VAX LISP/VMS
Editor Programming Guide

Order Number: AA-Y923C-TE

May 1986

This document contains information required by a LISP language
programmer to write programs that extend the VAX LISP Editor.

Operating System and Version: VAX/VMS Version 4.2

Software Version: VAX LISP/VMS Version 2.0

CONTENTS

PREFACE

Manual Objectives

The *VAX LISP/VMS Editor Programming Guide* provides the information needed to program the VAX LISP Editor in order to extend and customize its capabilities.

Intended Audience

Readers of this manual are assumed to have a working knowledge of LISP programming and to be able to use the VAX LISP Editor as provided.

- The VAX LISP language elements are described in *COMMON LISP: The Language.* *

- Instructions for using the VAX LISP Editor appear in the *VAX LISP/VMS User's Guide.*

Readers who are not familiar with LISP programming can use the VAX LISP Editor as provided, but should not attempt to customize it.

Structure of This Document

This manual is organized in three parts:

- **PART I, GUIDE TO EDITOR PROGRAMMING,** introduces the techniques of Editor programming in a task-oriented fashion. It contains six chapters, each covering a major area of Editor programming.

    - Chapter 1 provides an overview of the subsystems of the Editor and of the data types that each subsystem contains. It also describes the methods of accessing Editor objects.

---

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

- Chapter 2 describes the techniques of creating Editor commands.

- Chapter 3 describes the techniques of binding Editor commands to keyboard keys and pointer actions.

- Chapter 4 introduces the Editor's text operations subsystem and the techniques of extending it.

- Chapter 5 introduces the Editor's window and display operations subsystem and the techniques of extending it.

- Chapter 6 describes the techniques of modifying the Editor's styles and of creating new styles.

- **PART II, CONCEPTS IN EDITOR PROGRAMMING**, contains programming information arranged for quick reference. This part consists of separate, alphabetically arranged articles on each of the major concepts and data types used in Editor programming.

- **PART III, EDITOR OBJECT DESCRIPTIONS**, describes the individual functions, variables, and other objects provided with the Editor. The descriptions are arranged alphabetically by object name.

This manual also contains four appendixes:

- **Appendix A, Editor Objects by Category**, contains lists of the functions, variables, and other objects provided with the Editor, categorized by the major concepts and data types used in Editor programming.

- **Appendixes B, C,** and **D** list all the commands provided with the Editor, all bound keys, and other information that is useful in binding Editor commands to keys and key sequences. These appendixes also appear in the *VAX LISP/VMS User's Guide.*

Associated Documents

The following documents are relevant to programming the VAX LISP Editor:

- The *VAX LISP/VMS User's Guide* provides general information about using VAX LISP, and serves as a guide to generally helpful VMS documentation. This manual also presents information on using the VAX LISP Editor as provided.

- *COMMON LISP: The Language* provides a definition of the COMMON LISP language.

- The *VAX LISP/VMS Graphics Programming Guide* explains the use of the VAX LISP programming interface to VAXstation graphics.

Conventions Used in This Document

The following conventions are used in this manual:

Convention	Meaning
( )	Parentheses used in examples of LISP code indicate the beginning and end of a LISP form. For example: (SETQ NAME LISP)
UPPERCASE	LISP symbols are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters.
lowercase italics	Lowercase italics in function and macro descriptions and in text indicate arguments that you supply to a function or macro; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters.
. . .	In LISP code examples, a vertical ellipsis indicates that lines of code not pertinent to the example are omitted.
{ }	In function and macro format specifications, braces enclose elements that are considered to be one unit of code. For example: {*keyword value*}
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered to be one unit of code and that can be repeated zero or more times. For example: {*keyword value*}*
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments after it are defined to be optional. For example: COPY-MARK *mark* &OPTIONAL *mark-type* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.

Convention	Meaning
&REST	In function and macro format specifications, the word &REST indicates that an indefinite number of arguments may appear. For example:

    INVOKE-HOOK *name* &REST *args*

Do not specify &REST when you invoke the function or macro whose definition includes &REST.

| &KEY | In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: |

    LOCATE-ATTRIBUTE *mark-or-string attribute*
       &KEY :TEST :CONTEXT :DIRECTION :LIMIT

Do not specify &KEY when you invoke the function or macro whose definition includes &KEY.

| CTRL/*x* | CTRL/*x* indicates a control character generated when you hold down the key labeled CTRL while you press another key. For example: |

    CTRL/Z or CTRL/Y

SUMMARY OF NEW AND CHANGED INFORMATION

This section summarizes the ways in which the Version 2.0 VAX LISP/VMS Editor and its documentation differ from previous versions.

Document Supersession

This document supersedes the *VAX LISP Editor Manual*. The relationship of the *VAX LISP/VMS Editor Programming Guide* to the *VAX LISP Editor Manual* is as follows:

- Part I of the *Editor Programming Guide* is new material.

- Part II of the *Editor Programming Guide* is a reorganization of the material in the *Editor Manual*, Chapter 2.

- Part III of the *Editor Programming Guide* corresponds to the *Editor Manual*, Chapter 3, with additions and revisions as noted below.

- Appendix A of the *Editor Programming Guide* corresponds to the *Editor Manual*, Appendix A, with additions and revisions as noted below.

- Appendixes B, C, and D of the *Editor Programming Guide* are new material. They contain tables that are identical to tables included in the *VAX LISP/VMS User's Guide*.

- The *Editor Manual*, Chapter 1, has been revised and now appears as Chapter 3 of the *VAX LISP/VMS User's Guide*, Using the VAX LISP Editor.

New and Revised Editor Features

The major additions and changes to the VAX LISP Editor for VAX LISP/VMS Version 2.0 are the following:

- Improved speed of Editor operation

- Support for the pointing device on the AI VAXstation in the Editor environment

- Revision of the Editor's display architecture to support VAXstation windowing

- Recursive use of the ED function

- Use of a permanently displayed window for all Editor prompting

- Some miscellaneous features under the control of the programmer

  - A right margin setting that causes text inserted from the keyboard to wrap

  - A text overstrike mode

  - The ability to specify the number of anchored windows the Editor normally displays

  - The ability to specify the buffer onto which the Editor displays a window when it has no other window to display

New and Revised Editor Objects

The *VAX LISP/VMS Editor Programming Guide* documents the following Editor objects for the first time with **VAX LISP/VMS, Version 2.0.** These objects are described in Part III and listed under the appropriate categories in Appendix A.

New Functions

BIND-POINTER-COMMAND
BUFFER-HIGHLIGHT-REGIONS
ENQUEUE-EDITOR-COMMAND
GET-POINTER-STATE
POINTER-STATE-ACTION
POINTER-STATE-BUTTONS
POINTER-STATE-P
POINTER-STATE-TEXT-POSITION
POINTER-STATE-WINDOW-POSITION
POSITION-WINDOW-TO-MARK
UNBIND-POINTER-COMMAND

New Macro

RETURN-FROM-EDITOR

New Commands

"Close Outermost Form"
"Describe Word at Pointer"
"EDT Paste at Pointer"
"Exit Recursive Edit"
"Kill Enclosing List"
"Kill Next Form"
"Kill Previous Form"
"Kill Rest of List"
"Maybe Reset Select at Pointer"
"Move Point and Select Region"
"Move Point to Pointer"
"Page Previous Window"
"Select Enclosing Form at Pointer"
"Set Select Mark"
"Unset Select Mark"
"Yank at Pointer"

New Variable

EDITOR-DEFAULT-BUFFER

New Editor Variables

"Anchored Window Show Limit"
"Buffer Right Margin"
"Current Window Pointer Pattern"
"Current Window Pointer Pattern X"
"Current Window Pointer Pattern Y"
"Editor Initialization Hook"
"Information Area Pointer Pattern"
"Information Area Pointer Pattern X"
"Information Area Pointer Pattern Y"
"Noncurrent Window Pointer Pattern"
"Noncurrent Window Pointer Pattern X"
"Noncurrent Window Pointer Pattern Y"
"Prompt Rendition Complement"
"Prompt Rendition Set"
"Select Region Rendition Complement"
"Select Region Rendition Set"
"Text Overstrike Mode"

The *VAX LISP/VMS Editor Programming Guide* also documents the following
Editor objects that have been changed for VAX LISP/VMS, Version 2.0.

Functions

BUFFER-WRITABLE
ED
PROMPT-FOR-INPUT
REGION-READ-POINT
REMOVE-WINDOW
SHOW-WINDOW
SIMPLE-PROMPT-FOR-INPUT
WINDOW-DISPLAY-START

Commands

"EDT Deselect"
"EDT Select"
"Insert Close Paren and Match"
"Remove Current Window"

Editor Variable

"Default Window Lines Wrap"

The documentation of the following objects has been revised to correct
documentation errors or omissions.

Functions

ALTER-WINDOW-HEIGHT
CURRENT-WINDOW
DESCRIBE-OBJECT-COMMAND
EDIT-LISP-OBJECT-COMMAND
EXIT-EDITOR-COMMAND
INSERT-STRING
MAKE-BUFFER
MAKE-WINDOW
NEXT-WINDOW
UPDATE-WINDOW-LABEL
WINDOW-DISPLAY-COLUMN
WINDOW-DISPLAY-ROW
WINDOW-LABEL-RENDITION
WORD-OFFSET

Commands

"Beginning of Buffer"
"Beginning of Line"
"Describe Word"
"Downcase Region"
"EMACS Forward Search"
"Exit"
"Help"
"Pause Editor"
"Set Screen Width"

Editor Variable

"Buffer Exit Hook"

Buffers

EDITOR-HELP-BUFFER
EDITOR-PROMPTING-BUFFER

PART I

GUIDE TO EDITOR PROGRAMMING

CHAPTER 1

EDITOR OVERVIEW

The VAX LISP Editor is an interactive LISP program that enables the user to insert, display, and manipulate text. The behavior and capabilities of the Editor as they appear to the interactive user are described in the *VAX LISP/VMS User's Guide*.

The Editor is designed to be easily modified and extended. Since the Editor is written entirely in LISP, you can alter it by writing new LISP code that

- Modifies the behavior of Editor commands

- Binds commands to key sequences or actions of a pointing device

- Modifies the Editor's initial features, such as the labeling of its windows, the frequency of checkpointing, the size of the information area, and so on

- Adds new capabilities, such as justification of text, parsing of LISP code, recognition of the syntax of another programming language, and so on

To write Editor-related code, you use the same functions and data types that were used to develop the Editor originally. These include some specially defined "Editor objects"; you can also use any object defined in VAX LISP.

This chapter provides an overview of the Editor as a LISP program and of its data types. The intent is to orient you to the process of extending the Editor and to the range of possible extensions. This chapter also provides some basic programming information that is needed to follow the discussion in later chapters.

This chapter contains:

- Some illustrations of simple Editor extensions

- An overview of the Editor's subsystems and utilities

- An introduction to Editor data types and the means of referencing them

1.1 SOME COMMON EDITOR EXTENSIONS

This section includes the LISP code that implements several different kinds of Editor extensions. These simple examples serve to illustrate the process of programming the Editor.

The following Editor extensions are shown:

- Changing the frequency of checkpointing

- Changing the number of windows that the Editor normally displays at one time

- Changing the default major style

- Binding a DIGITAL-supplied command to a key sequence

- Defining a new command to change the width of the terminal screen

If you wish to make any of these changes in your own Editor, you simply execute the forms as shown. You can execute them either by typing them at top level LISP or by loading them from a file.

Note in these examples that the symbols for DIGITAL-supplied objects that relate to the Editor are referenced with the package prefix EDITOR:. For a full discussion of the package location of Editor objects, see Section 1.3.5 below.

1.1.1 Changing the Frequency of Checkpointing

The Editor checkpoints buffers associated with files after every 350 commands that alter text (see *VAX LISP/VMS User's Guide*). If you would like checkpointing to occur either more or less frequently, you can make this change by using SETF with the function CHECKPOINT-FREQUENCY.

```
(SETF (EDITOR:CHECKPOINT-FREQUENCY) 1000)
```

After you have executed this form, the Editor will checkpoint after every 1000 commands that alter text.

To disable checkpointing completely, you set the value to NIL:

```
(SETF (EDITOR:CHECKPOINT-FREQUENCY) NIL)
```

1.1.2 Changing the Number of Windows Displayed

The Editor normally displays up to two anchored windows at a time. If you call for a third anchored window (by selecting another buffer or editing another file or LISP object), the Editor removes a window from the screen to make room for the new window. (See *VAX LISP/VMS User's Guide*.)

If you would like the Editor to show up to three windows at a time, you change the value of an Editor variable called "Anchored Window Show Limit" from 2 to 3. If you want only one window shown at a time, you set the value to 1.

An Editor variable differs slightly from a LISP variable (see Section 1.3.4 below). You reference an Editor variable by means of a specifier (symbol or a string called a display name), and you access the variable's value with the function VARIABLE-VALUE. Using SETF, you can then change the value of the Editor variable.

The code that changes the number of anchored windows that the Editor can show is:

```
(SETF (EDITOR:VARIABLE-VALUE "Anchored Window Show Limit") 3)
```

After you have executed this form, the Editor will show up to three anchored windows at a time. If you call for a fourth anchored window, the Editor will remove a window from the screen to accommodate the new window.

1.1.3 Changing the Default Major Style

The Editor activates "EDT Emulation" as the major style in all the buffers that it creates for editing files and LISP objects. If you prefer the behavior and key bindings of an EMACS-based editor, you can make "EMACS" style the default major style instead. (See *VAX LISP/VMS User's Guide*.)

The default is stored as the value of the Editor variable "Default Major Style". The possible values are Editor objects called styles, which, like Editor variables, can be referenced by either their symbols or their display names.

The following form changes the Editor's default major style to "EMACS":

        (SETF (EDITOR:VARIABLE-VALUE "Default Major Style") "EMACS")

Any new buffers that the Editor creates after you have executed this form will have "EMACS" as their major style. Buffers that already exist are not affected.

1.1.4 Binding a Command to a Key Sequence

Many DIGITAL-provided commands are not bound to keys or key sequences and must therefore be invoked by name (see *VAX LISP/VMS User's Guide*). You might find it convenient to bind keys to the commands you use frequently, such as "Write Current Buffer".

To establish a key binding, you can use the function BIND-COMMAND and the display name of an Editor command. To specify the key or keys, you can use the normal LISP syntax for a character or a vector containing characters. (BIND-COMMAND is one of the few Editor-related symbols that is accessible in the "USER" package; you need not prefix it with the "EDITOR" package qualifier.)

        (BIND-COMMAND "Write Current Buffer" '#(#\^X #\^W))

The sequence CTRL/X CTRL/W will now execute "Write Current Buffer" in the Editor. To make sure that the key sequence you choose is not already bound to an Editor command, you can consult Appendix C to this manual, which lists the keys that are bound in the Editor as provided.

1.1.5 Defining a Command to Change Screen Width

In editing LISP code, you might occasionally want your terminal screen to display more than 80 columns so that lines do not truncate. One way to do this is to execute the command "Set Screen Width" after specifying a prefix argument (such as 132). If you adjust screen width frequently, you might prefer to have a command that you can execute in one step.*

To implement a new command, you use the macro DEFINE-COMMAND. A possible implementation for a command that widens the screen to 132 columns is:

_____

* On DIGITAL terminals without the Advanced Video Option, widening the screen reduces available screen height to 12 rows.

```
(EDITOR:DEFINE-COMMAND (WIDEN-SCREEN-COMMAND
                              :DISPLAY-NAME "Widen Screen")
                       (PREFIX)

    (SETF (EDITOR:VARIABLE-VALUE "Default Window Truncate Char")
          NIL)
    (SETF (EDITOR:SCREEN-WIDTH) (OR PREFIX 132)))
```

The function SCREEN-WIDTH returns the current width of the screen. Using SETF, you change the value to 132 or to a prefix value that you can supply interactively. (If no prefix value is supplied interactively, the Editor automatically passes a NIL value for this parameter and the OR form will then return the value 132.)

This example also sets the value of the Editor variable "Default Window Truncate Char" to NIL. This action dispenses with the character that normally appears on the screen to indicate line truncation. DEFINE-COMMAND creates a command named "Widen Screen" that executes these two SETF forms.

To have another command that sets the screen back to 80 columns and reestablishes > as the truncation character, you could write:

```
(EDITOR:DEFINE-COMMAND (SHRINK-SCREEN-COMMAND
                              :DISPLAY-NAME "Shrink Screen")
                       (PREFIX)

    (SETF (EDITOR:VARIABLE-VALUE "Default Window Truncate Char")
          #\>)
    (SETF (EDITOR:SCREEN-WIDTH) (OR PREFIX 80)))
```

The new commands created by these DEFINE-COMMAND forms can be invoked by name within the Editor. To make the commands accessible from the keyboard, you could bind them to key sequences:

```
(BIND-COMMAND "Widen Screen" '#(#\ESCAPE #\w))

(BIND-COMMAND "Shrink Screen" '#(#\ESCAPE #\s))
```

The new commands can now be invoked from the keyboard by means of ESCAPE w and ESCAPE s.

1.2 THE COMPONENTS OF THE EDITOR

This section introduces the various subsystems and utilities of the Editor. The purpose is to indicate the range of Editor behavior that can be programmed and to introduce the data types that each subsystem contains.

Section 1.3 discusses the nature of the Editor data types and the means of referencing them.

1.2.1 Text Operations

Text in the Editor is made up of the 256 characters in the ASCII 8-bit extended character set (the DEC Multinational Character Set). The Editor's text operations are the operations that insert, copy, and delete text and that indicate any given text position (for positioning the cursor, for instance).

The text operations subsystem contains the following specially defined data types, along with functions and macros that operate on them:

- Objects that contain text: BUFFERS, REGIONS, LINES

- Objects that indicate text positions: MARKS

- Objects that distinguish among characters for the purpose of searching through text: EDITOR ATTRIBUTES

The text operations subsystem also contains EDITOR VARIABLES and several LISP global variables.

Information on programming text operations appears in Chapter 4 and in the descriptions of the above data types in Part II.

1.2.2 Window and Display Operations

The Editor's window operations create, delete, and manipulate windows that open onto the contents of buffers. The display operations make windows (and thus buffer contents) visible on the screen or remove windows from the screen. Display operations also manage the allocation of the total screen area and the use of the information area at the bottom of the screen.

The window and display operations subsystem contains the following data types, along with the functions and macros that operate on them:

- Text-containing objects that can be displayed: BUFFERS

- Objects that translate text into displayable form: WINDOWS

The subsystem also contains EDITOR VARIABLES and LISP global variables, as well as functions that operate on the information area.

Information on programming window and display operations appears in Chapter 5 and in the descriptions of the above data types in Part II.

1.2.3 Binding Contexts

Contexts are separate programming environments within the Editor where bindings can take place. Certain types of objects may have different bindings simultaneously in different contexts:

- EDITOR VARIABLES

- EDITOR ATTRIBUTES

An Editor variable or an Editor attribute can reference more than one value if the variable or attribute is bound in more than one Editor context. The bindings of keyboard keys and pointer actions to Editor commands are also context-dependent.

Two Editor data types can serve as binding contexts:

- Any BUFFER

- Any STYLE

In addition, the Editor supports a global binding context.

To determine which of several bindings to use in a given situation, the Editor searches through the contexts in a predetermined order and uses the first binding it encounters. The search order is:

- The current buffer

- The styles active in the current buffer, beginning with the most recently activated minor style, if any, and ending with the major style, if any (see *VAX LISP/VMS User's Guide*)

- The global Editor context

When you reference a context-dependent object in LISP code, you can specify the appropriate context.

Editor contexts implement a form of scoping that is unlike either the dynamic or lexical scoping of COMMON LISP (see *COMMON LISP: The Language*). The binding context determines the scope of Editor variables, Editor attributes, keys, and pointer actions.

The extent of these context-dependent objects is indefinite (see *COMMON LISP: The Language*). That is, the objects have extent that begins when they are bound in a context and ends when they are unbound from that context. To "bind" an Editor variable or an Editor attribute is to establish it as usable in a certain context. You cannot assign values unless the variable or attribute is bound ("established") in one or more contexts -- buffer, style, or global.

The use of binding contexts is pervasive in Editor programming. Further detail and examples can be found in Chapter 3, Binding Commands, and in Chapter 6, Operations on Styles. See also the discussion of the context subsystem in Part II.

1.2.4 Other Subsystems and Utilities

The smaller subsystems of the Editor consist mainly of functions that enable you to control certain types of Editor behavior:

- Prompting the user for a value necessary for the execution of a command -- discussed in Chapter 2 and Part II

- Signaling errors in command execution and handling LISP errors -- discussed in Chapter 2 and Part II

- Checkpointing buffers to save their contents in the event of system failure -- discussed in Part II

In addition, the Editor has several low-level tools and utilities. The following items are all discussed in Part II:

- Input and output streams: LISP streams that permit normal COMMON LISP input and output operations to be performed within the Editor.

- String tables: specialized hash tables that store information indexed by a string (such as the display name of a command or other Editor object).

- Hooks: functions that are invoked automatically by certain Editor operations, such as activating a style or making a buffer or window.

- Rings: circular caches of values that are used, for instance, to store deleted text. Rings are used to implement the kill ring -- a facility like those in certain EMACS editors that stores deleted text.

1.3 REFERENCING EDITOR OBJECTS

The objects provided with the Editor include several new data types. The Editor also contains definitions of LISP functions, macros, and global variables. All these objects are LISP objects that can be referenced in any LISP code.

This section provides information on how to access these various kinds of objects. It introduces:

- Functions, macros, and variables

- Editor-specific data types

  - Named and unnamed objects

  - Context-independent and context-dependent objects

- The package location of Editor symbols

1.3.1 Functions, Macros, and LISP Variables

The Editor contains definitions of LISP functions, macros, and global variables. All the normal COMMON LISP rules concerning scope and extent apply to the identifiers of these objects.

1.3.2 Editor Objects

The specially defined Editor data types are:

- Editor attributes

- Buffers

- Commands

- Lines

- Marks

- Regions

- Rings

- String tables

- Styles

- Editor variables

- Windows

The methods of accessing Editor objects differ according to whether the object in question is:

- Named or unnamed

- Context-independent or context-dependent

These methods are outlined in the two sections that follow.

1.3.3 Named and Unnamed Editor Objects

Named objects are Editor objects that can have two special specifiers: a string called a display name and a symbol. The specifiers are associated with a named object at the time it is defined, and they serve as a means of accessing the object under certain circumstances.

It is important to recognize that the symbol specifier of a named Editor object cannot be treated as an ordinary LISP symbol. That is, the Editor object is not the symbol-value of the symbol. Editor object specifiers behave somewhat like the symbol and string specifiers of LISP packages. The function FIND-PACKAGE can take, for instance, the symbol 'USER or the string "USER" and return the package object; the symbol 'USER itself does not evaluate to the package object.

The reference list in Part III of this manual identifies both the display name and the symbol of all named objects provided with the Editor. Section 1.3.3.2 outlines the use of these specifiers in accessing named Editor objects.

The named object types are:

- Editor attribute

- Buffer

- Command

- Style

- Editor variable

The other Editor object types are unnamed. Unnamed Editor objects have no special specifiers.

- Line

- Mark

- Region

- Ring

- String table

- Window

1.3.3.1 Referencing Unnamed Objects - Any Editor object -- named or unnamed -- can be accessed in the usual LISP way: that is, by means of a form that evaluates to the object. Unnamed objects can be accessed only in this way.

For instance, the function CURRENT-WINDOW takes no arguments and returns the window that is current in the Editor. You can access the current window (an unnamed object) by writing:

    (EDITOR:CURRENT-WINDOW)

Similarly, you can access a string table by referencing the LISP global variable to which it is bound. For instance, evaluating

    EDITOR:*EDITOR-COMMAND-NAMES*

returns the string table that contains the names of the commands that are currently defined in the Editor.

1.3.3.2 Referencing Named Objects - Named Editor objects can be accessed in the same way as unnamed objects: by means of an expression that returns the object. For instance, the form

    (EDITOR:CURRENT-BUFFER)

returns the buffer (a named object) that is current in the Editor.

You can also reference named Editor objects by means of their specifiers (symbols or display names) in certain circumstances:

- Interactively, when the Editor prompts for the name of a command, buffer, or style, you supply the appropriate display name.

- In LISP code, when calling a function that takes a named Editor object specifier as an argument, you can supply any of three specifiers of the named object:

    - The display name
    - The symbol
    - Any form that evaluates to the object

In contrast, some functions take a named Editor object but *not* a specifier. When calling these functions, you *must* supply a form that evaluates to the object. The function descriptions in Part III of this manual distinguish between functions that can take specifiers (including objects) and functions that can only take objects.

For instance, the following functions are among those that can take specifier arguments: COMMAND-CATEGORIES, VARIABLE-VALUE, BUFFER-MAJOR-STYLE, FIND-STYLE, and BIND-ATTRIBUTE. The following examples show how you can call each of these functions from LISP code with the specifier of a named Editor object as the argument. In each case, you could use either the symbol or the display name of the named object; you could also, of course, use any form that evaluates to the object in question.

> Using a command specifier:
>
>   (EDITOR:COMMAND-CATEGORIES 'EDITOR:END-OF-LINE-COMMAND)
>   (EDITOR:COMMAND-CATEGORIES "End of Line")
>
> Using an Editor variable specifier:
>
>   (EDITOR:VARIABLE-VALUE 'EDITOR:TARGET-COLUMN)
>   (EDITOR:VARIABLE-VALUE "Target Column")
>
> Using a buffer specifier:
>
>   (EDITOR:BUFFER-MAJOR-STYLE 'EDITOR:EDITOR-HELP-BUFFER)
>   (EDITOR:BUFFER-MAJOR-STYLE "Help")
>
> Using a style specifier:
>
>   (EDITOR:FIND-STYLE 'EDITOR:EDT-EMULATION)
>   (EDITOR:FIND-STYLE "EDT Emulation")
>
> Using an Editor attribute specifier:
>
>   (EDITOR:BIND-ATTRIBUTE 'EDITOR:WORD-DELIMITER)
>   (EDITOR:BIND-ATTRIBUTE "Word Delimiter")

Because these functions evaluate their arguments, the argument can also be a form that evaluates to a specifier of a named Editor object. For instance, the following pair of forms has the same effect as the previous calls to BUFFER-MAJOR-STYLE.

>   (SETF b "Help")
>
>   (EDITOR:BUFFER-MAJOR-STYLE b)

1.3.3.3 **A Note On Efficiency** - The display names of named Editor objects are included for the convenience of the programmer and the Editor user. If you wish to maximize the efficiency of your program, however, you should be aware that accessing an object by using its display name is less efficient than using its symbol. Further, using either specifier is less efficient that using an expression that evaluates to the object.

For example, the following three forms are equivalent when the buffer named "Mybuffer.txt" is the current buffer. The forms are listed in order from the least to the most efficient:

```
(EDITOR:BUFFER-MAJOR-STYLE "Mybuffer.txt")

(EDITOR:BUFFER-MAJOR-STYLE 'MYBUFFER.TXT)

(EDITOR:BUFFER-MAJOR-STYLE (EDITOR:CURRENT-BUFFER))
```

The code examples in this manual frequently use display names for convenience and readability. When you reference named objects in your own code, however, you should consider the trade-off between convenience and efficiency in each instance.

1.3.4 Context-Independent and Context-Dependent Editor Objects

Editor objects are either context-independent or context-dependent. The context-dependent objects are actually specifiers that may be associated with different objects in different Editor contexts (the contexts are individual buffers, individual styles, and global).

Context-independent objects exist independently of Editor context. These objects are accessed according to the scoping rules defined in *COMMON LISP: The Language.*

1.3.4.1 **Referencing Context-Independent Objects** - All the unnamed Editor objects and most of the named objects (buffers, commands, and styles) are context-independent. A context-independent object, once created, exists within the Editor as a unique object, and the accessing functions appropriate to the data type locate and return that unique object.

For instance:

```
(EDITOR:FIND-BUFFER 'FACTORIAL)     ;Finds and returns the buffer
                                    ;object named FACTORIAL
```

```
(EDITOR:FIND-STYLE "VAX LISP")        ;Finds and returns the style
                                      ;object named "VAX LISP"

(EDITOR:NEXT-WINDOW)                   ;Finds and returns a unique
                                      ;window object (unnamed)
```

1.3.4.2 **Referencing Context-Dependent Objects** - The context-dependent
Editor objects are Editor attributes and Editor variables. Attributes
and variables are not unique objects. That is, a specifier can be
associated with different values (or, in the case of variables, also
with different functions) in different Editor contexts.

It is important to recognize that these multiple associations can
exist simultaneously; leaving an Editor context makes an association
temporarily inaccessible, but it does not destroy it.

The following functions are used to access a value or function
associated with a context-dependent object:

- VARIABLE-VALUE takes a variable specifier and an optional
  context and returns the value (if any) of that variable in
  that context.

- VARIABLE-FUNCTION takes a variable specifier and an optional
  context and returns the function definition (if any) of that
  variable in that context.

- CHARACTER-ATTRIBUTE takes an attribute specifier, a character,
  and an optional context and returns that character's value (if
  any) for that attribute in that context.

The functions FIND-ATTRIBUTE and FIND-VARIABLE are different from the
FIND-*object* functions for the context-independent data types. The
FIND-*object* functions for context-dependent objects take a specifier
(symbol or display name) and return the symbol of an attribute or
variable. They do not return a value or function object associated
with the specifier.

Chapters 3 and 6 contain code examples that illustrate the nature and
use of context-dependent objects. Further explanation also appears in
Part II.

1.3.5 The "EDITOR" Package

The symbols for the objects that are defined in the Editor are located
in the "EDITOR" package and are external in that package.

Most of these symbols are not exported to the "USER" package. (Only the functions ED and BIND-COMMAND are accessible in the "USER" package.) Any other symbols for DIGITAL-supplied Editor objects must be referenced in the "EDITOR" package when you use them in writing extensions.

There are three ways to reference symbols that are located in the "EDITOR" package:

● By using the package prefix

● By executing a USE-PACKAGE form

● By executing an IN-PACKAGE form

This section describes these three methods and the circumstances under which you would use them.

1.3.5.1 **The Package Prefix** - When you are working in the "USER" package, you can reference any symbol in the "EDITOR" package by prefixing it with the package qualifier EDITOR:. For instance, if you want to call VARIABLE-VALUE with the symbol of an Editor variable, you would prefix both symbols with EDITOR:.

     (EDITOR:VARIABLE-VALUE   'EDITOR:DEFAULT-MAJOR-STYLE)

This expression references two symbols in the "EDITOR" package, but it can be evaluated in the "USER" package.

Note that using the display name of a named Editor object instead of its symbol avoids the problem of package location, although at the expense of efficiency:

     (EDITOR:VARIABLE-VALUE   "Default Major Style")

1.3.5.2 **Using USE-PACKAGE** - To avoid the inconvenience of using qualified names, you can reference all the external symbols in the "EDITOR" package by executing either of the forms:

     (USE-PACKAGE "EDITOR") | (USE-PACKAGE 'EDITOR)

Note that the string argument ("EDITOR") must be upper case.

Executing either of these forms makes all Editor-related symbols accessible in the "USER" package for the remainder of your current LISP session.

However, before executing a USE-PACKAGE form, you should consider whether you will also be using symbols from other packages in the same LISP session. Because of possible name conflicts among packages, you should use qualified names in sessions where you will be referencing symbols in more than one package. In particular, there are several name conflicts in VAX LISP between the "EDITOR" package and the "UIS" package (see *VAX LISP/VMS Graphics Programming Guide*).

If you begin the file containing your completed Editor extensions with a USE-PACKAGE form, you should end the file with a call to UNUSE-PACKAGE. A call to USE-PACKAGE in your initialization file makes all symbols in that package accessible throughout every LISP session; these symbols may then interfere with symbols you want to use from other packages.

1.3.5.3 **Using IN-PACKAGE** - It is generally good programming practice to place your newly defined symbols in an appropriate package. You can place your completed Editor extensions in a specified package by heading the file that contains the extensions with a call to IN-PACKAGE. An IN-PACKAGE form makes the specified package current while your file is being loaded into LISP; it then returns you to the "USER" package for the remainder of your session.

You can place your completed Editor extensions in the "EDITOR" package by heading your file with either of the forms:

    (IN-PACKAGE "EDITOR") | (IN-PACKAGE 'EDITOR)

However, this use of the "EDITOR" package allows for possible name conflicts (overwriting) between user-defined extensions and present or future DIGITAL-supplied objects.

You can avoid overwriting by placing your extensions in a new, user-defined package. To do so, and to have the "EDITOR" package symbols accessible in the new package, you begin the file with the following forms:

    (IN-PACKAGE "EDITOR-EXTENSIONS")
    (USE-PACKAGE "EDITOR")
        .
        .
        .

These forms place your extensions in the package "EDITOR-EXTENSIONS", and make the symbols from the package "EDITOR" accessible in that package. They do not make the symbols from either of these packages accessible in the "USER" package.

CHAPTER 2

CREATING EDITOR COMMANDS

Commands are the means by which you control the VAX LISP Editor in an interactive session. It is by executing commands that you insert and revise text, display text or other information, activate a style, or bring about any other Editor operation. (See *VAX LISP/VMS User's Guide*.)

The primary way to customize the Editor is to alter its commands: replace existing commands or create entirely new ones. This chapter introduces the techniques of implementing Editor commands. The topics it covers are:

- Commands and their associated LISP functions

- Creating commands with DEFINE-COMMAND

- Including some special features in a new command

The techniques of binding commands to keyboard keys and pointer actions are covered in Chapter 3.

2.1 COMMANDS AND THEIR ASSOCIATED FUNCTIONS

A command is a named Editor object that is associated with a particular LISP function. For instance, the command "Forward Word" is associated with the function FORWARD-WORD-COMMAND, and the command "Execute Named Command" is associated with the function EXECUTE-NAMED-COMMAND-COMMAND. (The nature of named Editor objects is discussed in Chapter 1.)

Whenever you execute a command during an interactive Editor session, the Editor calls the associated function. Evaluating this function brings about the specified change in the Editor. For instance, when you execute the command "Forward Word" in the Editor, either by name or by means of the key sequence bound to it, the Editor invokes the function FORWARD-WORD-COMMAND. The result you see is that the cursor moves to the next word in the text.

2-1

CREATING EDITOR COMMANDS

To implement an Editor command, you create both a new LISP function and a named Editor command associated with it. Both these operations are performed by the macro DEFINE-COMMAND.

2.2 USING DEFINE-COMMAND

DEFINE-COMMAND is similar to DEFUN in that it creates a new LISP function from the specified argument list and forms. In addition, it creates a new Editor command with the specifiers (display name and symbol) that you supply. The new command definition is a side effect of a call to DEFINE-COMMAND; the return value is the associated function definition.

