the "stack" of overlapping windows. Its title bar is highlighted to indicate it has input focus. Any keystrokes you type appear in this window. When you select another window, the new window is given input focus. Only one window can have input focus at a time.



Window with Input Focus

Window Without Input Focus

ZK–0581A–GE

To select a window:

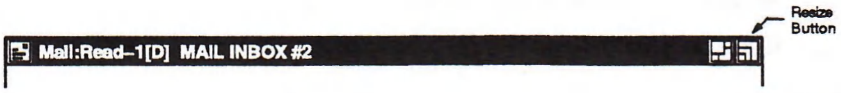
1. Point to a location in the window or title bar.

   In the FileView window, point to the title bar.

2. Click MB1.

A.5 Changing the Size of Windows

Sometimes you want to make one window very large so you can see everything in it. Other times, you might want to work with small windows, such as when several applications are running simultaneously. You can change the size of your windows to suit your needs by using the resize button.



ZK–0563A–GE

To change the size of a window:

1. Point to the window's resize button.
2. Press and hold MB1.

   The pointer changes into a small resize cursor.
3. Drag the resize cursor to the size you want.

   To make the window larger, drag the resize cursor beyond the window border. To make the window smaller, drag the resize cursor beyond the window border and then back in. The outline stops moving when the window is as small as it can get.
4. Release MB1.

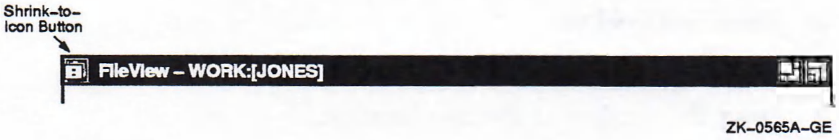
You can change the size of a window in one dimension (height or width) or in both dimensions simultaneously. To change the size in one dimension, drag the resize cursor across one border of the window. As long as you cross only one border, the outline that follows the resize cursor changes in only one dimension. If after crossing one border you cross an adjacent border, you see an outline that can change in both dimensions.

If you drag the resize cursor through one border and then through the *opposite* border, the first border you crossed reverts to its original location, and the other border becomes an outline that follows the resize cursor.

To cancel a window-resizing operation in progress, click another mouse button before releasing MB1. The outline disappears, and the window retains its original size.

A.6 Shrinking Windows

When you start an application, its icon appears in the Icon Box. You shrink a window to an icon if you want to free up space on your screen to run other applications without exiting from the application. When you shrink a window to an icon, the application it represents continues to run in memory and remains easily accessible. Any processes continue to execute while the application is stored as an icon.



When the application is running in a window, its icon is dimmed. The icon appears bold when the application is stored in the Icon Box. If the Icon Box contains more icons than can be displayed at once, scroll bars appear. You use scroll bars to view the text that does not fit in a single window.

To shrink a window to an icon:

1. Point to the window's shrink-to-icon button.
2. Click MB1.

   The window closes and its icon in the Icon Box appears bold.

You cannot shrink the Icon Box to an icon.

As you stop applications, your Icon Box develops gaps where icons used to be. You can rearrange the icons in the Icon Box in two ways:

- By moving the icons. To move an icon, drag it to a new location.
- By clicking on the Icon Box's shrink-to-icon button. Clicking on this button in an application window shrinks the window to an icon. Clicking on this button in the Icon Box, however, eliminates the gaps between icons that may develop when you stop applications.

A.7 Expanding Icons to Windows

When you expand an application icon, you open a window for that application. If you have more than one window open and expand an icon to a window, that new window is placed at the front of the stack of overlapping windows. If the window accepts text entry, it is also given input focus.

To expand an icon to a window:

1. Point to the icon in the Icon Box.
2. Click MB1.

A.8 Moving Windows

If one window partially obscures another, you might want to arrange them so that each is completely visible. To move a window, drag it by its title bar.

Title Bar
Mail:Read-1[D] MAIL INBOX #2

ZK-0579A-GE

To move a window:

1. Position the pointer anywhere in the window's title bar (except in the shrink-to-icon, push-to-back, or resize buttons).
2. Press and hold MB1.

   An outline of the window appears.
3. Drag the window to the new location.
4. Release MB1.

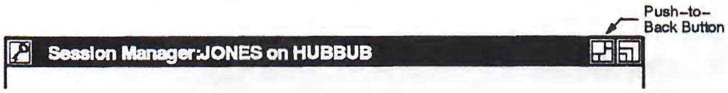
   If the window was partially obscured by other windows, it moves to the front of the stack of windows and is given input focus.

To cancel a window-moving operation in progress, click another mouse button before releasing MB1. The outline disappears and the window is not moved.

A.9 Stacking Overlapping Windows

When you are working with stacked windows and select a window, it moves to the front of the stack and is given input focus. But if a larger window completely obscures a smaller window, you cannot select that small window without moving the larger window out of the way. If you use the title bar to move the larger window, you disrupt your window arrangement.

By clicking on the visible window's push-to-back button, you can push that window to the back of the stack and expose the window underneath. Without moving the windows to another location on the screen, you can push one window out of the way or retrieve another to work with.



ZK-0564A-GE

To push the frontmost window to the back of a stack of overlapping windows:

1. Point to the frontmost window's push-to-back button.
2. Click MB1.

If you try this repeatedly with three or more windows, you see that the windows cycle through the stacking order, moving up one position in the stack each time a window is moved to the back of the stack.

A.9.1 Making Overlapping Windows Stick in Place

When you are working with overlapping windows and select one window, it moves to the front of the stack of windows and is given input focus. You can, however, prevent a partially obscured window from moving to the front of the stack when you select it by arranging the windows to display only what you need to see and then securing them in place.

For example, if you are working with a DECterm, Notepad, Mail, and FileView windows, you might arrange them so that only the portions you need to work with are visible. In the DECterm window, you probably need to see only the last few lines so you can enter commands. Once you secure the windows in place, you can switch from one window to another without disturbing their order in the stack.

To lock overlapping windows in the stacking order:

1. Point to a window's push-to-back button.
2. Shift click on the button.

   That window is pushed to the back of the stack.

The lower right-hand corner of the push-to-back button is filled to indicate that the window is fixed in the stacking order. Although the window is given input focus when you select it, it does not pop to the front of the stack.

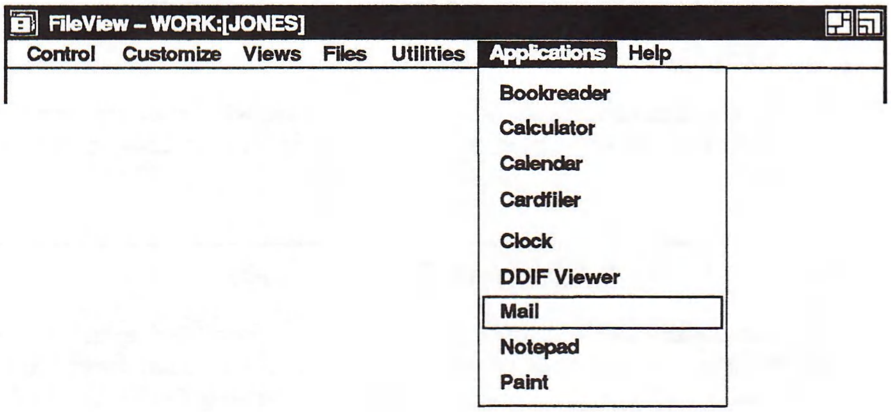
You can still, however, push to the back or bring to the front any window that you have fixed in the stacking order to see it unobscured. Clicking on a window's filled push-to-back button moves that window to the opposite position in the stack, but does not give it input focus. Clicking on a filled push-to-back button pushes an unobscured window to the back of the stack. Clicking on a filled push-to-back button moves a partially obscured window to the front of the stack.

A.9.2 Releasing Windows Locked in the Stacking Order

To release a window locked in the stacking order, shift click again on its push-to-back button. If the window was at the back of the stack, it moves to the front; if the window was at the front, it is pushed to the back. If you look at the window's push-to-back button, you will notice that it is no longer filled.

A.10 Choosing Items from Pull-Down Menus

The names of menus available in an application appear on the menu bar. When you press MB1 on a menu name, the menu's contents are displayed or pulled down.



ZK–0234A–GE

Some pull-down menus list commands. Others list the names of items you can work with. You tell DECwindows what you want to do or what you want to work with by choosing commands or items from pull-down menus. Any menu item followed by three periods (...) is your cue that a dialog box will be displayed if you choose that menu item.

To choose an item from a pull-down menu:

1. On the menu bar, point to the name of the menu you want to display.
2. Press and hold MB1.

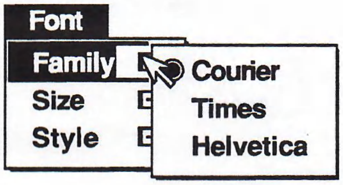
   This highlights the menu name and pulls down a menu.
3. While holding MB1, drag to the menu item you want.
4. Release MB1.

If you change your mind while looking at a pull-down menu, drag outside the menu and release MB1. The menu disappears and no action is taken.

Some applications offer rectangular push buttons to duplicate frequently used commands that are also available as menu items. Push buttons are usually found underneath an application's work area. To execute these commands quickly, click MB1 on the push button.