The format of DEFINE-COMMAND is also similar to that of DEFUN:

        DEFINE-COMMAND *name*
                    arglist
                    &OPTIONAL *command-documentation*
                    &BODY *forms*

An example follows of a DEFINE-COMMAND expression that implements a new Editor command named "My Next Screen". (This command differs slightly from the DIGITAL-supplied "Next Screen" command; the difference is clarified in Section 2.2.4 below.) The remainder of this section discusses the purpose and use of each of the parameters of DEFINE-COMMAND, using this expression as an example.

Recall that the symbols for DIGITAL-provided Editor objects must be referenced in the "EDITOR" package (see Chapter 1).

```
(DEFINE-COMMAND
    (MY-NEXT-SCREEN-COMMAND :DISPLAY-NAME "My Next Screen") ;name
    (PREFIX &OPTIONAL (WINDOW (CURRENT-WINDOW)))             ;arglist

    " Scrolls the current window down one screen. If        ;com-doc
    a positive integer prefix is supplied, it scrolls
    down by that many screens (up if prefix is negative)."

    " MY-NEXT-SCREEN-COMMAND prefix &OPTIONAL window         ;func-doc

    This function has an optional argument window which
    defaults to the current Editor window.  It scrolls
    the window down one screen if the prefix argument
    is NIL.  If a positive integer prefix is supplied,
    it scrolls down by that many screens (up if prefix
    is negative).  The modified window point is returned."


    (SCROLL-WINDOW WINDOW (* (OR PREFIX 1)                   ;forms
                        (1- (WINDOW-HEIGHT WINDOW))))))
```

2.2.1 Specifying the Names

A command can have two special specifiers: a display name, which is a string, and a symbol, which is identical to the symbol of the function defined in the same form.

The display name of a command is provided as a convenience for the interactive user. It serves, for instance, to invoke a command within the Editor. In LISP code, you can use either the display name or the symbol of a command, as well as the associated function itself, as an argument to a function that takes a command specifier argument. (See Chapter 1 for referencing named Editor objects, including commands.)

You specify the names of a new command and function in the *name* parameter to DEFINE-COMMAND. The *name* argument can be a symbol or a list of the form:

        (*symbol* :DISPLAY-NAME *string*)

The *symbol* argument serves the same purpose as the *name* argument for DEFUN -- it names the function being defined. In a call to DEFINE-COMMAND, *symbol* also becomes the symbol specifier of the new command. The *string* argument becomes the display name of the new command.

For example:

        (DEFINE-COMMAND
            (MY-NEXT-SCREEN-COMMAND :DISPLAY-NAME "My Next Screen") ;name
            .
            .
            .

This form creates a LISP function named MY-NEXT-SCREEN-COMMAND. It also creates an Editor command with the display name "My Next Screen" and the symbol MY-NEXT-SCREEN-COMMAND.

The display name can be any string you want to specify. For DIGITAL-supplied commands, the convention is that display names are identical to the associated symbols except for case and the omission of the hyphens and the final element -COMMAND. If you do not specify a display name, the default is the print name of the symbol. In this example, the default display name would be "MY-NEXT-SCREEN-COMMAND", a less convenient specifier than "My Next Screen".

2.2.2 Specifying the Argument List

When you execute a command within the Editor, the Editor always calls the associated function with exactly one argument. This is the prefix argument, which can be an integer or NIL. You can supply a prefix

value by previously executing the command "Supply Prefix Argument". If you execute a command without supplying a prefix value, the Editor passes NIL.

Because the Editor always passes one argument, the *argument list* of every DEFINE-COMMAND expression must have at least one parameter. By convention, the first parameter is designated as PREFIX. If you supply other parameters, they must be optional. (You can supply values for optional arguments only when calling the new function from LISP code, not when executing the new command in the Editor.)

The *argument list* for MY-NEXT-SCREEN-COMMAND specifies that this function can take two arguments: a prefix and a window.

```
(DEFINE-COMMAND
    (MY-NEXT-SCREEN-COMMAND :DISPLAY-NAME "My Next Screen") ;name
    (PREFIX &OPTIONAL (WINDOW (CURRENT-WINDOW)))              ;arglist
    .
    .
    .
```

The prefix argument usually means the number of times the action is to be repeated, although other meanings are possible. (In fact, the difference between "My Next Screen" and the DIGITAL-supplied "Next Screen" command is in the use they make of the prefix.) As with any function parameter, the meaning of the prefix argument to any particular command is specified in the body of that command's definition.

If you call the function MY-NEXT-SCREEN-COMMAND from LISP code, you can also specify the window that is to be operated upon. If you do not specify a window, then the function CURRENT-WINDOW will be evaluated and will return the current window. Since you cannot specify a window argument when you execute "My Next Screen" in the Editor, the Editor always applies the command's action to the current window.

2.2.3 Supplying Documentation Strings

DEFINE-COMMAND takes two optional documentation strings. The first is associated with the new command; the second, which is actually part of the body, is associated with the new function. If you supply only one documentation string, it becomes the *command-documentation*.

The *command-documentation* is normally used to describe the behavior of the Editor when you execute the new command. You can retrieve this documentation within the Editor by means of the "Describe" command, using the display name of the command in question. To retrieve documentation at top-level LISP, you can call either the DESCRIBE

function or the DOCUMENTATION function and pass it the symbol of the command. (If you use DOCUMENTATION, the *doc-type* is EDITOR-COMMAND.)

The function documentation is like the documentation string for DEFUN: it normally gives the function's format and return value and describes its behavior when called from LISP code. You can retrieve this documentation at top-level LISP by means of DESCRIBE or DOCUMENTATION, using the symbol of the function. (The *doc-type* is FUNCTION.)

The two kinds of documentation string -- one addressed to the person executing the command and the other to the person calling the function -- are illustrated below:

```
(DEFINE-COMMAND
    (MY-NEXT-SCREEN-COMMAND :DISPLAY-NAME "My Next Screen") ;name
    (PREFIX &OPTIONAL (WINDOW (CURRENT-WINDOW)))              ;arglist

    "  Scrolls the current window down one screen. If        ;com-doc
    a positive integer prefix is supplied, it scrolls
    down by that many screens (up if prefix is negative)."

    "  MY-NEXT-SCREEN-COMMAND prefix &OPTIONAL window         ;func-doc

    This function has an optional argument window which
    defaults to the current Editor window. It scrolls
    the window down one screen if the prefix argument
    is NIL. If a positive integer prefix is supplied,
    it scrolls down by that many screens (up if prefix
    is negative). The modified window point is returned."
    .
    .
    .
```

Note the placement of whitespace and newline characters in both the documentation strings in this example. As with DEFUN, you use these characters to affect the appearance of a string when it is displayed in response to "Describe", DESCRIBE, or DOCUMENTATION.

2.2.4 Specifying the Action

The *forms* that you supply to DEFINE-COMMAND are identical in purpose to the *forms* for DEFUN: they constitute the body of the LISP function that will be invoked when you execute the new command. The *forms* include the function documentation, if any, and they may include declarations.

Including the *forms* completes the definition of "My Next Screen":

```
(DEFINE-COMMAND
    (MY-NEXT-SCREEN-COMMAND :DISPLAY-NAME "My Next Screen") ;name
    (PREFIX &OPTIONAL (WINDOW (CURRENT-WINDOW)))              ;arglist
```

```
"   Scrolls the current window down one screen. If        ;com-doc
a positive integer prefix is supplied, it scrolls
down by that many screens (up if prefix is negative)."

"   MY-NEXT-SCREEN-COMMAND prefix &OPTIONAL window         ;func-doc

This function has an optional argument window which
defaults to the current Editor window.  It scrolls
the window down one screen if the prefix argument
is NIL.  If a positive integer prefix is supplied,
it scrolls down by that many screens (up if prefix
is negative).  The modified window point is returned."


(SCROLL-WINDOW WINDOW (* (OR PREFIX 1)                     ;forms
                          (1- (WINDOW-HEIGHT WINDOW)))))
```

This example uses the COMMON LISP functions *, OR, and 1- and the
Editor functions SCROLL-WINDOW and WINDOW-HEIGHT:

- SCROLL-WINDOW takes a window and a count. It scrolls the
  specified window by the number of rows indicated by the count
  and returns the window point. (The window point is an object
  that indicates the position of the screen cursor in the
  current window; see Chapter 5.)

- WINDOW-HEIGHT takes a window and returns the height (in rows)
  of that window.

The action of the function MY-NEXT-SCREEN-COMMAND is to scroll the
specified window (or default window) by a number of rows that equals
one less than the height of the window. That is, the last row of the
current text display becomes the first row of the new text display.
If you supply a prefix argument, the action is repeated that many
times. Because SCROLL-WINDOW moves the window point, the cursor will
appear within the new text display when you execute "My Next Screen".

To see the difference between "My Next Screen" and the
DIGITAL-supplied "Next Screen" command, compare the last form in the
above example with the last form in the definition of "Next Screen":

```
(SCROLL-WINDOW WINDOW
               (OR PREFIX (1- (WINDOW-HEIGHT WINDOW)))))
```

The prefix value in "My Next Screen" serves as a repetition count. In
"Next Screen" the prefix value is an alternative to the window height
in determining how many rows to scroll the window.

2.2.5 Modular Definition of Commands

You can define Editor commands of any degree of complexity. When defining a complex command, it is good programming practice to write the code in modules. You can, for instance, use DEFUN to create a new function and then use that function in a DEFINE-COMMAND expression.

For example:

```
(DEFUN PRINT-TIME (STREAM)

  "  PRINT-TIME stream

  Formats a record of the current date and time and writes
  that record to the specified stream."

  (MULTIPLE-VALUE-BIND
      (SECOND MINUTE HOUR DATE MONTH YEAR DAY-OF-WEEK)
      (GET-DECODED-TIME)
    (FORMAT STREAM
            "~D:~2,'0D:~2, '0D on  ~[Monday~;Tuesday~;Wednesday~
            ~;Thursday~;Friday~;~Saturday~;Sunday~],  ~
            ~D ~[~;January~;February~;March~;April~;May~;June~
            ~;July~;August~;September~;October~;November~
            ~;December~], ~D"
            HOUR MINUTE SECOND DAY-OF-WEEK DATE MONTH YEAR)))
```

This form creates the function PRINT-TIME, which writes the current date and time to a specified stream. To include this action in an Editor command, you might write:

```
(DEFINE-COMMAND
    (SHOW-TIME-COMMAND :DISPLAY-NAME "Show Time")
    (PREFIX)

  "  Displays the current time and date in the
  information area."

  "  SHOW-TIME-COMMAND prefix

  Displays the current time and date in the information
  area.  The prefix argument is ignored."

  (DECLARE (IGNORE PREFIX))

  (CLEAR-INFORMATION-AREA)
  (PRINT-TIME *INFORMATION-AREA-OUTPUT-STREAM*))
```

This form creates an Editor command named "Show Time". When you execute "Show Time", the Editor clears the information area of any previous text and then directs the record formatted by PRINT-TIME to the information area. Note the declaration that the prefix is

ignored, since the PREFIX parameter is not used in the body of the expression.

2.2.6 Commands and Context

Many commands are normally used within a single Editor context (buffer, style, or global), but commands are not context-dependent objects. That is, commands are not bound in Editor contexts, as keyboard keys and pointer actions are: any command can be invoked by name no matter which contexts are visible in the Editor. For instance, the command "EDT Change Case" is usually used in "EDT Emulation" style, but it could also be used in "EMACS" or "VAX LISP" styles.

However, the definition of a command may reference another object that is context-dependent: an Editor variable or an Editor attribute. (See Chapter 1 for a discussion of context-dependent objects.) If so, the command behaves differently when you execute it in contexts where the context-dependent object is bound differently or not bound.

An example is the command "EDT Move Word", which moves the cursor by one or more words. The body of this command begins with a test of whether the Editor variable "EDT Direction Mode" is set to :FORWARD. If so, it invokes FORWARD-WORD-COMMAND:

```
(IF (EQ (VARIABLE-VALUE "EDT Direction Mode") :FORWARD)
    (FORWARD-WORD-COMMAND PREFIX)
    .
    .
    .
```

If you execute this command outside of "EDT Emulation" style, it will not invoke FORWARD-WORD-COMMAND because "EDT Direction Mode" is unbound.

The behavior of FORWARD-WORD-COMMAND also differs in different contexts. This function references the Editor attribute "Word Delimiter", whose values are context-dependent. "Forward Word" behaves differently in "EDT Emulation", "EMACS", and "VAX LISP" styles because different characters are recognized as word delimiters in these styles.

NOTE

Unlike commands themselves, the key and pointer bindings of Editor commands are context-dependent (see Chapter 3). "EDT Change Case" can be invoked by name anywhere within the Editor, but, as provided, it is only in "EDT Emulation" style that this command can be invoked by means of keypad PF1 1.

2.3 SOME SPECIAL COMMAND FACILITIES

Most of the new commands that you implement are likely to pertain to text operations or to window and display management. Regardless of the command's primary purpose, however, you may also want to include in it such features as a prompt or a particular error response. You can also include the command in a command category, which facilitates certain kinds of testing that may take place during command processing.

This section introduces the following command subsystems/facilities:

● Errors

● Prompting

● Command categories

2.3.1 Errors

By using functions from the Editor's error subsystem, you can implement commands that take some action in response to errors in command processing. In addition, you can use the LISP variable *UNIVERSAL-ERROR-HANDLER* to modify the way the Editor handles LISP errors.

This section introduces the following error-related objects:

● The ATTENTION function

● The EDITOR-ERROR function

● The *UNIVERSAL-ERROR-HANDLER* variable

2.3.1.1 **Getting The User's Attention** - The ATTENTION function, the simplest of the error-related functions, can be included in the body of a command to gain the user's attention if the command's action is not performed. On DIGITAL VT100- and VT200-series terminals and the AI VAXstation, the action of ATTENTION is to ring the bell.

An example of the use of ATTENTION is:

```
(DEFINE-COMMAND (FORWARD-WORD-COMMAND
                    :DISPLAY-NAME "Forward Word")
                (PREFIX)

    "  Moves the buffer point forward one word. If a prefix
    argument is supplied, the point is moved forward that
    many words (backward if the prefix is negative)."
```

```
(UNLESS (WORD-OFFSET (CURRENT-BUFFER-POINT)
                     (OR PREFIX 1))
    (ATTENTION))
  (CURRENT-BUFFER-POINT))
```

The command "Forward Word" invokes the function WORD-OFFSET, which moves the current buffer point by one or more words. If this action cannot be performed -- if too few words remain in the buffer, for instance -- then the ATTENTION function is called to alert the user.

A command continues processing after evaluating ATTENTION. In this case, the next form (CURRENT-BUFFER-POINT) is evaluated to return the buffer point.

2.3.1.2 **Signaling An Error** - The most generally useful error-signaling function is EDITOR-ERROR. EDITOR-ERROR is typically used to indicate an invalid command operation, invalid or incomplete user input, or some other error that allows the Editor to continue operation after ceasing to process the currently executing command.

The EDITOR-ERROR function invokes ATTENTION to signal a problem in command processing. In addition, it can display an optional line of text in the information area to explain the nature of the problem. The arguments to EDITOR-ERROR are analogous to those for the LISP ERROR function. However, EDITOR-ERROR allows the user to remain in the Editor after it is called, rather than being placed in the Debugger.

Unlike ATTENTION, which allows the Editor to continue processing the command, EDITOR-ERROR terminates the processing of the current command. The Editor then awaits the next command.

The use of EDITOR-ERROR is illustrated in the DIGITAL-supplied command "EDT Special Insert". This command must be invoked with a prefix; it inserts as text the character whose code is the prefix value supplied.

```
(DEFINE-COMMAND (EDT-SPECIAL-INSERT-COMMAND
                  :DISPLAY-NAME "EDT Special Insert")
                (PREFIX)

  "  Takes the value supplied as the prefix argument
  and inserts the character whose ASCII code is that
  value at the current buffer point."

  (UNLESS PREFIX
    (EDITOR-ERROR "Character code not supplied"))
```

```
(UNLESS (AND (INTEGERP PREFIX)
             (<= 0 PREFIX 255))
        (EDITOR-ERROR "Invalid character code ~A" PREFIX))

(INSERT-CHARACTER (CURRENT-BUFFER-POINT)
                  (CODE-CHAR PREFIX)))
```

Two errors that can occur when you invoke this command are (1) no prefix value supplied, and (2) prefix value supplied that is not a valid ASCII (extended) character code. Before attempting to evaluate the INSERT-CHARACTER form, the command tests for each of these possible errors. If an error has occurred, the appropriate explanation string is displayed in the information area and the processing of this command stops.

A somewhat more complex error-signaling function is EDITOR-ERROR-WITH-HELP. This function resembles EDITOR-ERROR except that it takes an additional optional string argument. The additional string, which supplies further information about the error, is displayed if the user executes the command "Help on Editor Error" (see Part III).

2.3.1.3 **Error Handling** - When implementing a command, you can also modify the way the Editor handles LISP errors that occur during command processing.

As provided, the Editor responds to a LISP error by clearing the screen, displaying the error message, and asking if you want to save modified buffers. It then gives you the choice of entering the Debugger or returning to top-level LISP.

You can alter this behavior by defining a new error-handling function and binding it to the variable *UNIVERSAL-ERROR-HANDLER* (see *VAX LISP/VMS User's Guide*). You can then reference this variable in a command definition to invoke the new error-handling function.

For instance, suppose that you want to alter the command "Insert File" to respond in a particular way when your response to the prompt is not a valid file name. To achieve this, you define a new error-handling function, and then write a file-insertion command that invokes this function if it receives an invalid file name.

The following example shows a skeletal version of a function to be invoked when the Editor cannot insert a file:

```
(DEFUN INSERT-FILE-ERROR-HANDLER (&REST ARGS)

  (WITH-OUTPUT-TO-MARK (*ERROR-OUTPUT*
                         (BUFFER-POINT
                           (FIND-BUFFER "Error Record")))
    (APPLY #'PRINT-SIGNALED-ERROR ARGS))
  (EDITOR-ERROR "Error reading file..."))
```

CREATING EDITOR COMMANDS

This function creates an output stream by means of the macro WITH-OUTPUT-TO-MARK and binds it to the variable *ERROR-OUTPUT*. This stream is directed to a user-defined buffer named "Erro Record". The VAX LISP function PRINT-SIGNALED-ERROR formats an error message from the supplied arguments and writes that message to *ERROR-OUTPUT*. The message text is thus inserted at the buffer point of the "Error Record" buffer. Once the formatting is done, INSERT-FILE-ERROR-HANDLER calls EDITOR-ERROR to print a brief explanation and return to the Editor command loop.

To write a new command that invokes INSERT-FILE-ERROR-HANDLER instead of the Editor's default error handler when a LISP error occurs, you bind this new function to *UNIVERSAL-ERROR-HANDLER* in the definition of the command. For instance, the relevant portion of a file-inserting command might look like:

```
(DEFINE-COMMAND (MY-INSERT-FILE-COMMAND ...) (PREFIX)
   .
   .
   .
   (LET ((*UNIVERSAL-ERROR-HANDLER* #'INSERT-FILE-ERROR-HANDLER))
     (INSERT-FILE-AT-MARK (PATHNAME...)
                            (CURRENT-BUFFER-POINT)))
   .
   .
   .
```

This command invokes the Editor function INSERT-FILE-AT-MARK to insert a specified file at the current buffer point. This action occurs within the scope of a LET form that binds *UNIVERSAL-ERROR-HANDLER* to INSERT-FILE-ERROR-HANDLER. If any LISP error occurs during the file-insertion operation, the error system calls INSERT-FILE-ERROR-HANDLER instead of the default error handler.

2.3.2 Prompting

The Editor's prompting subsystem enables you to write commands that prompt for any additional user input needed for their execution. For example, when you invoke the command "Select Buffer", the Editor prompts for the name of the buffer that is to become current.

Commands prompt by invoking one of the following functions:

● SIMPLE-PROMPT-FOR-INPUT

● PROMPT-FOR-INPUT

Both functions display a prompt in the prompting window, which is a window onto the DIGITAL-supplied buffer "General Prompting". User interaction, including editing the response to the prompt, occurs in

this buffer. The more versatile of the two functions, PROMPT-FOR-INPUT, also enables you to include some additional prompt-related behavior, such as input completion, alternatives, and help.

2.3.2.1 **Simple Prompting** - SIMPLE-PROMPT-FOR-INPUT is less versatile than PROMPT-FOR-INPUT, but it is generally more straightforward to use. SIMPLE-PROMPT-FOR-INPUT prompts for input and returns the user's input as a string. Its format is:

    SIMPLE-PROMPT-FOR-INPUT &OPTIONAL *prompt default*

The *prompt* argument is a string to be displayed as the prompt; the *default* argument is a string to be returned by SIMPLE-PROMPT-FOR-INPUT if the user presses RETURN without typing any input. The default value for both arguments is a null string.

An example of a new command that invokes SIMPLE-PROMPT-FOR-INPUT is "Visit File". This command is similar to the DIGITAL-supplied "View File" command, except that it allows the user to edit the specified file.

    (DEFINE-COMMAND (VISIT-FILE-COMMAND
                    :DISPLAY-NAME "Visit File")
                (PREFIX &OPTIONAL (FILE-NAME NIL))

      " Prompts for a file name and then edits the specified
      file. If the specified file is associated with a buffer,
      it simply switches to that buffer; otherwise a new buffer
      is created."

      (DECLARE (IGNORE PREFIX))

      (UNLESS FILE-NAME
        (SETF FILE-NAME
              (SIMPLE-PROMPT-FOR-INPUT "Enter file name: ")))

      (EDIT-FILE-COMMAND NIL FILE-NAME))

The function VISIT-FILE-COMMAND, when called from LISP code, takes an optional *file-name* argument that can be a pathname or a string. When you invoke "Visit File" in the Editor, however, you cannot supply this argument. To obtain the value, the command displays the specified prompt, "Enter file name: ". SIMPLE-PROMPT-FOR-INPUT returns your response to the prompt as a simple string. The string is bound to the variable FILE-NAME and then passed to the function EDIT-FILE-COMMAND.

Even though SIMPLE-PROMPT-FOR-INPUT always returns a simple string, you can use this function when the argument needed is some other data type. In such a case, the command must coerce the user's input into the appropriate data type.

For example, the command "EDT Special Insert", shown above in Section 2.3.1.2, takes an integer argument. If you fail to execute "Supply Prefix Argument" beforehand, "EDT Special Insert" displays an error message and stops processing. You could rewrite this command to prompt for the needed value instead of signaling an error. The code for such a command might be:

```
(DEFINE-COMMAND (MY-SPECIAL-INSERT-COMMAND
                  :DISPLAY-NAME "My Special Insert")
               (PREFIX)

  " Takes the prefix value and inserts the character whose
  ASCII code is that value at the current buffer point.  If
  no prefix is supplied, it prompts for a value."

  (UNLESS PREFIX
    (SETF PREFIX
          (READ-FROM-STRING
             (SIMPLE-PROMPT-FOR-INPUT " Enter ASCII code:  "))))

  (IF (AND (INTEGERP PREFIX) (<= 0 PREFIX 255))
      (INSERT-CHARACTER (CURRENT-BUFFER-POINT)
                        (CODE-CHAR PREFIX))
      (EDITOR-ERROR "Invalid character code:  ~A" PREFIX)))
```

The COMMON LISP function READ-FROM-STRING is used here to coerce the user's input string to an integer. This integer is then bound to PREFIX and passed to CODE-CHAR. Only if the input supplied does not convert into a valid ASCII extended character code will this command display an error message.

2.3.2.2 **General Prompting** - General prompting differs from simple prompting in that (1) the prompting function can return any data type (not only a string), and (2) you can include a greater range of prompt-related behavior by specifying a number of keyword arguments. The function you use for general prompting is PROMPT-FOR-INPUT.

What follows is a brief introduction to the use of PROMPT-FOR-INPUT. Part III of this manual contains a fuller description of this function and of its keyword arguments.

The basic format of PROMPT-FOR-INPUT is:

PROMPT-FOR-INPUT *validation*

CREATING EDITOR COMMANDS

The one required argument to PROMPT-FOR-INPUT is a *validation* function. This function takes the user's input string and returns a value that will be returned by PROMPT-FOR-INPUT. If the *validation* function returns NIL, PROMPT-FOR-INPUT signals an error and awaits further input.

For instance:

    (PROMPT-FOR-INPUT #'FIND-BUFFER)

This form prompts the user with a default prompting message and passes the user's input string to the function FIND-BUFFER. If the input string is not a valid buffer name, FIND-BUFFER returns NIL. PROMPT-FOR-INPUT then displays a default error message and waits for a valid buffer name before command processing continues.

PROMPT-FOR-INPUT can also make available the facilities for input completion and alternatives to assist the user. By providing string tables as arguments to the keywords :COMPLETION and :ALTERNATIVES, you specify that those string tables are to be searched if the user requests assistance from either of these facilities. In the above example, the appropriate string table is bound to the variable *EDITOR-BUFFER-NAMES*, and the form would look like:

    (PROMPT-FOR-INPUT #'FIND-BUFFER
                      :COMPLETION *EDITOR-BUFFER-NAMES*
                      :ALTERNATIVES *EDITOR-BUFFER-NAMES*)

(These two keyword arguments can also be values other than string tables; see the description of PROMPT-FOR-INPUT in Part III of this manual.)

Other keywords allow you to specify, for instance:

- The prompt to be displayed

- An error message to be displayed if the validation function returns NIL

- Help text to be displayed if the user requests help

- Whether user input is required

- A default value to be returned if you specify that user input is not required

These and other arguments to PROMPT-FOR-INPUT are described in full in Part III.

What follows is a comparatively simple example of this function, using only a few of its possible keyword arguments. The new command "My Insert Buffer" calls PROMPT-FOR-INPUT to prompt for a buffer name.

The command then inserts the text of that buffer into the current
buffer. Its code is:

```
(DEFINE-COMMAND (MY-INSERT-BUFFER-COMMAND
                  :DISPLAY-NAME "My Insert Buffer")
                (PREFIX)

  (DECLARE (IGNORE PREFIX)

  (INSERT-REGION
      (CURRENT-BUFFER-POINT)
      (BUFFER-REGION
          (PROMPT-FOR-INPUT #'FIND-BUFFER

              :PROMPT "Enter Buffer Name:   "
              :REQUIRED T
              :COMPLETION *EDITOR-BUFFER-NAMES*
              :ALTERNATIVES *EDITOR-BUFFER-NAMES*))))
```

This command displays the string argument to :PROMPT in the prompting
window. Because the value of :REQUIRED is T, the user must enter a
string for the action to continue (no default value can be returned by
PROMPT-FOR-INPUT). As in the example above, the string table
arguments to :COMPLETION and :ALTERNATIVES make available to the user
the names of all existing buffers.

The user's input string is passed to the validation function,
FIND-BUFFER, which returns a buffer object if the input is a valid
buffer name. The buffer object returned by FIND-BUFFER is passed to
BUFFER-REGION, which returns the text-containing region of that
buffer. The region is passed to INSERT-REGION, which inserts it at
the buffer point of the current buffer. (The region-manipulating
functions and other text operations objects are described in Chapter 4
of this manual.)

2.3.3 Command Categories

A command category indicates some property of a command that another
command may need to test for. The test is performed by checking
whether the command is a member of a specified category.

For example, the command "EMACS Forward Search" checks to see if the
last command executed was in the category :EMACS-SEARCH. If so, it
means that that command was also a search command and that the user
has already entered a search string. "EMACS Forward Search" will
therefore use the previous string rather than prompt again for one.
If the last command executed was not in the :EMACS-SEARCH category,
then "EMACS Forward Search" prompts for a search string.

CREATING EDITOR COMMANDS

The categories provided with the Editor are :GENERAL-PROMPTING, :LINE-MOTION, :MOVE-TO-POINTER, :EMACS-SEARCH, :EMACS-PREFIX, and :KILL-RING. Categories can also be user-defined.

You can place a command in one or more categories by including the keyword :CATEGORY and a symbol or list of symbols as part of the *name* argument of a DEFINE-COMMAND form. You can use existing categories, or you can define new categories simply by specifying their symbols. For example:

```
(DEFINE-COMMAND (EMACS-BACKWARD-SEARCH-COMMAND
                :DISPLAY-NAME "EMACS Backward Search"
                :CATEGORY :EMACS-SEARCH)
   ...)
```

Or,

```
(DEFINE-COMMAND (MY-NEW-COMMAND-COMMAND
                :DISPLAY-NAME "My New Command"
                :CATEGORY (:LINE-MOTION  'MY-NEW-CATEGORY)
   ...)
```

To check whether a given command is included in a specified category, you call the function COMMAND-CATEGORIES. This function takes a command specifier and returns a list of the categories that include that command (or NIL if none is found). The variable *PREVIOUS-COMMAND-FUNCTION* is bound to the function associated with the last command executed; this variable is a command specifier acceptable to COMMAND-CATEGORIES.

For instance, the following form tests whether the previous command executed was in the category :EMACS-SEARCH.

```
(IF (MEMBER :EMACS-SEARCH
            (COMMAND-CATEGORIES *PREVIOUS-COMMAND-FUNCTION*)
            :TEST #'EQ)
   .
   .
   .
```

What follows is a full command definition that illustrates both (1) placing a command in a category, and (2) testing the previously executed command for membership in that category. The example, "My EMACS Forward Search", is a simplified version of the DIGITAL-provided command "EMACS Forward Search".

```
(DEFINE-COMMAND (MY-EMACS-FORWARD-SEARCH-COMMAND
                :DISPLAY-NAME "My EMACS Forward Search"
                :CATEGORY :EMACS-SEARCH)
                (PREFIX)
```

```
"   Searches forward once or the number of times specified
by the prefix argument.  Prompts for a search string only
if the previous command was not a searching command."

(IF (MEMBER :EMACS-SEARCH
            (COMMAND-CATEGORIES *PREVIOUS-COMMAND-FUNCTION*)
            :TEST #'EQ)

     (FORWARD-SEARCH-COMMAND
         PREFIX
         (VARIABLE-VALUE "Last Search String"))

     (FORWARD-SEARCH-COMMAND PREFIX)))
```

The Editor sets the user's response to a search-command prompt to the
value of the Editor variable "Last Search String". "My EMACS Forward
Search" calls FORWARD-SEARCH-COMMAND, but only after determining
whether the previous command executed in the Editor was also in the
category :EMACS-SEARCH.

- If so, "My EMACS Forward Search" calls FORWARD-SEARCH-COMMAND
  with two arguments: the prefix and a string that is the
  current value of "Last Search String".

- If not, "My EMACS Forward Search" calls FORWARD-SEARCH-COMMAND
  with only a prefix argument, thus requiring FORWARD-SEARCH-
  COMMAND to prompt for the needed string.

CHAPTER 3

BINDING COMMANDS TO KEYS AND POINTER ACTIONS

The most common way to extend the VAX LISP Editor is to bind Editor commands to keys and key sequences. You can then use the bound keys or sequences to invoke the commands within the Editor.

Many of the commands provided with the Editor are bound. The bindings may be to graphic or control characters, keyboard escape sequences, function or keypad keys, or to some combination of these keys. Commands can also be bound to actions of a pointing device provided with the AI VAXstation. All DIGITAL-supplied bindings are listed in Appendixes B and C of this manual, arranged both by command name and by key or key sequence.

A key binding or pointer-action binding exists within an Editor context, that is, within a particular style or buffer or in the global Editor context. The key or pointer action will invoke the command only when the appropriate context is active in the Editor. If more than one context is active at a time and if a key or pointer action is bound to different commands in these contexts, only one binding will be visible. (See VAX LISP/VMS User's Guide for a discussion of context and shadowing as they appear to the interactive user.)

You can change the DIGITAL-supplied bindings and bind commands that are not currently bound.

- To bind a key or key sequence to an Editor command, you call the function BIND-COMMAND from LISP code. To delete a key binding, call UNBIND-COMMAND.

- To bind a pointer action to an Editor command, you call the function BIND-POINTER-COMMAND from LISP code. To delete a pointer-action binding, call UNBIND-POINTER-COMMAND.

This chapter introduces the techniques of binding commands with these two functions. The topics covered are:

- Using BIND-COMMAND

  - The command to be bound
  - The key or key sequence to be bound
  - The binding context

- Using BIND-POINTER-COMMAND

  - Specifying a pointer action
  - Specifying a button state
  - Getting the state of the pointer

NOTE

The Editor's cancel character (initially, CTRL/C) is not established with BIND-COMMAND and is not context-dependent. This global association cannot be shadowed by Editor command bindings to the same character. To change the Editor's cancel character, you use SETF with the function CANCEL-CHARACTER (see Part III for a description of this function).

3.1 USING BIND-COMMAND

BIND-COMMAND takes a command specifier, a key or key sequence, and an optional context specifier. Its format is:

    BIND-COMMAND command key-sequence &OPTIONAL context

BIND-COMMAND binds the key-sequence to the command in the specified (or default) context. For example:

    (BIND-COMMAND "View File" #\^V)

This form binds the key CTRL/V to the DIGITAL-supplied command "View File", which has no binding in the Editor as provided. Since no context argument is specified, the binding is global by default.

Note that BIND-COMMAND is one of the few Editor-related symbols that are accessible in the "USER" package. If you include any other Editor-related symbols in a BIND-COMMAND form, you must reference them in the "EDITOR" package (see Chapter 1).

The sections that follow discuss each of the parameters of BIND-COMMAND in turn:

- The command to be bound

- The key or sequence to be bound

- The binding context

3.2 THE COMMAND TO BE BOUND

You can specify any Editor command as an argument to BIND-COMMAND, including:

- New user-defined commands

- DIGITAL-defined commands that are not bound

- Any command that is currently bound to another key or sequence.

You reference the command to be bound by means of any of the three kinds of command specifier (see Chapter 1):

- The command's display name

- The command's symbol

- A form that evaluates to the function associated with the command

For instance, the following three forms are equivalent. All three bind the command "View File" to CTRL/V in the global context. (The first and third of these forms are equal in efficiency; the middle form, which uses the symbol specifier, is very slightly faster.)

        (BIND-COMMAND "View File" #\^V)

        (BIND-COMMAND 'VIEW-FILE-COMMAND #\^V)

        (BIND-COMMAND (FIND-COMMAND "View File") #\^V)

You change an existing binding to a command in the same way that you establish a new binding. You may prefer to delete the old binding (using the function UNBIND-COMMAND) before rebinding a command, but this is not required. For instance, if you were to bind the command "Pause Editor" to CTRL/A, then both CTRL/A and the original binding (CTRL/X CTRL/Z) would invoke "Pause Editor".