A.11 Choosing Items from Submenus

A menu item with a submenu icon—an arrow pointing to the right—indicates that a corresponding submenu is available. If you choose that menu item, you need to refine your choice by displaying its submenu and choosing a menu item from that submenu.



ZK-0529A-GE

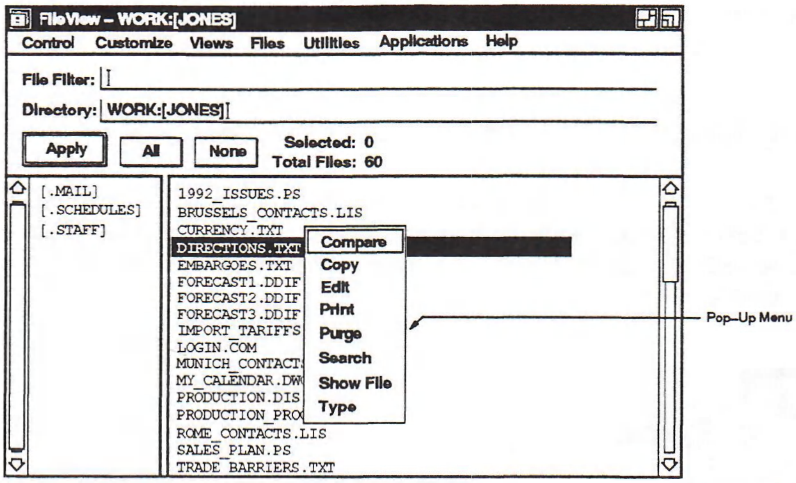
To display a submenu and choose a menu item from it:

1. On the menu bar, point to the name of the menu you want to display.
2. Press and hold MB1.
3. Drag to the menu item you want.
4. Drag onto the submenu icon.

   A submenu is displayed to the right of the menu.
5. Drag to the menu item you want to choose from the submenu.
6. Release MB1.

A.12 Choosing Items from Pop-Up Menus

DECwindows provides pop-up menus to make it easier for you to work with files and applications. Pop-up menus duplicate commands and functions available on pull-down menus. Unlike pull-down menus, which require you to move the pointer to the menu bar, you can display a pop-up menu anywhere in an application's work area. By reducing your use of the mouse, pop-up menus give you quick, direct access to an application's commands and functions.



File Filter:
Directory: WORK:[JONES]

Selected: 0
Total Files: 60

```
[.MAIL]
[.SCHEDULES]
[.STAFF]
```

```
1992_ISSUES.PS
BRUSSELS_CONTACTS.LIS
CURRENCY.TXT
DIRECTIONS.TXT
EMBARGOES.TXT
FORECAST1.DDIF
FORECAST2.DDIF
FORECAST3.DDIF
IMPORT_TARIFFS
LOGIN.COM
MUNICH_CONTACT
MY_CALENDAR.DW
PRODUCTION.DIS
PRODUCTION_PRO
ROME_CONTACTS.LIS
SALES_PLAN.PS
TRADE_BARRIERS.TXT
```

ZK-0574A-GE

To display a pop-up menu:

1. Press and hold MB2 on the application's work area.

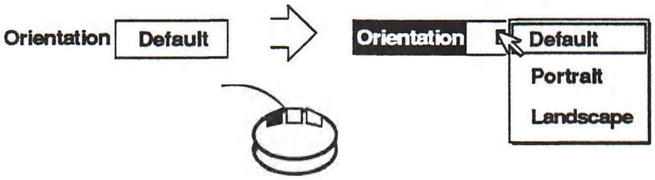
   In the FileView window, press and hold MB2 on the file type whose pop-up menu you want to display.

2. Drag to the menu item you want.
3. Release MB2.

If you change your mind while looking at a pop-up menu, drag outside the menu and release MB2. The menu disappears and no action is taken.

A.13 Choosing Items from Option Menus

An option menu is a pop-up menu that appears in a dialog box. An option menu lets you choose one option from many. In the dialog box, only the current option is displayed. To see the other options from which you can choose, you display the option menu.



ZK-0562A-GE

To display an option menu:

1. Press and hold MB1 on the current option.

   The option menu is displayed.

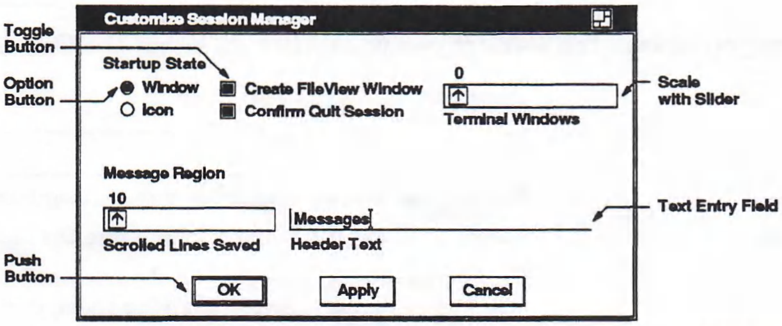
2. Drag to the menu item you want.
3. Release MB1.

The option menu disappears. The option you chose is now the current option.

If you decide not to change the original option, drag outside the menu and release MB1. The menu disappears and no changes occur.

A.14 Supplying Information in Dialog Boxes

DECwindows displays a dialog box whenever it needs additional information from you to carry out a task. Sometimes you need to type text; other times, you need only click on a button to change a setting. Some dialog boxes display settings you chose earlier.



Customize Session Manager

Startup State

● Window

○ Icon

■ Create FileView Window

■ Confirm Quit Session

Terminal Windows: 0

Message Region

Scrolled Lines Saved: 10

Header Text: Messages

ZK–0550A–GE

Dialog boxes offer various ways to supply information to an application:

- By typing text in a text entry field. The blinking text cursor shows you where the text you type will appear. What you type appears to the left of the text cursor.
- By clicking on option buttons or square toggle buttons. Option buttons let you select one option from many. Toggle buttons let you turn a setting on or off.
- By dragging the slider in a scale. Dialog boxes often contain a scale and slider when you need to supply a numeric value. The arrow in the slider points to the current value.
- By selecting choices, for example, file names, from a list box. A list box contains a list of items from which you can choose. Scroll bars appear if the choices do not fit in the list box.
- By clicking on push buttons. Push buttons, such as OK, Cancel, or Filter, let you tell DECwindows what to do with the information you supplied in the dialog box.

  In a dialog box, the OK button is usually the default option. You can press the Return key to achieve the same result as clicking on OK.

A.14.1 Moving and Changing Settings in a Dialog Box

How you move in a dialog box depends on the object, for example, a toggle button or text field, you want to work with. The following table describes the ways in which you can move and change settings in a dialog box.

To	Do this
Move forward between text fields	Press the Tab key or point to the field to which you want to move and click MB1.
Move backward between text fields	Press the Shift/Tab keys or point to the field to which you want to move and click MB1.
Move the text cursor within a text field	Point to where you want the text inserted and click MB1 or use the right and left arrow keys to move the text cursor right or left. New characters push existing ones to the right.
Change the numeric value on a scale	Drag the slider on the scale right or left or point to another location on the scale and click MB1.
Change an option or toggle button setting	Point to the option or toggle button and click MB1.

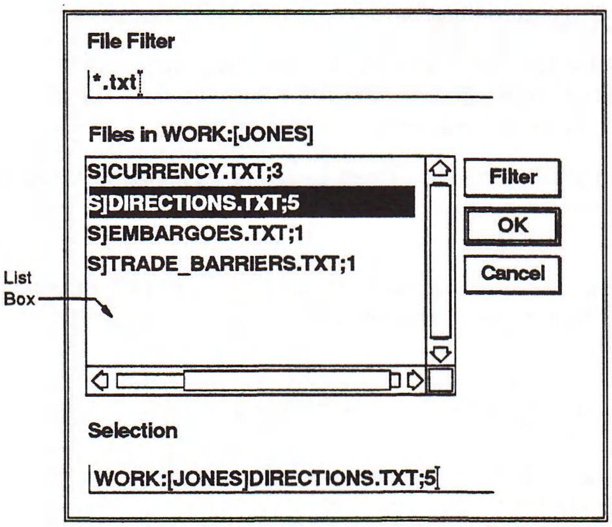
Once you change the settings you want, click on either the OK, Apply, or Cancel buttons.

Click on	To
OK	Record your choices and dismiss the dialog box.
Apply	Record your choices without dismissing the dialog box.
Cancel	Dismiss the dialog box without changing any settings. If you made any changes without applying them, clicking on the Cancel button cancels those changes.

A.14.2 Making Selections from List Boxes

A list box is part of a dialog box that contains a list of items, often file names, from which you can choose. Many applications display a list box when you open or save a file.

To select an item from a list box, point to the item and click MB1. The item you select is highlighted.



File Filter

*.txt

Files in WORK:[JONES]

S]CURRENCY.TXT;3

S]DIRECTIONS.TXT;5

S]EMBARGOES.TXT;1

S]TRADE_BARRIERS.TXT;1

Selection

WORK:[JONES]DIRECTIONS.TXT;5

ZK-0548A-GE

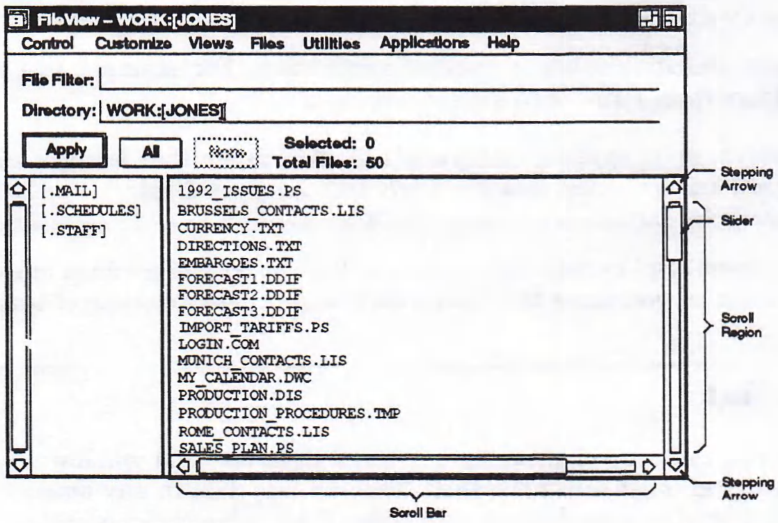
If you selected a file name, you can click OK to open the file. DECwindows also gives you a shortcut for opening files from list boxes: Double clicking on the file name produces the same results as selecting that file name and clicking on OK.

If you want to work with a file that does not appear in the list box, type the name of the file in the Selection text entry field and click on OK. Or, use the File Filter text entry field to list a subset of files that you can then select from. For example, to list all the files in another directory with the file type TXT, enter the complete directory specification—[JONES.LETTERS]*.TXT—in the File Filter text entry field and click on the Filter button. The list of files that meet the qualifications are displayed. Double click on the name of the file you want to open.

A.15 Scrolling

Some windows display scroll bars, which let you view the text that does not fit in a single window. Some windows have both horizontal and vertical scroll bars.

A scroll bar consists of stepping arrows at either end of the scroll region. The slider is the thicker box that overlays the scroll region. If the slider is at the top of the scroll region, the beginning of the file or list is visible. If the slider is at the bottom of the scroll region, the end of the file or list is visible.



File Filter:
Directory: WORK:[JONES]

Selected: 0
Total Files: 50

```
[.MAIL]
[.SCHEDULES]
[.STAFF]
```

```
1992_ISSUES.PS
BRUSSELS_CONTACTS.LIS
CURRENCY.TXT
DIRECTIONS.TXT
EMBARGOES.TXT
FORECAST1.DDIF
FORECAST2.DDIF
FORECAST3.DDIF
IMPORT_TARIFFS.PS
LOGIN.COM
MUNICH_CONTACTS.LIS
MY_CALENDAR.DWC
PRODUCTION.DIS
PRODUCTION_PROCEDURES.TMP
ROME_CONTACTS.LIS
SALES_PLAN.PS
```

ZK–0557A–GE

The size of the slider is relative to the total amount of text in the document and indicates how much more text remains to be displayed. For example, a small slider indicates that much text remains to be displayed. A large slider that completely fills the scroll bar indicates that all the text is displayed.

The following table describes how to use scroll bars.

To scroll	Do this
One line at a time	Click MB1 on the stepping arrows.
Forward one windowful of text at a time	Point to the scroll region below the slider and click MB1.
Back one windowful of text at a time	Point to the scroll region above the slider and click MB1.

To scroll	Do this
Continuously through the list or file one line at a time	Press and hold MB1 on either stepping arrow.
Continuously through the list or file one windowful of text at a time	Press and hold MB1 in the scroll region.
To another location in the list or file	Drag the slider to a position in the scroll region that corresponds to the general location you want to see. If the slider is at the top of the scroll region, you are viewing the beginning of the list or file. If the slider is in the middle of the scroll region, you are viewing the middle of the list or file. Cancel the drag by clicking another mouse button before releasing MB1.

A.16 Editing Text

DECwindows provides many ways to edit text, which saves you from retyping long file names or large blocks of text. Most applications let you move and copy text

- From one place in a window to another.
- From one window to another window. For example, you can copy text from one Create-Send window in Mail to another.
- From one application to another application. For example, you can move a picture from Paint onto a Cardfiler card.

In addition, most applications provide an Edit menu that lets you cut, copy, and paste text and graphics. See the *VMS DECwindows Desktop Applications Guide* for more information about using the Edit menu in specific applications.

Finally, most applications define specific keys to let you perform basic text editing. These keys let you move the cursor and delete small amounts of text efficiently.

A.16.1 Selecting Text

Before you can copy or move text to other locations in a window or between windows, you must select the text. You can copy text in any amount, including a word, a line, or a paragraph at a time. Successive clicks of MB1 increase the amount of text you are selecting. The following table describes how to select text.

To	Do this
Position the cursor where you want the selection to start	Point to the location and click MB1.
Select a word	Point to the word and double click MB1.
Select a line	Point to the line and triple click MB1.
Select continuous text, from the original selection point to the point where the button is released	Press and hold MB1 and drag the pointer through the text.
Extend the current selection	Simultaneously press and hold the Shift key and MB1 while dragging the pointer through the additional text.
Extend the current selection to where the pointer is positioned	Press and hold the Shift key and click MB1.

In addition, some applications provide a way for you to select larger blocks of text at a time. For example, you can select a paragraph of text in EVE by pointing to the paragraph and clicking MB1 four times. You can select an entire mail message or the contents of a Notepad file by pointing to the text and clicking MB1 five times.

You can select only one piece of text at a time. By selecting text in one application, you cancel any other text selection established in the same window or in another application.

A.16.2 Copying Text

If you can type text in a window, you can select and copy that text from one place to another in the same window, between windows of the same application or between different applications.

You can also copy text from a FileView window—including FileView's file list—to an application that supports text entry.

To copy text in a window, between windows in the same application, or between applications:

1. Select the text you want to copy, using the text selection techniques described in Section A.16.1.
2. Position the cursor where you want the text copied by pointing and clicking MB1.
3. Click MB3.

   The text is copied to the new location.

The window from which you selected text takes input focus. If you copied that text back to your current window, you had to reestablish input focus for that window. Sometimes, however, you just want to grab a piece of text from another window without worrying about which window has input focus. That is where QuickCopy comes in handy: You use QuickCopy to copy text from another window to your current window without losing input focus in your current window.

To use QuickCopy:

1. In the current window, position the cursor where you want the text copied by pointing and clicking MB1.
2. In the other window, point to the text you want to copy.
3. Press and hold MB3.
4. Drag across the text you want to copy.

   The text is underlined as you drag across it.
5. Release MB3.

   The text is copied to the new location in your current window.

A.16.3 Moving Text Between Windows

DECwindows also lets you work in one window, select text from another, and move that text to the current window without losing input focus in your current window. The text is deleted from its original location.

To move text from one window to another:

1. In the current window, position the cursor where you want the text pasted by pointing and clicking MB1.

   Make sure the window has input focus.

2. In the other window, point to the text you want to move.
3. Press and hold Ctrl/MB3.
4. Drag across the text you want to move.

   The text is underlined as you drag across it.

5. Release Ctrl/MB3.

   The text is moved to the new location and deleted from the old.

A.16.4 Deleting Text with Pending Delete

When you mark text for pending delete, you can delete large blocks of text with one keystroke instead of pressing the Delete key repeatedly. You mark text for pending delete by selecting it as described in Section A.16.1. The selected text is deleted when you press any key. You can then type new text.

To cancel a pending delete selection (once you select the text but before you press a key), point to the selected text and click MB1.

A.16.5 Editing Text in Dialog Boxes

You can use the text editing techniques described in the following table to move the cursor or delete text in dialog boxes in any DECwindows application. Use the arrow keys to move the cursor one character at a time.

To	Press
Move the cursor to the next word	Shift/→
Move the cursor to the previous word	Shift/←
Move the cursor to the beginning of the line	F12 or Ctrl/H
Move the cursor to the end of the line	Shift/F12 or Ctrl/E
Move the cursor forward between text fields	Tab
Move the cursor backward between text fields	Shift/Tab
Delete the character to the left of the cursor and move all text to the right of the deleted character one space to the left	⌫

To	Press
Delete the character after the cursor and move all text to the right of the deleted character one space to the left	Shift/⌫. In overstrike mode, Shift/⌫ deletes the character under the block cursor. Not enabled in EVE.
Delete all characters to the start of the line	Ctrl/U

A.17 Composing Special Characters

In DECwindows, you can use compose sequences to create special characters. A compose sequence is a series of keystrokes that creates characters that do not exist as standard keys on your keyboard. See the list of ISO Latin-1 compose sequences in the *VMS DECwindows Desktop Applications Guide*.

Depending on the keyboard type, you compose characters in either of the following ways:

- Using three-stroke sequences on a VT200- or VT300-series keyboard.
- Using two-stroke sequences on any VT200-series keyboard except North American.

To compose a character, using the list of compose sequences in the *VMS DECwindows Desktop Applications Guide*:

1. Find the character you want to create in column 1.
2. To compose a three-stroke sequence, press and hold the Compose key while you press the space bar, and then type the two characters in column 2.

   To compose a two-stroke sequence, type the two characters in column 3. The desired character is displayed.

To cancel a compose sequence, press and hold the Compose key while you press the space bar, or press the Delete key, Tab key, Return key, or Enter key.

See the *VMS DECwindows Desktop Applications Guide* for information about composing characters in DECterm.