However, if you modify a DIGITAL-supplied command, any key bindings to the original command continue to invoke the function associated with original command rather than the function associated with the new command. For instance, if you were to implement your own version of a "Next Window" command, the sequence CTRL/X CTRL/N would continue to invoke the DIGITAL-supplied function NEXT-WINDOW-COMMAND. To have the sequence CTRL/X CTRL/N invoke the function associated with your new "Next Window" command, you would need to rebind that key sequence to the new command by means of a subsequent call to BIND-COMMAND.

3.3 THE KEY OR KEY SEQUENCE TO BE BOUND

Commands are actually bound not to keys but to the characters generated by those keys. Most keyboard keys generate single characters; function keys and keypad keys generate sequences of characters.

You can bind a command to any character in the 8-bit extended ASCII character set (the DEC Multinational Character Set), with the few exceptions noted below. You can also bind a command to any valid LISP sequence of these characters. LISP sequences include lists and vectors containing characters, as well as strings.

The remainder of this section discusses the *key-sequence* argument to BIND-COMMAND:

- How to choose a key or sequence to bind

- How to specify character keys, function and keypad keys, and combinations of these keys

3.3.1 Choosing a Key or Sequence

In choosing a character key or key sequence to bind to a command, there are several considerations to keep in mind:

- You cannot bind the characters CTRL/S and CTRL/Q, which lock and unlock your terminal. This is a limitation of the operating system.

- You should not bind the current cancel character, which is initially CTRL/C.

- It is generally not good practice to bind graphic characters or sequences that begin with graphic characters. Every graphic character key is bound to the command "Self Insert", which inserts that character as text. Rebinding a character will supersede the "Self Insert" binding and leave you unable to insert that character as text except by quoting it.

As the last item suggests, the operation of BIND-COMMAND destructively modifies any previous binding of a key or sequence to a command (in the same context). For instance, if you were to rebind the sequence CTRL/X CTRL/Z to a new command (assuming the global context), then that sequence will no longer invoke "Pause Editor". In choosing keys to bind, take care not to modify any previous bindings that you wish to keep.

Similarly, you will lose bindings if you bind a key or sequence that begins another bound sequence. For instance, if you were to bind CTRL/X to a command, then the DIGITAL-supplied bindings (in the same context) that begin with CTRL/X (such as CTRL/X CTRL/Z for "Pause Editor") will be inaccessible.

3.3.2 Specifying a Character Key or Sequence

A character key can be specified with the usual LISP character syntax. For control characters and other nongraphic characters, you use the printed representation. For example, to bind the graphic character A, you write:

    (BIND-COMMAND "Self Insert" #\A)

To bind the nongraphic character CTRL/A, you write:

    (BIND-COMMAND "Transpose Previous Characters" #\^A)

To bind a sequence of characters, you use the LISP syntax for the LISP sequence you intend to use: vector, string, or list. A character sequence can include a graphic character without interfering with the "Self Insert" binding as long as the graphic character does not begin the sequence. The following two examples show vectors that combine graphic and nongraphic characters:

    (BIND-COMMAND "Name of Command" '#(#\^X #\w))

    (BIND-COMMAND "Name of Command" '#(#\ESCAPE #\a))

The first form binds the specified command to the sequence CTRL/X w; the second binds it to the sequence ESCAPE a. Note that case does not matter in specifying the printed representations of nongraphic characters (such as CTRL/X and ESCAPE). Case does matter, however, in specifying graphic characters (such as w and a).

3.3.3 Specifying a Function Key, Keypad Key, or Sequence

There is no essential difference between binding a command to a character sequence and binding it to a function key or keypad key (or sequence), since these keys generate sequences of characters.

Appendix D to this manual identifies the character sequences that are
generated by the function keys and keypad keys on DIGITAL VT100 and
LK-201 keyboards. You can specify these sequences as vectors (or
other LISP sequences) in BIND-COMMAND forms, as shown in the previous
section.

For instance, the GOLD key (keypad PF1) generates the character
sequence <ESCAPE>OP and keypad 1 generates <ESCAPE>Oq. The form that
binds the sequence keypad PF1 1 to the command "EDT Change Case" is:

```
(BIND-COMMAND "EDT Change Case"
              '#(#\ESCAPE #\O #\P #\ESCAPE #\O #\q))
```

Note that this binding takes place in the Editor's global context,
rather than in "EDT Emulation" style. See Section 3.4 for the means
of specifying a context argument to BIND-COMMAND.

You can also combine character keys and function or keypad keys in a
LISP sequence and pass that sequence to BIND-COMMAND. For instance,
the following form binds a vector that contains the characters
generated by keypad PF1 and the character h.

```
(BIND-COMMAND "Name of Command" '#(#\ESCAPE #\O #\P #\h))
```

The command will now be invoked by pressing the key sequence PF1 h.

3.4 THE BINDING CONTEXT

BIND-COMMAND binds a key or sequence to a command within a particular
Editor context. The key or sequence will invoke the specified command
only when that context is active in the Editor. For instance, if you
try to use "EDT Emulation" keypad bindings when only "EMACS" style is
active, the Editor will consider the keys unbound.

The binding context can be:

- Global, the default context, which means that the key binding
  exists universally within the Editor

- A style, which means that the key will invoke the specified
  command only when that style is active is the current buffer

- A buffer, which means that the key will invoke the specified
  command only when that buffer is current in the Editor

Since more than one context is often active in the Editor at any given
time -- global, a major style, one or more minor styles, and a buffer,
for instance -- some command bindings can be shadowed by other
bindings to the same keys in different contexts. The Editor searches
through the active contexts in a predetermined order to identify the
correct command for a key sequence.

This section describes the means of specifying a context in LISP code, as well as the Editor's search hierarchy for locating the correct binding when you use a key sequence to invoke a command.

3.4.1 Specifying the Binding Context

The context argument to BIND-COMMAND is specified with either:

- The keyword :GLOBAL

- A list beginning with the keyword :STYLE followed by a style specifier

- A list beginning with the keyword :BUFFER followed by a buffer specifier

BIND-COMMAND can take only one context argument at a time. To bind a command in more than one style or other context, you need to write a BIND-COMMAND form for each context.

3.4.1.1 Global - The global context is used for very basic Editor commands that enable you to function in the Editor even with no style active. Examples are the commands bound to the arrow keys, the RETURN key, and the DELETE key, as well as "Self Insert", "Pause Editor", and "Execute Named Command".

Since :GLOBAL is the default context argument for BIND-COMMAND, the following two examples are equivalent:

    (BIND-COMMAND "Pause Editor" '#(#\^X #\^Z) :GLOBAL)

    (BIND-COMMAND "Pause Editor" '#(#\^X #\^Z))

The two forms are also equal in efficiency.

3.4.1.2 Style - Styles are the most commonly used binding contexts. Your major style would usually include bindings to all the commands you commonly invoke for general editing. Minor styles can be seen as smaller sets of special-purpose bindings, such as those you use only for editing the syntax of a particular language.

A style argument to BIND-COMMAND is specified as a list beginning with the keyword :STYLE followed by a style specifier. For example:

```
'(:STYLE "EDT Emulation")

'(:STYLE EDT-EMULATION)

(LIST :STYLE (VARIABLE-VALUE "Default Major Style"))

`(:STYLE ,(VARIABLE-VALUE "Default Major Style"))
```

The Editor variable "Default Major Style" is set to a particular style object (initially, "EDT Emulation").

3.4.1.3 **Buffer** - Some commands may be used only in the context of a certain buffer, and it may be convenient to have their key bindings local to that buffer. For instance, the DIGITAL-supplied buffer "General Prompting" contains buffer-local bindings of commands that pertain to interactive user input, such as "Prompt Complete String" and "Prompt Help".

A buffer argument to BIND-COMMAND is specified as a list beginning with the keyword :BUFFER followed by a buffer specifier. For example:

```
'(:BUFFER "General Prompting")

'(:BUFFER EDITOR-PROMPTING-BUFFER)

(LIST :BUFFER (CURRENT-BUFFER))
```

The function CURRENT-BUFFER returns the buffer that is current in the Editor.

3.4.2 Search Order and Shadowing

To locate the correct command binding for a key sequence, the Editor searches through all the active contexts in the following order:

1. Current buffer

2. Minor styles active in that buffer beginning with the most recently activated

3. Major style of that buffer

4. Global context

The Editor will use the first command binding that it encounters in this search; any other bindings will be shadowed.

For instance, if you have "EMACS" style active in the current buffer, as either major or minor style, then you cannot use the global binding of CTRL/Z to invoke "Execute Named Command". Because style precedes global in the search order, the "EMACS" binding of CTRL/Z to "Scroll Window Down" shadows the global binding.

For keys that do not have multiple bindings in the active contexts, no shadowing occurs. For instance, the sequence CTRL/X CTRL/N invokes its global binding, "Next Window", even when you have "EDT Emulation" active as major style and both "EMACS" and "VAX LISP" as minor styles. None of these styles has a conflicting binding for that sequence.

Because of the small number of conflicting bindings involved, it is feasible to use all three DIGITAL-provided styles -- "EDT Emulation", "EMACS", and "VAX LISP" -- at once. If only "EDT Emulation" and "VAX LISP" are active, then most global bindings are visible as well. "EMACS", however, shadows a greater number of global bindings.

3.5 USING BIND-POINTER-COMMAND

By calling BIND-POINTER-COMMAND, you can bind various actions of a mouse or other pointing device to Editor commands. When the pointer cursor is in the current Editor window, the Editor will respond to pointer actions by invoking the bound commands. You cannot program the Editor to respond to pointer actions that occur when the pointer cursor is outside the current Editor window.

BIND-POINTER-COMMAND must be referenced in the "EDITOR" package. Its format is:

        BIND-POINTER-COMMAND command pointer-action &KEY :CONTEXT
                                                         :BUTTON-STATE

The command argument is a specifier of the command to be bound. Any valid command argument for BIND-COMMAND can also be used with BIND-POINTER-COMMAND (see Section 3.2 above).

The possible values for the :CONTEXT keyword are identical to those for the context parameter to BIND-COMMAND (see Section 3.4 above). The default binding context is :GLOBAL.

For instance, to invoke a command by means of a specified pointer action in "VAX LISP" style, you would write:

        (BIND-POINTER-COMMAND "Describe Word at Pointer"
                        pointer-action
                        :CONTEXT '(:STYLE "VAX LISP"))

The remainder of this section discusses the *pointer-action* parameter and the :BUTTON-STATE keyword. This section also explains the procedure for storing and retrieving the state of the pointing device at a given point in time.

3.5.1 Specifying a Pointer Action

The pointer actions you can use to invoke Editor commands are:

- A movement of the pointer cursor

- A transition (depressing or releasing) of a pointer button

3.5.1.1 Pointer Cursor Movement - A pointer movement in the Editor is defined as a movement across at least one character in any direction. Small movements of the pointer cursor (within a character) are not significant.

You specify movement of the pointer cursor by supplying the keyword :MOVEMENT as the *pointer-action* argument. For instance:

    (BIND-POINTER-COMMAND "Name of Command" :MOVEMENT)

If you want to have the command invoked by a movement only when one or more buttons is depressed, you supply a value for :BUTTON-STATE. See Section 3.1.2 below.

3.5.1.2 Pointer Button Transitions - The buttons on a supported pointing device are indicated by the symbols for button constants. The symbols are in the package "UIS", and they take the form POINTER-BUTTON-n, beginning with POINTER-BUTTON-1 for the leftmost button. (See *VAX LISP/VMS Graphics Programming Guide* for further information on button constants.)

To specify a downward transition of a particular button, you simply supply the appropriate button constant as the *pointer-action* argument. For instance, to bind a command to a downward transition of the middle button on a three-button mouse, you would write:

    (EDITOR:BIND-POINTER-COMMAND "Name of Command"
                         UIS:POINTER-BUTTON-2)

Note that this form uses symbols from both the "EDITOR" package and the "UIS" package.

To specify an upward transition, you supply a list of one element that is the appropriate button constant. For instance, to bind a command to an upward transition of the middle button on a three-button mouse, you would write:

```
(EDITOR:BIND-POINTER-COMMAND "Name of Command"
                            (LIST UIS:POINTER-BUTTON-2))
```

Or,

```
(EDITOR:BIND-POINTER-COMMAND "Name of Command"
                            `(,UIS:POINTER-BUTTON-2))
```

These are the most common methods of specifying button transitions. For other methods, see the description of BIND-POINTER-COMMAND in Part III of this manual.

3.5.2 Specifying a Button State

In the examples above, the assumption is that all pointer buttons except one specified in the *pointer-action* argument are in the up state. The :BUTTON-STATE keyword permits chording of pointer buttons to invoke commands. That is, you can specify that the *pointer-action* argument is to invoke the command only if one or more pointer buttons are depressed at the time the pointer action occurs.

For instance, in the Editor as provided:

- Depressing the middle button invokes the command "EDT Cut" (in "EDT Emulation" style)

- Depressing the middle button with the left button depressed invokes the command "EDT Paste at Pointer" (in "EDT Emulation" style)

The value for the :BUTTON-STATE keyword is a button constant or the LOGAND of two or more button constants. These indicate the button(s) that must be in a down state at the time the specified *pointer-action* occurs.

For instance:

- To specify that the left button is depressed:

    ```
    :BUTTON-STATE   UIS:POINTER-BUTTON-1
    ```

- To specify that the left and right buttons are depressed:

    ```
    :BUTTON-STATE   (LOGAND  UIS:POINTER-BUTTON-1
                            UIS:POINTER-BUTTON-3)
    ```

If the *pointer-action* argument is a button transition, then any value supplied for that button in the :BUTTON-STATE argument is ignored.

The binding of "EDT Paste at Pointer" -- depressing the middle button with the left button depressed -- is established by:

```
(EDITOR:BIND-POINTER-COMMAND "EDT Paste at Pointer"
                        UIS:POINTER-BUTTON-2
                        :BUTTON-STATE   UIS:POINTER-BUTTON-1
                        :CONTEXT '(:STYLE "EDT Emulation"))
```

The global binding of "Move Point and Select Region" -- move pointer with left button depressed -- is established by:

```
(EDITOR:BIND-POINTER-COMMAND "Move Point and Select Region"
                        :MOVEMENT
                        :BUTTON-STATE   UIS:POINTER-BUTTON-1)
```

The button state in a chorded pointer binding is a static state of the button or buttons indicated. You should, however, consider the transitions (prior pressing and subsequent releasing) that establish and end that state. Either of these transitions might be bound to a DIGITAL-supplied command (see *VAX LISP/VMS User's Guide* for initial pointer bindings). Or, you might wish to bind one or both transitions to commands.

3.5.3 Getting the State of the Pointer

You can retrieve the state of the pointing device -- which includes the position of the pointer cursor, the up-or-down state of each button, and other information -- for a given point in time by calling GET-POINTER-STATE. This function is described in full in Part III.

GET-POINTER-STATE returns a pointer-state object that contains information about the state of the pointer at the time the function is called. If GET-POINTER-STATE is called from within an Editor command *and* if that command was invoked by a pointer action, the function returns the state of the pointer at the time the pointer action occurred. If the pointer action that invoked the command was a button transition, then the pointer-state object contains the state of the buttons at the end of the transition.

GET-POINTER-STATE is useful in commands that take different actions depending on some feature of the pointer state. For instance, the DIGITAL-supplied command "Yank at Pointer" tests to see whether the pointer cursor is indicating a text position (line and character position).

- If so, "Yank at Pointer" moves the current buffer point to that text position and inserts the current region in the kill ring at the modified buffer point.

- If the pointer cursor is indicating an empty position in a line, "Yank at Pointer" moves the current buffer point to the last character position in that line and inserts the kill region.

- If the pointer cursor is not indicating a line, "Yank at Pointer" moves the current buffer point to the last character position in the current buffer and inserts the kill region.

"Yank at Pointer" calls GET-POINTER-STATE to store the pointer-state information, and calls POINTER-STATE-TEXT-POSITION to retrieve the text position (line and character position) stored in the pointer-state object. A possible way to implement "Yank at Pointer" is:

```
(DEFINE-COMMAND (YANK-AT-POINTER-COMMAND
                      :DISPLAY-NAME "Yank at Pointer")
               (PREFIX)

  (DECLARE (IGNORE PREFIX))
  (LET ((STATE (GET-POINTER-STATE)))

    ;; Get the text position of the pointer cursor and bind the
    ;; two values to LINE and CHARPOS.
    (MULTIPLE-VALUE-BIND (LINE CHARPOS)
                      (POINTER-STATE-TEXT-POSITION STATE)

      ;; If there is a line, move buffer point to the CHARPOS or
      ;; to the end of that line.
      (IF LINE
          (MOVE-MARK-TO-POSITION (CURRENT-BUFFER-POINT)
                                 (OR CHARPOS
                                     (LINE-LENGTH LINE))
                                 LINE)

          ;; If there is no line, move buffer point to end of
          ;; buffer.
          (BUFFER-END (CURRENT-BUFFER-POINT)))

      ;; After buffer point is modified, call YANK-COMMAND.
      (YANK-COMMAND NIL))))
```

If "Yank at Pointer" has been invoked by means of its pointer binding in "EMACS" style (depress middle button with left button depressed), the command uses the pointer state at the time this pointer action occurred. If "Yank at Pointer" has been invoked by name, it uses the pointer state at the time the command executes.

The command "Yank at Pointer" calls POINTER-STATE-TEXT-POSITION to retrieve the line and character position of the pointer cursor. Other information contained in the pointer-state object and the corresponding accessing functions are:

- The window position (display row and display column in a given window) -- POINTER-STATE-WINDOW-POSITION

- The pointer action, if any, that invoked the currently executing command -- POINTER-STATE-ACTION

- The state of the pointer buttons at the time GET-POINTER-STATE was called -- POINTER-STATE-BUTTONS

Further information on GET-POINTER-STATE and on each of these accessing functions appears in Part III of this manual.

Note that the function POINTER-STATE-BUTTONS can be used to implement chording in pointer bindings. When called from within an Editor command, the following form

    (POINTER-STATE-BUTTONS (GET-POINTER-STATE))

returns the state (up or down) of each pointer button at the time the command was invoked (see Part III). You can use this information to have the command take different actions depending on whether a specified button was depressed at the time a pointer action invoked the command.

CHAPTER 4

TEXT OPERATIONS

Text consists of the characters that you normally see when you enter the Editor and display a buffer. The essential operations that any editor must allow you to perform on text are:

- Indicating the position occupied by any given character

- Inserting, deleting, and changing characters

- Moving from one character position to another

This chapter introduces the data types and functions you use to program these kinds of operations in the VAX LISP Editor.

You can envision text in the VAX LISP Editor as a group or region of contiguous characters. The characters can be any of those in the ASCII 8-bit extended set (the DEC Multinational Character Set); they can include whitespace and nongraphic characters as well as alphanumeric characters.

Each character in the Editor occupies a specifiable position. You can access the characters either individually (that is, at one position) or in groups (that is, between two positions). You can also manipulate characters -- insert them, delete them, copy them -- either individually or in groups. Finally, you can move around within text either by accessing specified character positions or by searching for particular characters or sequences of characters.

These sets of text-related capabilities in the VAX LISP Editor are introduced in the following order:

- Operations on a particular character position

- Operations on a group of contiguous characters

- Moving and searching operations

- Miscellaneous operations

TEXT OPERATIONS

This chapter introduces the following Editor data types:

- Marks

- Regions

- Attributes

- Lines

Part II of this manual describes these objects in more detail. Reference information concerning the functions and macros that operate on these objects appears in Part III.

Buffers are another relevant Editor data type: most text is contained in buffers. However, most text operations are not operations on a buffer object, but rather on marks, regions, lines, or characters that may be contained in a buffer object. Operations on buffers are covered in Chapter 6 and in Part II.

Recall that the symbols for DIGITAL-provided Editor objects must be referenced in the "EDITOR" package.

4.1 OPERATIONS ON A CHARACTER POSITION

Editor objects called marks are used to reference the position of any character in text. An example of a mark is the buffer point, which is the point of attention in each buffer where most text operations occur. In the current buffer, this mark -- the current buffer point -- is tracked by the screen cursor.

Every buffer you enter or create in the Editor contains at least three marks. Besides the buffer point, each buffer contains two marks that point to the beginning and end of text in that buffer. To reference positions in text, you can use the existing marks, or you can create new marks. Creating marks is covered in Section 4.4.1 below.

By using a mark, you can:

- Retrieve and change a character

- Insert a character

- Insert a string of characters

- Delete one or more characters

In the examples in this section, the variable MARK is assumed to be bound to a mark.

4.1.1 Retrieving and Changing a Character

For the purpose of text operations, you should think of a mark as pointing between two adjacent characters. A mark can also point before the first character in a buffer or after the last character.

You can retrieve the characters on either side of a mark by means of the functions NEXT-CHARACTER and PREVIOUS-CHARACTER. Using SETF, you can also change the specified character.

```
(SETF (NEXT-CHARACTER MARK) #\W)
```

This form changes the character to the right of the mark to W. If your text consists of ABCD and the mark is pointing between B and C, this form changes the C to a W. The text then reads ABWD.

4.1.2 Inserting a Character

You can add a new character to text by means of the function INSERT-CHARACTER. INSERT-CHARACTER takes a mark and a character. It inserts the specified character at the mark, that is, between existing characters.

For instance:

```
(INSERT-CHARACTER MARK #\W)
```

If the mark is pointing between B and C in the text ABCD, then executing this form changes the text to ABWCD.

To perform the same operation at the current buffer point, you could reference that mark by calling the function CURRENT-BUFFER-POINT with no arguments:

```
(INSERT-CHARACTER (CURRENT-BUFFER-POINT) #\W)
```

4.1.3 Inserting a String of Characters

INSERT-CHARACTER allows you to insert only one character at a time. By using INSERT-STRING, you can insert any number of characters at the specified mark.

```
(INSERT-STRING MARK "ABCD EFGH IJKL")
```

If your text consists of XX and the mark is pointing between the two characters, this form changes the text to XABCD EFGH IJKLX.

If the string argument to INSERT-STRING contains newline characters, then multiple lines of text are inserted.

```
(INSERT-STRING MARK "ABCD
EFGH")
```

This form inserts ABCD at the mark, breaks the line, and inserts EFGH at the beginning of the next line. Any text following the mark in the original line will appear after EFGH.

An example of a string that might be inserted in text is the string that the user enters in response to a prompt. For instance:

```
(INSERT-STRING (CURRENT-BUFFER-POINT)
               (SIMPLE-PROMPT-FOR-INPUT "Enter input: "))
```

This form takes the user's response to the prompt and inserts it as text at the current buffer point.

4.1.4 Deleting Characters

DELETE-CHARACTERS takes a mark and an optional integer that defaults to 1. It deletes the specified number of characters after the mark, or before the mark if the integer is negative. If there are not enough characters after (or before) the mark, DELETE-CHARACTERS does not modify the text.

For example, to delete the next 5 characters after a specified mark, you would write:

```
(DELETE-CHARACTERS MARK 5)
```

To delete the character preceding the current buffer point, you would write:

```
(DELETE-CHARACTERS (CURRENT-BUFFER-POINT) -1)
```

When deleting a character, you may want to save the character so that you can reinsert it later. The following forms show a "delete and save" operation and a subsequent reinsertion operation:

```
;;; Define a variable to which to bind a deleted character.

(DEFVAR *SAVED-CHARACTER*)

;;; Bind the character following the current buffer point to the
;;; variable.

(SETF *SAVED-CHARACTER* (NEXT-CHARACTER (CURRENT-BUFFER-POINT)))
```

```
;;; Delete the character following the current buffer point.

(DELETE-CHARACTERS (CURRENT-BUFFER-POINT))

     .
     .
     .

;;; Later, insert the character bound to the variable at the
;;; then-current buffer point.

(INSERT-CHARACTER (CURRENT-BUFFER-POINT) *SAVED-CHARACTER*)
```

4.2 OPERATIONS ON A GROUP OF CHARACTERS

Editor objects called regions indicate groups of contiguous characters. The text in a region can be accessed and manipulated as a unit.

A region is defined by two marks that indicate the character positions where the region begins and ends. To create a region, you write:

```
(MAKE-REGION MARK1 MARK2)
```

Every buffer contains at least one region, which is defined by the marks that indicate the positions where text begins and ends in that buffer. This region is called the buffer region.

Any number of regions can be created within a buffer region. They may overlap in arbitrary ways, and one may be completely contained within another. Since regions may share text, any alterations you do to the text in one region will affect other regions that share that text.

Using regions, you can:

- Insert a block of text

- Copy a block of text

- Delete a block of text

- Delete and save a block of text

- Write a block of text to a file

You can perform these operations on any region. To perform these operations on a buffer, you perform them on the buffer region of that buffer.

4.2.1 Inserting a Region

The function INSERT-REGION enables you to insert a specified block of text as a unit. INSERT-REGION takes a region and a mark at which to insert that region in text. The text inserted is a copy of the specified region; the original region is not altered.

For example:

```
(INSERT-REGION (CURRENT-BUFFER-POINT) (MAKE-REGION MARK1 MARK2))
```

This form defines a region from the two specified marks, which allows you to treat the text between those marks as a single unit. The text in this region is copied, and the copy is inserted at the current buffer point.

4.2.2 Copying a Region

The function COPY-REGION takes a region and returns a new region that contains a copy of the text in the specified region. The new region is "disembodied," in that it is not contained in a buffer. Operations performed on the copy do not affect the original region, and vice versa.

The following forms illustrate the process of copying and saving a specified region for later insertion elsewhere. The original region is not deleted or otherwise altered.

```
;;; Define a variable to which to bind a region.

(DEFVAR *SAVED-REGION*)

;;; Copy a region and bind it to the variable.

(SETF *SAVED-REGION* (COPY-REGION (MAKE-REGION MARK1 MARK2)))

;;; Later, insert the copied region at the current buffer point.

(INSERT-REGION (CURRENT-BUFFER-POINT) *SAVED-REGION*)
```

4.2.3 Deleting a Region

Regions are commonly used to indicate blocks of text to be deleted. You can delete the text in a region by calling either DELETE-REGION or DELETE-AND-SAVE-REGION with a region argument.

The function DELETE-REGION takes a region and deletes the text in it, leaving an empty region.

```
(DELETE-REGION (MAKE-REGION MARK1 MARK2))
```

If you wish to retain a copy of the text in a deleted region so that you can reinsert it elsewhere, you call DELETE-AND-SAVE-REGION. This function deletes the text in a region and returns a disembodied region that contains a copy of the deleted text.

```
(INSERT-REGION (CURRENT-BUFFER-POINT)
               (DELETE-AND-SAVE-REGION
                  (MAKE-REGION MARK1 MARK2)))
```

In this example, DELETE-AND-SAVE-REGION deletes the text in the region between MARK1 and MARK2 and returns a copy of the deleted text. The disembodied region containing the copied text is passed to INSERT-REGION, which inserts it at the current buffer point.

4.2.4 Writing a Region to a File

The function WRITE-FILE-FROM-REGION takes a file name (pathname or namestring) and a region. It writes the specified region to the specified file. For instance:

```
(WRITE-FILE-FROM-REGION "Myfile.lsp"
                        (MAKE-REGION MARK1 MARK2))
```

This form writes the text between the specified marks to a file named "Myfile.lsp".

4.2.5 Operating on Buffers

Text operations that appear to be performed on buffers are actually performed on the buffer regions of those buffers. Some operations you can perform on buffer regions are:

- Deleting the text in a buffer

- Inserting the contents of one buffer into another

- Writing the contents of a buffer to a file

- Inserting the contents of a file into a buffer

These operations use the same region-manipulating functions that apply to smaller regions within a buffer. The difference here is that the region argument you supply is the buffer region. The function BUFFER-REGION takes a buffer and returns the buffer region of that buffer.

4.2.5.1 Deleting The Text In A Buffer - To delete all the text in a buffer, you simply delete the text in the associated buffer region.

        (DELETE-REGION (BUFFER-REGION (CURRENT-BUFFER)))

This form deletes the text in the current buffer. The buffer itself and the empty buffer region remain.

4.2.5.2 Inserting One Buffer Into Another - To insert the text from one buffer into another buffer, you call the function INSERT-REGION. As arguments you supply a mark in one buffer and the buffer region of another buffer.

        (INSERT-REGION (CURRENT-BUFFER-POINT) (BUFFER-REGION BUFFER2))

This form inserts a copy of the buffer region of BUFFER2 into the current buffer at the current buffer point. The content of BUFFER2 is not affected by this operation, and subsequent changes to the text in BUFFER2 and in the inserted region do not affect one another.

4.2.5.3 Writing A Buffer To A File - To write a buffer to a file, you call WRITE-FILE-FROM-REGION and pass it a file name (pathname or namestring) and the buffer region of a specified buffer.

        (WRITE-FILE-FROM-REGION "Myfile.lsp"
                                (BUFFER-REGION (CURRENT-BUFFER)))

This form writes the contents of the current buffer to a file named "Myfile.lsp".

4.2.5.4 Inserting A File Into A Buffer - The function INSERT-FILE-AT-MARK is similar to INSERT-STRING in that it inserts text at a specified mark. The mark can indicate any text position in a buffer or disembodied region; the text inserted is the content of a specified file. INSERT-FILE-AT-MARK is commonly used to insert a file into a buffer.

For example:

        (INSERT-FILE-AT-MARK "Myfile.lsp" (CURRENT-BUFFER-POINT))

This form inserts the contents of the file "Myfile.lsp" into the current buffer at the current buffer point.

4.3 MOVING AND SEARCHING OPERATIONS

A number of functions exist that "move" marks. That is, these functions modify a mark so that it specifies a different text position.

There are three basic ways to move marks:

- By specifying a new character position

- By searching for a specified string of characters

- By searching for a character with a particular property

4.3.1 Moving by Character Positions

Several functions take a mark and alter it to point to a specified character position. The character position can be specified either by "counting" from the mark's initial position or by referencing another mark.

To move a mark one or more character positions away from its current position, you call:

- MOVE-MARK-AFTER - moves a mark to the position that follows its initial position

- MOVE-MARK-BEFORE - moves a mark to the position that precedes its initial position

- CHARACTER-OFFSET - takes a count and moves the mark forward that many positions (backward if the count is negative).

You can also move a mark to point to the position specified by another mark. Some functions you can use are:

- BUFFER-END - moves a mark to. the end of the text in a specified buffer

- BUFFER-START - moves a mark to the beginning of the text in a specified buffer

- MOVE-MARK - moves a mark to the position occupied by any other specified mark

Moving by character position is illustrated in a new function that transposes the pair of characters before a specified mark. This function also illustrates accessing and manipulating individual characters.

```
(DEFUN TRANSPOSE-CHARACTERS (MARK)

  " Transpose the pair of characters before the specified mark."

  ;; Access the character before the mark and bind it to CHAR2.

  (LET ((CHAR2 (PREVIOUS-CHARACTER MARK)))

    ;; If there is a character before the mark, delete that
    ;; character.

    (WHEN CHAR2
      (DELETE-CHARACTERS MARK -1)

      ;; If there is a character in the position now
      ;; preceding the mark, move to the position preceding
      ;; that character and reinsert the deleted character.

      (COND ((PREVIOUS-CHARACTER MARK)
             (INSERT-CHARACTER (MOVE-MARK-BEFORE MARK)
                               CHAR2)
             ;; Move the mark back to its initial position.

             (MOVE-MARK-AFTER MARK))

            ;; If there is no character in the position
            ;; preceding the mark, reinsert the deleted
            ;; character at its initial position.

            (T
             (INSERT-CHARACTER MARK CHAR2)))))))
```

The action of this function differs slightly from that of the DIGITAL-supplied command "Transpose Previous Characters". One difference is that the command suppresses screen display of the separate text operations, showing only the completed action. Display-related operations are discussed in Chapter 5. Also, the command creates a new mark for the operation and disposes of the mark after the operation is completed. Creating marks is discussed in Section 4.4.1 below.

4.3.2 Searching by Pattern

Searching by pattern enables you to move a mark to a specified string of characters within a region of text. The search can be forward or backward from the mark's initial position, and it can either consider or ignore case in determining whether a text string matches the search string.

To perform a search by pattern, you call two functions:

- MAKE-SEARCH-PATTERN - computes a pattern, including the string to be matched, the direction of the search, and whether the search is case-sensitive

- LOCATE-PATTERN - initiates a search operation beginning at a specified mark and searching according to the parameters of the specified pattern

This section illustrates the use of these functions to implement search operations.

4.3.2.1 Making A Search Pattern - Before beginning a search operation, you call MAKE-SEARCH-PATTERN, which computes and returns a search pattern. Its format is:

MAKE-SEARCH-PATTERN *kind direction string* &OPTIONAL *reuse-pattern*

The *kind* argument can be either :CASE-SENSITIVE or :CASE-INSENSITIVE. The *direction* argument can be either :FORWARD or :BACKWARD. The *string* argument is the string to be searched for. (The optional *reuse-pattern* argument is described in Part III.)

For instance, to search forward for the string ABCDE, disregarding case, you could begin with the following pattern:

(MAKE-SEARCH-PATTERN :CASE-INSENSITIVE :FORWARD "abcde")

4.3.2.2 Locating A Search Pattern - To initiate the search, you call LOCATE-PATTERN. This function takes a search pattern, such as that specified above, as well as a mark at which to begin the search:

LOCATE-PATTERN *mark search-pattern*

LOCATE-PATTERN searches for a text string that matches the specified search pattern. If one is found, it changes the mark to point to the beginning of the matched string.

A very simple search operation is illustrated by the following command:

```
(DEFINE-COMMAND (MY-SIMPLE-SEARCH-COMMAND
                    :DISPLAY-NAME "My Simple Search")
                (PREFIX)

    (DECLARE (IGNORE PREFIX))

    (LOCATE-PATTERN (CURRENT-BUFFER-POINT)
                    (MAKE-SEARCH-PATTERN
                        :CASE-INSENSITIVE
                        :FORWARD
                        (SIMPLE-PROMPT-FOR-INPUT
                            "Search for: "))))
```

"My Simple Search" prompts for a search string, which it uses in making a search pattern. It then searches forward for that string, beginning at the current buffer point and disregarding case.

Note that this command searches only once. To locate more than one occurrence of the string, you would need to invoke the command repeatedly. To search in the opposite direction, you would need to write another command with :BACKWARD as the *direction* argument to MAKE-SEARCH-PATTERN.

4.3.2.3 **Replacing A Pattern** - You can also program the Editor to replace strings that it locates through a search operation. The function REPLACE-PATTERN is similar to LOCATE-PATTERN except that it takes a *replacement* argument -- a new string with which to replace the string it locates in the text:

REPLACE-PATTERN *mark search-pattern replacement* &OPTIONAL *n*

For example:

```
(REPLACE-PATTERN (CURRENT-BUFFER-POINT)
                 (MAKE-SEARCH-PATTERN
                     :CASE-SENSITIVE
                     :BACKWARD
                     "This is a")
                 "This is not a")
```

This form searches backward through text from the current buffer point for every case-matched instance of the search string. It deletes each matching string and replaces it with the specified replacement string. (Unlike LOCATE-PATTERN, REPLACE-PATTERN does not move the mark.)