A.18 Getting Help

You can get help in any DECwindows application by using the Help menu. Help provides brief information about screen objects, concepts, or tasks you can perform in applications. Some applications also let you get help on specific screen objects, for example, scroll bars and menu items, by using the Help key and MB1.

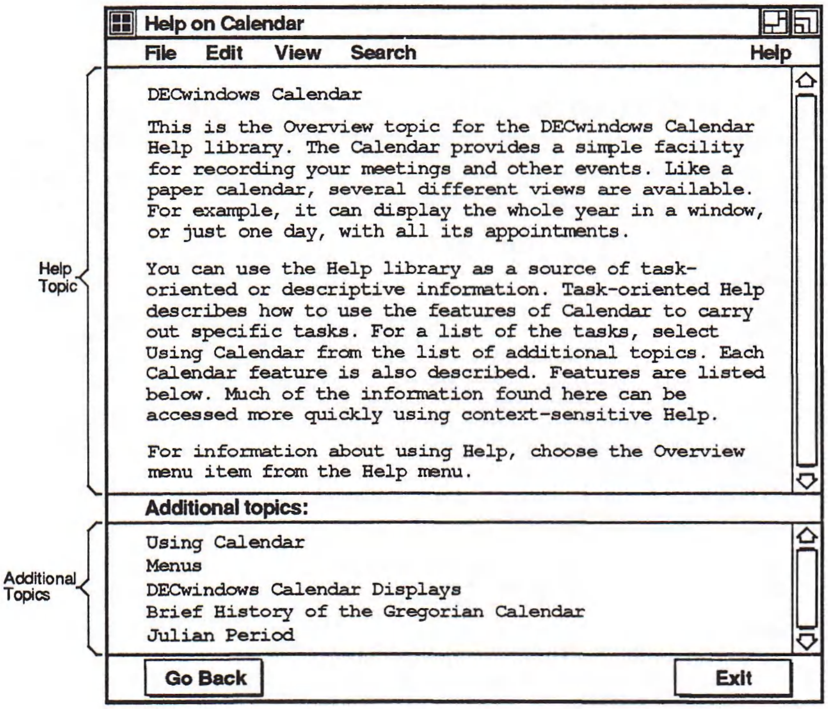
Help is designed to let you request general information on an application and quickly narrow the focus of your inquiry. In Help, you can

- Navigate quickly through help topics. Help keeps track of the path you used to get to a particular topic, which makes it easy for you to retrace your steps and follow a different path.
- Search Help for a keyword or topic supplied by the application.

A.18.1 Invoking Help

To get help on tasks in DECwindows applications, choose Overview from the application's Help menu.

A help window opens with the Overview topic displayed, including a list of additional topics that explain how to do common tasks.



```
DECwindows Calendar

This is the Overview topic for the DECwindows Calendar
Help library. The Calendar provides a simple facility
for recording your meetings and other events. Like a
paper calendar, several different views are available.
For example, it can display the whole year in a window,
or just one day, with all its appointments.

You can use the Help library as a source of task-
oriented or descriptive information. Task-oriented Help
describes how to use the features of Calendar to carry
out specific tasks. For a list of the tasks, select
Using Calendar from the list of additional topics. Each
Calendar feature is also described. Features are listed
below. Much of the information found here can be
accessed more quickly using context-sensitive Help.

For information about using Help, choose the Overview
menu item from the Help menu.
```

```
Using Calendar
Menus
DECwindows Calendar Displays
Brief History of the Gregorian Calendar
Julian Period
```

ZK-0498A-GE

- The help topic describes the task or object about which you requested help. Scroll bars appear if the text does not fit in one frame.
- The Additional topics list contains related topics that you can select to display more information. You select these topics by pointing to them and double clicking MB1.
- The help buttons, Go Back and Exit, let you display the previous help frame or exit from Help.

To get help on screen objects, such as menu names, scroll bars, and dialog boxes, point to the screen object and press and hold the Help key on your keyboard while you click MB1. Note that help on objects is not available in all DECwindows applications.

To get help on a menu item, press the Help key while you press and hold MB1 on the menu item, then release MB1.

A help window opens, displaying information on the object you specified.

You can display product information about your application, such as the software version number, by choosing About from the application's Help menu. In some applications, the Help menu also contains a Glossary menu item, which you can use to look up terms specific to your application.

For more information about using Help, choose Help from the Help menu in each application's help window.

A.18.2 Navigating in Help

When you select a topic from the Additional topics list in the Overview window, you start down a path that is limited only by your own curiosity. You can follow a path of topics by continuing to select additional topics, retrace your steps and branch off to a different topic, or return to the Overview frame and start down another path.

To display an additional topic:

1. Select the item from the list of additional topics.
2. Choose Go To from the View menu.

Faster still, just double click on the topic you want. Double clicking on a topic is a shortcut for selecting the topic and choosing Go To.

Help displays the selected topic. You can continue to select other topics from the Additional topics list or redisplay the topic you last saw by clicking on the Go Back button.

If you want to display the current topic and a new topic simultaneously:

1. Select another topic from the list of additional topics.
2. Choose Visit from the View menu.

Instead of replacing the current help topic with the new topic, another help window opens displaying information about the new topic. You can then explore other topics from the new help window and keep the current topic open for reference.

To see the path you followed to get to your current topic:

1. Choose History... from Help's Search menu.

   Help displays a dialog box that lists the topics you have already seen.
2. Double click on a topic to replace the current topic or select a topic and click on the Visit button to open another help window without replacing the current topic.

When you finish looking at a topic and want to close the help window, click on the Exit button. To return to the Overview frame, choose Go To Overview from the View menu in any help window.

A.18.3 Searching Help for Titles and Keywords

You can search Help for words or phrases to see whether they appear in topic titles or in help text. For example, you might want to see whether there is a topic title in Mail that contains the word "Sending", or a topic in which the phrase "message" appears.

To search for a word or phrase contained in a topic title:

1. Choose Title... from Help's Search menu.

   Help displays a dialog box.
2. In the Title text field, type the word or phrase you want to search for.

Leave this field empty if you want Help to display a list of every topic title.

3. Click on Apply.

   Help displays the topics whose titles contain the word or phrase you specified.

To display the topic whose title contains the word or phrase you searched for, double click on the topic or select it and click on the Visit button. Help displays the topic in another window. The Search Topic Titles dialog box remains open for you to continue your topic search.

Help for each application has predefined words or phrases called keywords that you can search for. To search for a keyword used in a topic:

1. Choose Keyword... from Help's Search menu.

   Help displays a dialog box that lists the keywords defined for that application.

2. Double click on the keyword you want to search for.

   Help lists the topics in which the keyword is used.

To display the topic in which the keyword is used, double click on the topic or select it and click on the Visit button. Help displays the topic in another window. The Search Topic Keywords dialog box remains open for you to continue your keyword search.

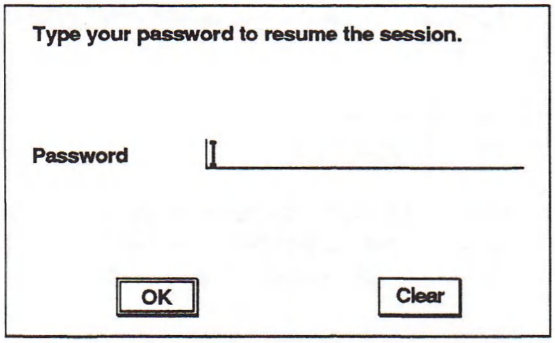
A.18.4 Exiting from Help

To exit from Help, click on the Exit button. If you have multiple help windows open, you must close each one.

A.19 Putting a Session on Hold

At any time, you can put your current session on hold indefinitely and lock your workstation without ending your session. When you put your session on hold, your screen is cleared, but your session is maintained exactly as it was. Any applications you started continue to run.

To put your current session on hold, choose Pause from the Session Manager's Session menu. Your screen is cleared and the Continue Session dialog box is displayed.



Type your password to resume the session.

ZK–0249A–GE

To continue your session, type your password and press Return. Once the system verifies your password, your session resumes. If the Continue Session dialog box remains on your screen, you probably made a typing mistake. Click on the Clear button and type your password again.

A.20 Ending a Session

You can end a session at any time. When you end a session, DECwindows stops all applications and clears the screen.

To end your session:

1. Choose Quit from the Session Manager's Session menu.

   The Session Manager displays a dialog box asking you to confirm that you want to end the session.

2. Click on OK or press Return.

Appendix B

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 more time or memory
- Data structure representations

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.

B.1 Data Structures

This section describes how to optimize the use of data structures in your code.

B.1.1 Integers

Fixnum arithmetic is much faster than bignum arithmetic. Therefore, if possible, use numbers in the range -2**28 to 2**28-1. The MOST-NEGATIVE-FIXNUM is -268435456; the MOST-POSITIVE-FIXNUM is 268435455. (The range of integers represented as fixnums in V3.0 was cut in half from V2.2.) 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 one instruction for each addition or subtraction operation and two 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 since it must be either T or NIL.

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 one longword for a header and enough space to represent the number in two's complement format with a minimum of two longwords. Therefore, working with bignums consumes much more time and space 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.

B.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.

B.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 have a 1-longword header.

B.1.4 Complex Numbers

The real part and the imaginary part of any complex number must be of the same type: both must be integers, single-floats, double-floats, or long-floats. Complex numbers occupy 2 longwords; they have a 1-longword header.