The optional *n* argument to REPLACE-PATTERN allows you to specify how many occurrences of the string should be replaced (see the full

description of REPLACE-PATTERN in Part III). The default action is to replace every instance in the direction specified in the search pattern.

4.3.3 Searching by Attribute

Searching by attribute enables you to locate text entities such as words, whitespace, LISP forms, and so on. The Editor recognizes these entities by the characters that define or delimit them. For instance, the Editor locates a word by searching for a character that it recognizes as a word delimiter.

Characters acquire these added properties by means of Editor objects called attributes. Some attributes provided with the Editor are "Word Delimiter", "Whitespace", and "LISP Syntax". (Attributes can also be user-defined.) Once an attribute is established in the Editor, then all 256 characters have a value for that attribute.

NOTE

> Editor attributes should not be confused with character attributes in COMMON LISP. The Editor ignores all COMMON LISP bit and font information about characters. Editor attributes capture other, Editor-specific information about the meaning of individual characters.

You can think of Editor attributes on the analogy of a human attribute, such as "Political Party Member". In contexts where you apply this attribute to people, every person has a value for it. The values might be specified as REPUBLICAN, DEMOCRAT, WHIG, TORY, and so on. Another possible value is NOT-A-MEMBER.

Similarly, every character in the Editor has a value for the attribute "Whitespace". The possible values in this case are 1 and 0, which you can think of as IS-WHITESPACE and IS-NOT-WHITESPACE, respectively.

To carry out a search by attribute, the Editor tests each character in turn until it locates one that has a particular value for the specified attribute. For example:

- To skip over whitespace to the next non-whitespace character, the Editor searches for the next character with the value 0 for the attribute "Whitespace".

- To find word breaks, the Editor searches for characters with the value 1 for the attribute "Word Delimiter".

- To find the next list in LISP code, the Editor searches for the next character with the value :LIST-INITIATOR for the attribute "LISP Syntax".

To search by attribute in the Editor, you call the function LOCATE-ATTRIBUTE. This section introduces:

- Using LOCATE-ATTRIBUTE

- Mark and cursor behavior in an attribute search

- Using LOCATE-ATTRIBUTE repeatedly

This discussion focuses on using DIGITAL-provided attributes and attribute values. Information on creating new attributes and on changing attribute values appears in Chapter 6 and in Part II.

4.3.3.1 **Using LOCATE-ATTRIBUTE** - The function LOCATE-ATTRIBUTE is used to locate a character with a particular attribute value. This function is described in full in Part III of this manual. Its format, with only a few of its parameters, is:

```
LOCATE-ATTRIBUTE mark attribute &KEY
                               :TEST
                               :DIRECTION
```

LOCATE-ATTRIBUTE scans the text in the specified direction to find a character with a particular value for the specified *attribute*. The value of interest is one for which the specified test function returns a non-NIL value. If such a character is found, LOCATE-ATTRIBUTE moves the *mark* to point to that character. The default value for the keyword argument :DIRECTION is :FORWARD; the default function used as the :TEST is PLUSP.

For example, to find the next word delimiter in a region of text, you could write:

```
(LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT)
                  "Word Delimiter"
                  :TEST #'PLUSP
                  :DIRECTION :FORWARD)
```

This form moves the current buffer point forward to the next character whose "Word Delimiter" value is 1. The values are tested by passing them to the predicate function PLUSP. PLUSP returns NIL if its argument is 0 and T if its argument is greater than 0. The first character whose "Word Delimiter" value is 1 satisfies the test, and the search stops.

LOCATE-ATTRIBUTE behaves the same way when called with the argument :BACKWARD, but the search direction is reversed.

```
(LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT)
                  "Word Delimiter"
                  :DIRECTION :BACKWARD)
```

In this case, LOCATE-ATTRIBUTE moves the current buffer point backward to the first character whose "Word Delimiter" value satisfies the default test PLUSP.

To find the next character that is not a word delimiter, you change the test function:

```
(LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT)
                  "Word Delimiter"
                  :TEST #'ZEROP)
```

This form moves the current buffer point to the next character that has the "Word Delimiter" value 0. The function ZEROP returns non-NIL only when its argument is 0.

4.3.3.2 **Mark And Cursor Behavior** - When using LOCATE-ATTRIBUTE to move a mark, it is important to remember that marks point between characters rather than to characters. Depending on the direction of the search, the modified (or "moved") mark points either just before or just after the character that has satisfied the test. The screen cursor, on the other hand, always appears on the character just after the mark it is tracking (the current buffer point).

A mark's behavior in an attribute search is symmetrical -- "mirror-image" -- forward and backward. The cursor's behavior is not symmetrical.

For instance, imagine that the following text string contains the current buffer point in the position between J and K; the cursor appears on K.

A B C D E F G H I J K L M N O P Q R S T

MLO-250-86

Then call LOCATE-ATTRIBUTE (with its default arguments) to search ahead for the first character with the value 1 for the attribute "Word Delimiter".

```
(LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT) "Word Delimiter")
```

The test succeeds at the space after L, and the search stops. The mark is left pointing just before the space and the cursor is on it.

A B C D   E F G H   I J K L M N O P   Q R S T
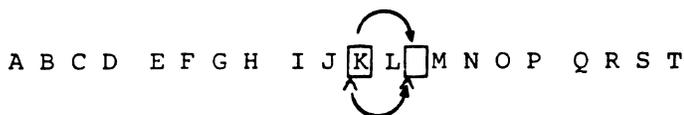
MLO-251-86

If you evaluate the form again, LOCATE-ATTRIBUTE does not move the mark. It may appear, since the cursor is on the space, that the next word delimiter is the space after P. However, the mark is actually positioned just before the space indicated by the cursor. This character satisfies the test. (The following section discusses how to call LOCATE-ATTRIBUTE repeatedly.)

The backward-searching behavior of LOCATE-ATTRIBUTE is a mirror image of its forward-searching behavior. The symmetry is apparent when you consider mark positions, but less apparent when you consider only cursor positions.

For instance:

```
(LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT)
                  "Word Delimiter"
                  :DIRECTION :BACKWARD)
```

A B C D   E F G H   I J K L   M N O P   Q R S T
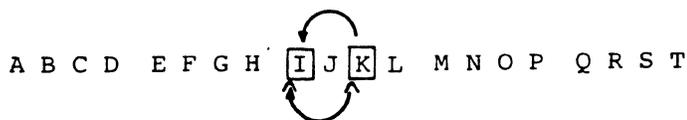
MLO-252-86

If the mark points, as before, between J and K, the first character to satisfy the test is the space between H and I. Because the search direction is backward, LOCATE-ATTRIBUTE moves the mark to the position between the space and the I. The mark indicates the space character to its left, but the cursor, which is always to the right of the mark, stops on the I. (Compare this cursor behavior with the cursor behavior when LOCATE-ATTRIBUTE searches forward.)

Again, LOCATE-ATTRIBUTE does not move the mark if called a second time. The character that the mark is indicating is the space, which satisfies the test.

4.3.3.3 Using **LOCATE-ATTRIBUTE** Repeatedly – Some higher level functions and commands may invoke LOCATE-ATTRIBUTE more than once. For instance, WORD-OFFSET takes an optional count that indicates the

number of word breaks to be located. It invokes LOCATE-ATTRIBUTE the number of times specified.

When you invoke LOCATE-ATTRIBUTE repeatedly, you need to consider the cases where the mark is already indicating a character with the attribute value in question. As shown above, LOCATE-ATTRIBUTE does not move the mark when the first character in the specified direction satisfies the test.

It is necessary, therefore, to include a test of the mark's position before invoking LOCATE-ATTRIBUTE. An example of such a test is the following:

```
(DEFMACRO NEXT-CHAR-IN-WORD-P (MARK)
  `(LET ((NEXT (NEXT-CHARACTER ,MARK)))
     (AND NEXT
          (ZEROP (THE FIXNUM (CHARACTER-ATTRIBUTE
                                "Word Delimiter" NEXT)))))))
```

The macro NEXT-CHAR-IN-WORD-P tests whether the character after the mark is part of a word, and thus not a word delimiter. That is, it tests whether that character has the value 0 for the attribute "Word Delimiter".

Using this test, you can now write a command that invokes LOCATE-ATTRIBUTE:

```
(DEFINE-COMMAND (CAPITALIZE-WORD-AND-TRAVEL-COMMAND
                   :DISPLAY-NAME "Capitalize Word and Travel")
                (PREFIX)

  ;; Repeat the action if a prefix argument is supplied.

  (DOTIMES (INDEX (OR PREFIX 1))

    ;; If the next character is a word delimiter, find the next
    ;; one after it that is not a word delimiter.

    (UNLESS (NEXT-CHAR-IN-WORD-P (CURRENT-BUFFER-POINT))
      (LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT)
                        "Word Delimiter"
                        :TEST #'ZEROP))

    ;; If the next character is not a word delimiter, capitalize
    ;; the word that contains it and move to the next word.

    (WHEN (NEXT-CHAR-IN-WORD-P (CURRENT-BUFFER-POINT))
      (CAPITALIZE-WORD-COMMAND 1)
      (FORWARD-WORD-COMMAND 1))))
```

4-17

4.4 MISCELLANEOUS TEXT OPERATIONS

The preceding sections have assumed that you are working with existing marks. You can, however, create new marks when you are programming text operations.

For some operations, you can also work with lines as text-containing objects. Lines are sometimes less convenient to use than marks and regions, but line operations may be more efficient to execute.

This section introduces the techniques of:

● Creating marks

● Operating on lines

4.4.1 Creating Marks

Marks are used primarily to indicate positions for insertions and deletions in text. A single mark can indicate the position for an insertion operation; a pair of marks can indicate a region of text to be deleted or inserted.

The Editor supplies a number of marks automatically -- buffer points and the marks that indicate the limits of buffer regions. If none of these marks is suitable for the operation you want to perform, you can create one or more new marks.

New marks are most often created by "copying" existing marks. Both the function COPY-MARK and the macro WITH-MARK create a new mark that indicates the same text position as a specified mark. You can also specify the type of mark you want to create; a mark's type determines its behavior in a text operation.

This section introduces:

● Mark types and their behavior

● Using COPY-MARK

● Using WITH-MARK

4.4.1.1 **Mark Types And Their Behavior** - Whenever you create a new mark, you need to consider what becomes of that mark after a text operation is performed on it. Marks are of two basic types:

- Temporary marks, which become invalid after any operation upon them

- Permanent marks, which remain valid after any operation upon them

Temporary marks are useful for one-time operations; after any operation that affects the mark or the text to which it points, the mark becomes invalid and should not be reused. Temporary marks are more efficient than permanent marks for some applications because they require less overhead to make and use.

Permanent marks remain valid after any operation on them, including deletion of the text to which they point. For instance, the current buffer point and the two marks that indicate the beginning and end of text in a buffer are permanent marks. If you delete all the text in a buffer, these three marks (and any other permanent marks in that buffer) continue to point into that (empty) buffer. Permanent marks can be removed only with the function DELETE-MARK.

When you insert text at a permanent mark, the mark's behavior depends on whether it is left-inserting or right-inserting:

- A left-inserting mark appears at the end of a new insertion. That is, new text is inserted to the left of the mark. The current buffer point and the buffer-end mark are permanent left-inserting marks.

- A right-inserting mark appears at the beginning of a new insertion. That is, new text is inserted to the right of the mark. The buffer-start mark is a permanent right-inserting mark.

You can specify the type of mark you want to create when you call COPY-MARK or WITH-MARK. Each takes an optional *mark-type* argument which may be :TEMPORARY, :LEFT-INSERTING, or :RIGHT-INSERTING.

4.4.1.2 **Using COPY-MARK** - The function COPY-MARK creates and returns a new mark that points to the same position as a specified mark.

COPY-MARK *mark* &OPTIONAL *mark-type*

For instance:

(COPY-MARK (CURRENT-BUFFER-POINT) :TEMPORARY)

This form creates a new temporary mark that specifies the same text position as the current buffer point.

If no *mark-type* argument is specified, the new mark is of the same type as the specified mark. In the following example, the new mark is a permanent left-inserting mark (like the current buffer point).

```
(COPY-MARK (CURRENT-BUFFER-POINT))
```

COPY-MARK is often used in defining a region between the current buffer point and some other character position in the buffer. The procedure is:

1. Indicate the position of the buffer point by placing a new mark there.

2. Move the buffer point with any of the mark-moving functions.

3. Define a region from the new mark and the modified buffer point.

For instance, to make a region from the current buffer point to the end of a buffer, you could write:

```
(MAKE-REGION (COPY-MARK (CURRENT-BUFFER-POINT))
             (BUFFER-END (CURRENT-BUFFER-POINT)))
```

This form copies the current buffer point and then moves the buffer point to the end of the buffer. It uses the copied mark and the altered buffer point to define a region.

If you perform an operation on this region, both marks will remain valid because both are permanent marks. For instance:

```
(DELETE-REGION
      (MAKE-REGION (COPY-MARK (CURRENT-BUFFER-POINT))
                   (BUFFER-END (CURRENT-BUFFER-POINT))))
```

This form deletes all the text between the initial position of the current buffer point and the end of the buffer. Both the buffer point and the new mark are left pointing to the end of the buffer. However, because DELETE-REGION returns NIL, you have no access to either the new mark or the new region.

4.4.1.3 **Using WITH-MARK** - When you are programming text operations, you may want to dispose of a new mark after it has served its purpose. In this situation, you can create the mark with the macro WITH-MARK.

WITH-MARK copies a mark and binds the new mark to a specified variable. The new mark can be of any mark type; the default is :TEMPORARY. The variable can be referenced within the body of the macro. Upon exit from the form, the mark is deleted and the variable becomes unbound.

WITH-MARK is analogous to the COMMON LISP macro WITH-OPEN-FILE. Its use guarantees that the overhead of creating a mark ends on exit from the macro.

An example of the use of WITH-MARK follows. This form copies the current buffer point for a file-insertion operation. Inserting a file moves the current buffer point to the right and leaves it at the end of the new insertion. If you want the current buffer point to end up at the beginning of the new insertion, you could write:

```
;; Place a new right-inserting mark at the same position as the
;; current buffer point.

(WITH-MARK ((NEW-MARK (CURRENT-BUFFER-POINT) :RIGHT-INSERTING))

    ;; Insert the file at the current buffer point.

    (INSERT-FILE-AT-MARK FILE (CURRENT-BUFFER-POINT))

    ;; Move the current buffer point back to the position of the
    ;; new mark.

    (MOVE-MARK (CURRENT-BUFFER-POINT) NEW-MARK))
```

This form creates a new mark, bound to the variable NEW-MARK. Because NEW-MARK is right-inserting, it remains in its initial position when you insert a file. The current buffer point, which is left-inserting, moves to the end of the new text. After the insertion, the buffer point moves back to the position indicated by the new mark. When the action is completed, NEW-MARK becomes unbound and the new mark is deleted.

4.4.2 Operating on Lines

A line is an Editor object that points to a string of characters. The text string in a line normally corresponds to the line of text displayed on the screen. A line also contains pointers to the lines that precede and follow it.

All text operations result, directly or indirectly, in the alteration of lines or of their relative positions. For instance, marks point into lines, and operations on marks alter the lines into which they point. Also, the two marks that define a region point into the same or different lines, and operations on the region alter the line or lines that the region contains.

You may prefer, for reasons of efficiency, to perform some text operations directly on lines. These operations include:

- Retrieving and altering the string of text in a line

- Retrieving and altering a particular character

- Moving from one line to another

- Testing the relative positions of lines

Lines are created as a result (side effect) of Editor operations such as reading files, breaking lines, and so on. In the following examples, the variables LINE, LINE1, and LINE2 are assumed to be bound to lines.

4.4.2.1 **Retrieving And Altering The Text In A Line** - The function LINE-STRING takes a line and returns the string that is the text contained in that line.

    (LINE-STRING LINE)

This form returns the text in the specified line as a string.

You can use SETF with LINE-STRING to modify the text in a line.

    (SETF (LINE-STRING LINE) "abcde")

This form accesses the text in LINE and replaces it with abcde.

To change the string from abcde to ABCDE, you could write:

    (SETF (LINE-STRING LINE) (NSTRING-UPCASE (LINE-STRING LINE)))

Note that you must use SETF to have the destructive operation performed by NSTRING-UPCASE appropriately reflected in LINE.

4.4.2.2 **Retrieving And Altering A Single Character** - The function LINE-CHARACTER takes a line and an integer that indicates a character position in that line. The character positions are numbered from the left beginning with 0. LINE-CHARACTER returns the character in the specified position (or NIL if no character is found). You can use SETF with this function to alter the specified character.

For example:

    (SETF (LINE-CHARACTER LINE 4) #\W)

In this form, LINE-CHARACTER returns the character in position 4 in LINE. Setting that value to W changes the text string in LINE from ABCDE to ABCDW.

You can break a line, and thus create a new line, by replacing a character with a newline character:

```
(SETF (LINE-CHARACTER LINE 2) #\NEWLINE)
```

This form replaces the C in LINE with a newline character. The result of this operation is two lines, one containing the text AB and the next containing the text DW.

4.4.2.3 Moving By Line

4.4.2.3 **Moving By Line** - A line contains pointers to the lines that precede and follow it. You can move from line to line by following these links. LINE-NEXT and LINE-PREVIOUS take a line and return the next line or the previous line (or NIL if no line is found).

For example, assume that LINE1 is followed by LINE2. Then,

```
(LINE-NEXT LINE1)              ;returns LINE2

(LINE-PREVIOUS LINE2)          ;returns LINE1
```

4.4.2.4 Testing Relative Line Positions

4.4.2.4 **Testing Relative Line Positions** - To check the relative positions of two lines, or to see if the two are the same line, you use the functions LINE=, LINE<, LINE>=, and so on. These functions are listed in Appendix A and described in full in Part III of this manual.

For instance, the following form returns T only if LINE1 follows LINE2:

```
(LINE> LINE1 LINE2)
```

Because LINE2 is the second of the two lines in the example, this form returns NIL.

4.4.2.5 Retrieving And Testing Mark Positions

4.4.2.5 **Retrieving And Testing Mark Positions** - Marks have been introduced as objects that indicate positions in text. A mark indicates a text position by pointing to a line and to an integer that is a character position in that line. (Recall that character positions are numbered from the left beginning with 0.)

You can determine a mark's position by means of the functions MARK-LINE and MARK-CHARPOS:

- MARK-LINE takes a mark and returns the line into which that mark points.

- MARK-CHARPOS takes a mark and returns its character position, that is, the number of characters to the left of the mark in the same line.

To check the relative positions of two marks, you call a function such as MARK=, MARK>, and so on. These functions are listed in Appendix A and described in full in Part III of this manual.

4.4.2.6 **Example Of An Operation On Lines** - The following example implements a function that performs a text operation directly on lines. The new function, PREFIX-LINES, takes a string and a region; it adds the specified string to the front of each line in the specified region. This function could be used, for instance, to indent text (by inserting a specified number of spaces at the beginning of each line) or to indicate comments in any code (by inserting the appropriate comment-delimiters and spaces).

```
(DEFUN PREFIX-LINES (STRING REGION)

   " Adds the specified string to the beginning of each line
in the specified region."

   ;; Access each line in turn, beginning with the line that
   ;; contains the mark that starts the region.

   (DO* ((LINE (MARK-LINE (REGION-START REGION))
               (LINE-NEXT LINE)))

        ;; When LINE contains the mark that ends the region,
        ;; end the loop.

        ((OR (NULL LINE)
             (LINE> LINE (MARK-LINE (REGION-END REGION)))))

        ;; Prefix each line with the specified string.

        (SETF (LINE-STRING LINE)
              (CONCATENATE 'STRING STRING (LINE-STRING LINE)))))
```

CHAPTER 5

WINDOW AND DISPLAY OPERATIONS

Whenever you enter the VAX LISP Editor and select a buffer, the Editor makes a window onto that buffer and displays it on the screen. A window is an Editor object that translates some portion of the text in a buffer into a form that is displayable. Displaying the window makes that text visible.

NOTE

Editor windows are similar to virtual displays in the VMS screen management facility (SMG).* Creating a window is a separate operation from displaying it, and windows can exist without being displayed. As these features suggest, Editor windows are quite different from windows in traditional EMACS editors, where a window is a section of the screen.

During an interactive session, the Editor makes and displays windows as a result of executing certain commands. For instance, you see a window appear whenever you edit a file or function, select a buffer, or execute "Help" or "Describe". Each of these windows is an Editor object that includes certain information, such as its size, screen position, display type (anchored versus floating), and the content and position of its label. In LISP code, you can alter the features of a particular window, and you can program the Editor to make windows with the features you specify.

When multiple windows are displayed, a display manager within the Editor determines how they are arranged on the screen. Depending on the number of windows and their display types, some windows may overlap others, and some may be resized or repositioned on the screen. In all situations, space at the bottom of the screen is reserved for

----

* See *VAX/VMS Run-Time Library Routines Reference Manual*.

the information area, which the Editor uses to report on its activities and to signal errors, and for the prompting window.

A few DIGITAL-provided Editor commands enable you to operate directly on windows or to override the automatic display management. For instance, you can resize, scroll, and split the window you are working in; you can resize the display area or remove windows from it; and you can move the cursor from one window to another. If you want to exert finer control over window and display operations, you can use the functions and other objects in the Editor's display subsystem to implement new commands.

This chapter introduces the techniques of programming the Editor to perform window and display operations. It also covers operations on the display area (screen) and on the information area (a section of the screen).

The topics covered in this chapter are:

● Accessing windows

● Operations on a window's text content

● Operations on window appearance: video rendition and labeling

● Managing display, window size, and window screen position

● Making and deleting windows

Part III of this manual contains more detailed information concerning the individual objects in the display subsystem. An extended example at the end of this chapter (Section 5.6) illustrates the use of many of these objects.

5.1 ACCESSING WINDOWS

Windows are not named Editor objects; that is, you can access an Editor window only by means of an expression that evaluates to that window object. This section introduces the functions that you use to access some particular windows in LISP code:

● The current window

● The windows onto a buffer

● All the windows on the screen

● The "next" window on the screen

Many of the examples in this chapter use these functions to access windows. Otherwise, the variables WINDOW, WINDOW1, WINDOW2, and so on are assumed to be bound to Editor windows.

Recall that the symbols for DIGITAL-provided Editor objects must be referenced in the "EDITOR" package.

5.1.1 The Current Window

Windows are always associated with buffers, and more than one window can open onto a single buffer. One window onto the current buffer is the current window. This is the window that contains the cursor; it is the "active" window where text operations commands are executed.

The current window is returned by the function CURRENT-WINDOW, which takes no arguments. You can use SETF with CURRENT-WINDOW to make another window the current window:

    (SETF (CURRENT-WINDOW) WINDOW2)

This form makes WINDOW2 the current window. If WINDOW2 is not already displayed, the display manager makes it visible on the screen. The buffer associated with WINDOW2 becomes the current buffer, and the cursor moves to WINDOW2.

5.1.2 The Windows onto a Buffer

The function BUFFER-WINDOWS returns a list of the windows that open onto a specified buffer. The list contains all windows onto that buffer, including any that are not currently displayed.

For instance, another way to access the current window is:

    (FIRST (BUFFER-WINDOWS (CURRENT-BUFFER)))

The current window is always the first element in the list of windows onto the current buffer.

5.1.3 All the Windows on the Screen

The function VISIBLE-WINDOWS returns a list of all windows that are currently displayed, regardless of what buffers they open onto. Visible windows are those that have been displayed but not removed. Even if a window is completely overlapped by other windows, it is still considered "visible" and is in the list returned by VISIBLE-WINDOWS.

An example using VISIBLE-WINDOWS is the DIGITAL-provided command "Remove Other Windows". This command checks each element of the list returned by VISIBLE-WINDOWS to determine if that window is the current window. (The current window may be anywhere on the list.) Each window that is not the current window is removed from the screen by means of the function REMOVE-WINDOW.

```
(DEFINE-COMMAND (REMOVE-OTHER-WINDOWS-COMMAND
                :DISPLAY-NAME "Remove Other Windows")
               (PREFIX)

 "  Removes all windows from the screen except the current
window."

 (DECLARE (IGNORE PREFIX))

 ;; Display only the result of the operation, not intermediate
 ;; states.
 (WITH-SCREEN-UPDATE

   (LET ((CURRENT (CURRENT-WINDOW)))
     (DOLIST (WINDOW (VISIBLE-WINDOWS))
       (UNLESS (EQ WINDOW CURRENT)
         (REMOVE-WINDOW WINDOW)))))
 T)
```

5.1.4 The "Next" Window

The function NEXT-WINDOW returns a visible window other than the current window (or NIL, if no other window is found). The format of NEXT-WINDOW is:

    NEXT-WINDOW &OPTIONAL *window-type count*

The sequence in which windows are accessed is undefined, except that you can limit the search to windows of a given *window-type*. The *window-type* argument :ANCHORED or :FLOATING causes NEXT-WINDOW to return a window of that type (or NIL, if none is found). The default argument T causes NEXT-WINDOW to return a window of the same type as the current window; if no such window is visible, then NEXT-WINDOW returns one of the opposite type. If only one window is displayed, then NEXT-WINDOW with argument T returns NIL (the prompting window is not considered as a possible return value).

The optional *count* argument is an integer specifying the number of windows to advance in the sequence to find the window to return. The default *count* is 1. An argument of 0 returns the current window; a negative argument advances through the sequence of windows in reverse order.

If called repeatedly with the same arguments, NEXT-WINDOW returns the same window -- it does not advance through the sequence of windows. However, you can circulate through all the displayed windows, or through all those of a specified type, by repeatedly setting the current window to the "next" window. This is the action of the DIGITAL-provided command "Next Window", which moves the cursor to another window on the screen. A possible implementation is:

```
(DEFINE-COMMAND (NEXT-WINDOW-COMMAND
                    :DISPLAY-NAME "Next Window")
               (PREFIX)

    "  Switches the current window to be the next visible window
    on the screen.  If a prefix argument n is supplied, it goes
    to the nth visible window."

    (SETF (CURRENT-WINDOW)
          (NEXT-WINDOW T (OR PREFIX 1))))
```

Since the *window-type* argument to NEXT-WINDOW is T, this command circulates through all the visible windows when you execute it repeatedly.

The command "Previous Window" circulates in reverse order. Its implementation is identical to that for "Next Window" except that the *count* argument to NEXT-WINDOW is negative:

```
(NEXT-WINDOW T (- (OR PREFIX 1)))
```

5.2 WINDOW CONTENT

You can think of a window as a rectangular opening onto a portion of text in a buffer: the window opens onto a group of contiguous lines and a maximum number of characters per line. That portion of text is the "content" of the window. To view other text in the buffer -- either other lines or more characters per line -- you perform certain operations on the window, not on the text.

The operations you can perform on window content are:

- Retrieving window position in the buffer, that is, determining where in the buffer the window begins and ends

- Repositioning a window within a buffer, that is, "moving" a window so that it contains different text lines

- Wrapping text within a window, that is, altering a window so that it contains all the characters in text lines that extend beyond the window

Another way you can view text that overflows a window, either in length or in width, is to alter the dimensions of the window -- making it higher or wider. However, depending on screen size, window type, and the number of windows to be displayed at once, the Editor's display manager may limit or override a window's specified dimensions. The techniques of resizing windows are covered in Section 5.4 below, along with a discussion of display management.

5.2.1 Window Position in a Buffer

The portion of text included in a window is delimited by two marks. These marks are returned by the functions WINDOW-DISPLAY-START and WINDOW-DISPLAY-END; both take a window argument.

Like all marks (see Chapter 4, "Text Operations"), each of these marks indicates a character position in a buffer:

- The display-start mark points to the beginning (character position 0) of the first line in the window.

- The display-end mark points just after the last character in the window.

The text between these two marks is the portion of the buffer's text that the window translates into displayable form.

You can use the display-start and display-end marks is to indicate text positions to which to move another mark. For instance, the DIGITAL-supplied commands "Beginning of Window" and "End of Window" move the current buffer point to coincide with the display-start mark and the display-end mark, respectively, of the current window. A possible implementation of "Beginning of Window" is:

```
(DEFINE-COMMAND (BEGINNING-OF-WINDOW-COMMAND
                     :DISPLAY-NAME "Beginning of Window")
                (PREFIX &OPTIONAL (MARK (CURRENT-BUFFER-POINT))
                        (WINDOW (CURRENT-WINDOW)))

    "Moves the cursor to the beginning of the current window."

    (DECLARE (IGNORE PREFIX))

    (MOVE-MARK MARK (WINDOW-DISPLAY-START WINDOW)))
```

The code for "End of Window" can be the same except that it calls WINDOW-DISPLAY-END.

You cannot move a window to a different position in the buffer by moving the display-start and display-end marks. For the techniques of moving windows, see Section 5.2.3 below.

5.2.2 The Window Point

In addition to the display-start and display-end marks, each window also contains a third mark called the window point. This mark is created at the time the window is created, and you can access it by means of the function WINDOW-POINT. The window point of a window always indicates a character position that is within the text contained in the window.

When the window is current, its window point is the same mark as the current buffer point. That is, the following form always evaluates to T:

```
(EQ (WINDOW-POINT (CURRENT-WINDOW))
    (CURRENT-BUFFER-POINT))
```

The screen cursor, which is always visible in the current window, tracks the position of this mark.

When the window is not current, its window point is not the same mark as the buffer point of the associated buffer. Instead, the window point indicates the last position occupied by the current buffer point when the window was current. If the window becomes current again, the buffer point (and therefore the cursor) move to the position indicated by the window point.

You can move the window point by means of any of the mark-moving functions described in Chapter 4. If you move the window point to a text position that is not within the window, the Editor moves the window so that it always contains the window point. For instance:

```
(BUFFER-START (CURRENT-BUFFER-POINT))
```

Or,

```
(BUFFER-START (WINDOW-POINT (NEXT-WINDOW T)))
```

Both these forms move the window point to the first text position in the window's associated buffer. If this position is not within the window, the window moves automatically so that it does contain this text position:

- In the first form, the cursor remains visible in the current window because the window content changes.

- In the second form, window content changes on the screen; if you immediately make this window current, the cursor appears at the new window-point position at the beginning of the buffer.

5.2.3 Moving a Window in the Buffer

When the text in a buffer is longer (i.e, has more lines) than a window opening onto it, the window can be moved forward or backward through the buffer. It appears that the text is moving past the window, but in the VAX LISP Editor it is actually the window that is moving.

As shown in the preceding section, you can cause a window to move within the buffer by operating on its window point. You can also operate directly on a window to alter the portion of a buffer's text that it opens onto. The two ways to move a window are:

- Scrolling, or moving line-by-line in the buffer

- Moving to a specified position in the buffer

5.2.3.1 **Scrolling** - The function SCROLL-WINDOW moves a specified window within its buffer by a specified number of rows. This action changes the text line that appears in the first row of the window. A positive count argument to SCROLL-WINDOW indicates the number of rows to move forward in the buffer; a negative count indicates backward movement.

For instance:

        (SCROLL-WINDOW (CURRENT-WINDOW) -20)

This form scrolls the current window backward through the text (text moves down on the screen) for 20 rows. If this action moves the window beyond the position indicated by the current buffer point, the Editor automatically moves that mark to a position within the new content of the window. The position of the updated mark is near the center of the window.

The window to be scrolled need not be the current window, of course. For example:

        (SCROLL-WINDOW (NEXT-WINDOW) 10)

This form scrolls the "next" window on the screen forward by 10 rows. If this action moves the window beyond the position indicated by its window point, then the Editor automatically moves that mark to indicate a position within the new window content (again, near the center of the window).

5.2.3.2 **Moving To A Specified Position** - You can also move a window to a specified position within its associated buffer. The function POSITION-WINDOW-TO-MARK moves a specified window to the line that contains a specified mark. That is, the line containing the mark is placed in the first row of the window.

For instance:

```
(POSITION-WINDOW-TO-MARK (CURRENT-WINDOW)
                         (REGION-START (BUFFER-REGION
                                        (CURRENT-BUFFER))))
```

This form moves the current window to the beginning of the current buffer. The Editor automatically updates the window point if necessary to keep it within the window.

5.2.4 Wrapping the Lines in a Window

A window can be narrower (fewer characters per line) than the text it opens onto. Windows always include the beginnings of lines; that is, the display-start mark is always at character position 0 of the line it indicates. Any lines that are longer than the width of the window are, by default, truncated on the right.

To view text in positions beyond the width of the window, you cannot move the window to the right. Instead, you set the window to "wrap" text lines onto one or more additional window rows.

The function WINDOW-LINES-WRAP-P takes a window and returns NIL if that window truncates lines or T if it wraps lines. Using SETF, you can change the behavior of a specified window:

```
(SETF (WINDOW-LINES-WRAP-P (CURRENT-WINDOW)) T)

(SETF (WINDOW-LINES-WRAP-P (CURRENT-WINDOW)) NIL)
```

The first form causes the current window to wrap lines that are wider than the window. The second form resets the current window to truncate lines.

The above examples alter a particular existing window. If you want to specify the line-handling behavior of newly created windows, you can reset the value of the Editor variable "Default Window Lines Wrap". Its possible values are T and NIL, which indicate wrapping and truncating, respectively:

```
(SETF (VARIABLE-VALUE "Default Window Lines Wrap") T)
```

This form causes all newly created windows to wrap lines unless otherwise specified.

Truncation and wrapping in a window are indicated by certain
characters that appear at the end of an affected line. The default
characters are an underlined > for truncation and an underlined < for
wrapping. You can change these characters for a specified window
using the functions WINDOW-TRUNCATE-CHAR and WINDOW-WRAP-CHAR. Both
these functions take a window and return a character, and both can be
used with SETF:

```
(SETF (WINDOW-TRUNCATE-CHAR WINDOW1) #\+)

(SETF (WINDOW-WRAP-CHAR WINDOW2) #\|)
```

Assuming that WINDOW1 is set to truncate, the first form establishes
an underlined + as the character that signals that a line has been
truncated. Assuming that WINDOW2 is set to wrap, the second form
establishes an underlined | to signal wrapping.

To change the truncation or wrapping characters in newly created
windows, you can reset the Editor variables "Default Window Truncate
Char" and "Default Window Wrap Char". For example:

```
(SETF (VARIABLE-VALUE "Default Window Truncate Char") #\+)

(SETF (VARIABLE-VALUE "Default window Wrap Char") #\|)
```

The underlining is a special video rendition of the selected
characters; you cannot change this feature. On terminals without
advanced video capabilities, the characters appear in reverse video
instead of underlined.

5.3 WINDOW APPEARANCE

Window appearance refers to the "look" of a window when it is
displayed: its video rendition and whether it is bordered and
labeled. All these features are included in the window object itself.
You can change a window's appearance by using the functions and
variables introduced in this section.