B.1.5 Characters

String characters use 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.

B.1.6 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 or 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 procedure 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.

Symbols occupy six longwords each.

B.1.7 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 procedure 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 tests for the MEMBER, ASSOC, and RASSOC functions are EQ and EQL, not EQUAL or = . The default test is 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 keys that are unknown when the structure is defined.

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, and automatically restore the values.

B.1.8 Strings, General Vectors, and Bit Vectors

Simple arrays of one dimension, [(SIMPLE-ARRAY * 1)], are processed faster than nonsimple vectors (vectors with fill pointers, adjustable vectors, or displaced vectors). Vectors that are simple 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 the EXPLODE or IMPLODE functions of earlier dialects of LISP). Strings are fully supported in this language, unlike these earlier dialects. 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 or with structures. 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 slightly 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 general vectors that hold characters or bits.

B.1.9 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 upon first use after any 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.

B.1.10 Functions

Compiled code is faster than interpreted code; when interpreted code is evaluated, much consing occurs.

Calling a compiled function that is a closure takes several instructions more than calling a compiled function that is not a closure. A function is a closure when it has free lexical variable references.

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 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.

B.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.

The format for using the OPTIMIZE declaration and its qualities with the PROCLAIM and DECLARE forms 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. Explicit type checking is ignored.

$x>y$ 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.

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. For example,

```
(let (( v ...))
  (declare (type (simple-array (unsigned-byte-8) 1)
                 v))
  (the fixnum (+ (aref v 0)
                 (aref v 1))))
```

only takes a few instructions when compiled with the (OPTIMIZE (SPEED 3) (SAFETY 0)) declarations. Declaring structures is equally helpful.

The PROCLAIM and DECLARE forms may be 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 most common cases. Even if the element type is not known at compile time, adding (SIMPLE-ARRAY * 1) declarations helps.

Multidimensional array operations benefit from 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.

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 the arithmetic operation is 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 as 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)
          (char-upcase (aref y i j k)))))))
```

B.3 Program Structure

In tight inner loops, use macros or in-line functions rather than called functions. Always compile macro definitions, functions proclaimed 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 after providing a definition, a compiler warning will result because the compiler will not have remembered the definition for the in-line function.

The FUNCALL and APPLY functions are slower than calls to functions whose names are known at compile time, 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

Using APPLY to call a function is usually two to three times slower than a compiled call to a named function with a fixed number of arguments.

The CATCH special form and operations that use the catch-throw mechanism are slower than a function call.

Using the &OPTIONAL keyword in a lambda-list costs a few instructions. 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; there is run-time code to parse the argument list and assign the proper values for the given keywords, and evaluate the defaults for the unsupplied 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 doing any I/O.

Similarly for output, calls to WRITE, PRINT, PRIN1, and PRINC are slower than calls to WRITE-CHAR, WRITE-LINE, and WRITE-STRING due to the need to determine what is being printed.

The FORMAT and PPRINT functions are slower than calls to the WRITE, PRINT, PRINC, and PRIN1 functions when pretty printing and *PRINT-CIRCLE* are disabled.

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.

B.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 lexical 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 functions 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 C

Customizing DECwindows from VAX LISP

This appendix describes how to customize the DECwindows-based development environment in VAX LISP V3.0.

C.1 Customizable Attributes

Your DECwindows-based development environment has a particular look. This look is made up of certain attributes, such as color, font, and the position of the windows. You can change the look of the DECwindows-based development environment by customizing the common attributes for all utilities and some specific attributes for a given utility. The attributes are independent of the syntax used for changing them. (See Section C.3 for information on syntax.) This section describes all the attributes.

C.1.1 Common Attributes

With the exceptions noted, you can set the following attributes for each utility, potentially for each window in a utility.

Geometry
The position, width, and height in pixels of the main window in a utility. For the Inspector, you can set only the geometry for the Inspector History window; the Inspect windows have more specific attributes (see Section C.1.2.1). For the Editor, you cannot set the geometry arbitrarily—there are more specific attributes for it (see Section C.1.2.5).

Color
The color of the foreground and background of each of the windows. On color systems, you can set these to arbitrary values, within the constraints of the number of colors available on your system. This setting is ignored on monochrome systems. The defaults are the values that you set with the DECwindows Session Manager. For information on named colors available to you, see *VMS DECwindows Guide to Xlib Programming*.

Monochrome reverse
Reverses the foreground and background colors on monochrome systems, black foreground on a light background or vice versa. This setting is ignored on color systems. The default is NIL—no change.

Font
The font used for all input and output in a utility. Individual windows in a utility can have different fonts. If you have specified a font description that does not exist on the server where LISP is displayed, the specified font is ignored and the

server's default font is used. If you change to a variable-width font, text may not appear to be column aligned as it is by default. The default for all utilities except the Editor is Courier Bold 14. The default for the Editor is Terminal 14, which is the same as the small DECterm font. The Editor disallows the use of a variable-width font. For information on how to find the font names that exist on your current system, see Section C.2.

Object recording
With object recording turned on, the position of an object is stored when it is written to the screen and pointer feedback is displayed. Turning off this attribute decreases the amount of system overhead for writing to the window. Turning off object recording in the Inspector, Trace List, Debugger Calling Stack, or Debugger Variable Bindings windows is not useful, because their syntax and function fundamentally rely on object information. This attribute is useful only for the Listener, Debug I/O window, and the Trace Output region of the Trace window. This attribute does not apply to the Editor because it does not do object recording. The default is T—object recording is turned on.

C.1.2 Specific Attributes and Restrictions

Some utilities have additional attributes that you can customize. The Editor has restrictions on customizing common attributes. These specific customizable attributes and restrictions are described below.

C.1.2.1 Inspector Attributes

Inspect windows count
An integer representing the number of Inspect windows that the Inspector rotates through as you perform subsequent inspections. The default is 5.

Inspect windows geometry
The size of the Inspect windows when they are first created and the position of the first Inspect window. The positions of subsequent windows are calculated from this position and the Inspect window geometry offset attribute.

If you resize, close, and reopen a window (that is, remove the window with the Close item and bring it back through a later call to INSPECT), it comes up the size to which you resized it. If you change the position portions of this value, only windows created after the change will be affected.

Inspect windows position offsets
The value in pixels indicating where subsequent windows are drawn relative to the initial Inspect window. For example, if the initial position is 800,100 and the offset is –50,100, the first window appears toward the upper right corner of your workstation screen (assuming it is 1024 pixels wide). The next window appears lower and to the left, at position 750,200. The next window is drawn along the same line, at 700,300.

Sequence length threshold
An integer indicating how many elements of a sequence to inspect before prompting you to continue. Only elements up to the threshold are computed and printed, at which time you are prompted for whether you would like to see more. The default is 75.

C.1.2.2 Listener Attributes

Evaluation history limit

An integer value denoting how many evaluations to store for command-line recall. If you raise the limit, your transcript and command recall ring grow. If you lower the limit, the oldest (OLD-VALUE–NEW-VALUE) evaluations are immediately removed from the transcript and the recall ring. The default is 40.

Prompt on Exit

When this attribute is turned on, you are prompted when you try to exit the Listener through the File menu Exit item. The default is for you to be prompted.

C.1.2.3 Debugger Attributes

Evaluation history limit

An integer value denoting how many evaluations to store for command-line recall. The default is 40.

Auxiliary windows initial states

Open or closed for the three Debugger auxiliary windows: Control Stack, Variable Bindings, and Debugger Commands, these values apply only to the first time you enter the Debugger. If you open or close any of them during a debugging session, their state as you leave the Debugger is how they will appear when you reenter it. The default is to have each window open.

Prompt on entry

True or false denoting whether you are prompted about entering the Debugger when an error or continuable error is signaled. The default is true, you are prompted about entering the Debugger.

Prompt on exit

True or false denoting whether you are prompted when you try to quit the Debugger through a menu item, command button, or without specifying the `T` argument to the Debugger `QUIT` command. The default is true, you are prompted about quitting the Debugger.

C.1.2.4 Trace Attributes

Prompt on untrace all

True or false denoting whether you are prompted when you choose the Untrace menu item with no selection. The default is true, you are prompted about untracing all items in the Trace List.

C.1.2.5 Editor Restrictions

The Editor does not have additional customizable attributes; rather, there are some restrictions on Editor customization.

Restrictions on Common Customizations

Editor width and height are specified in character-cell rows and columns, not in pixels. The Editor cannot handle a window that is not aligned on character cell boundaries. The font must be fixed width. The Editor does not correctly display text or draw the cursor, using a variable-width font.

Conflict with Older Editor Customizations Through LISP

For backward compatibility, any customizations made to the Editor through the DECwindows interface are overridden by any LISP initializations as supported in earlier versions of VAX LISP. For example, suppose that you change the default Editor screen width through customization. If, however, you still have a form like

```
(setf (editor:screen-width) 132)
```

in a LISP file that you load, then 132 is your Editor screen width, regardless of customization settings. If you want your stored customization settings to take precedence, you must remove all the LISP forms that set editor variables or SETF forms that modify the Editor's appearance.

C.2 Getting Information on Fonts

VAX LISP provides the LIST-FONTS function to get information on fonts.

LIST-FONTS function

You can use the LIST-FONTS function to find out which font names exist on the current system. Elements in this list are legal new values for the SETF forms with DECW-UTILS:CUSTOMIZE and the :FONT attribute argument. This function should be used only for customizing the development environment. For writing CLX programs you should use CLX:LIST-FONT-NAMES with the appropriate arguments, since the CLX:DISPLAY used by your program may not be the same one used by the development environment.

Format

DECW-UTILS:LIST-FONTS &OPTIONAL *fontname*

fontname

A string that names a font. The string may contain multiple occurrences of the wildcard character (*) to avoid supplying all components of the font name. Case is ignored.

Return Value

A list of strings that name the fonts available on the current X server whose names match the specified *fontname*, if supplied. Otherwise, all fonts known to the server are listed.

Examples

1.
```
Lisp> (decw-utils:list-fonts "*helvetica*-12-*")
("-Adobe-Helvetica-Medium-R-Normal--12-120-75-75-P-67-ISO8859-1"
 "-Adobe-Helvetica-Bold-R-Normal--12-120-75-75-P-70-ISO8859-1"
 "-Adobe-Helvetica-Bold-O-Normal--12-120-75-75-P-69-ISO8859-1"
 "-Adobe-Helvetica-Medium-O-Normal--12-120-75-75-P-67-ISO8859-1")
```

   This example asks for all the 12-point fonts in the Helvetica family. The order in which they are returned is dependent on the X server, so you should not count on it to always be the same. (The R means Roman and the O means Oblique.)

2.
```
Lisp> (decw-utils:list-fonts "*-bold-i-*-8-*")
("-Adobe-New Century Schoolbook-Bold-I-Normal--8-80-75-75-P-56-ISO8859-1"
 "-Adobe-Times-Bold-I-Normal--8-80-75-75-P-47-ISO8859-1")
```

   This example asks for all the bold and italic 8-point fonts.

C.3 Getting and Modifying Attributes

VAX LISP provides the CUSTOMIZATION function to modify the screen display.

CUSTOMIZATION function

This function returns the value of the attribute specified by its arguments. You can use it with SETF to change the value. With the exceptions noted below, when you use this routine with SETF, the appearance of the associated window changes immediately (if it is visible—if not, the change is apparent the next time the window is brought up). Changing the geometry with the pointer and the window manager is the same as doing it by calling the SETF form. Subsequent calls to the function will show this new value.

Format

DECW-UTILS:CUSTOMIZATION *object-keyword attribute-keyword*

object-keyword
This argument must be a keyword that specifies some object in the user interface (either a utility, a group of windows, a window, or a region of a window). The value of this argument restricts which *attribute-keyword* is applicable, as follows:

Table C–1: Object Keywords for DECW-UTILS:CUSTOMIZATION Function

Object Keyword	Attribute Keyword
:DEBUGGER	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :OBJECT-RECORDING :EVALUATION-HISTORY-LIMIT :PROMPT-ON-ENTRY :PROMPT-ON-EXIT
:CALLING-STACK-WINDOW	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :INITIAL-STATE
:VARIABLE-BINDINGS-WINDOW	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :INITIAL-STATE

(continued on next page)

Table C–1 (Cont.): Object Keywords for DECW-UTILS:CUSTOMIZATION Function

Object Keyword	Attribute Keyword
:DEBUGGER-COMMANDS-BOX	:FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :POSITION :INITIAL-STATE
:STEPPER-COMMANDS-BOX	:FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :POSITION :INITIAL-STATE
:EDITOR	:FONT :BOLD-FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :POSITION
:INSPECTOR	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :SEQUENCE-LENGTH-THRESHOLD
:INSPECT-WINDOWS	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :POSITION-OFFSETS :COUNT
:LISTENER	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :OBJECT-RECORDING :EVALUATION-HISTORY-LIMIT :PROMPT-ON-EXIT
:APROPOS-WINDOWS	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :POSITION-OFFSETS

(continued on next page)

Table C–1 (Cont.): Object Keywords for DECW-UTILS:CUSTOMIZATION Function

Object Keyword	Attribute Keyword
:DESCRIBE-WINDOWS	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :POSITION-OFFSETS
:TRACE	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE :GEOMETRY :OBJECT-RECORDING :PROMPT-ON-UNTRACE-ALL
:TRACE-LIST	:FONT :FOREGROUND-COLOR :BACKGROUND-COLOR :MONOCHROME-REVERSE

attribute-keyword

This argument must be a keyword that specifies an attribute. The value of this argument restricts the type of new value allowed when this function is used with SETF. The following table shows both the *object-keyword* with which each possible *attribute-keyword* can be used, as well as the types allowable for new values.

Table C–2: Attribute Keywords for Customization Functions

Attribute Keyword	Object Keyword†	New Value Type
:FONT	All objects except: :DEBUGGER-COMMANDS-BOX :STEPPER-COMMANDS-BOX	A CLX:FONT or string naming a font (wildcard characters allowed).
:BOLD-FONT	:EDITOR	A CLX:FONT or string naming a font (wildcard characters allowed).
:FOREGROUND-COLOR	All objects.	Either a CLX:COLOR, CLX:PIXEL, or string naming a color.
:BACKGROUND-COLOR	All objects.	Either a CLX:COLOR, CLX:PIXEL, or string naming a color.
:MONOCHROME-REVERSE	All objects.	T or NIL.

† "All objects" indicates all allowable values for the *object-keyword*.

(continued on next page)

Table C–2 (Cont.): Attribute Keywords for Customization Functions

Attribute Keyword	Object Keyword†	New Value Type
:GEOMETRY	All objects except: :EDITOR :DEBUGGER-COMMANDS-BOX :STEPPER-COMMANDS-BOX :TRACE-LIST	(x y width height) where each symbol's value is a fixnum.
:POSITION	:EDITOR :DEBUGGER-COMMANDS-BOX :STEPPER-COMMANDS-BOX	(x y) where each symbol's value is a fixnum.
:POSITION-OFFSETS‡	:INSPECT-WINDOWS :APROPOS-WINDOWS :DESCRIBE-WINDOWS	(delta-x delta-y) where each symbol's value is a fixnum.
:OBJECT-RECORDING	:DEBUGGER :LISTENER :TRACE	T or NIL.
:EVALUATION-HISTORY-LIMIT	:DEBUGGER :LISTENER	A fixnum.
:PROMPT-ON-ENTRY	:DEBUGGER	T or NIL.
:PROMPT-ON-EXIT	:DEBUGGER :LISTENER	T or NIL.
:PROMPT-ON-UNTRACE-ALL	:TRACE	T or NIL.
:INITIAL-STATE‡	:CALLING-STACK-WINDOW :VARIABLE-BINDINGS-WINDOW :DEBUGGER-COMMANDS-BOX :STEPPER-COMMANDS-BOX	One of the keywords :OPEN or :CLOSED.
:SEQUENCE-LENGTH-THRESHOLD‡	:INSPECTOR	A fixnum.
:COUNT‡	:INSPECT-WINDOWS	A fixnum.

† "All objects" indicates all allowable values for the *object-keyword*.

‡ This attribute does not immediately affect the appearance of any window.

Return Value

The return value of the function depends on the *attribute-keyword* (see Table C–2).

Examples

1.
```
Lisp> (decw-utils:customization :listener :font)
#<Font -Adobe-Courier-Medium-R-Normal--14-140-75-75-M-90-ISO8859-1>
Lisp> (setf (decw-utils:customization :listener :font)
            "*courier-bold-*14*")
"*courier-bold-*14*"
Lisp> (decw-utils:customization :listener :font)
#<Font -Adobe-Courier-Bold-R-Normal--14-140-75-75-M-90-ISO8859-1>
```

The first evaluation returns the font currently being used by the Listener, the next evaluation sets this to a new font, and the last evaluation returns the new font.

2.
```
Lisp> (decw-utils:customization :listener :geometry)
(200 200 750 500)
```

This shows where your listener is located. If you move it by hand, you might see:

3.
```
Lisp> (decw-utils:customization :listener :geometry)
(203 245 750 500)
```

This gives you the exact window position. It is hard to move the windows exactly and in this case you actually moved to the right by 3 pixels. If you wanted to move a window down by exactly 50 pixels, it is more accurate to do it through the form, as follows:

4.
```
Lisp> (setf (decw-utils:customization :listener :geometry)
            (let ((geometry-list
             (decw-utils:customization :listener :geometry)))
              (incf (second geometry-list) 50)
              geometry-list))
(200 250 750 500)
```

Here you get the return value; modify and use it to set the new value. The window will move immediately to the new position.

5.
```
Lisp> (setf (decw-utils:customization :listener :geometry)
            '(nil 0 nil nil))
(200 0 750 500)
```

This example uses NIL to indicate that this field of the geometry list should be unchanged. In this example, the window is moved to the left edge of the screen.

6.
```
Lisp> (decw-utils:customization :listener :foreground-color)
#<Color red: 0.00 :green 0.00 :blue 0.00>
```

This example shows that the Listener is writing text in black.

7.
```
Lisp> (setf (decw-utils:customization :listener :foreground-color)
            "blue")
"blue"
Lisp> (decw-utils:customization :listener :foreground-color)
#<Color red: 0.00 :green 0.00 :blue 0.98>
```

This might be the result if you tried to set the Listener foreground to blue. The exact color set depends on the server implementation and how many colors remain unallocated in the default colormap.

8.
```
Lisp> (setf (decw-utils:customization :listener :background-color)
            (clx:make-color :red .9 :green .9))
#<Color red: 0.90 :green 0.90 :blue 0.00>
```

This example sets a color to an arbitrary mix of RGB values. Again, the exact color depends on the server.

9.
```
Lisp> (setf (decw-utils:customization :listener :background-color) 239)
239
```

   In this case, if you got the CLX:PIXEL value 239 from some other window or widget that used the default colormap, it would change the Listener background to this color.

10.
```
Lisp> (setf (decw-utils:customization :listener :monochrome-reverse) t)
```

   On a monochrome system, this example reverses the current light-on-dark or dark-on-light appearance of the window.

11.
```
Lisp> (decw-utils:customization :inspect-window :count)
5
Lisp> (setf (decw-utils:customization :inspect-window :count) 3)
3
```

   This example shows how to determine and change the number of Inspect windows used in the Inspector.

12.
```
Lisp> (setf (decw-utils:customization :calling-stack-window
                                      :initial-state)
            (decw-utils:customization :variable-bindings-window
                                      :initial-state))
:CLOSED
```

   This example sets the Debugger Calling Stack window's initial state to the same value as the initial state of the Debugger Variable Bindings window.

C.4 Saving Customizations

VAX LISP provides the SAVE-CUSTOMIZATIONS function to save current customizations.

SAVE-CUSTOMIZATIONS function

This function saves all the current customizations into the file specified by merging *file* against its default. You can load files written out by the function by using DECW-UTILS:LOAD-CUSTOMIZATIONS.

Format

DECW-UTILS:SAVE-CUSTOMIZATIONS &KEY *file*

file
A string or pathname that names a file or directory into which your customizations should be written. The default value is "DECW$USER_DEFAULTS:LISP$DEFAULTS.DAT".

Return Value

The namestring of the file actually written.

Examples

1.
```
Lisp> (decw-utils:save-customizations)
"LISPZ$:[JONES]LISP$DEFAULTS.DAT;3"
```

   This example overwrites previously saved customizations in the default file.

2.
```
Lisp> (decw-utils:save-customizations :file "defdir:")
"LISPZ$:[JONES.DEFAULTS]LISP$DEFAULTS.DAT;1"
```

   This example specifies a directory to keep defaults.

3.
```
Lisp> (decw-utils:save-customizations :file "lisp-bigfonts")
"LISPZ$:[JONES]LISP-BIGFONTS.DAT;42"
```

   This example writes out a specialized defaults file.

4.
```
Lisp> (decw-utils:save-customizations
        :file "defdir:lisp-widewindows.init")
"LISPZ$:[JONES.DEFAULTS]LISP-WIDEWINDOWS.INIT;3"
```

   This example writes out a specialized defaults file for a certain geometry setup and puts it in a preferred directory.

C.5 Recalling Customizations

Every time LISP starts up it reads the previously saved information. If no user customizations have been stored, the system defaults are read from DECW$SYSTEM_DEFAULTS:LISP$DEFAULTS.DAT. If you make modifications and would like to reset to your previously saved settings, use the `LOAD-CUSTOMIZATIONS` function.

LOAD-CUSTOMIZATIONS function

This function loads customizations previously saved with `DECW-UTILS:SAVE-CUSTOMIZATIONS`. By default, this function searches, in order, the default directory, DECW$USER_DEFAULTS: and DECW$SYSTEM_DEFAULTS: for the file LISP$DEFAULTS.DAT. (By default, this function should always find DECW$SYSTEM_DEFAULTS:LISP$DEFAULTS.DAT, which is supplied by the VAX LISP system.) If *file* is supplied, it is merged against "DECW$USER_DEFAULTS:LISP$DEFAULTS.DAT". If the result specifies a directory, only that directory is searched; otherwise, the search path described above is used. If the file is not found, an error is signaled. This function is called automatically, without arguments, every time the development environment is started. If you have previously saved your customizations into LISP$DEFAULTS.DAT in either your default directory or DECW$USER_DEFAULTS: with `DECW-UTILS:SAVE-CUSTOMIZATIONS`, your saved customizations are in effect from the start.

Format

DECW-UTILS:LOAD-CUSTOMIZATIONS &KEY ***file***

file

A string or pathname that names a file into which your customizations have been written. If you do not name a file, the function loads customizations from either DECW$USER_DEFAULTS:LISP$DEFAULTS.DAT or DECW$SYSTEM_DEFAULTS:LISP$DEFAULTS.DAT, which is supplied by VAX LISP.

Return Value

The namestring of the file actually read.

Examples

(These examples refer to information in the examples in the `DECW-UTILS:SAVE-CUSTOMIZATIONS` routine description.)

1.
```
Lisp> (load-customizations)
"DECW$SYSTEM_DEFAULTS:LISP$DEFAULTS.DAT;1"
```

This is the result if you never previously saved customizations.

2.
```
Lisp> (load-customizations)
"LISPZ$:[JONES]LISP$DEFAULTS.DAT;1"
```

This is the result if you saved customizations into the default file.

3.
```
Lisp> (load-customizations :file "lisp-bigfonts")
"LISPZ$:[JONES]LISP-BIGFONTS.DAT;42"
```

This example loads the specialized defaults file.

4.
```
Lisp> (load-customizations :file "defdir:lisp-widewindows.init")
"LISPZ$:[JONES.DEFAULTS]LISP-WIDEWINDOWS.INIT;3"
```

This example loads the specialized geometry.

Appendix D

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 causes 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. Additional EMACS related examples are in LISP$EXAMPLES:EDINIT.

This appendix is organized as follows:

- Section D.1 explains to a new user how to learn about the Editor.
- Section D.2 describes how to activate the "EMACS" style as a minor or major style, thus making the "EMACS" key bindings available to you.
- Section D.3 lists the key bindings in the "EMACS" style.

D.1 Introduction to the Editor

To learn about using the Editor from a terminal interface, read Chapter 3. To learn about using the Editor in the DECwindows environment, read Chapter 8. Most of the information in these chapters 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 D–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 D–1 is arranged in the approximate order that the key bindings and commands are presented in Chapter 3. (Table D–1 lists only those commands listed in Chapter 3. The full set of "EMACS" style key bindings is presented in Section D.3.)

Section 3.2.1 contains information on editing using 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 D–1: Commands for Manipulating Buffers and Windows

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

D.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.

D.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"))
```