Most windows that the Editor creates are shown with no special video
rendition -- they share the video setting (dark-on-light or
light-on-dark) of the terminal or other display device. The window
onto the "Help" buffer, however, is shown in bold. Depending on the
video capabilities of your display device, you can specify that a
window be shown in reverse video (the reverse of terminal setting) or
that the text in the window appear bold, underlined, or blinking.

You can see all these video options on a VT200-series terminal, an AI
VAXstation, and on VT100-series terminals with the Advanced Video
Option. For the video rendition capabilities of foreign terminals
that are supported by the VAX LISP Editor, consult your terminal
manual.

You can also specify the video rendition of a region of text. The special rendition of a region can be either:

- Relative to the window that contains the region -- for instance, bold if the window is not bold and vice versa

- Absolute -- for instance, always bold or always not bold, regardless of the window rendition

Finally, you can specify whether a window is to have borders and a label when it is displayed. You can also determine the content of a window's label, the label's position on the window, and the label's video rendition.

This section introduces the techniques of:

- Altering window rendition

- Making highlight regions

- Operating on window labels and borders

5.3.1 Altering Window Rendition

The function WINDOW-RENDITION takes a window and returns a keyword or a list of keywords that define the video characteristics of that window when it is displayed. The keywords are :NORMAL, :BLINK, :BOLD, :REVERSE, and :UNDERLINE.

You can use SETF to change the rendition of a specified window to one or more of the possible values. For instance:

        (SETF (WINDOW-RENDITION (CURRENT-WINDOW)) :UNDERLINE)

Or,

        (SETF (WINDOW-RENDITION (NEXT-WINDOW :FLOATING))
              '(:REVERSE :BLINK))

The first form changes the rendition of the current window to underlined. The second form alters the "next" floating window on the screen to reverse video (the reverse of the terminal setting) with blinking.

You can also specify the default window rendition features of newly created windows, including those that the Editor creates automatically. To do so, you set the value of the Editor variable "Default Window Rendition" to the desired keyword or list of keywords:

```
        (SETF (VARIABLE-VALUE "Default Window Rendition"
                               '(:BUFFER "Help"))
              :REVERSE)
```

This buffer-local binding causes all newly created windows onto the "Help" buffer to have the rendition value :REVERSE unless otherwise specified. (The global binding of this variable in the Editor as provided is :NORMAL.)

Recall that removing a window from the screen does not delete the window object. If a window onto the "Help" buffer already exists, changing the value of "Default Window Rendition" does not affect the rendition of that window.

5.3.2 Making Highlight Regions

Sometimes you might want to alter the rendition of a particular block of text in a window, rather than the entire window. For example, the select regions that the Editor makes in "EDT Emulation" and "EMACS" styles are shown in reverse video (the reverse of the window).

To alter the rendition of a block of contiguous text, you use the function MAKE-HIGHLIGHT-REGION. This function is similar to MAKE-REGION (described in Chapter 4) except that it allows you to specify the video rendition that the region will have when a window containing it is displayed. Highlight regions can be used and treated like any other Editor region, and all the region-manipulating functions operate on them.

The format of MAKE-HIGHLIGHT-REGION is:

    MAKE-HIGHLIGHT-REGION start end &OPTIONAL set complement

Like MAKE-REGION, MAKE-HIGHLIGHT-REGION takes two marks that indicate the text positions where the region begins and ends. If you do not supply optional arguments, the function makes a region with no special video features.

The optional set and complement arguments specify the rendition feature or features that you want the region to have. They can be any of the keywords :BOLD, :BLINK, :REVERSE, or :UNDERLINE, or a list of these keywords. (The default for both is NIL.) In deciding whether to provide a set argument, a complement argument, or both, you need to consider the desired rendition of the region in relation to the rendition of the window where the region will be displayed.

You can think of a set argument with no complement argument as turning "on" the specified feature in a highlight region. The region will have that video feature regardless of the rendition of the window that contains the region.

For instance:

        (MAKE-HIGHLIGHT-REGION MARK1 MARK2 :REVERSE)

This form makes a reverse-video region of the text between the two
specified marks.   If this region is displayed in a reverse-video
window, then no difference will be apparent between the region and the
other text in the window -- all text in the window will appear in
reverse video. If this region is displayed in a blinking window, then
all the text in the window will blink, including that in the highlight
region. (In the latter case, the region will be distinguished from
the other text by its reverse video.)

You can achieve finer control over the video rendition of highlight
regions by providing a complement argument, either alone or in
conjunction with a set argument.

You use the complement parameter alone if you want the region to
contrast with the window rendition on the specified feature or
features but to share the window's value for any other rendition
features. For instance:

        (MAKE-HIGHLIGHT-REGION (COPY-MARK (CURRENT-BUFFER-POINT)
                                          :RIGHT-INSERTING)
                               (CURRENT-BUFFER-POINT)
                               NIL :REVERSE)

This form is essentially the definition of the select regions that the
Editor makes.   The form makes a new right-inserting mark at the same
position as the current buffer point and then makes a highlight region
from that mark and the buffer point. If you then move the buffer
point, the text in the region between the two marks is always shown in
reverse video with respect to the window that contains the region.
That is, if the window is dark-on-light, the region is light-on-dark,
and   vice   versa.   The region shares any other special video
characteristics of the window -- bold, blink, or underline.   (Note
that moving one of the region-defining marks causes the display of the
region to track the mark's position.)

If, on the other hand, you want the highlight region not to share
specified rendition features that a window might happen to have -- in
this example, the bold, blink, and underline -- you use the set and
complement parameters in conjunction. You can think of the complement
argument as turning "off" a video feature that is turned "on"
elsewhere -- either in the set argument or in the window that contains
the region.

For instance:

        (MAKE-HIGHLIGHT-REGION MARK1 MARK2
                               '(:BOLD :BLINK :UNDERLINE)
                               '(:BOLD :BLINK :UNDERLINE :REVERSE))

In this form, the :REVERSE value of the region "complements" the value of the window for this feature, as in the form above. The other three features are turned "on" by the set argument, but then turned "off" by the *complement* argument. They remain "off" regardless of window rendition; that is, if this region is displayed in a blinking window, the text in the highlight region does not blink.

Like any regions, highlight regions can overlap or one can be contained within another. The effect of overlapping on the video rendition of the text shared between the regions is, however, unpredictable.

To remove the highlighting of a region, you use the function REMOVE-HIGHLIGHT-REGION. This function takes a highlight region and deletes the region object. The text in the region is not affected by this operation, but its special video rendition is removed.

If you use the normal region-deleting functions, DELETE-REGION and DELETE-AND-SAVE-REGION with a highlight region, the text is removed from the region but the highlight region remains. If you insert new text into the region, it will be displayed with the specified rendition features of the region.

5.3.3 Operations on Window Labels and Borders

Editor windows can have borders on all four sides and a label on one of these borders. A border is a solid line that surrounds the text-displaying area of the window. The border occupies the screen rows above and below the window's text area and the screen columns to the right and left of the text area. A window label is a string of text that overlays part or all of one of the window borders.

Most windows that the Editor makes have borders and labels. However, depending on window size, window type, and the number of windows on the screen at once, one or more of the borders -- including the one with the label -- may "spill off" the display area or be obscured by another window. Nonvisible borders and labels still exist as part of the window object, however, and they might be made visible under other display circumstances. In contrast, the prompting window, which appears near the bottom of the screen, has no borders and no label.

This section introduces the functions and Editor variables that enable you to perform operations on window borders and labels. Section 5.4.2.4 below identifies the circumstances under which a border or label may be obscured when a window is displayed.

Some operations you can perform on window borders and labels are:

- Adding and removing borders and labels

- Specifying label content

- Specifying label position

- Specifying label rendition

5.3.3.1 **Borders, Labels, And Label Content** - The function WINDOW-LABEL takes a window and returns either a string, a function, or NIL. The value returned indicates whether the specified window has borders, whether it has a label, and, if it has a label, what text that label contains.

You can use SETF with WINDOW-LABEL to alter any of these features for a specified window. That is, you can add or remove borders, add or remove a label, and specify label content for an existing window. The meaning of each of possible return values of WINDOW-LABEL is shown in the following examples.

If you supply the value NIL, the specified window then has no borders and no label:

```
(SETF (WINDOW-LABEL WINDOW1) NIL)      ; WINDOW1 has no borders
                                       ; and no label.
```

If you supply a null string (""), the window then has borders but no label:

```
(SETF (WINDOW-LABEL WINDOW2) "")       ; WINDOW2 has borders but
                                       ; no label.
```

If you specify label content, then the window has borders and a label with the content specified. One way to specify label content is to specify the actual string that you want to label to contain:

```
(SETF (WINDOW-LABEL WINDOW3) "String") ; WINDOW3 has borders and
                                       ; a label that contains
                                       ; the specified string.
```

It is usually more useful, however, to have window label contents vary according to the buffer the window opens onto. The label can say, for instance, the name of the file or function being edited in the buffer, and it can list the styles active in the buffer. To achieve this, you specify a function that returns a string; the string becomes the window label.

For instance, the following DEFUN form defines a simple function named LABELER, which returns a string. LABELER is then used with SETF and WINDOW-LABEL in a form corresponding to those shown above:

```
(DEFUN LABELER (WINDOW)
  (LET ((BUFFER (WINDOW-BUFFER WINDOW)))
    (FORMAT NIL "LISP EDITOR ~A"
      (BUFFER-NAME BUFFER))))

(SETF (WINDOW-LABEL WINDOW4) #'LABELER)
```

LABELER invokes WINDOW-BUFFER to access the buffer object associated with the specified window and BUFFER-NAME to find the name of that buffer. LABELER returns a string that looks like "LISP EDITOR Name-of-Buffer". The SETF form labels WINDOW4 with the string returned by LABELER for that window.

The above examples all deal with alterations to a single existing window. By resetting the value of the Editor variable "Default Window Label", you can make corresponding specifications for newly created windows. For example:

```
(SETF (VARIABLE-VALUE "Default Window Label" :GLOBAL) 'LABELER)
```

The possible values of "Default Window Label" -- NIL, a null string, a string, or a function -- have the same meanings that they have as return values of WINDOW-LABEL. (Note that the function LABELER in this example is set to the value slot of the variable.) In the Editor as provided, the value of this variable is set to different functions in "EDT Emulation" style and "EMACS" style.

5.3.3.2 **Label Position** - For a given window that has a label, you can specify which border the label is on and where on the border the label is placed. The relevant functions are WINDOW-LABEL-EDGE and WINDOW-LABEL-OFFSET.

In deciding where to place window labels, you need to consider whether the border you choose will be visible when the window is displayed and, if so, whether the entire label will be visible on the border. For instance, the top borders of anchored windows are never visible; a label placed on that border can never be seen. Or, a floating window that spills off the screen on the right may "lose" the end of a label that is placed on its top or bottom border. These considerations are outlined in the Section 5.4 on display management.

WINDOW-LABEL-EDGE takes a window and returns the border on which that window's label appears. The possible values are :TOP, :BOTTOM, :LEFT, and :RIGHT. You can use SETF with this function to alter the placement of the label:

```
(SETF (WINDOW-LABEL-EDGE WINDOW1) :TOP)
```

Assuming that WINDOW1 has a label, this form places the label on the top border of WINDOW1.

The function WINDOW-LABEL-OFFSET, used with SETF, allows you to specify where on a border the label is to appear. The value NIL causes the label to be centered on the border; a nonnegative integer value indicates how many character positions the label is offset from the beginning of the border.

For instance, to center the label of WINDOW1 (which the previous example placed on that window's top border), you would write:

```
(SETF (WINDOW-LABEL-OFFSET WINDOW1) NIL)
```

To place a label flush left (if it is on a top or bottom border) or to make the label begin at the top of a side border, you would write:

```
(SETF (WINDOW-LABEL-OFFSET WINDOW) 0)
```

The above examples all deal with label placement in a single existing window. To change the default label placement in windows created in the future, you use the Editor variables "Default Window Label Edge" and "Default Window Label Offset". The global bindings of these variables in the Editor as provided are :BOTTOM and NIL, respectively. Thus, window labels appear centered on the bottom border of a window unless otherwise specified.

5.3.3.3 **Label Rendition** - For a given window that has a label, you can retrieve and alter the video rendition of that label. The function WINDOW-LABEL-RENDITION takes a window and returns one of the keywords :NORMAL, :REVERSE, :BLINK, :BOLD, or :UNDERLINE, or a list of those keywords.

The rendition of a window label is an absolute value that is not relative to the rendition of the window itself. For instance, the rendition of a reverse-video label is always the reverse of the terminal setting (light-on-dark or dark-on-light), regardless of whether the window rendition is normal or reverse. A window label set to blink will always blink, regardless of whether the window also blinks.

WINDOW-LABEL-RENDITION is acceptable to SETF:

```
(SETF (WINDOW-LABEL-RENDITION WINDOW) :UNDERLINE)
```

This form underlines the label of WINDOW.

To change the default rendition of the labels of newly created windows, you reset the value of the Editor variable "Default Window Label Rendition". The possible values are any of the keywords listed above, or a list of those keywords. The global binding in the Editor as provided is :REVERSE.

5.4 DISPLAY MANAGEMENT

Once a window exists as an Editor object, you can cause it to become visible on the screen. You can also remove a window from the screen without destroying the window object, and you can redisplay that window at any time. Each window object contains information concerning its size and the screen position it occupies when it is displayed.

However, the Editor's display manager retains control over many display-related decisions. The Editor guarantees, for instance, that the display area is always filled, and that anchored windows do not obscure each other's text content. The display manager will reposition, resize, and even remove some windows from the screen in order to meet these requirements.

Within the constraints set by the display manager, you have considerable freedom to determine the total appearance of the screen: the size of the display area, the number of windows displayed, and the size and screen positions of some individual windows.

This section introduces the following sets of techniques, each in conjunction with the constraints set by the Editor's automatic display management:

- Operations on the display area -- including the information area and the prompting window

- Window types and their behavior -- visibility, size, and screen position as they relate to a window's display type

- Displaying windows and removing windows from the display

5.4.1 The Display Area

The Editor's display area is the total space available on your display device for showing Editor windows and the information area. On VT100- and VT200-series terminals, the display area is, by default, the full terminal screen. Both these terminal screens are 24 rows in height, and both permit screen widths of 80 or 132 columns.*

The AI VAXstation permits an Editor display area of up to 66 rows by 167 columns. The Editor display area provided by default on the AI VAXstation is 50 rows by 80 columns.

You can think of the display area as an x-y coordinate system. You use these coordinates to specify the screen positions at which windows are displayed. Both the columns (x coordinate) and the rows (y coordinate) are numbered from the upper left corner beginning with 1 (see Figure 5-1).
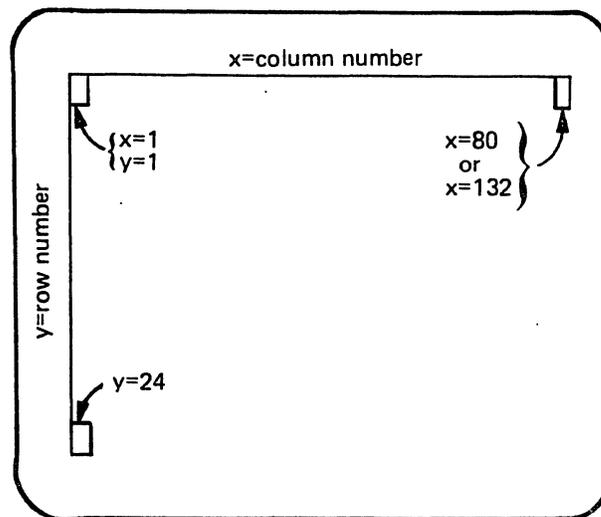
MLO-253-86

Figure 5-1: Display Area Coordinates

You have some latitude to alter the dimensions of the Editor's display area, and thus the total screen area available for Editor-related displays. Within this total area, the Editor always reserves some space for the information area and for the prompting window. The space that remains in the display area after the prompting window and the information area are accounted for is the total space available for displaying other windows.

This section introduces the following concepts and techniques:

- Display area dimensions - retrieving and altering the height and width of the display area

- The reserved display area - operating on the information area and the prompting window

- The available display area - displaying and removing other Editor windows

5.4.1.1 **Display Area Dimensions** - The functions SCREEN-HEIGHT and SCREEN-WIDTH return integers that are the number of rows and columns, respectively, in the display area. The dimensions returned are not necessarily those of the screen. By default, the display area occupies the full screen on DIGITAL terminals, but you can use SETF with SCREEN-HEIGHT and SCREEN-WIDTH to alter either dimension:

        (SETF (SCREEN-HEIGHT) 20)

        (SETF (SCREEN-WIDTH) 60)

If you execute these two forms, the display area is reduced to the dimensions shown in Figure 5-2:
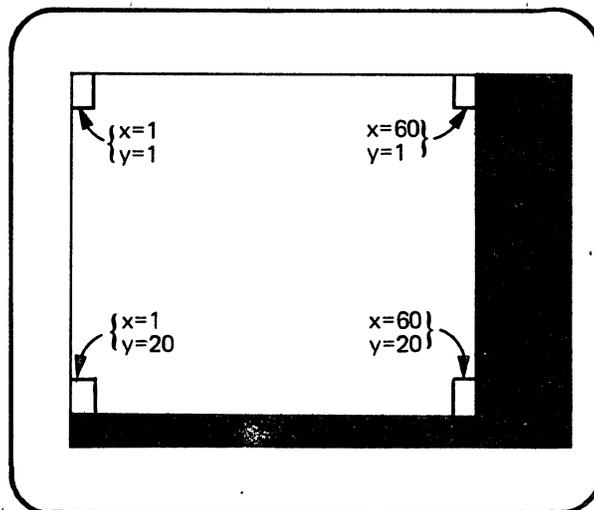
MLO-254-86

Figure 5-2: Altered Display Area Dimensions

The coordinate numbering does not change when you change the dimensions of the display area. The rows and columns that become unavailable are those at the bottom and on the right. The upper left corner of the display area still corresponds to the upper left corner of the screen, and the upper left position is still designated as 1,1.

You cannot make the display area larger than the screen. If you supply a value in either of the above SETF forms that is larger than the maximum screen dimension, the display area dimension is altered to the maximum screen dimension.

---

* For screen sizes of foreign (non-DIGITAL) terminals supported by the VAX LISP Editor, consult your terminal manual.

You can, however, use SETF with SCREEN-WIDTH to alter the width setting of the screen on DIGITAL terminals; the width setting can be either 80 columns ("normal") or 132 columns ("wide"). Widening the screen makes more columns available for the Editor's display area. Note that if your terminal does not have the Advanced Video Option, widening the screen limits screen height to 12 rows.

Suppose that your terminal is set to the "normal" width of 80 columns. Then execute:

      (SETF (SCREEN-WIDTH) 120)

This form resets VT100- and VT200-series terminals to "wide" width -- 132 columns; at the same time, it makes 120 of those columns available as the display area. The 12 right-most columns of the screen will be blank.

On the AI VAXstation, this form simply sets to display area width to 120 columns out of the 167 columns available on the screen.

With the AI VAXstation, you can adjust display area dimensions before the Editor is started (using SETF with SCREEN-HEIGHT or SCREEN-WIDTH). When you start the Editor, its display area will be the size you specified. Performing such operations on a VT100- or VT200-series terminal before the Editor is started produces no effect.

5.4.1.2 **The Reserved Display Area** - The Editor always reserves some part of the total display area for the information area and the prompting window. The information area is always at the bottom of the display area, and the prompting window is always just above the information area. By default, both these areas are the full width of the display area, and each is 1 row in height.

The information area is managed directly by the Editor's display manager, which uses it to display error messages and to report on Editor activity. The information area is not an Editor window, and none of the window-related functions operate on it. You cannot delete the information area, and no Editor windows can overlap it. It is, therefore, an area of guaranteed visibility within the display area.

You can, however, increase the space alloted for the information area, and you can direct output to it.

The function INFORMATION-AREA-HEIGHT returns the height (in screen rows) of the information area. Using SETF, you can increase the height to more than 1 row, but you cannot decrease it to 0 rows.

To direct output to the information area, you can use the LISP global variable *INFORMATION-AREA-OUTPUT-STREAM*. This variable is bound to an output stream that is directed to the information area. You can

use the variable as the *stream* argument to any of the LISP functions
that take a stream, such as WRITE-STRING, FORMAT, or PRINC. For
example:

```
(WRITE-STRING "Operation completed. "
              *INFORMATION-AREA-OUTPUT-STREAM*)
```

Some other functions that operate on the information area are
CLEAR-INFORMATION-AREA, which removes any current text, and
EDITOR-ERROR, which directs error messages to the information area.
EDITOR-ERROR is discussed in Chapter 2 of this manual.

The prompting window is also permanently displayed. The full
prompting area consists of a small section of the screen where prompts
are displayed, followed on the same row by an Editor window where user
input is displayed. By default, the full prompting area is 1 row in
height and the full width of the display area.

Although the prompting window is an Editor window, some of the
window-related functions do not operate on it. The display manager
ignores any attempt to remove this window from the screen, to
reposition it on the screen, or to overlap it with another window.
Thus, the prompting area is also an area of guaranteed visibility.

You can, however, alter the appearance of the prompting window by
using the functions described in the previous section. (The screen
area where prompts are displayed is not affected by operations on the
prompting window.)

To alter the video rendition of the prompting window, you might write:

```
(SETF (WINDOW-RENDITION
        (FIRST (BUFFER-WINDOWS
                (FIND-BUFFER "General Prompting"))))
      :REVERSE)
```

This form alters the rendition of the prompting window to reverse
video. The rendition of the area where prompts are displayed can be
altered by resetting the Editor variables "Prompt Rendition Set" and
"Prompt Rendition Complement" (see Part III).

You can also alter the dimensions of the prompting window, in the same
way that you resize other windows. See Section 5.4.2.2 for the
techniques of resizing windows.

The screen space -- number of rows -- that you have allotted to the
prompting window and the information area is always reserved for them,
and thus not available for displaying other windows.

5.4.1.3 **The Available Display Area** - Whatever space is not reserved for the information area and the prompting window is the "available" display area. This is the area you use to display Editor windows, and its dimensions constrain your decisions on sizing and positioning windows if you do not want them to overlap one another or overflow the screen.

In the following sections on displaying, sizing, and positioning windows, the term *display area* should be taken to mean the available display area.

5.4.2 Window Types and Their Behavior

There are two types of Editor windows: anchored and floating. The Editor normally provides anchored windows for ordinary text editing. Floating windows are used for displaying information or for other special purposes. For instance, the windows onto the "Help" and "General Prompting" buffers are floating windows.

The content and appearance of a window are not affected by the display type of that window. The distinction between the two types lies in how they are treated in display management. This section outlines the effect of window type on three aspects of display management behavior:

- Whether a window overlaps or is overlapped by other windows

- Sizing or resizing a visible window

- Positioning or repositioning a window in the display area

You can access or change the type of a specified window. The function WINDOW-TYPE takes a window and returns a keyword, which can be either :ANCHORED or :FLOATING. Using SETF, you can change the window's type:

        (SETF (WINDOW-TYPE WINDOW) :FLOATING)

The Editor variable "Default Window Type", which specifies the default type of newly created windows, can be set to either of these keywords. In the Editor as provided, the default window type is :ANCHORED.

In the examples that follow, the variable ANCHORED-WINDOW is assumed to be bound to an anchored window, and FLOATING-WINDOW is assumed to be bound to a floating window.

5.4.2.1 **Display Behavior By Window Type** - The major difference between anchored and floating windows lies in whether they can overlap, or be overlapped, by other windows:

- A floating window can overlap or completely obscure any other window in the available display area, either anchored or floating.

- An anchored window never obscures the text area of another window, either anchored or floating.

When a floating window is displayed, it appears in the size and at the screen position contained in the window object. Any other windows that occupy any part of that space are overlapped.

If more than one anchored window is displayed at once, the display manager moves and resizes them as necessary so that no text is obscured. The size and screen position contained in the window object are ignored.

The Editor's display rules concerning window size and screen position follow from this basic rule concerning window overlaps. Thus, automatic display management relates to anchored windows only; floating windows, which need not avoid overlaps, are under the user's control and not subject to automatic resizing and repositioning.

5.4.2.2 **Window Size And Display Behavior** - Each window object has height (number of rows) and width (number of columns). You can retrieve and alter these features in any window, but it is only in floating windows that the features have significance.

You can access window dimensions with the functions WINDOW-HEIGHT and WINDOW-WIDTH, and you can use SETF to alter these dimensions:

    (SETF (WINDOW-HEIGHT FLOATING-WINDOW) 12)

    (SETF (WINDOW-WIDTH FLOATING-WINDOW) 40)

These forms alter the specified floating window to be 12 rows in height and 40 columns wide. The dimensions of a window refer to the text area only; if the window has borders, the borders occupy 2 additional rows and 2 additional columns outside the text area.

The minimum size for a floating window is 1 row by 2 columns. If you attempt to alter either dimension to less than the minimum, an error is signaled. If you make either dimension larger than the available display area, the window will overflow the display area to the right or to the bottom.

For anchored windows, the dimensions contained in a window object are ignored by the Editor's display manager. An anchored window always occupies the full width of the display area. The window's height depends on how many anchored windows are visible at once.

If an anchored window is the only anchored window displayed, its text area occupies the full height of the available display area, minus 1 row for the window's bottom border, if any. (The top and side borders of anchored windows are always obscured.) You cannot adjust the height of an anchored window when it is the only anchored window on the screen.

If two or more anchored windows are displayed at once, the Editor automatically adjusts their heights to be about equal. You can override this adjustment for a specified window by means of the function ALTER-WINDOW-HEIGHT. This function takes a window and a positive or negative integer that is the number of rows by which to adjust the window's height. (The argument window need not be visible at the time the form is evaluated, and it can be either an anchored or a floating window.) The integer argument specifies a change in size, rather than an absolute size as does the return value of WINDOW-HEIGHT.

        (ALTER-WINDOW-HEIGHT ANCHORED-WINDOW -2)

This form makes the specified anchored window, when displayed with at least one other anchored window, two rows shorter than the height determined by the display manager. The display manager adjusts the height of any other visible anchored windows to accommodate the altered height of the argument window and still fill the display area.

The minimum height to which you can adjust an anchored window is 1 row of text area. (If the window is bordered, a second screen row is reserved for the bottom border.) The maximum height is determined by the rule that no anchored window can obscure another's text. The argument window cannot be made so high that another visible anchored window would be reduced to less than 1 text row (or two screen rows if bordered). If the integer argument to ALTER-WINDOW-HEIGHT violates these rules, then the Editor adjusts the argument window only as much as possible.

5.4.2.3 **Window Position And Display Behavior** - Editor windows contain information on the screen position at which they are to be displayed. Display position is specified as the x-y coordinates of the screen position occupied by the top-left character in the window. (Recall that the x-y coordinate numbering of the screen and the display area begin at the same point in the upper left corner.)

You can retrieve the screen position of a specified window by means of the functions WINDOW-DISPLAY-COLUMN and WINDOW-DISPLAY-ROW. (The argument window need not be currently visible.)

        (WINDOW-DISPLAY-COLUMN WINDOW)

        (WINDOW-DISPLAY-ROW WINDOW)

If these forms return 10 and 5, respectively, then the upper left corner of the window's text area is displayed at column 10, row 5 of the display area (see Figure 5-3). If the window is bordered, the borders lie outside this position.
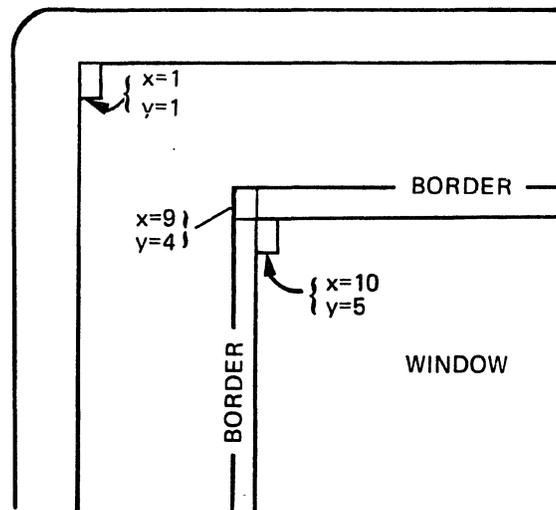
MLO-255-86

Figure 5-3: A Window Display Position

As with window size, the values that indicate window display position are significant only for floating windows. Anchored windows are always displayed beginning in column 1. The display row position of an anchored window is determined by the display manager.

You can alter the display position of a floating window by means of the function MOVE-WINDOW. MOVE-WINDOW takes a window and a row and column to which to move the upper left corner of that window's text area:

        (MOVE-WINDOW FLOATING-WINDOW 8 25)

This form alters the floating window's values for display row and display column to those specified. If the window is displayed, it moves to that position; if the window is not displayed, it appears at that position when it is next displayed.

An example using MOVE-WINDOW is a function MY-MOVE-WINDOW-VERTICALLY. This function moves a window's display row by the number of rows specified, or by fewer rows if the specified offset value would result in spilling the window (or its border) off the display area. Its code is:

```
(DEFUN MY-MOVE-WINDOW-VERTICALLY (WINDOW OFFSET)

   ;; Compute the new display row.
   (LET ((NEW-ROW
             (+ (WINDOW-DISPLAY-ROW WINDOW) OFFSET)))

      ;; Move the window to the display row, or as far as possible
      ;; without overflowing.  Retain the window's display column.
      (MOVE-WINDOW WINDOW
             (MIN (MAX NEW-ROW 1)
                    (- (SCREEN-HEIGHT) (WINDOW-HEIGHT WINDOW)
                       (IF (WINDOW-LABEL WINDOW)
                           1
                           0)))
                    (WINDOW-DISPLAY-COLUMN WINDOW))))
```

To use this function in an Editor command, you could write:

```
(DEFINE-COMMAND (MY-MOVE-WINDOW-UP-COMMAND
                   :DISPLAY-NAME "My Move Window Up")
             (PREFIX)

   "  Moves the current window up one or more rows, without
   spilling it off the screen.  Cannot be used with
   anchored windows."

   (IF (EQ (WINDOW-TYPE (CURRENT-WINDOW) :FLOATING))
        (MY-MOVE-WINDOW-VERTICALLY (CURRENT-WINDOW)
                                   (- (OR PREFIX 1)))
        (ATTENTION)))
```

You could write a similar function to move a window horizontally, and similar commands to move a window down, right, and left.

5.4.2.4 **Window Borders And Display Behavior** - The Editor's display rules concerning the overlapping of window text areas do not apply to window borders. A bordered window can sometimes be sized or positioned in a way that obscures one or more of its borders.

For anchored windows, the top and side borders are always nonvisible. If an anchored window with a border is the only anchored window displayed, its text area fills the available display area, leaving 1 row for its bottom border. Its top and side borders overflow the display area. For this reason, you should always place the labels of anchored windows on their bottom borders.

If two or more anchored windows with borders are displayed at once, the bottom border of one window obscures the top border of the window displayed below it. The bottom border of an anchored window never overflows the display area, and it cannot be obscured by another anchored window. It can, however, be obscured by a floating window.

5-27

For floating windows, the borders are more often visible since
floating windows can be sized and positioned well within the display
area. However, if a floating window equals or exceeds the size of the
display area, then any or all of its borders can spill off. The
"Help" window, for instance, is the full width of the display area but
shorter in height. Therefore, its top and bottom borders are visible,
but its side borders are not visible. If a floating window,
regardless of its size, is positioned on the screen in such a way that
it overflows the display area, then the border of the affected edge
cannot be seen.

5.4.3 Displaying and Removing Windows

Three functions enable you to display and remove windows from the
screen. These functions are:

- SHOW-WINDOW

- PUSH-WINDOW

- REMOVE-WINDOW

The behavior of these functions varies with the display type of the
argument window and of other visible windows. This section introduces
these behavior variations; you can find further information in the
description of each function in Part III of this manual.

5.4.3.1 Using SHOW-WINDOW - SHOW-WINDOW takes a window and displays
it on the screen. Its format is:

SHOW-WINDOW *window* &OPTIONAL *row column*

If the window is a floating window, you can supply an optional row and
column at which its upper left character will appear. If you do not
specify a position, the window is placed at the position contained in
the window object.

A newly displayed floating window obscures any other window that is
currently displayed in the same position. If the argument (floating)
window is already displayed but is obscured by one or more windows,
SHOW-WINDOW redisplays it "on top of" the obscuring window(s).

If the argument window is an anchored window, its position is
determined by the display manager. If you supply *row* and *column*
arguments, they are ignored.

You can continue to add anchored windows to the screen with
SHOW-WINDOW up to the number that is the value of the Editor variable

"Anchored Window Show Limit". For instance, if the value is 2 (the global default), then you can show 2 anchored windows on the screen at a given time. Adding a third anchored window with SHOW-WINDOW causes the least recently used anchored window to be removed.

A newly displayed anchored window appears at the bottom of the screen, and any other anchored windows that remain on the screen are moved up. All the visible anchored windows are resized so that they are about equal in height. If a window is removed to accommodate the newly displayed window, the new window is made the same size as the window that was removed.

5.4.3.2 **Using PUSH-WINDOW** - PUSH-WINDOW is like SHOW-WINDOW, except that it does not automatically remove anchored windows when their number exceeds the value of "Anchored Window Show Limit". Also, PUSH-WINDOW takes two optional arguments that enable you to override some features of the Editor's automatic treatment of anchored windows.

The format of PUSH-WINDOW is:

PUSH-WINDOW *window* &OPTIONAL *companion insert-above*

If the argument window is floating, PUSH-WINDOW has the same effect as SHOW-WINDOW, except that you cannot specify a display position. The window appears at the position contained in the window object. If you supply optional arguments, they are ignored.

If the argument window is anchored, PUSH-WINDOW adds it to the display without removing any previously displayed anchored windows. The Editor resizes all the visible anchored windows to make them about equal in height.

The optional arguments enable you to specify the position of a newly displayed anchored window in relation to a visible anchored window. If you specify a *companion* argument -- a visible anchored window -- the newly displayed window appears just below the companion. If you supply both a *companion* argument and an *insert-above* argument of T, the new window appears just above the companion window.

5.4.3.3 **Using REMOVE-WINDOW** - REMOVE-WINDOW removes a window from the display. Its format is:

REMOVE-WINDOW *window* &OPTIONAL *new-current*

If the argument window is floating, REMOVE-WINDOW has no effect on the remaining visible windows. If the argument window is anchored, the Editor automatically resizes and repositions the remaining anchored windows so that they fill the available display area.

If the window being removed is the current window, you can supply a
new-current argument to specify the window that is to become current.
If you do not supply a *new-current* argument, then the Editor invokes
NEXT-WINDOW (with argument T) to identify the window that becomes
current.

By using REMOVE-WINDOW repeatedly, you can remove from the available
display area all but one of the visible windows. You cannot empty the
available display area completely, however; the one window that will
remain opens onto the buffer bound to the variable
EDITOR-DEFAULT-BUFFER. If you have not bound a buffer to this
variable, the Editor displays a window onto the "Basic Introduction"
buffer when it has nothing else to show.

5.5 MAKING AND DELETING WINDOWS

Most of the windows that the Editor displays are made by the Editor as
a result of executing certain commands. You can, however, make a
window directly in LISP code, and you can specify all the features you
want that window to have. If the new window is an anchored window,
its specified size and screen position are ignored by the Editor's
display manager.

To create a new window, you call the function MAKE-WINDOW. Its format
is:

```
MAKE-WINDOW buffer-or-mark &KEY :HEIGHT :WIDTH
                               :DISPLAY-ROW :DISPLAY-COLUMN
                               :TYPE
                               :LINES-WRAP
                               :LABEL
```

The one required argument to MAKE-WINDOW is *buffer-or-mark*. This
argument indicates the text content of the window; that is, it
indicates the buffer onto which the window opens, as well as the text
position within that buffer where the window begins.

- If the argument is a buffer, the window opens onto that
  buffer, beginning with the line that contains the buffer
  point.

- If the argument is a mark, the window opens onto the buffer
  that contains that mark, beginning with the line that the mark
  indicates.

For instance:

    (MAKE-WINDOW (FIND-BUFFER "Help"))

Or,

    (MAKE-WINDOW (WINDOW-POINT (FIRST (BUFFER-WINDOWS "Buffer"))))

The first form makes a window onto the "Help" buffer, starting with
the line that contains the buffer point of that buffer. The second
form makes a window onto the buffer associated with another window,
beginning with the line indicated by the window point of the other
window.

Some of the keyword arguments to MAKE-WINDOW -- :TYPE, :LINES-WRAP,
and :LABEL -- have defaults that are the values of the corresponding
Editor variables. For instance, the default window type is :ANCHORED
-- the global value of the Editor variable "Default Window Type".

The keyword arguments that pertain to window size and screen position
all take integer values. These values are significant only for
floating windows. For anchored windows, any values you supply are
ignored by the display manager.

   ● :HEIGHT is the number of rows of text in the window, excluding
     borders. The default is the height of the available display
     area (minus one row if the window is bordered).

   ● :WIDTH is the number of columns of text in the window,
     excluding borders. The default is the value of the Editor
     variable "Default Window Width".

   ● :DISPLAY-ROW and :DISPLAY-COLUMN indicate the screen position
     of the upper-left corner of the window's text area (excluding
     borders) when the window is displayed. The defaults are 1 and
     1. You can override these values by supplying row and column
     arguments to SHOW-WINDOW.

To delete a window object, you call the function DELETE-WINDOW and
pass it a window argument. The window can be visible or not visible;
if it is visible, DELETE-WINDOW first removes it from the display and
then deletes it. When an Editor window is deleted, it is destroyed
and cannot be used again.

5.6 EXAMPLE OF WINDOW AND DISPLAY OPERATIONS

The following example illustrates the Editor objects that you can use
to create, display, and remove a window. The DEFINE-COMMAND form
implements a new command named "Clock", which displays a window that
contains the current date and time. The command obtains the current

date and time by calling the function FORMAT-CLOCK, whose code is shown afterward.

```
(DEFINE-COMMAND (CLOCK-COMMAND  :DISPLAY-NAME "Clock")
                (PREFIX)

  "  Displays the current date and time in a window."

  (DECLARE (IGNORE PREFIX))

  ;; Find or make a buffer named "Clock".
  (LET ((BUFFER (FIND-BUFFER "Clock")))
    (UNLESS BUFFER
      (SETF BUFFER
            (MAKE-BUFFER
                '(CLOCK-BUFFER :DISPLAY-NAME "Clock")
                :MAJOR-STYLE NIL :MINOR-STYLES NIL
                :VARIABLES NIL)))

    ;; Find or make a window onto the "Clock" buffer.
    (LET ((WINDOW (FIRST (BUFFER-WINDOWS BUFFER))))
      (UNLESS WINDOW
        (SETF WINDOW
              (MAKE-WINDOW BUFFER
                           :TYPE :FLOATING
                           :HEIGHT 2 :WIDTH 30
                           :LABEL "Clock"
                           :DISPLAY-ROW 2 :DISPLAY-COLUMN 48))
        (SETF (WINDOW-LABEL-EDGE WINDOW) :TOP)
        (SETF (WINDOW-LABEL-RENDITION WINDOW) :BLINK)
        (SETF (WINDOW-RENDITION WINDOW) :BOLD))

      ;; Delete any previous text in the "Clock" buffer.
      (DELETE-REGION (BUFFER-REGION BUFFER))

      ;; Insert the string returned by FORMAT-CLOCK onto
      ;; "Clock" at the buffer point.
      (INSERT-STRING (BUFFER-POINT BUFFER) (FORMAT-CLOCK))

      ;; Display the window.
      (SHOW-WINDOW WINDOW 2 48)

      ;; Force the window to reflect the current contents of
      ;; the buffer.
      (UPDATE-DISPLAY)

      ;; Leave the window on the screen for 3 seconds and
      ;; then remove it.
      (SLEEP 3.0)
      (REMOVE-WINDOW WINDOW))))
```

Note that there is a redundancy in this form: the window's display position is specified both in the MAKE-WINDOW form and in the SHOW-WINDOW form. You can choose either place to make this specification.

The function MAKE-BUFFER, which creates and returns a new buffer, is described in Chapter 6 and in Part III.

FORMAT-CLOCK, which returns a string containing the current day, date, and time, could be implemented as follows:

```
(DEFUN FORMAT-CLOCK ()

   "  Returns the current time of day and the current date."

     (MULTIPLE-VALUE-BIND (SECOND MINUTE HOUR DAY MONTH YEAR
                                   WEEK-DAY)
                          (GET-DECODED-TIME)

       (DECLARE (FIXNUM SECOND MINUTE HOUR DAY MONTH YEAR WEEK-DAY))

       (LET ((MONTHS '#( "Jan" "Feb" "Mar" "Apr" "May" "Jun"
                         "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"))
             (WEEK-DAYS '#( "Monday" "Tuesday" "Wednesday"
                            "Thursday" "Friday" "Saturday"
                            "Sunday"))
             (DISPLAY-HOUR (IF (= HOUR 12)
                               HOUR
                               (MOD HOUR 12))))

         (DECLARE (SIMPLE-VECTOR MONTHS WEEK-DAYS)
                  (FIXNUM DISPLAY-HOUR))

         (FORMAT NIL
                 "~A ~2D-~A-~4D~%~2D:~2,'0D:~2,'0D"
                 (SVREF WEEK-DAYS WEEK-DAY)
                 DAY
                 (SVREF MONTHS (1- MONTH))
                 YEAR HOUR MINUTE SECOND)))))
```

CHAPTER 6

OPERATIONS ON STYLES

Styles in the VAX LISP Editor act as sets of Editor capabilities that you can turn on and off in the buffers where you are editing. For instance, in "EDT Emulation" style, you use the same key sequences as with DIGITAL's EDT editor to execute similar Editor commands. If you prefer the behavior and key bindings of an EMACS-based editor, you can use the Editor's "EMACS" style instead in any or all buffers. When "VAX LISP" style is also active, the Editor recognizes LISP syntax, knows how to indent LISP code, can evaluate selected regions of code, and so on. (See *VAX LISP/VMS User's Guide*.)

In programming terms, a style is a named Editor object that serves as a binding context. A style object can contain bindings for:

- Editor variables

- Editor attributes

- Keyboard keys and pointer actions

The particular bindings of variables, attributes, keys, and pointer actions within a style are responsible for the Editor's distinctive behavior when that style is active.

The difference in key bindings from one style to another is obvious: the LINEFEED key, for instance, invokes "EDT Delete Previous Word" in "EDT Emulation" style, but in "VAX LISP" style, it invokes "New LISP Line." The differences in variable and attribute bindings are less obvious when you are using the Editor, but they are equally important in determining the Editor's behavior. For instance:

- When "EDT Emulation" style is active, the Editor "knows" in what direction it is to execute movement and search commands by the current value of the Editor variable "EDT Direction Mode" (:FORWARD or :BACKWARD). Outside of "EDT Emulation" style, this variable is unbound and you must provide needed directional information in some other way.

● When "VAX LISP" style is active, the Editor "knows" that a semicolon is the beginning of a LISP comment because this character has the value :COMMENT-DELIMITER for the Editor attribute "LISP Syntax". Outside of "VAX LISP" style, this attribute is unbound and the Editor does not recognize any characters as significant in LISP syntax.

The Editor's distinctive behavior in a style also arises from the particular commands that you normally invoke in that style. For instance, the command "EDT Delete Previous Word" differs slightly from the "Delete Previous Word" command bound in "EMACS", just as the text-deleting commands in DIGITAL's EDT differ slightly from the delete commands in EMACS editors. While commands themselves are not context-dependent, many commands are normally used only within a particular style:

● Commands are frequently invoked by means of key sequences or pointer actions, and these bindings are context-dependent (see Chapter 3).

● Many commands reference Editor variables and Editor attributes, which are also context-dependent. Such a command will behave differently where the context-dependent object is unbound or bound differently (see Chapter 2).

This chapter introduces several kinds of operations that you can perform on styles when you are customizing the VAX LISP Editor. You can:

● Choose the styles that are active in any or all buffers

● Modify or extend a DIGITAL-provided style

● Create a new Editor style

More information about styles can be found in Part II of this manual.

Recall that the symbols for DIGITAL-provided Editor objects must be referenced in the "EDITOR" package. For the methods of specifying named Editor objects, including styles, see Chapter 1.

6.1 ACTIVATING AND DEACTIVATING STYLES

For the bindings in a style to be visible in an interactive session, that style must be "active" in the current buffer. Styles are activated in each buffer at the time the buffer is created; you can later access and change the active styles in a buffer at any time.

A buffer can have zero or one major style and zero or more minor styles. There is no inherent difference in style objects that makes

them major or minor. The difference arises from the way a style is activated in a buffer. The difference between major and minor activation becomes significant when the Editor searches for the proper binding of a key sequence, a pointer action, a variable, or an attribute.

In searching for the proper binding for a context-dependent object, the Editor searches the currently active contexts in the following order:

- The current buffer

- The minor styles of the current buffer, if any, beginning with the most recently activated

- The major style, if any, of the current buffer

- The global Editor context

The Editor uses the first binding it encounters in this search for the object in question. If the object is bound in more than one of these contexts, then all but one of the bindings are inaccessible, or "shadowed". For instance, if you have "EDT Emulation" active as the major style and "VAX LISP" active as a minor style, the LINEFEED key will invoke "New LISP Line". The binding of that key in "EDT Emulation" style ("EDT Delete Previous Word") is shadowed.

Further information on context search and shadowing can be found in Part II of this manual and in the *VAX LISP/VMS Users Guide*.

This search order suggests that, in general, your major style should be a general-purpose style that determines a wide range of Editor capabilities -- how the Editor manipulates text, moves the cursor, manages the display, reads in and writes to files, and so on. The two DIGITAL-provided styles that are suitable for use as major styles are "EDT Emulation" and "EMACS".

A minor style is typically a more limited set of bindings that you use to alter some details of the major style in particular circumstances. "VAX LISP" style, for instance, enhances either "EDT Emulation" or "EMACS" to enable you to edit LISP code. The general-purpose style is activated as the major style so that it is the last style searched for bindings. The special-purpose style (added as a minor style) can add variations to the major style because it shadows the major style.

This section outlines the methods of activating styles in buffers:

- Activating styles in a newly created buffer

- Setting the Editor's default styles

- Accessing and altering the styles in an existing buffer

6.1.1 The Styles in a New Buffer

Most buffers are created automatically by the Editor whenever you begin to edit a file or function. You can also make buffers yourself in LISP code. In either case, the new buffer can be created with specified style(s) active.

The function MAKE-BUFFER takes a buffer-name and returns a new buffer and T (or NIL if a buffer of that name already exists). The optional keywords :MAJOR-STYLE and :MINOR-STYLES let you specify the styles that are to be active in the new buffer.

- The *name* argument can be a symbol or a list containing a symbol and a string argument to the keyword :DISPLAY-NAME. (This naming convention is the same for all named Editor objects; for further detail, see the discussion of naming Editor commands in Chapter 2.)

- The :MAJOR-STYLE argument can be a style specifier or NIL.

- The :MINOR-STYLES argument can be a list of style specifiers or NIL.

For example:

```
(MAKE-BUFFER '(MYBUFFER :DISPLAY-NAME "Mybuffer.lsp")
            :MAJOR-STYLE  "EDT Emulation"
            :MINOR-STYLES '("VAX LISP"))
```

This form creates a buffer named MYBUFFER, with the alternative specifier "Mybuffer.lsp". Its major style is "EDT Emulation" and its one minor style is "VAX LISP".

If you do not specify style arguments, the Editor supplies default values. If you want the buffer to have no minor styles (or no major style), you supply the argument NIL to the appropriate keyword. The defaults, and the techniques of changing them, are presented in the next section.

6.1.2 The Editor's Default Styles

The Editor supplies default values for the major and minor styles of a newly created buffer unless otherwise specified in the MAKE-BUFFER form. You can access and change these default values.

Note that changing a default value does not affect buffers that already exist. Only buffers that are created after the default has changed will have the new default styles active.

6.1.2.1 **The Default Major Style** - The Editor's default major style is stored as the value of the Editor variable "Default Major Style." In the Editor as provided, this value is "EDT Emulation."

You can use SETF to change the default major style:

        (SETF (VARIABLE-VALUE "Default Major Style") "EMACS")

Note that only the global value of this variable is used.

6.1.2.2 **The Default Minor Style(s)** - The Editor's default minor styles are stored as the value of the Editor variable "Default Minor Styles". The possible values are a list of style specifiers or NIL. In the Editor as provided, this variable is used only globally, and its value is NIL.

You can establish a default minor style by resetting the value of "Default Minor Styles". If, for instance, you want to retain "EDT Emulation" as the Editor's default major style but add "EMACS" as the default minor style, you could write:

        (SETF (VARIABLE-VALUE "Default Minor Styles") '("EMACS"))

Note, however, that if you had previously established a default minor style, the form as written would remove that style as the default and replace it with "EMACS". To add "EMACS" without removing the previous default, you could use PUSH.

        (PUSH "EMACS" (VARIABLE-VALUE "Default Minor Styles"))

This form adds "EMACS" to the front of the list of default minor styles. The minor styles are activated in a buffer in reverse order to their position in the list. That is, the first style in the list is the last activated and thus the first searched when the Editor conducts a context search. In this example, "EMACS" will shadow other minor styles (as well as the major style) that are active in the same buffer.

You can access and change the entire minor style list if you want to change the order of the elements or add another style somewhere other than to the front of the list. For instance, suppose you have established "VAX LISP" as the default minor style and you now want to add "EMACS". If you added "EMACS" with PUSH, it would shadow "VAX LISP". To have "VAX LISP" shadow "EMACS", you would write:

        (SETF (VARIABLE-VALUE "Default Minor Styles")
              '("VAX LISP" "EMACS"))

This form makes "VAX LISP" the last-activated (and therefore the first-searched) of the minor styles in any buffer that has the default minor styles.

6.1.2.3 **Default Minor Style(s) By Type Of Buffer** – If a minor style is a special-purpose style, you may want to have it active only in the buffers where the special capabilities are needed. For instance, "VAX LISP" style is activated automatically in buffers that are associated with LISP objects or with files of the filetype LSP.

If you want to activate a minor style in buffers associated with a LISP object, you reset the value of the Editor variable "Default LISP Object Minor Styles". The value is a list of style specifiers, such as:

        (SETF (VARIABLE-VALUE "Default LISP Object Minor Styles")
              '("My New Style" "VAX LISP"))

This form specifies that two minor styles, "My New Style" and "VAX LISP", are to be activated in any buffer that is associated with a LISP object. These styles will be searched in the order shown; they will be searched before any styles in the "Default Minor Styles" list.

If you want to activate a minor style in buffers associated with a specified type of file, you reset the value of the Editor variable "Default Filetype Minor Styles". The value is an association list of the form:

        ((FILETYPE-STRING . MINOR-STYLE-LIST) ...)

For instance, the initial value of this variable is:

        ("LSP" . "VAX LISP")

Again, minor styles specified by this variable are activated after any styles specified by "Default Minor Styles", and are therefore searched first.

6.1.2.4 **Example Of Activating Default Styles** – This section illustrates the activation of multiple default styles in several buffers. The search order, which is the reverse of the order of activation, is then shown for each buffer.

Suppose you have five styles to work with: "EDT Emulation", "EMACS", "VAX LISP", and two user-defined styles, "LISP Variation" and "FORTRAN". One way to set your default activation values is as follows:

```
(SETF (VARIABLE-VALUE "Default Major Style") "EDT Emulation")

(SETF (VARIABLE-VALUE "Default Minor Styles") '("EMACS"))

(SETF (VARIABLE-VALUE "Default LISP Object Minor Styles")
      '("VAX LISP"))

(SETF (VARIABLE-VALUE "Default Filetype Minor Styles")
      '(("LSP" . ("LISP Variation" "VAX LISP"))
        ("FOR" . "FORTRAN")))
```

In buffers that have the default styles, the search order is as follows:

In a buffer named "Myfile.txt":

1. "EMACS"
2. "EDT Emulation"

In a buffer named LISP-FUNCTION:

1. "VAX LISP"
2. "EMACS"
3. "EDT Emulation"

In a buffer named "Myfile.lsp":

1. "LISP Variation"
2. "VAX LISP"
3. "EMACS"
4. "EDT Emulation"

In a buffer named "Myfile.for":

1. "FORTRAN"
2. "EMACS"
3. "EDT Emulation"

6.1.3 The Styles in an Existing Buffer

You can access and change the styles in a specified buffer at any time.

Note that changing the active style(s) in one buffer has no effect on any other buffer.

6.1.3.1 **A Buffer's Major Style** - The function BUFFER-MAJOR-STYLE
takes a buffer specifier and returns the major style of that buffer
(or NIL if the buffer has no major style). You can use SETF with this
function to change the major style active in a buffer:

        (SETF (BUFFER-MAJOR-STYLE "Mybuffer.txt") "EMACS")

This form deactivates the major style, if any, of "Mybuffer.txt" and
activates "EMACS" instead.

6.1.3.2 **A Buffer's Minor Style(s)** - You can also access and alter the
minor style or styles active in a specified buffer.

To determine whether a specified buffer has minor styles active, you
can use the function BUFFER-MINOR-STYLE-LIST. This function takes a
buffer object and returns a list of the minor styles active in that
buffer:

        (BUFFER-MINOR-STYLE-LIST (FIND-BUFFER "Mybuffer.txt"))

BUFFER-MINOR-STYLE-LIST is an accessing function only. Because it is
not a place form acceptable to SETF, you cannot use it to alter the
list of minor styles active in a buffer.

To alter the list, you use the function BUFFER-MINOR-STYLE-ACTIVE.
This function takes a buffer specifier and a style specifier. It
returns T if the specified style is active as a minor style in the
specified buffer; otherwise NIL. This function can be used with SETF
to add or remove a style from the minor style list of the buffer.

For instance, to activate "VAX LISP" as a minor. style in the current
buffer, you could write:

        (SETF (BUFFER-MINOR-STYLE-ACTIVE (CURRENT-BUFFER) "VAX LISP") T)

This form adds "VAX LISP" to the front of the minor style list for the
current buffer. "VAX LISP" will then shadow all other active styles.
This form is the essential action of the DIGITAL-provided command
"Activate Minor Style", which prompts for a style name and activates
that style as a minor style in the current buffer.

To deactivate "VAX LISP" you would end the above SETF form with NIL:

        (SETF (BUFFER-MINOR-STYLE-ACTIVE (CURRENT-BUFFER) "VAX LISP")
              NIL)

This form is the essential action of the DIGITAL-provided command
"Deactivate Minor Style".

If you activate a minor style that is already active in the specified buffer, the style moves to the front of the minor style list. The action actually deactivates and then reactivates the style, making it the most recently activated (and thus the first-searched).

6.2 MODIFYING A DIGITAL-PROVIDED STYLE

The three styles provided with the Editor can be extended and customized in any way you like. The operations that you can perform to modify a style are:

- Binding keys and pointer actions in the style

- Binding Editor variables in the style and assigning values or function definitions to them

- Binding Editor attributes in the style and assigning values for each attribute to all characters

You can also define new variables and attributes and bind them in any style. Only when an Editor variable or an Editor attribute is bound in a style can you assign values (see Chapter 1).

6.2.1 Binding Keys and Pointer Actions

A common way to extend a style is to bind keys or pointer actions to commands in that style. The command to be invoked can be:

- A new user-defined command

- A DIGITAL-provided command that is not currently bound in the style

- Any command that is currently bound to another key or pointer action in the style

6.2.1.1 Finding Key Bindings - A complete list of the key bindings in the Editor as provided appears as Appendix C of this manual. This list is organized by key or sequence, and it includes all bindings for each key (buffer, style(s), and global).

To find the current key bindings in the Editor, including any you have added or changed, you can execute the command "List Key Bindings". The Editor displays all visible bindings unless you specify a style (or other context) in response to the prompt.

If you want to find the current key bindings from the LISP
interpreter, you can call the function MAP-BINDINGS. This function is
described in full in Part III of this manual. Basically, MAP-BINDINGS
finds all key bindings and applies to them a function that you supply
as its argument. To get a list of the bindings, you would supply a
printing function.

For instance, if you wanted to print a list of the bindings in "VAX
LISP" style, you could start by defining a new function such as:

```
(DEFUN LISP-BINDINGS (KEY COMMAND CONTEXT)
  (WHEN (EQUAL CONTEXT (LIST :STYLE (FIND-STYLE "VAX LISP")))
    (FORMAT T "~% ~{ ~:C~} ~30,10T ~A"
        (COERCE KEY 'LIST) (COMMAND-NAME COMMAND)))))
```

You then call MAP-BINDINGS with the new function as its argument:

```
(MAP-BINDINGS #'LISP-BINDINGS)
```

The result is a screen display of all the keys and key sequences that
are bound in "VAX LISP" style, along with the name of the command
bound to each.

6.2.1.2 **Review Of BIND-COMMAND** - You bind keys in a style according
to the procedures outlined in Chapter 3 of this manual. You call the
function BIND-COMMAND with the command, key, and context arguments
that you want. Recall that to specify a style as a context argument,
you supply a list that begins with the keyword :STYLE, followed by a
style specifier (symbol or display name). For instance:

```
'(:STYLE "EMACS")
```

Or,

```
(LIST :STYLE 'VAX-LISP)
```

To bind the key CTRL/V to "View File" in "EDT Emulation" style, you
would write:

```
(BIND-COMMAND "View File" #\^V '(:STYLE "EDT Emulation"))
```

The binding procedure is the same regardless of whether the key
sequence or the command is already bound in the style. Rebinding a
key sequence destroys any previous binding of the key sequence;
binding another key sequence to a command leaves the previous key
binding to that command intact (see Chapter 3).

The procedure for binding pointer actions is similar. See the
discussion of BIND-POINTER-COMMAND in Chapter 3.

6.2.1.3 **Choosing Commands To Bind** - When binding a key or pointer action to a command in a style, you should first invoke the command by name in that style to make sure that it behaves as expected. Any command can be bound in any style, but a command that references a context-dependent object (an Editor variable or an Editor attribute) may behave differently in different contexts.

For instance, if you invoke "EDT Move Word" in "EMACS" style, the command does not behave as it does in "EDT Emulation" style. "EDT Move Word" references both the Editor variable "EDT Direction Mode" and the Editor attribute "Word Delimiter"; both are unbound in "EMACS" style. It would not be worthwhile, therefore, to bind a key to "EDT Move Word" in "EMACS" style.

You can also define new commands with a particular style in mind and then bind keys to them in that style. These procedures are explained in detail in Chapters 2 and 3 of this manual.

6.2.2 Binding Variables and Setting Variable Values

Both the value slot and the function slot of an Editor variable can be set in the context of a style, but only if the variable is first bound in that style. Binding an Editor variable in a context establishes the variable as usable in that context. Only then can its value or function definition be set.

The function slot is commonly used for hook functions, which are discussed in Part II.

This section discusses:

- Finding which variables are bound in a style

- Altering variable values in a style

- Binding a variable in a style

You can also define a new Editor variable and bind it in a style.

6.2.2.1 **Finding Style Variables** - The function STYLE-VARIABLES takes a style object and returns a list of the variables that are bound in that style. For instance, the form

    (STYLE-VARIABLES (FIND-STYLE "EDT EMULATION"))

returns the list:

```
(EDT-DELETED-LINE   EDT-DELETED-WORD   EDT-DELETED-CHARACTER
 SELECT-REGION-RENDITION-SET   SELECT-REGION-RENDITION-COMPLEMENT
 EDT-DIRECTION-MODE   DEFAULT-WINDOW-LABEL   EDT-PASTE-BUFFER)
```

To check whether a specified variable is bound in a specified style, you can use VARIABLE-BOUNDP. This function takes a variable specifier and an optional context that defaults to the current context. It returns T if the variable is bound in the context, otherwise NIL.

For instance, the first form below returns T; the second returns NIL:

```
(VARIABLE-BOUNDP "EDT Paste Buffer" '(:STYLE "EDT Emulation"))

(VARIABLE-BOUNDP "EDT Paste Buffer" '(:STYLE "EMACS"))
```

6.2.2.2 **Altering Variable Values** - To access the current value of an Editor variable that is bound in a specified style, you call the function VARIABLE-VALUE. Depending on the variable, the value might be any object, including a function. (The function VARIABLE-FUNCTION accesses the function slot; see Part III.)

For instance, to determine the current value of the variable "Select Region Rendition Complement" in "EDT Emulation" style, you would write:

```
(VARIABLE-VALUE "Select Region Rendition Complement"
                '(:STYLE "EDT Emulation"))
```

In the Editor as provided, this form returns :REVERSE.

Using SETF, you can change the value of any variable in a specified style. In the example above, you can change the video rendition of select regions in "EDT Emulation" style. (See Chapter 5 for the meaning of the various region rendition values.) To make select regions appear in bold if the window where they are displayed is non-bold, and vice versa, you would write:

```
(SETF (VARIABLE-VALUE "Select Region Rendition Complement"
                      '(:STYLE "EDT Emulation"))
      :BOLD)
```

If the value of a variable is a function, you proceed in the same way. You access the value with VARIABLE-VALUE:

```
(VARIABLE-VALUE "Default Window Label" '(:STYLE "EMACS"))
```

This form returns the function EMACS-WINDOW-LABEL. This function could be defined as:

```
(DEFUN EMACS-WINDOW-LABEL (WINDOW)
   (LET ((BUFFER (WINDOW-BUFFER WINDOW)))
     (FORMAT NIL " ~A ~@[(~{~S~^ ~}) ~]"
            (IF (EQ (BUFFER-TYPE BUFFER) :FILE)
                (NAMESTRING (BUFFER-OBJECT BUFFER))
                (BUFFER-NAME BUFFER))
            (MAPCAR #'STYLE-NAME (BUFFER-MINOR-STYLE-LIST
                                     BUFFER)))))))
```

This function specifies the default label content for any new windows
created in buffers where the style "EMACS" is active (and not
shadowed). It labels new windows with the name of the buffer
(namestring or object-name) and with any minor styles active in that
buffer.

You can rewrite this function in any way you like or define an
entirely new labeling function. To set the value of "Default Window
Label" in "EMACS" style to the new function, you would write:

```
(SETF (VARIABLE-VALUE "Default Window Label" '(:STYLE "EMACS"))
      'MY-EMACS-LABELER)
```

6.2.2.3 **Binding A Variable In A Style** - A variable cannot have a
value (or function definition) in a style unless the variable itself
is first bound in that style.

If a given variable is not initially bound in a style, you can include
it with the function BIND-VARIABLE. Its format, with only a few of
its keywords, is:

    BIND-VARIABLE symbol &KEY :CONTEXT
                             :INITIAL-VALUE

The symbol argument is an Editor variable specifier (symbol or display
name). The optional keyword arguments are a context specifier (the
default is :GLOBAL) and an initial value for the variable in the
context (the default is NIL).

For instance, suppose you want the Editor to show only one anchored
window at a time in "VAX LISP" style, but up to two when you are not
using "VAX LISP" style. The maximum number of anchored windows the
Editor shows at a time is determined by the value of the Editor
variable "Anchored window Show Limit". In the Editor as provided,
this variable is bound globally and its value is 2; the variable is
not bound in any other context.

If you want "Anchored Window Show Limit" to have the value 1 in "VAX
LISP" style, you must bind the variable in that style:

```
(BIND-VARIABLE "Anchored Window Show Limit"
               :CONTEXT '(:STYLE "VAX LISP")
               :INITIAL-VALUE 1)
```

This form binds "Anchored Window Show Limit" in "VAX LISP" style with the initial value 1. If you later want to change the value to 3, you need only use SETF because the variable is already bound in the style:

```
(SETF (VARIABLE-VALUE "Anchored Window Show Limit"
                      '(:STYLE "VAX LISP"))
      3)
```

This form changes the value of the specified variable to 3 in "VAX LISP" style. The Editor will show up to three anchored windows at once when this style is active. When "VAX LISP" style is not active, the effective value of the variable will be its global value (2) unless you also bind it in other contexts.

6.2.2.4 **Defining New Variables** - If some action that you want the Editor to perform requires a new Editor variable, you can create a variable with the macro DEFINE-EDITOR-VARIABLE. You then proceed as above to bind the variable in one or more contexts and to adjust its value as you like.

DEFINE-EDITOR-VARIABLE is described in full in Part III of this manual. Basically, it creates a variable with the specified name and an optional documentation string. For instance:

```
(DEFINE-EDITOR-VARIABLE (LISP-COMMENT-COLUMN
                         :DISPLAY-NAME "LISP Comment Column")

 "  When bound in \"VAX LISP\" style, this variable specifies
    an integer value that indicates the position in a line where
    a LISP comment should begin.")
```

This form is the one used to create the DIGITAL-provided Editor variable "LISP Comment Column". The naming convention for Editor variables is the same as that used for all named Editor objects. For more detail on specifying names, see the discussion of naming Editor commands in Chapter 2.

Before you can use a new variable, you must bind it in a context. For instance:

```
(BIND-VARIABLE "LISP Comment Column"
               :CONTEXT '(:STYLE "VAX LISP")
               :INITIAL-VALUE 49)
```

This form binds the variable in "VAX LISP" style and gives it an initial value. (BIND-VARIABLE also allows you to set the function

slot of the variable if you like. See Part III.) The variable can now be referenced by functions and commands in "VAX LISP" style.

6.2.3 Binding Attributes and Setting Attribute Values

Another way to modify a style is to alter the treatment of Editor attributes in the context of that style.

Like Editor variables, any attribute can be bound in any context.* Once an Editor attribute is bound in a style (or other context), then every character has a value for that attribute in that context. The values for an attribute serve to distinguish characters from one another for the purpose of searching through text.

For instance, to find whitespace the Editor passes over every character with the value 0 for the attribute "Whitespace" and accepts the first character with the value 1 for this attribute. (See the discussion of searching by attribute in Chapter 4.)

You can access and change the value that a character has for a specified attribute in a specified style -- but, as with Editor variables, only if the attribute is itself bound in the style. This section discusses:

- Finding the attributes and attribute values in a style

- Altering attribute values in a style

- Binding an attribute in a style

You can also define a new Editor attribute and bind it in a style.

6.2.3.1 **Finding Style Attributes** - The function CHARACTER-ATTRIBUTE takes an attribute specifier, a character, and an optional context. It returns the value that the character has for the attribute in the context. (If you do not supply a context argument, the Editor performs a normal context search to find the proper attribute value.)

The Editor provides no attribute-related functions similar to STYLE-VARIABLES or MAP-BINDINGS. That is, there is no way provided to determine which Editor attributes are bound in a style or what values all the characters have for an attribute in a style.

---

* An exception is "Print Representation", which can only be bound globally.

To obtain this information, you might define a new function such as the following:

```
(DEFUN LIST-ATTRIBUTE-VALUES (ATTRIBUTE CONTEXT)

    ;; Print a heading.
    (FORMAT T "~%ATTRIBUTE VALUES OF ~S IN CONTEXT ~S ~2%"
            ATTRIBUTE CONTEXT)


    ;; Define a local error handler in case attribute in unbound.
    (LET ((*UNIVERSAL-ERROR-HANDLER*
            #'(LAMBDA (&REST ARGS)
                (DECLARE (IGNORE ARGS))
                (FORMAT T
                        "~% The attribute ~S is not bound in ~
                            context ~S.~%"
                        ATTRIBUTE CONTEXT)
                (RETURN-FROM LIST-ATTRIBUTE-VALUES
                            (VALUES)))))

      ;; Print the value of each character for the attribute in
      ;; the context.
      (DOTIMES (INDEX 255)
        (FORMAT T "~C~15,5T<=>      ~S ~%"
                (CODE-CHAR INDEX)
                (CHARACTER-ATTRIBUTE ATTRIBUTE INDEX CONTEXT)))
    (VALUES)))
```

The new function LIST-ATTRIBUTE-VALUES takes an attribute specifier and a style (or other context) specifier. If you execute it at top-level LISP, it displays on the screen a list of the values of all 256 characters for that attribute in that context. If no values are found, the result is a screen message that the attribute is not bound in the specified context. For instance:

```
(LIST-ATTRIBUTE VALUES "LISP Syntax" '(:STYLE "VAX LISP"))

(LIST-ATTRIBUTE VALUES "LISP Syntax" :GLOBAL)
```

The first from results in a screen display of the values of all characters for the attribute "LISP Syntax" in "VAX LISP" style. The second form results in a message that the attribute "LISP Syntax" is not bound globally.

Note the use of *UNIVERSAL-ERROR-HANDLER* in this form. If no special error handler were defined, the function would call the VAX LISP error handler when it encountered an unbound attribute. The default error handler places you in the Debugger; this error handler returns you to top-level LISP.

6.2.3.2 **Altering Attribute Values** - CHARACTER-ATTRIBUTE is a place form acceptable to SETF. You can use it to change the value that a character has for a specified attribute in a specified style.

For instance, the Editor recognizes a hyphen as a word delimiter in the global context but not in "EDT Emulation" style. If you want the hyphen to be a word delimiter in "EDT Emulation", you change the value of that character for that attribute in that style from 0 to 1:

```
(SETF (CHARACTER-ATTRIBUTE "Word Delimiter" #\-
                          '(:STYLE "EDT Emulation"))
      1)
```

After you execute this form, the Editor will recognize the hyphen as a word delimiter in "EDT Emulation" style.