D.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"))
```

D.3 "EMACS" Style Key Bindings

Table D–2 lists the key bindings supplied in the "EMACS" style. Appendix E 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 E 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 D–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
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

(continued on next page)

Table D–2 (Cont.): "EMACS" Style Key Bindings

Key(s)	Command
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
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

(continued on next page)

Table D–2 (Cont.): "EMACS" Style Key Bindings

Key(s)	Command
	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 E

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 E.1 lists the Editor commands, along with each command's key bindings and a brief description of the command.
- Section E.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.

E.1 Editor Command Descriptions

Table E–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, see the *VAX LISP/VMS Editor Programming Guide*.

Table E–1: Editor Commands and Key Bindings

Name	Binding(s)	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.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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.
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-parenthesis characters at the cursor position.
Copy from Pointer[2]	`:GLOBAL` ○ ○ ↑	Sets the end of secondary selection and copies the text to the window that has input focus; check that input focus is correctly set before initiating this command.

[2] Available only in DECwindows Pointer Syntax.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
Copy to Pointer[2]	`:GLOBAL` [○ ○ ↓]	Moves the current buffer point to the position indicated by the pointer and inserts the text from the primary selection at that location. If pointer is beyond the end of a line, inserts the text at the end of the line. If pointer is beyond the end of the buffer region, inserts the text at the end of the buffer region.
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[3]	`(:STYLE "VAX LISP")` [○ ○ ↓]	Calls the `DESCRIBE` function with the word at the pointer position as the argument.

[2] Available only in DECwindows Pointer Syntax.

[3] Available only in UIS Pointer Syntax.

[○ ↓ ●] 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
Downcase Region	None	Makes all alphabetic characters in the current select region lowercase.
Downcase Word	(:STYLE "EMACS") Escape l	Makes all alphabetic characters in the word at the cursor position lowercase.
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") keypad 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 [5]and F12 [4]	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 the number of lines specified by the prefix argument.
EDT Change Case	(:STYLE "EDT Emulation") PF1 keypad 1	Changes from lowercase to uppercase (or vice versa) all characters in the select region or, if no select region is defined, the character at the cursor position.
EDT Cut	(:STYLE "EDT Emulation") keypad 6 and Remove [4]and ○ ↓ ○ [3]	Deletes the current select region and replaces the contents of the paste buffer with it.
EDT Delete Character	(:STYLE "EDT Emulation") keypad .	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.

[3] Available only in UIS Pointer Syntax.

[4] Key available only on LK201 keyboard.

[5] Key available only on VT100 terminal.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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 [5]and F13 [4]	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.
EDT Delete to End of Line	`(:STYLE "EDT Emulation")` PF1 keypad 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")` keypad -	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 keypad .	Cancels the current select region; equivalent to `"Unset Select Mark"`.
EDT End of Line	`(:STYLE "EDT Emulation")` keypad 2	Moves the cursor to the end of the current, next, or previous line, depending on starting cursor position and current direction; moves the number of lines specified by the prefix argument.
EDT Move Character	`(:STYLE "EDT Emulation")` keypad 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")` keypad 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")` keypad 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 keypad 6 and Insert Here [4]	Inserts the contents of the paste buffer at the cursor location.
EDT Paste at Pointer[3]	`(:STYLE "EDT Emulation")` ● ↓ ○	Inserts the contents of the paste buffer at the pointer cursor location.

[3] Available only in UIS Pointer Syntax.

[4] Key available only on LK201 keyboard.

[5] Key available only on VT100 terminal.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
EDT Query Search	`(:STYLE "EDT Emulation")` PF1 PF3 and Find [4]	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 keypad 9	Replaces the current select region with the contents of the paste buffer.
EDT Scroll Window	`(:STYLE "EDT Emulation")` keypad 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.
EDT Select	`(:STYLE "EDT Emulation")` keypad . and Select [4]	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")` keypad 5	Sets the current direction to backward.
EDT Set Direction Forward	`(:STYLE "EDT Emulation")` keypad 4	Sets the current direction to forward.
EDT Special Insert	`(:STYLE "EDT Emulation")` PF1 keypad 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 keypad ,	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 keypad -	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/A	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.

[4] Key available only on LK201 keyboard.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
End Keyboard Macro	:GLOBAL Ctrl/X)	Ends the collection of keystrokes for a keyboard macro.
End of Buffer	(:STYLE "EDT Emulation") PF1 keypad 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 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.
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 [4] (:STYLE "EDT Emulation") PF1 keypad 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	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.

[4] Key available only on LK201 keyboard.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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 [4]	Displays help on your current situation.
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.

[4] Key available only on LK201 keyboard.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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.
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 [○ ↓ ○] [3]	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[1]	:GLOBAL [↑ ○ ○]	If the pointer cursor has not moved, cancels the select region that was started with [↓ ○ ○] ; if the pointer cursor has moved since [↓ ○ ○] , does nothing.
Move Point and Select Region[1]	:GLOBAL [● ○ ○] →	Moves the text cursor with the pointer cursor, marking a select region.
Move Point to Pointer[1]	:GLOBAL [● ○ ○]	Moves the text cursor to the pointer cursor.

[1] Available in both DECwindows and UIS Pointer Syntax.

[3] Available only in UIS Pointer Syntax.

[○ ↓ ●] 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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.
Next Line	:GLOBAL ↓ (:STYLE "EMACS") Ctrl/N	Moves the cursor to the next line or down 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 [4]	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 keypad 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 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 the number of lines specified by the prefix argument; keeps the cursor in the same column if possible.

[4] Key available only on LK201 keyboard.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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 [4] (:STYLE "EMACS") Escape v	Moves the cursor up in the buffer by one screenful or 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/Space	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 for whatever is being prompted.
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 let you 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 \ (:STYLE "EMACS") Ctrl/X q	Inserts the next character typed at the cursor location without Editor interpretation.
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 (:STYLE "EMACS") Ctrl/L	Erases and redisplays everything on the screen.
Remove Current Window	:GLOBAL Ctrl/X Ctrl/R (:STYLE "EMACS") Ctrl/X d	Removes the current window from the screen; does not delete the associated buffer.
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 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 number of rows specified by the prefix argument.

[4] Key available only on LK201 keyboard.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
Secondary Select Region[2]	:GLOBAL ○ ○ ●	Sets the beginning of secondary selection (used in Copy from Pointer command).
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[1]	(:STYLE "VAX LISP") ↓ ○ ○	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/Space	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 DECwindows Pointer Syntax	None	Unbinds the UIS pointer bindings and binds the DECwindows pointer bindings.
Set UIS Pointer Syntax	None	Unbinds the DECwindows pointer bindings and binds the UIS pointer bindings.
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.
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") keypad . (:STYLE "EMACS") Ctrl/Space	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 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 cataloged 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.

[1] Available in both DECwindows and UIS Pointer Syntax.

[2] Available only in DECwindows Pointer Syntax.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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 keypad . (:STYLE "EMACS") Escape Ctrl/Space	Cancels the current select region.
Upcase Region	None	Changes all alphabetic characters in the current select region to uppercase.
Upcase Word	(:STYLE "EMACS") Escape u	Changes all alphabetic characters in the word at the cursor location to uppercase.
View File	(:STYLE "EMACS") Ctrl/X Ctrl/F	Prompts for a file specification, then reads the specified file into a read-only buffer.
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[3]	(: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.

[3] Available only in UIS Pointer Syntax.

○ ↓ ● 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.

(continued on next page)

Table E–1 (Cont.): Editor Commands and Key Bindings

Name	Binding(s)	Description
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.

○ ↓ ● 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.

E.2 Editor Key Bindings

Table E–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. If you have bound a key or key sequence to a command in the context of a particular buffer, and that buffer is the current buffer, 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 E–2: Editor Key Bindings

Key(s)	Context	Command
	Single Keys	

Ctrl/Space	`(:BUFFER "General Prompting")`
	`(:STYLE "EMACS")`
		Prompt Complete String
		Set Select Mark

Ctrl/A	`(:STYLE "EMACS")`	Beginning of Line

(continued on next page)

Table E–2 (Cont.): Editor Key Bindings

Key(s)	Context	Command
	Single Keys	
Ctrl/B	`(:STYLE "EMACS")`	Backward Character
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")`
	`(:STYLE "VAX LISP")`
	`(:STYLE "EDT Emulation")`
		New Line
		New LISP Line
		EDT Delete Previous Word

Ctrl/K	`(:STYLE "EMACS")`	Kill Line
Ctrl/L	`(:STYLE "EMACS")`	Redisplay Screen

Return or Ctrl/M	`(:BUFFER "General Prompting")`
	`(:STYLE "EMACS")`
	`:GLOBAL`
		Prompt Read and Validate
		New Line
		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")`
	`(:STYLE "EDT Emulation")`
		Supply EMACS Prefix
		EDT Delete Previous Line

Ctrl/V	`(:BUFFER "General Prompting")`
	`(:STYLE "EMACS")`
		Prompt Scroll Help Window
		Next Screen

Ctrl/W	`(:STYLE "EMACS")`
	`(:STYLE "EDT Emulation")`
		Kill Region
		Redisplay Screen

Ctrl/Y	`(:STYLE "EMACS")`	Yank

Ctrl/Z	`(:STYLE "EMACS")`
	`:GLOBAL`
		Scroll Window Down
		Execute Named Command

Ctrl/\	`(:STYLE "EMACS")`	EMACS Forward Search
Ctrl/?	`(:STYLE "VAX LISP")`	Describe Word

Delete or ⌫	`(:STYLE "EMACS")`
	`(:STYLE "EDT Emulation")`
	`:GLOBAL`
		Delete Previous Character
		Delete Previous Character
		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

(continued on next page)

Table E–2 (Cont.): Editor Key Bindings

Key(s)	Context	Command
	Single Keys	
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") (:STYLE "EDT Emulation") :GLOBAL	Prompt Help Help Help
keypad PF3	(:STYLE "EDT Emulation")	EDT Search Again
keypad PF4	(:STYLE "EDT Emulation")	EDT Delete Line
↑	:GLOBAL	Previous Line
↓	:GLOBAL	Next Line
→	:GLOBAL	Forward Character
←	: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

(continued on next page)

Table E–2 (Cont.): Editor Key Bindings

Key(s)	Context	Command
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") :GLOBAL	Ed Execute Keyboard Macro
Ctrl/X Ctrl/F	(:STYLE "EMACS")	View File
Ctrl/X Tab or Ctrl/X Ctrl/I	(:STYLE "EMACS") (:STYLE "VAX LISP")	Insert File 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") :GLOBAL	Read File 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") :GLOBAL	Shrink Window 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
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

(continued on next page)

Table E–2 (Cont.): Editor Key Bindings

Key(s)	Context	Command
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
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 ⌫	(:STYLE "EMACS")	Delete Previous Word

(continued on next page)

Table E–2 (Cont.): Editor Key Bindings

Key(s)	Context	Command
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
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

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua NH 03061
Rest of U.S.A. and Puerto Rico[1]	800–DIGITAL	
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order Desk
Internal orders (for software documentation)	—	Software Supply Business (SSB) Digital Equipment Corporation Westminster MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Services (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro MA 01532

[1] Prepaid orders from Puerto Rico, call Digital's local subsidiary (809–754–7575)

Reader's Comments

VAX LISP/VMS Program Development Guide
AA–MK69A–TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as described)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

What I like best about this manual: ____________________

____

What I like least about this manual: ____________________

____

I found the following errors in this manual:

Page Description

____ ____________________

____ ____________________

____ ____________________

____ ____________________

My additional comments or suggestions for improving this manual:

____

____

____

Please indicate the type of user/reader that you most nearly represent:

□ Administrative Support
□ Computer Operator
□ Educator/Trainer
□ Programmer/Analyst
□ Sales

□ Scientist/Engineer
□ Software Support
□ System Manager
□ Other (please specify) ____________________

Name/Title ____________________ Dept. ____________

Company ____________________ Date ____________

Mailing Address ____________________

____________________ Phone ____________

— — Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — — —

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
PKO3-1/30D
129 PARKER STREET
MAYNARD, MA 01754-2198

— — Do Not Tear — Fold Here —

Cut Along Dotted Line