The attribute "LISP Syntax" differs slightly from "Word Delimiter" in that its values are keywords. Most characters in "VAX LISP" style have the value :CONSTITUENT for the attribute "LISP Syntax" -- they can be constituents of LISP symbols, but they have no syntactical significance. The characters that are significant as LISP syntax have appropriate keyword values: the open parenthesis has the value :LIST-INITIATOR, the backquote has the value :READ-MACRO, and so on. (The values for "LISP Syntax" are listed in Part III of this manual.)

These keyword values can be altered in the same way as the zero-one values illustrated above. For instance, if you want to use square brackets instead of parentheses around LISP forms, you could write:

```
(SETF (CHARACTER-ATTRIBUTE "LISP Syntax" #\[
                          '(:STYLE "VAX LISP"))
      :LIST-INITIATOR)
```

And,

```
(SETF (CHARACTER-ATTRIBUTE "LISP Syntax" #\(
                          '(:STYLE "VAX LISP"))
      :CONSTITUENT)
```

These forms, along with comparable forms for the close bracket and close parenthesis, make the Editor recognize the brackets as the characters that initiate and terminate a list in "VAX LISP" style. The Editor will no longer recognize parentheses as significant in LISP syntax.

6.2.3.3 **Binding An Attribute In A Style** - Characters cannot have values for an attribute in a style unless the attribute is bound in that style.

If a given attribute is not initially bound in a style, you can include it with the function BIND-ATTRIBUTE. Its format is:

```
BIND-ATTRIBUTE attribute &KEY :TYPE
                              :CONTEXT
                              :INITIAL-VALUE
```

In addition to the desired attribute and context specifiers, you can supply to BIND-ATTRIBUTE a :TYPE argument and an :INITIAL-VALUE argument. The type argument defines the data types of the possible values of the attribute in the context. The argument can be any LISP type specification (see *COMMON LISP: The Language*); the default is (MOD 2).

The initial-value argument becomes the value of all 256 characters for the specified attribute in the specified context. You can then use SETF with CHARACTER-ATTRIBUTE to change the value assigned to any of the characters.

For instance, suppose you have established "EDT Emulation" as your default major style and "EMACS" as your default minor style. This action makes the Editor behave like an EMACS editor that also has an EDT-like keypad. Any conflicting bindings will be resolved in favor of "EMACS".

However, the Editor attribute "Word Delimiter" is not bound in "EMACS" style. When "EMACS" is the only style active, the global values for this attribute are visible. When "EDT Emulation" is interposed in the search order between "EMACS" and the global context, then references to "Word Delimiter" produce the "EDT Emulation" values. As a result, the Editor recognizes words the way DIGITAL's EDT does, rather than the way that EMACS editors do.

You can alter this behavior by binding "Word Delimiter" in "EMACS" style and assigning the characters the values you want them to have. To bind the attribute, you write:

```
(BIND-ATTRIBUTE "Word Delimiter" :TYPE '(MOD 2)
                                 :CONTEXT '(:STYLE "EMACS")
                                 :INITIAL-VALUE 0)
```

When you execute this form, "Word Delimiter" becomes bound in "EMACS" style, and all characters have the value 0 for this attribute in this context. You need not include the :TYPE argument in this form since (MOD 2) is the default; it specifies that the possible values for the attribute are 0 and 1.

You then select the characters that you want the Editor to recognize as word delimiters and change their values to 1. If you want the values to be set as they are in the Editor's global context, you can get a list of those values by calling LIST-ATTRIBUTE-VALUES (see Section 6.2.3.1) with the context argument :GLOBAL.

For instance, many punctuation marks are word delimiters in the global context but not in "EDT Emulation". To have these characters be word delimiters in "EMACS", you write the following form for each:

```
(SETF (CHARACTER-ATTRIBUTE "Word Delimiter" #\;
                           '(:STYLE "EMACS"))
      1)
```

You perform no operation on characters that you do not want recognized as word delimiters in "EMACS". These characters already have the value 0 (the initial value) for this attribute in this style.

6.2.3.4 **Defining New Attributes** - If some action that you want the Editor to perform requires a new Editor attribute, you can create an attribute with the macro DEFINE-ATTRIBUTE. This macro is similar to DEFINE-EDITOR-VARIABLE: it creates a new object with the specified name and optional documentation string. You then proceed as above to bind the attribute in one or more contexts and to adjust characters' values for the attribute as you like.

The following form is the one used to create the DIGITAL-provided Editor attribute "Page Delimiter":

```
(DEFINE-ATTRIBUTE (PAGE-DELIMITER :DISPLAY-NAME "Page Delimiter")

   "  When bound, this attribute can have the value 1 for
   characters that separate pages.")
```

This form creates the attribute "Page Delimiter". The attribute cannot be used, however, until it is bound in some context. For instance:

```
(BIND-ATTRIBUTE "Page Delimiter" :INITIAL-VALUE 0)
```

This form binds "Page Delimiter" in the default context (global) with the default type specification (MOD 2). The initial value for all characters in the global context for this attribute is 0.

To distinguish the character(s) that you want the Editor to recognize as page delimiters, you change their values from 0 to 1:

```
(SETF (CHARACTER-ATTRIBUTE "Page Delimiter" #\formfeed) 1)
```

This form gives the formfeed character (CTRL/L) the value 1 for "Page Delimiter" in the global context. When the Editor performs an attribute search to find the next page delimiter, the formfeed character will satisfy the test (see Chapter 4).

6.3 CREATING A NEW STYLE

To create a new Editor style, you first make a new style object. You
then bind in the new style all the features that you want it to have.

Creating a new style brings together all the techniques discussed so
far in this manual:

- Defining new commands, variables, and attributes as necessary
  to perform text operations, display operations, and other
  Editor operations

- Binding keys, pointer actions, variables, and attributes in
  the new style

- Activating the new style in any or all buffers

You can also include in a new style some specially defined functions
that are invoked whenever the style is activated or deactivated in a
buffer. These "hook functions" create some useful feature in buffers
where the style is active and remove that feature whenever the style
is deactivated.

All these procedures are illustrated in this section in relation to a
new Editor style.

6.3.1 Making a Style Object

To create a new style object, you use the macro MAKE-STYLE.   This
macro is described in full in Part III of this manual. Its format is:

```
MAKE-STYLE name &OPTIONAL documentation
              &KEY :ACTIVATION-HOOK :DEACTIVATION-HOOK
```

For example:

```
(MAKE-STYLE (TEXT-MODE :DISPLAY-NAME "Text")

  "  Used when editing narrative text.  It emulates the
  behavior of word-processing programs. in formatting text.")
```

If you evaluate this form, the Editor will have a new style named
TEXT-MODE or "Text". The style will contain no bindings until you add
them with calls to BIND-VARIABLE, BIND-ATTRIBUTE, BIND-COMMAND, or
BIND-POINTER-COMMAND.

Before you make the style, however, you should decide whether you want
it to have activation and deactivation hooks. These functions can be
attached to a style in the MAKE-STYLE form; they cannot be added
later.

6.3.2 Style Activation and Deactivation Hooks

A style activation hook can be used to perform some operation that you want done every time the style is activated.

For instance, in a text-related style you need to be able to set margins -- the character positions in each line where text is to begin and end. You can define new Editor variables to store margin settings and then bind the variables in each buffer that has "Text" style active. This action enables each buffer to store its own margin settings.

The new variables might look like:

```
(DEFINE-EDITOR-VARIABLE (LOCAL-LEFT-MARGIN
                            :DISPLAY-NAME "Local Left Margin")

    "  Specifies the first character position where text can begin
    in each line.")
```

And,

```
(DEFINE-EDITOR-VARIABLE (LOCAL-RIGHT-MARGIN
                            :DISPLAY-NAME "Local Right Margin")

    "  Specifies the last character position that text can occupy
    in each line.")
```

It would be convenient to have these variables bound automatically in each buffer that has "Text" style active. To achieve this, you can define a function that binds the variables in a buffer and then specify that function as the activation hook of the style "Text".

The style activation hook is invoked whenever its style is activated in a buffer. The hook function is called with two arguments -- the style and the buffer. A function that binds the above variables on a per-buffer basis might look like:

```
(DEFUN BIND-MARGINS (STYLE BUFFER)

    ;; The function does not use the STYLE paramenter.
    (DECLARE (IGNORE STYLE))

    ;; CONTEXT is the buffer where the style is being activated.
    (LET ((CONTEXT (LIST :BUFFER BUFFER)))

        ;; Make the left margin 0 in the buffer.
        (BIND-VARIABLE "Local Left Margin"
                    :CONTEXT CONTEXT
                    :INITIAL-VALUE 0)
```

```
;; Make the right margin 1 less than the screen width in
;; the buffer.
(BIND-VARIABLE "Local Right Margin"
               :CONTEXT CONTEXT
               :INITIAL-VALUE (1- (SCREEN-WIDTH)))))
```

A style deactivation hook is similar: it is invoked whenever a style is made inactive in a buffer. If you deactivate "Text" style in a given buffer, you would have no further use for the margin settings. A deactivation hook that unbinds the margin variables might look like:

```
(DEFUN UNBIND-MARGINS (STYLE BUFFER)
  (DECLARE (IGNORE STYLE))
  (LET ((CONTEXT (LIST :BUFFER BUFFER)))
    (UNBIND-VARIABLE "Local Left Margin" CONTEXT)
    (UNBIND-VARIABLE "Local Right Margin" CONTEXT)))
```

Once you have defined the hook functions and the variables they reference, you are ready to create the new style "Text":

```
(MAKE-STYLE (TEXT-MODE :DISPLAY-NAME "Text")

  " Used when editing narrative text. It emulates the
  behavior of word-processing programs in formatting text."

  :ACTIVATION-HOOK #'BIND-MARGINS
  :DEACTIVATION-HOOK #'UNBIND-MARGINS)
```

This form creates the new "Text" style and establishes BIND-MARGINS and UNBIND-MARGINS as its activation and deactivation hooks.

6.3.3 Adding Capabilities to the Style

Once you have created a new style, you can add features to it at any time. You can add capabilities to a style by:

- Binding keys or pointer actions to commands in that style. This may involve defining new functions and commands; you can also use existing commands.

- Binding Editor variables in the style. The variables to be bound can be new or existing variables. Any variable that is referenced by a command used in the new style must be bound in that style, unless a binding will be visible from another context when the style is active.

- Binding Editor attributes in the style. The attributes to be bound can be new or existing attributes. Any attribute that is referenced by a command used in the new style must be bound in that style, unless a binding will be visible from another context when the style is active.

OPERATIONS ON STYLES

For instance, the "Text" style might include a key binding for a command that allows you to reset the margins in any buffer. The new Editor variables "Local Left Margin" and "Local Right Margin" serve to store margin settings once they are bound in a buffer. The activation hook function BIND-MARGINS binds these variables in a buffer and sets their initial values whenever "Text" style is activated in the buffer.

To implement a command called "Set Margins", you can use the new margin variables, along with various Editor objects and other LISP objects. Such a command might look like:

```
(DEFINE-COMMAND (SET-MARGINS-COMMAND :DISPLAY-NAME "Set Margins")
                (PREFIX)

    " Prompts for new margin values and resets the left
  and right margins to the new values."

  (DECLARE (IGNORE PREFIX))

  (LET ((NEW-MARGIN (OR "Local Left Margin" 0)))
    (SETQ NEW-MARGIN
          (SIMPLE-PROMPT-FOR-INPUT
              (FORMAT NIL
                      "Current left margin at ~D.  ~
                        Enter new value: " NEW-MARGIN)
              NEW-MARGIN))

    (UNLESS (INTEGERP NEW-MARGIN)
      (SETF (VARIABLE-VALUE "Local Left Margin")
            (PARSE-INTEGER NEW-MARGIN)))

    (SETQ NEW-MARGIN (OR "Local Right Margin"
                         (1- (SCREEN-WIDTH)))
          NEW-MARGIN
          (SIMPLE-PROMPT-FOR-INPUT
              (FORMAT NIL
                      "Current right margin at ~D.  ~
                        Enter new value: " NEW-MARGIN)
              NEW-MARGIN))

    (UNLESS (INTEGERP NEW-MARGIN)
      (SETF (VARIABLE-VALUE "Local Right Margin")
            (PARSE-INTEGER NEW-MARGIN)))

    (CLEAR-INFORMATION-AREA)
    (FORMAT *INFORMATION-AREA-OUTPUT-STREAM*
            "Left margin ~D, right margin ~D"
            "Local Left Margin" "Local Right Margin")))
```

With the new text-formatting style in mind, you could also write commands that wrap text, that move to the left margin to begin new lines, and that fill and justify text to the right margin.

Once the new commands are defined, you can bind keys to them in "Text" style:

```
(LET ((CONTEXT (LIST :STYLE "Text")))
   (BIND-COMMAND "Set Margins" '#(#\X #\M) CONTEXT)
   (BIND-COMMAND ... )
   (BIND-COMMAND ... ))
```

6.3.4 Activating the Style

Once your style has enough capabilities bound in it to be useful, you can then decide how and where you want to activate the style.

As a special-purpose style, "Text" is suitable for minor activation. You would not want to assign it to "Default Minor Styles", since its behavior (wrapping, filling, and so on) is inappropriate for most code editing.   It would be best to activate "Text" in buffers associated with the filetypes you normally use for narrative text editing.

Recall that the value of "Default Filetype Minor Styles" is an association list. You can add items to this list with PUSH:

```
(PUSH '("TXT" . "Text")
      (VARIABLE-VALUE "Default Filetype Minor Styles"))

(PUSH '("RNO" . "Text")
      (VARIABLE-VALUE "Default Filetype Minor Styles"))
```

These forms establish "Text" as the last-activated (first-searched) of the minor styles in buffers associated with filetypes TXT and RNO.

PART II

CONCEPTS IN EDITOR PROGRAMMING

CONCEPTS IN EDITOR PROGRAMMING

This part is an "encyclopedia" of the major concepts and data types used in programming the VAX LISP Editor. It consists of separate, alphabetically arranged articles on the following topics:

ATTRIBUTES
BUFFERS
CHARACTERS
CHECKPOINTING (Subsystem)
COMMANDS
CONTEXT (Subsystem)
DEBUGGING SUPPORT
EDITOR VARIABLES
ERRORS (Subsystem)
HOOKS
INFORMATION AREA
LINES
MARKS
NAMED EDITOR OBJECTS
PROMPTING (Subsystem)
REGIONS
RINGS
STREAMS
STRING TABLES
STYLES
WINDOWS

See Appendix A for a list of the functions and other Editor objects that relate to each of the object types described in this part.

ATTRIBUTES

Attributes make up the primary character-related information stored by the Editor. An attribute is a named Editor object having a LISP type specification in some context. Each of the 256 characters can be assigned a value of the specified type for an attribute. The function LOCATE-ATTRIBUTE is used to to locate a character that satisfies a test on the value it has for an attribute.

Example

The following form defines an Editor attribute called WHITESPACE or "Whitespace":

```
(DEFINE-ATTRIBUTE (WHITESPACE :DISPLAY-NAME "Whitespace")

  " Used to determine which characters can be considered
  word delimiters.")
```

You can then bind this attribute in a context. For instance:

```
(BIND-ATTRIBUTE 'WHITESPACE
              :TYPE '(MOD 2)
              :CONTEXT :GLOBAL
              :INITIAL-VALUE 0)
```

This form creates an instance of the "Whitespace" attribute that can take on the values 0 or 1.

If we set

```
(SETF (CHARACTER-ATTRIBUTE 'WHITESPACE #\SPACE) 1)

(SETF (CHARACTER-ATTRIBUTE 'WHITESPACE #\TAB) 1)
```

then executing

```
(LOCATE-ATTRIBUTE position "Whitespace" :TEST #'PLUSP)
```

locates the first space or tab character following the specified position.

Attributes are powerful tools in processing syntax-dependent text. An attribute value can be of any LISP data type. However, the test function may assume that attribute values are of a certain type. For instance, the values for the attribute "LISP Syntax" are keywords, whereas other DIGITAL-provided attribute have integer values. The test functions for the latter are normally one-argument predicates.

Attributes, like Editor variables, can be bound in any Editor context. In the example above, an Editor style might create a new binding of

ATTRIBUTES (cont.)

the "Whitespace" attribute. This new binding would shadow the global binding of "Whitespace". When the style was later made inactive, the global definition would again be in effect.

BUFFERS

A buffer is the only Editor object that can be displayed and that can be associated with a file or a LISP object.

The text contained in a buffer is defined by a region associated with the buffer -- the buffer region. Although there may be many regions that mark off sections of the buffer's text, it is the buffer region that defines the beginning and end of the text in a buffer. It is an error to alter the marks that define the buffer region.

Each buffer has a permanent mark associated with it called the buffer point. The buffer point is a left-inserting mark that is a point of attention for the buffer (where most text operations commands are executed). The underlying display functions of the Editor cause the screen cursor to track the buffer point when that buffer is the "current" one. Moving the cursor is therefore accomplished by moving the buffer point mark. Most normal character insertion and deletion operations are performed with respect to the buffer point. It is an error to change the type of the buffer point or to change the point so that it points to text not contained in the buffer.

The Editor can maintain a number of buffers simultaneously. The limit on the number of buffers depends on the size of available LISP dynamic space. It is also possible to simultaneously display portions of more than one buffer or different portions of the same buffer.

The current buffer is the buffer you are currently working on. The screen cursor is always displayed in one of the display windows for the current buffer. The concept of the current buffer is important because dynamic context is determined by the setting of that buffer.

CHARACTERS

Characters in the Editor are normal VAX LISP string characters; that is, STRING-CHAR-P returns T for all characters stored in Editor buffers. Characters are not, however, independent atomic objects in the context of the Editor; they are always constituents of Editor lines.

VAX LISP recognizes and accepts all characters from the 8-bit extended ASCII character set. All COMMON LISP font and bit information is ignored by the Editor.

Not all characters can be displayed directly on a terminal, of course. Any character that cannot be displayed directly on a screen is converted to a string of printing characters. Such a string is displayed on the terminal as a representation of the actual character -- for example, the ASCII escape character is displayed as <ESCAPE>. See the description of the "Print Representation" attribute in Part III for more detail.

CHECKPOINTING (Subsystem)

The VAX LISP Editor provides a mechanism for protecting the results of
an editing session from catastrophic failure such as a system crash.
Without such a mechanism, you could lose the results of many hours of
work if the system were to fail. The protection mechanism adopted by
the VAX LISP Editor is called checkpointing.

Checkpointing involves writing to disk the full contents of any buffer
that was modified since the last checkpoint. The buffer is written to
a file having a different (and distinctive) name from the file name
that is associated with the buffer source. By default, the checkpoint
file name is:

        Source.Filetype_Version_LSC

where Source, Filetype, and Version correspond to the file name of the
source file being edited. For example,

        MYPROG.LSP_2_LSC

is the name of the checkpoint file created when you are editing
version 2 of a file named MYPROG.LSP.

Buffers that are not associated with files do not, by default, have
checkpoint files associated with them.

You can set or change the checkpoint file name explicitly by using
SETF with the BUFFER-CHECKPOINTED function. (If you change the
checkpoint file name to NIL, checkpointing is not performed for that
buffer. The checkpoint file name is also changed automatically
whenever the pathname of the buffer's associated file is changed.

When to checkpoint is determined by maintaining a count of the number
of commands that have caused modifications in the buffer text. The
count is kept on a global basis (otherwise it would be possible that
many modified files would never be checkpointed). You can determine
the frequency (number of commands) of checkpointing by calling the
function CHECKPOINT-FREQUENCY. This function is also a SETF form that
allows you to change checkpoint frequency. The default value is 350.
If it is set to NIL, all checkpointing is disabled.

Should there be a catastrophic failure of the system during an editing
session, you can recover a file in its most current state by looking
for its checkpointing file. Checkpointing files are deleted when
modified buffers are written. If a checkpoint file exists, it is
guaranteed to be the latest available copy of the buffer contents.
The user can rename a checkpoint file to the buffer file name.
Editing this file gets the most recent information that was in the
Editor before the crash. Only modifications made to text between the
time of the last checkpoint and the system failure are lost.

6

CONCEPTS IN EDITOR PROGRAMMING

COMMANDS

Commands are similar to function objects in that they can be invoked to produce changes in the state of the Editor. They are unlike ordinary LISP functions in how they are invoked and in the context rules for their execution. Every command is associated with a LISP function; invoking a command within the Editor causes the Editor to invoke the command's associated function.

Binding an Editor command can be thought of as creating a bridge between a character or sequence of characters and a command in a particular context. You accomplish this by executing the BIND-COMMAND function. If you then enter a key or key sequence from the terminal, the Editor makes a normal context search to find the correct command to execute. You can also bind actions of a pointing device to Editor commands, using the function BIND-POINTER-COMMAND.

There are three ways to invoke an Editor command:

- Entering a previously bound character or sequence of characters from the keyboard, or performing a pointer action, when you are in the Editor

- Using the "Execute Named Command" command and specifying the name of a command when you are in the Editor

- Calling the associated function directly from LISP

The first of these is the fastest method. These methods are discussed individually.

- The BIND-COMMAND function is used to bind an Editor command to a character or sequence of characters in a particular context. When you enter this character or sequence of characters, the Editor initiates a normal context search for a command object bound to that sequence. For example, by default, all the individual graphic characters are bound to the named command "Self Insert" in the global context. The result of your typing any of these characters is the execution of a function that inserts the characters at the buffer point of the current buffer.

  The action of BIND-POINTER-COMMAND is similar except that the specified command is invoked by an action of the pointing device, such as depressing a particular button or moving the pointer cursor. As with bound characters, the Editor performs a context search to determine which command to invoke in response to a pointer action.

  The binding of a command, like that of a variable or an attribute, can be shadowed by another binding to the same key

7

COMMANDS (cont.)

(or key sequence) in a local context. For example, when "VAX LISP" style is active in the current buffer, the right parenthesis character is bound to a function that finds and displays the matching left parenthesis before inserting the right parenthesis. This binding shadows the binding of the right parenthesis to "Self Insert" in the global context.

NOTE

> If you redefine a command that is bound in some context, you must rebind the appropriate key sequence or pointer action to that command in order to have the new command executed.

- Some commands (like "Delete Current Buffer") are either infrequently used or are potentially too dangerous to be bound to keys (where they might be invoked by accident). The VAX LISP Editor has a command, "Execute Named Command", that allows you to enter the name of a command and have the corresponding function executed.

- The third method used to invoke a command is to directly call the associated LISP function from another LISP function. To make use of existing commands when writing a new Editor command, you must use the function associated with the command.

Categories

Commands can also have a list of categories associated with them. These categories are user-defined and can be retrieved, tested, and altered. Examples of command categories are :GENERAL-PROMPTING and :LINE-MOTION. A command might examine the categories of itself or of a previously invoked command and perform different actions depending on the categories found.

Prefix Argument

Every function that implements a command takes at least one argument. This argument is called the prefix argument, and it usually tells the function how many times the operation is to be done. For example, if the "Self Insert" command is called with a prefix argument of 5, it inserts the most recently typed character five times. The prefix argument is automatically reset to NIL each time through the command loop. You use the "Supply Prefix Argument" command to set the prefix argument for the next command to be executed.

CONTEXT (Subsystem)

The VAX LISP Editor maintains a hierarchical search space that is used to locate all Editor key bindings, pointer action binds, variables, and attributes. The Editor must search this hierarchy in order to determine the correct command for a key sequence or pointer action, the correct value or function of an Editor variable, or the correct value of an attribute.

The binding of commands, variables, and attributes must take place in some context. The context can be

- Global, which means that the object is always defined.

- A style, which means that the object is defined in buffers that use the style as either the major style or a minor style.

- Specific to a particular buffer.

The search order of the hierarchy is

- Current buffer

- Minor styles active in that buffer in the order of most recently activated to least recently activated

- Major style of that buffer

- Global definitions

Only if the entire search fails is the command, variable, or attribute considered unbound.

By default, the standard order is used to locate the value of an object. It is possible to specify an explicit context for an accessing function (for example, VARIABLE-VALUE). In this case the normal searching operation is bypassed, and the object is accessed in the specified context only. Every function that binds an Editor object has an optional argument to define the context in which the created object is stored for later access. In such situations, a context argument is always specified with one of the following:

- :GLOBAL -- The object is defined in the global context and is universally accessible.

- A 2-element list consisting of the :STYLE keyword followed by a style reference. The object is defined in the context of the named style.

    Example: '(:STYLE "EDT Emulation")

9

CONTEXT Subsystem (cont.)

- A 2-element list consisting of the :BUFFER keyword followed by a buffer reference. The object is defined in the context of the specified buffer; that is, it is local to that buffer.

     Examples: '(:BUFFER "filename.lsp")

                        or

                (LIST :BUFFER (CURRENT-BUFFER))

     where the function CURRENT-BUFFER returns an Editor buffer.

As a result of the context and searching rules, the named objects can be thought of as forming a hierarchy:

```
                    Buffers
                   /    \
                 Styles   \
                    \  >   \
            /         \      \
     Commands    Variables   Attributes
```
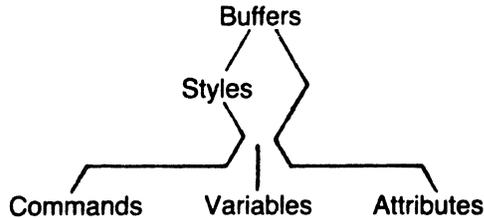
MLO-247-86

That is:

- Buffers can include active styles and bindings of commands, variables, and attributes, but not other buffers.

- Styles can include bindings of commands, variables, and attributes, but not buffers or other styles.

- Commands, variables, and attributes cannot contain bindings of one another.

Use of Context

A few conventions regarding the use of context by different users of the Editor follow:

- The global context is established by DIGITAL when the Editor is built. You can, of course, alter it, but you must be aware of any hook functions or default variables that are supplied with the Editor. Styles supplied with the Editor often assume the existence of certain variables and hook functions. Such assumptions are listed with the descriptions of the appropriate variables and commands.

CONTEXT Subsystem (cont.)

If you want to alter the global context, you should check for any predefined hooks or variables and make certain that they are either retained, or their use is not necessary.

- Styles are the province of writers of Editor extensions. A writer of an extension should feel free to make whatever alterations, bindings, variables, or attributes that are appropriate for implementing the desired style. A style should not alter global context or local buffer context without care. In particular, command bindings should be established only within the style.

- Buffer local bindings should be used for special-purpose buffers such as "General Prompting". These buffers exist to provide special capabilities that are not needed during normal text editing. The commands related to help, alternatives, and completion are bound locally in the "General Prompting" buffer because these commands have meaning only while the user is being prompted.

  It can, however, be appropriate to bind Editor variables locally in a buffer. Such bindings are proper when certain information necessary to the operation of a style needs to be kept in each buffer. The "Buffer Select Mark" is an example of such an Editor variable. The "EDT Emulation" style must keep track of any selected regions in each buffer having the style active. A style-local variable cannot be used because it would lose its value whenever a region was selected in some other buffer. Each buffer, therefore, keeps its own binding of the "Buffer Select Mark".

DEBUGGING SUPPORT

The normal action taken by the Editor's display subsystem when you execute the command "Pause Editor" is to save the current state of the screen and clear the screen. Conversely, when the Editor resumes after a pause, the current state of the screen is lost and the display is reset to the appearance it had at the time the pause went into effect.

This is reasonable behavior when you have been doing normal editing operations. It becomes a problem, however, for anyone implementing new Editor commands, because it means that Editor functions cannot reasonably be called from the LISP top level. The previous screen state is lost, and any windows created from top level are lost when the Editor is reentered.

In order to have effective debugging support for command implementers, the Editor provides a variable to aid in debugging extensions to the Editor:

    EDITOR-RETAIN-SCREEN-STATE

EDITOR-RETAIN-SCREEN-STATE is a LISP special variable that controls the action taken by the display subsystem when an Editor pause is executed. If the value is NIL (default), the display subsystem takes its normal action of saving the current state and clearing the screen; if the value is non-NIL, it does not save the screen state and clear the screen.

The display subsystem clears the screen and restores the old state only if the display was saved at the last pause. This behavior allows a command implementer to call Editor commands and functions from the LISP top level without losing changes made when the Editor resumes (by means of a call to the ED function).

EDITOR VARIABLES

Editor variables are distinct from VAX LISP special variables. They are similar to VAX LISP variables in that they can have both values and functions attached to them. The scope and extent rules for Editor variables, however, are different from LISP variables.

The scope of an Editor variable is defined by the Editor context searching rules. An Editor variable has extent that begins when it is bound in some context and ends when it is unbound from that context.

Editor variables are named objects, and special functions exist for accessing and setting the value and function slots of variable objects. You can use the functions VARIABLE-VALUE and VARIABLE-FUNCTION for accessing the value or function associated with an Editor variable. You can use them with SETF to change the value or function.

The LISP symbol corresponding to the Editor variable (the variable name) has its value and function slots set according to the current context -- that is, the symbol can be used as a special variable. Its value changes according to the current context. It becomes unbound in any context in which the Editor variable is not bound. By using the LISP symbol you can improve the access time to the value or function of an Editor variable.

Similarly, the LISP symbol can be used as a LISP function inside an Editor command. The function slot of the symbol is set to the function of the Editor variable bound in the current context. If there is no function definition, the LISP symbol has no function definition (FBOUNDP is NIL).

ERRORS (Subsystem)

In your extensions to the Editor, the Editor's error subsystem lets you handle errors during the execution of Editor commands and notify the user of such errors. In addition, a facility is provided to handle LISP-level errors (signaled from ERROR or CERROR, for example) and place the user in a usable debugging environment.

Errors Signaled from LISP

When you invoke the Editor (by means of the ED function), the variable *UNIVERSAL-ERROR-HANDLER* is bound to an Editor function that intercepts any LISP errors that occur (those signaled by ERROR, CERROR, and ASSERT, for example). This function first asks you if modified buffers should be saved. If you reply "Y," the Editor attempts to save any buffers that were modified, although the nature of the error may prevent some or all buffers from being saved. The Editor then asks if you want to enter the VAX LISP Debugger. If you reply "Y," the Debugger is invoked; you have access to all the normal Debugger features. If you reply "N," control returns to the LISP top level.

You treat this error just as you would a LISP error at top level and can take whatever actions are appropriate to the error signaled. Throwing* to top level (by pressing CTRL/C or quitting the Debugger) causes the Editor to quit the current command and pause. Continuing a continuable error causes a return to the interrupted Editor function. The Editor screen state is not automatically updated, but the display device is placed back in the mode required for operation of the Editor.

Errors Signaled from the Editor

The Editor provides two error functions that you can use when writing Editor commands -- EDITOR-ERROR and EDITOR-ERROR-WITH-HELP. EDITOR-ERROR is similar to the LISP ERROR function but is more appropriate to the Editor environment. When called, the function displays an optional line of text in the information area of the screen, calls the ATTENTION function to alert you to a problem, and executes a THROW to the top-level command loop of the Editor.

This function is typically used to indicate an illegal command operation, invalid user input, or some other such error that allows the Editor to continue normal operation, having discarded some improper data.

---

* See *COMMON LISP: The Language.*

ERROR Subsystem (cont.)

The second error function used by the Editor is similar to the VAX
LISP CERROR function. The EDITOR-ERROR-WITH-HELP function looks like
EDITOR-ERROR but takes an additional format string, which is used to
provide additional information to a user about the error that has
occurred. You can retrieve this additional formatted string by using
the "Help on Editor Error" command.

For example, when the Editor is writing a file, some error might occur
such as the quota being exceeded. The Editor signals an error and
displays a message in the information area notifying the user of that
fact. The EDITOR-ERROR-WITH-HELP function formats detailed
information about the error (the RMS error message), which the user
can retrieve if the problem is unknown by using the "Help on Editor
Error" command.

CONCEPTS IN EDITOR PROGRAMMING

HOOKS

It is often desirable, when you are writing extensions to the Editor, to have operations performed automatically when some particular part of the Editor state changes. Such automatically executed operations are called hooks; the functions that implement them are called hook functions.

The VAX LISP Editor implements hooks by attaching these functions to the function slots of Editor variables. Such variables are called hook variables, and their names, by convention, end with -HOOK. Any binding of a hook variable in an Editor context can have only one function associated with it (not a list of functions).

Two functions allow you to treat an ordinary Editor variable as a hook variable:

- INVOKE-HOOK

- REVERSE-INVOKE-HOOK

The arguments for each of these functions are an Editor variable optionally followed by additional arguments to be passed to the hook functions.

Normally, reference to an Editor variable results in a context search to locate a single instance of a variable. The invoking of a hook produces different behavior. A context search is made to locate all the instances of the hook variable in the context search list (buffer local, minor styles, major style, and global). Then *all* the functions (if any) attached to the instances of these variables are called in the INVOKE-HOOK or REVERSE-INVOKE-HOOK call. This is a major difference between hook variables and other Editor variables.

It is important to note that in the normal case (a call to INVOKE-HOOK) the functions are called in the reverse order of the context search -- global, major style, minor styles (from oldest to newest), and buffer local. The purpose of this ordering is to allow writers of styles and individual Editor users to modify effects of more global hook changes rather than to supplant them completely.

The REVERSE-INVOKE-HOOK function behaves identically to INVOKE-HOOK except that it calls the functions in normal context search order. All hooks built into the VAX LISP Editor are called with the INVOKE-HOOK function.

Setting a hook variable to a function in an Editor context will result in the loss of any previous setting of the function slot of that variable in that context.

To set hook variables to new functions without losing existing hooks, you can set the variables in the context of a user-defined style.

16

HOOKS (cont.)

(See Chapter 6 of this manual for information on creating styles.) When the new style is active, a reference to a hook variable results in evaluating the new hook function as well as any other hooks that are attached to that variable in other active contexts.

For instance, you can create a new style called "My LISP Hooks" or "My Text Hooks". You then define the hook functions you want and set them to the function slots of the appropriate hook variables in the new style.

```
;;; Create a style to serve as a binding context.

(MAKE-STYLE (MY-LISP-HOOKS :DISPLAY-NAME "My LISP Hooks")

   "  This style contains hooks related to editing LISP code.")


;;; Define hook functions.

(DEFUN HOOK-1  ... )
(DEFUN HOOK-2  ... )


;;; Bind the appropriate hook variables in the new style,
;;; specifying the initial function definitions.

(BIND-VARIABLE "Name of Hook Variable"
            :CONTEXT '(:STYLE "My LISP Hooks")
            :INITIAL-FUNCTION #'HOOK-1)


;;; Once the hook variables are bound in the style, they
;;; can be changed at any time using SETF.

(SETF (VARIABLE-FUNCTION "Name of Hook Variable"
                         '(:STYLE "My LISP Hooks"))
      #'HOOK-2)
```

You can activate the new style in any given buffer by executing "Activate Minor Style" command. You can also have the style activated automatically by adding it to the lists that are the values of the Editor variables "Default Minor Styles", "Default LISP Object Minor Styles", or "Default Filetype Minor Styles".

INFORMATION AREA

The Editor supports a dynamic multiwindow display. Windows can be displayed and moved to arbitrary locations. There is a reserved area at the bottom of the screen, however, that is never deleted or overlapped by an Editor window. This is the information area. This area is always at least one row in height and is the full width of the screen; its size can be increased.

The purpose of the information area is to have a location with guaranteed visibility where data can be displayed. Error messages are displayed here, as are other messages such as those telling you what file was just written. There is a global variable, *INFORMATION-AREA-OUTPUT-STREAM*, that is bound to an output stream for this area.

The information area is not an Editor window and cannot be treated as such. This means that there are no key bindings or Editor buffers associated with it. The Editor window functions do not operate on the information area. The information area should be used primarily as an information display area for the user.

The CLEAR-INFORMATION-AREA function erases any text currently in the information area.

The INFORMATION-AREA-HEIGHT function tells you the current height of the information area (in number of rows). You can change this value by using SETF with this form.

LINES

The line is the basic unit of text in the Editor; it contains a character string that normally corresponds to a single displayed line of text. The string is exactly what would be returned if you executed a READ-LINE function on a text file. A line also contains information concerning its own relative position within a group of lines, as well as within a buffer that might contain a group of lines.

Execution of most of the Editor functions results, directly or indirectly, in the alteration of either lines or their relations to other lines. The display subsystem of the Editor displays groups of specified lines.

Lines are created as by-products of certain Editor operations (such as making empty regions, breaking lines, and reading files). They can be accessed either through marks that point into them or through following the forward and backward links between lines. You can alter lines by inserting or deleting characters in the line, replacing individual characters in the line, deleting a region that is a portion of a line, or replacing the entire text of a line. You can delete lines by deleting regions that encompass them.

A line is never shared among buffers or disembodied regions. Altering or removing a line in one buffer cannot affect lines in another buffer. But because it is possible for regions to overlap or be completely contained in other regions, altering or removing lines in one region can affect the contents of another region in the same buffer or disembodied region.

CONCEPTS IN EDITOR PROGRAMMING

MARKS

The ability to indicate any given position in text is central to the operation of any editor. The VAX LISP Editor has a special type of LISP object for this purpose that is known as a mark.

A mark contains two items of information that allow Editor functions to access or address specific characters in the text -- a pointer to a line, and a number indicating the character position on the line. If you think of a single line of text as beginning at the leftmost position on the screen, then you can think of the representation of a character position as the number of characters "to the left" of the character position of interest.

For purposes of text manipulation, you should think of the mark as pointing between two characters. Any character inserted at the position of a mark is always placed between the characters. With respect to the number representing the character position, the mark points between positions n and $n+1$. The mark can also point between the beginning of a line and the first character ($n = 0$), or between the last character of a line and the end of the line.

Marks are of two types -- permanent and temporary.

There are two kinds of permanent marks. They differ with respect to whether text is inserted following the mark (right-inserting) or preceding the mark (left-inserting). The two kinds of permanent marks are designated by the keywords :LEFT-INSERTING and :RIGHT-INSERTING. Regardless of text insertions or deletions made before or after them, a right-inserting mark remains "attached" to the character that was to its left just prior to the operation; and a left-inserting mark remains "attached" to the character that was to its right.

A temporary mark, on the other hand, becomes invalid after any operation affecting the character it points to. You define a temporary mark by using the keyword :TEMPORARY. Temporary marks are used primarily in operations that require a mark to be used just once. They are used because these marks require less overhead in their creation and use than do permanent marks, and so are much more efficient in some applications.

If the line that a temporary mark points into is deleted, the mark becomes invalid and should no longer be used. If the line that contains a temporary mark is affected by insertion, being copied, deletion, or being relocated, the temporary mark becomes invalid and should no longer be used.

Marks are used primarily to indicate positions for character insertions or deletions. Unlike many LISP functions, the functions that manipulate marks are usually destructive operations on the mark. Moving a mark, for example, alters the mark so that it points to a new location. Only the accessing functions MARK-LINE and MARK-CHARPOS do

MARKS (cont.)

not alter the mark. Marks can be shared among regions. A given mark can be used to delimit any number of regions.

When a region of text is deleted, any permanent marks within that region (including the beginning and ending marks of the deleted region) then point to the location that is the junction of the text that preceded the deleted text and the text that followed the deleted text.

For example, in the following text

```
          A B C D E F G H I J K L M N
               ^       ^     ^
               1       2     3
```

MLO-248-86

the marks 1, 2, and 3 point to the indicated positions. If the region defined by marks 1 and 3 is deleted, the resulting text and mark positions become

```
          A B C   K L M N
               ^^^
               123
```

MLO-249-86

21

NAMED EDITOR OBJECTS

Several types of Editor object are called named objects. A named object is a special kind of LISP object. Once a named object is created, it can be referred to in any of three ways:

- By means of an expression whose evaluation results in the actual object. For example, (STYLEP *variable*) is true if, and only if, the value of the variable is an Editor style.

- By means of a symbol defined at the time the object was created. For example, (FIND-COMMAND 'WRITE-CURRENT-BUFFER-COMMAND) returns non-NIL only if the WRITE-CURRENT-BUFFER-COMMAND was specified as the name of a command when the command was created.

- By means of a string that is the display name defined at the time the object was created. For example, (FIND-BUFFER "JONES.LSP") returns non-NIL only if the string "JONES.LSP" was specified as a display name when the buffer was created.

The specification of the name when you are creating a named object is the same for each of the different types:

> *name* | (*name* :DISPLAY-NAME *string*)

where

- The *name* argument is a symbol.

- The *string* argument is a character string that can be specified as an alternate access string to the object. If a display name is not specified, the print name of the symbol is used as a display name.

Each named object can have a documentation string associated with it. Such a string appears when the symbol of the object is described, or the DOCUMENTATION function is used. The following documentation type is used for named objects:

> EDITOR-type -- Gets the documentation string of the specified object.

where type is one of the object types listed below.

The display names of all created named objects are stored in string tables. The string table associated with each type of named object is bound to a special variable of the form, *EDITOR-type-NAMES*. Use the string tables to find the symbol associated with a given display name.

NAMED EDITOR OBJECTS (cont.)

The following object types are named Editor objects:

- BUFFER

- STYLE

- VARIABLE

- COMMAND

- ATTRIBUTE

Buffers, styles, and commands are context-independent LISP objects --
that is, their creation functions (MAKE-BUFFER, MAKE-STYLE, and MAKE-
COMMAND) create and return LISP objects of these types. The other two
named object types (variables and attributes) are context-dependent
objects. That is, once defined, they must be bound in a specific
Editor context before they are used. In addition, the current value
of these objects depends on the current Editor context.

PROMPTING (Subsystem)

Often, the user must be prompted for information necessary to the operation of some function or command. The operation involves first telling the user what information is needed and then soliciting the input data. For example, the "Write Named File" command needs to ask the user to specify a file for the contents of the buffer to be written to. The VAX LISP Editor makes two functions available to you for creating a prompt:

    PROMPT-FOR-INPUT

        and

    SIMPLE-PROMPT-FOR-INPUT

Both functions make full use of the Editor capabilities for text processing and display. They assume the availability of a buffer with the display name "General Prompting". This is a normal Editor buffer that is created with the value of the Editor variable "Default Major Style" as its major style. You can change this style as you can for any other Editor buffer. This buffer also has a number of buffer-local command bindings and Editor variables that alter its normal behavior to provide additional prompting services to the user.

There is a window associated with the "General Prompting" buffer. This window is always visible in the Editor (in the row or rows above the information area), and most user interaction occurs in this window. Although the window is a normal Editor window, it can be manipulated with only a few of the normal Editor window functions. Specifically, the prompting window cannot be removed from the screen or moved to a different screen position.

Simple Prompting

The SIMPLE-PROMPT-FOR-INPUT is the more basic of the prompting mechanisms. It displays an optional prompting string in the prompting window and solicits a response from the user, which echoes in the prompting window. The prompt function reads the user's input as a simple string; there is no Editor interpretation of the individual characters. If the user supplies a null input string, an optional default argument is returned.

General Prompting

A much more general mechanism is provided with the PROMPT-FOR-INPUT function. This function has special capabilities that you can use to develop elaborate prompting schemes when you are creating commands.

PROMPTING Subsystem (cont.)

Validating User Input - The one required argument to the PROMPT-FOR-INPUT function is the validation function. This must be a function that accepts a string argument and produces some value that will be returned by the PROMPT-FOR-INPUT function.

The validation function indicates that the input string is invalid by returning NIL. In such an instance, PROMPT-FOR-INPUT signals an error to the user and awaits further input. If the string input is a null string, and the value of the :REQUIRED keyword is NIL, the value of the :DEFAULT keyword parameter is returned. You can actually allow the validation function to return NIL as a valid value by returning multiple values of NIL and T.

An example of a function you can use for validation is FIND-COMMAND. This function returns a command function if the string is the name of a command, and returns NIL if the string is not the name of a command.

Providing Input Completion - The PROMPT-FOR-INPUT function provides you with facilities that can attempt to complete partial user input. For example, the user might be generally familiar with a set of Editor commands, but not remember the exact display name of the one needed. By using the completion facility, the user can type a portion of the name of a command and ask the facility to complete the name automatically. The user normally requests input completion by typing a CTRL/space (the null character).

There are three ways you can supply such completion assistance to a user:

- If the argument to the :COMPLETION keyword is a string, it is just inserted into the prompting buffer.

- If the argument to the :COMPLETION keyword is a string table, the completion function uses the text entered by the user as the key to the string table and attempts to return a completed string that will automatically be inserted into the prompting buffer.

  The string table routines complete as much of the text as they can -- supplying the rest of the text string or only as much of it as is uniquely identifiable. The user is informed of whether the input is now complete or if other entries can be found starting with the same string. If no entry can be found to match the user input, the facility deletes characters from the end of the user input until some entry (possibly ambiguous) can be found in the string table. This mechanism is used in the "Execute Named Command" command.

PROMPTING Subsystem (cont.)

● If the argument given to the :COMPLETION keyword is a function, that function is called and passed any arguments specified in the :COMPLETION-ARGUMENTS keyword. You have complete control over the displayed contents of the prompting buffer. This method is used by the "Edit File" command, which attempts to complete user input by performing a directory search for a matching file name.

Providing Alternatives - The alternatives option to general prompting is designed to supply additional help to a user responding to a prompt. This feature is designed to help the user choose among a set of alternative possibilities. For example, there are many named commands in the Editor. When being asked for a command name, the user might not know the exact spelling of the name. Upon entering some input and asking for completion help, if the Editor cannot respond with an exact command name, the user needs to be able to get a list of possible names based upon the typed input. In such an instance, the calling of the alternatives option (pressing keypad PF1 PF2) should yield a displayed list of commands whose names begin with the typed string.

The general prompting facility uses the :ALTERNATIVES and :ALTERNATIVES-ARGUMENTS arguments to enable this form of help. The argument to :ALTERNATIVES can be a string table or a function. If the argument is a string table, that table is searched to find all possible entries that start with the string the user has typed. The list is automatically displayed in the "Help" buffer.

If the argument is a function, that function is called and passed any arguments that were given in the :ALTERNATIVES-ARGUMENTS argument. This function can perform any operations it needs and should provide a display of the user's options appropriate to the command. For example, such a function might do a wild-card directory search for possible file names.

Providing Help - If, at any point in PROMPT-FOR-INPUT, the user invokes the "Prompt Help" command (presses keypad PF2), the function takes action based on the value of the :HELP keyword. If the value is a string, that string is displayed in the information area or in the "Help" buffer if the text has more lines than will fit in the information area.

If the value is a function, that function is called and passed any arguments that were specified in the :HELP-ARGUMENTS keyword. This method, like that for completion, gives you, the command writer, all the flexibility necessary for supplying assistance tailored to the needs of the user and the command.

CONCEPTS IN EDITOR PROGRAMMING

REGIONS

A region contains a portion of text, which can be part or all of one or more lines in a group of related lines. The region is defined by two marks, which indicate the beginning and ending positions of the region. Regions are treated as blocks of text that can be manipulated as units -- deleted or inserted, for example.

The marks that delimit a region can be either temporary or permanent. You can use temporary marks for one-time operations on regions. If you use permanent marks, delimit the beginning of the region with a right-inserting mark and the end of the region with a left-inserting mark. If you use a left-inserting mark at the beginning of a region or a right-inserting mark at the end, and if you insert text at the beginning or end, the results can be unpredictable.

Regions can be of two types. The most commonly used region is a portion of text inside a buffer. The region is defined by beginning and ending marks. Regions of this type can share text with other regions. Regions can overlap in arbitrary ways or be entirely contained within other regions. Since the text of multiple regions can be shared, any alterations done in one region affect the text of any other region containing the same text.

The second type of region is a disembodied region -- a region of text that is not associated with any buffer. This type of region can be created only by means of the MAKE-EMPTY-REGION or DELETE-AND-SAVE-REGION function. It can be used with any of the normal text manipulation functions; for instance, INSERT-CHARACTER would work if given a mark that points into a disembodied region. Such a region cannot, however, be displayed. Disembodied regions are often used as storage areas for deleted text such as traditional cut-and-paste regions.

Highlight Regions

A highlight region is a special type of region that can be defined only for text in a buffer. A highlight region can be used just as any other region in the Editor is, and all the region manipulation functions operate on them. In addition, when any of the text defined by the highlight region is visible in a window, that text is displayed with any special display attributes specified when the region was created. The possible highlight atttributes are any combination of reverse video, bold, blinking, or underline. Highlight regions can overlap, but the resulting display attribute for the overlapped section is not predictable.

Special functions are available to you for creating and removing highlight regions. Once created, the highlighting remains in effect until the region is removed by means of REMOVE-HIGHLIGHT-REGION or deleted by means of either DELETE-REGION or DELETE-AND-SAVE-REGION.

REGIONS (cont.)

Removing a highlight region does not alter the text of the region, but only the display attributes of the text. If either the beginning or ending mark of a highlight region is moved, the display of the region tracks the motion of the mark.

RINGS

A ring is a specialized data structure that implements a circular cache of data values. Items of data can be retrieved either from the start of the ring (Last In/First Out) or from the end of the ring (First In/First Out). In addition, since the ring is circular, it can be rotated so as to move its start/end position.

Rings have general utility in editors -- for example, to store a record of deleted text. A set of utility routines is included to let you create and manipulate ring structures. Rings and ring functions can also be used outside the Editor environment.

STREAMS

VAX LISP I/O can be directed into and out of Editor regions by the creation of streams to these objects.

Establishing an Editor input stream allows text to be read from a region with standard LISP read operations. All normal LISP input functions can be used. The usual COMMON LISP end-of-file action is taken whenever an attempt to read past the end of the region occurs.

An Editor output stream allows normal VAX LISP write operations to put text into a region at a particular mark. All normal LISP output functions can be used. You can create a new line in the region by using the TERPRI function or by writing a newline character. Writing a carriage return - linefeed pair does not automatically break a line. These characters are inserted as ordinary nonprinting characters.

There are two additional functions that direct file operations into and out of Editor regions. INSERT-FILE-AT-MARK inserts the contents of a file at the designated mark. WRITE-FILE-FROM-REGION writes the contents of an Editor region to a file.

STRING TABLES

String tables are specialized hash tables that are used to store information indexed by a string. String tables are of general utility (they can be used outside the Editor environment) and are used for such actions as completing partial user input (of a command name, for example). There are functions that access these tables to retrieve data based on a specified string. The mapping of Editor names to LISP objects is accomplished through use of these tables.

The following special variables are bound to tables holding information on named Editor objects:

- *EDITOR-ATTRIBUTE-NAMES*

- *EDITOR-BUFFER-NAMES*

- *EDITOR-COMMAND-NAMES*

- *EDITOR-STYLE-NAMES*

- *EDITOR-VARIABLE-NAMES*

In addition to your being able to access information by means of the entire string, you can use functions to do a search based on a partial string (for example, the first four letters of a buffer name). These functions help in writing commands that attempt to complete a partial string that is specified.

For example, you might want to execute a named command. The Editor can accept a partial command name and, by means of the string table *EDITOR-COMMAND-NAMES*, complete the partial name; or the function might complete the string only to the point where it becomes ambiguous.

Example

If you are prompted for a command and enter Del, you can type a CTRL/space. There are two commands that begin with Del -- "Delete Buffer" and "Delete Window." The Editor therefore completes the string as far as "Delete ". You then have to enter at least B or W (indicating an unambiguous command) before typing the CTRL/space again.

You are not limited to this set of string tables. The facility is general and there are functions for creating new string tables. Strings are case-sensitive when stored or returned, but case-insensitive during string matching.

STYLES

A style is a collection of bindings of Editor keys, pointer actions, variables, and attributes, coupled with functions executed when a style is either activated or deactivated.

When a style is active in a buffer, it alters the current behavior of the Editor. An example of a style is one that causes the Editor to recognize the structure and syntax rules of LISP code. This behavior is appropriate only when you are editing LISP source code. Properly editing code written in FORTRAN or PL/I would require different Editor styles to be active.

Any number of styles can be active at one time when you are editing in a particular buffer. The styles can interact with one another to some extent, but one style can also shadow (hide) the behavior of another.

For example, you might be using the style called "VAX LISP" for editing LISP code, but you would like to specify your own command for indenting LISP text. The new indent command can be bound in another style called "LISP Indent" and "LISP Indent" can be made active in the current buffer. The binding of the indent command in "LISP Indent" shadows the binding of the indent command in "VAX LISP," but all other commands defined by "VAX LISP" are visible. Deactivating "LISP Indent" would "unshadow" the original indent command binding and make it visible again.

General Style Writing

The writer of an Editor style must take steps to ensure that any needed Editor support is present. For example, if the new style needs Editor variables bound in the style or in any buffers that use the style, the style must bind them directly or through use of a BUFFER-CREATION-HOOK function defined in that style. Editor variables that are defined with the VAX LISP Editor can be freely bound where needed.

A variable should be bound in a style whenever the style writer wants to retain information able to be used in any buffer having that style active. For example, the variable "EDT Paste Buffer" is bound in the EDT Emulation style. With this binding, any text that is cut from one buffer using "EDT Emulation" style can be pasted into any other buffer that also has "EDT Emulation" style.

Command bindings defined for a style should be thought of as recommendations. It is quite possible for the user to change the bindings local to that style. This means, for example, that help functions associated with a style should not assume that a particular key sequence is bound to a particular command.

STYLES (cont.)

Major/Minor Style Distinction

Any Editor Style can be bound as a major or minor style on a
per-buffer basis. The decision is normally made on the basis of the
extent of behavior changes introduced by the style. You make this
decision when you bind the style to a buffer. A buffer can have only
one major style active at a time, but any number of minor styles
active at the same time.

The set of global bindings of commands is extremely limited in the VAX
LISP Editor. This fact implies that any generally useful editing
session must have a powerful major style bound for each buffer. As
supplied in the VAX LISP Editor, the default major style is "EDT
Emulation," which supplies a set of commands and bindings that make
the Editor behave as EDT does. You may want to replace this default
style with "EMACS" or with one of your own that would make the Editor
behave in a different manner.

Minor styles are intended to be variations of the major style (or
other minor styles) that tailor the Editor behavior to more specific
needs. For example, the Editor comes with a "VAX LISP" style, which
modifies the Editor so that it has more knowledge of the syntax of
LISP. When this style is active, typing the) key not only inserts
the character but also locates and displays the corresponding (
character. Most of the editing capabilities are still vested in the
major style of the buffer.

Activation of Styles

There are two methods by which styles can be automatically activated
when a new buffer is created. One method works for all created
buffers; the other method can be tailored for specific attributes of
buffers.

The first method involves the Editor variables "Default Major Style"
and "Default Minor Styles." When a buffer is created, its major style
is set to the current value of "Default Major Style." As supplied, the
value of this variable is "EDT Emulation." If you change this value,
it changes the major style of the "Help" and "General Prompting"
buffers to the new style. The minor styles of the new buffer are set
from the list of styles contained in "Default Minor Styles." The
global value of this variable is initially NIL.

The second method allows you to activate minor styles in a new buffer
either according to the file-type of the associated file or whenever a
LISP object is being edited in the buffer.

The Editor variable "Default File Type Minor Styles" contains an
association list (a-list; see *COMMON LISP: The Language*). Each key

STYLES (cont.)

in the a-list is a string that is compared with the file type of the new buffer's associated file. Each element contains a list of minor styles to activate in any buffer with a file type matching the key. As supplied, this variable is bound in the global context and has a single element of the form ("LSP" . ("VAX LISP")). This means that the "VAX LISP" style is activated in any buffer containing a file having a file type of LSP.

A second Editor variable, "Default LISP Object Minor Styles," contains a list of minor styles to be activated in any buffer having an object being edited directly from LISP. As supplied, this variable is bound in the global context and contains the list ("VAX LISP"). This means that the "VAX LISP" style is automatically activated in any buffer used to edit a LISP function.

Order of Activation

When a buffer is created, first the major style is activated from the current value of "Default Major Style" (unless the major style is otherwise specified). Note that only the global value of this variable is used. The minor styles are then activated from the list found in "Default Minor Styles." The order of activation is in reverse order of the list. When the operation is complete, the order of search of the minor styles is the same as that of the list.

If the buffer contains a LISP object, the minor styles in the "Default LISP Object Minor Styles" list are next activated in reverse order. Any styles present in this list will be searched before any of the styles found in the "Default Minor Styles" list.

If the buffer has an associated file, the association list (a-list) contained in "Default File Type Minor Styles" is searched for an entry whose key matches the file type of the associated file. The styles contained in the entry are activated in reverse order so that the expected search order is maintained. Any styles present in this list will be searched before any of the styles found in the "Default Minor Styles" list.

Activation and Deactivation Functions

Activation and deactivation functions are associated with each style. When a style is made active, its activation function is executed; when a style is made inactive, its deactivation function is executed. You make a style active by using the SETF macro with BUFFER-MAJOR-STYLE or BUFFER-MINOR-STYLE-ACTIVE.

STYLES (cont.)

If the "EDT Emulation" major style is defined, and it is activated with

        (SETF (BUFFER-MAJOR-STYLE "TYPESET.LSP") "EDT Emulation"),

then the deactivation function of the old major style and the activation function of the new major style are executed. This process occurs unless the new and old styles are the same. Setting the major style to NIL causes the old major style to become inactive.

If the "VAX LISP" minor style is made active in the FACTORIAL buffer with

        (SETF (BUFFER-MINOR-STYLE-ACTIVE 'FACTORIAL "VAX LISP") T),

then its activation function is executed, and the new style is pushed onto the front of the list of active minor styles.

If an active minor style is again activated, and it is not the most recently activated minor style, then the following actions occur:

- The deactivation function associated with the style is executed.

- The original entry for the style is deleted from the list of active minor styles.

- The activation function associated with the style is executed.

- The style is pushed onto the front of the list of active minor styles.

If a style that is active in any Editor buffer is modified (for example, if a new variable is bound in that style), the modifications take effect in those buffers immediately.

CONCEPTS IN EDITOR PROGRAMMING

WINDOWS

A window is both an Editor object and the display mechanism of the Editor. Each window is a rectangular "opening" into a portion of an Editor buffer. This opening can be displayed on the screen of your display device, thereby showing you the current state of text within viewing range. As windows are Editor objects, they can be manipulated by various Editor functions.

Windows need not be the full height or width of the screen. Multiple windows can be on the screen at the same time. Moreover, windows can fully or partially overlap one another. The dimensions of a window are dynamic and can be changed either automatically by the Editor or under program control by a function you write.

Only text that lies within a buffer region can be displayed. A buffer can have multiple independent windows pointing into it. Since the text contained within a buffer can be both longer than a window (more lines) and wider (more characters per line), some provisions have been made to handle both circumstances.

Windows that are shorter than a buffer can be "moved" forward and backward through the buffer. This is known as "scrolling." In the VAX LISP Editor, it is the window that scrolls in the direction you specify and not the text. For example, when you scroll the window down (or forward) through the buffer, the text appears to move up to accommodate the new window display; actually, the window is moving down in the buffer. Windows can also be positioned absolutely in a buffer (at the beginning or end of a buffer, or at a particular line).

A window that is narrower than the text of the buffer is treated differently. The displayed text lines are either truncated on the right wherever the window ends (that is, only as many characters as will fit in the width of a window are displayed); or the lines "wrap around" (that is, the entire line of text is displayed even if it overflows onto one or more additional rows). Truncation and wrapping are indicated by special characters at the end of an affected line. The default is an underlined > for truncation, and an underlined < for wraparound, but you can specify different characters for any window.

The physical location of a window on the screen can be moved without affecting the portion of the buffer that the window is displaying; that is, you affect only where the text is displayed, not what is being displayed. A window can also exist as an Editor object but not be currently displayed. The Editor provides mechanisms for automatically placing windows in and removing windows from the display. You can also do this under the control of your program.

A window has an optional label -- a line of text that accompanies the window and that is displayed with it. The line can be displayed at the top, bottom, or either side of a window. By default, the label is placed at the bottom of the window and can be of any length up to the

WINDOWS (cont.)

length of whatever edge of the window the label is displayed on. It can contain any text you want -- for example, a buffer name or file name.

The label can be highlighted to give a visual separation from the buffer text being displayed. By default, this is done with reverse video on terminals that support this feature (VT100 compatible). The highlighting can be changed under program control. By default, the label is centered on whatever edge of the window is used for this display. You can control the label's position on a line, however, by specifying a starting position for the label -- an offset value that is the number of characters from the start of the window side (from the top of the window, if the label is on the right or left; or from the left-hand side, if the label is on the top or bottom).

Editor windows are of two types -- floating and anchored. Display of text in a window is unaffected by the type of the window. The distinction between the two lies in how they are treated by the display subsystem.

The simplest distinction is that floating windows are always displayed "on top" of (overlaying) any anchored windows, possibly obscuring them.

NOTE

With such overlaying, it is possible that the cursor that appears to be in the floating window is, in fact, indicating a position in the overlaid anchored window.

An anchored window cannot obscure a floating window. Another difference is that anchored windows are subject to automatic resizing and repositioning by the display subsystem. Floating windows are treated independently of other windows.

The two types of windows are identified with the keywords :FLOATING and :ANCHORED. By default, created windows are anchored if they are the full width of the screen and are displayed starting in column 1 (the left-hand side of the screen). Otherwise, they are floating. You can specify an explicit type for any windows you create and can also change the type of an existing window.

The display subsystem allows you to gain full control over the treatment of windows on the display -- where they are, what they overlap, and which are displayed. You can allow the Display Subsystem to exercise automatic control over the display of anchored windows. Floating windows are always assumed to be under program control.

37

WINDOWS (cont.)

The automatic treatment of anchored windows follows the rules given below:

1. Text in one anchored window is never obscured by text in another anchored window.

2. The bottom border of an anchored window is never obscured by another anchored window, but the top and side borders can be obscured.

3. Anchored windows are automatically adjusted in height when other anchored windows are added to, or removed from, the display. The adjustment is such that the text areas of anchored windows do not overlap one another, and the total height of all the anchored windows on the screen is the full height of the screen minus the height of the information area and the prompting window.

4. Any of the functions that manipulate windows on the screen assume that, unless explicit directions are given for the treatment of anchored windows (such as specifying height or relative position), all the currently displayed anchored windows are subject to automatic manipulation.

A record is also kept of the time a window is created. You can retrieve this information with the WINDOW-CREATION-TIME function. It returns a value in universal time.

PART III

EDITOR OBJECT DESCRIPTIONS

EDITOR OBJECT DESCRIPTIONS

This part describes each of the objects provided with the VAX LISP Editor. The objects are listed by name in alphabetical order.

The Editor objects listed include the following types:

- Functions

- Macros

- LISP global variables

- Named Editor objects

  - Buffers

  - Commands

  - Editor attributes

  - Editor variables

  - Styles

The other objects provided with the Editor are unnamed: lines, marks, regions, string tables, streams, windows. These objects are not described here, except for those that are bound to LISP variables. For instance, the string table bound to *EDITOR-COMMAND-NAMES* and the stream bound to *INFORMATION-AREA-OUTPUT-STREAM* are described here.

The following conventions are used in the individual object descriptions:

Named Editor Objects

Named Editor objects can have both a symbol and a display name, which is a string. For instance:

- EDT-EMULATION and "EDT Emulation" both refer to the same style object

- EDITOR-PROMPTING-BUFFER and "General Prompting" both refer to the same buffer object

The description of each named Editor object identifies both its symbol and its display name. The descriptions are alphabetized according to the objects' display names.

Functions Associated with Commands

The functions associated with commands are listed according to the display names of the commands. For instance, the function INDENT-LISP-REGION-COMMAND is described along with the command "Indent LISP Region". The symbol of the function and the symbol of the command are identical.

The command descriptions give the full format of the associated functions, including all optional arguments. Whether you can supply values for optional arguments depends on whether you are executing a command in the Editor or calling its associated function from LISP code:

- When executing a command within the Editor, you can supply a value only for the *prefix* parameter (by previously executing a command such as "Supply Prefix Argument" or "Supply EMACS Prefix"). If the Editor needs additional values to execute the command, it will either use default values or prompt for a needed value.

- When calling a command-associated function from LISP code, you can supply a value for any parameter.

Functions That Take Named Editor Objects

The descriptions of functions that take named Editor objects as arguments distinguish between those that can take an object specifier and those that can take only the object itself. In each case, the function description identifies the argument as either *object-type* or *object-type specifier*.

Object specifiers include the display names and symbols of named Editor objects. Functions that take objects (not object specifiers) cannot take a display name or symbol specifier. Recall that the symbol of a named Editor object does not evaluate to the object (see Chapter 1).

The distinction between functions that take only objects and functions that take specifiers is illustrated by BUFFER-WRITABLE and BUFFER-MAJOR-STYLE:

41

EDITOR OBJECT DESCRIPTIONS

- BUFFER-WRITABLE takes an Editor buffer. The argument can be
  specified only by a form that evaluates to a buffer object.
  For instance:

        (BUFFER-WRITABLE (CURRENT-BUFFER))
            or
        (BUFFER-WRITABLE (FIND-BUFFER 'EDITOR-HELP-BUFFER))
            or
        (BUFFER-WRITABLE *EDITOR-DEFAULT-BUFFER*)

- BUFFER-MAJOR-STYLE takes an Editor buffer specifier.   The
  argument can be specified by any of the following:

  - A buffer display name

        (BUFFER-MAJOR-STYLE "Mybuffer.lsp")

  - A buffer symbol

        (BUFFER-MAJOR-STYLE 'EDITOR-HELP-BUFFER)

  - A form that evaluates to a buffer object, such as

        (BUFFER-MAJOR-STYLE (CURRENT-BUFFER))
            or
        (BUFFER-MAJOR-STYLE (FIND-BUFFER 'EDITOR-HELP-BUFFER))
            or
        (BUFFER-MAJOR-STYLE *EDITOR-DEFAULT-BUFFER*)

ACTIVATE MINOR STYLE Command

Prompts the user for the name of a style and then activates that style as a minor style in the current buffer. Alternatives and completion are available during the prompt.

Category

> :GENERAL-PROMPTING

Display Name Format

> Activate Minor Style

Function Format

> ACTIVATE-MINOR-STYLE-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> The new minor style

ALTER-WINDOW-HEIGHT Function

Increases (for a positive *delta-value* argument) or decreases (for a negative *delta-value* argument) the height of the specified window by the specified number of rows. If the window is currently displayed and is of the anchored type, the heights of other displayed anchored windows are adjusted accordingly. The new height of the window cannot be less than 1. If the new height is too large to fit on the screen with the other displayed anchored windows, the height is set to the maximum height permissible. Calling this function causes the "Window Modification Hook" to be invoked.

Format

> ALTER-WINDOW-HEIGHT *window delta-value*

Arguments

window

> An Editor window. It need not be currently displayed.

delta-value

    An integer

Return Value

    The new height of the window

ANCHORED WINDOW SHOW LIMIT Editor Variable

Specifies the maximum number of Editor windows that can be displayed simultaneously by repeated calls to the function SHOW-WINDOW. If the number of anchored windows already displayed is greater than or equal to the value of this variable, then SHOW-WINDOW will remove the least recently used window when it displays another window. The default global value is 2.

The action of PUSH-WINDOW is not affected by this variable.

Display Name Format

    Anchored Window Show Limit

Symbol Format

    ANCHORED-WINDOW-SHOW-LIMIT

APROPOS Command

Displays a list of objects of the specified type in the "Help" buffer. Only objects whose name contains the specified string are listed. If the object type or string is NIL, the user is prompted for it in the Editor prompting window. An object type of T signifies that all Editor objects containing the specified string are to be displayed.

Category

    :GENERAL-PROMPTING

Display Name Format

    Apropos

Function Format

    APROPOS-COMMAND *prefix* &OPTIONAL *type string*

Arguments

prefix

Ignored

type

- A named Editor object type (ATTRIBUTE, BUFFER, COMMAND, VARIABLE, or STYLE)

- SYMBOL to search all LISP objects

- T to search all named editor objects

string

The string that is to be matched in the object names

Return Value

None

APROPOS-STRING-TABLE Function

Searches the specified string table for all entries whose key contains the specified string as a substring. It returns a list of all such keys in alphabetical order. If the string is of zero length, all the keys in the string table are returned.

Format

APROPOS-STRING-TABLE *string string-table*

Arguments

string

A string to be used as a search string

string-table

A string table to be searched

Return Value

An alphabetical list of keys

APROPOS WORD Command

Does an APROPOS of the word at the mark and displays the result in the Help buffer. If the mark is not supplied, it defaults to the current buffer point.

Display Name Format

Apropos Word

Function Format

APROPOS-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

Ignored

mark

An Editor mark that defaults to the current buffer point

Return Value

Undefined

ATTENTION Function

Gains the attention of the user.

Format

ATTENTION

Arguments

None

Return Value

NIL

ATTRIBUTE-NAME Function

Takes an attribute specifier as an argument and returns the display name of the attribute.

Format

ATTRIBUTE-NAME *attribute*

Arguments

attribute

An attribute specifier

Return Value

A string that is the display name of the attribute

BACKWARD CHARACTER Command

Moves the point in the current window back one character if the prefix argument is NIL. If you specify an integer prefix argument, the point is moved backward (or forward, if the prefix is negative) by the number of characters you indicated. An error is signaled if the point is at the beginning of a buffer.

Display Name Format

Backward Character

Function Format

BACKWARD-CHARACTER-COMMAND *prefix*

Arguments

prefix

A fixnum specifying how many characters to move

Return Value

The updated buffer point mark

BACKWARD KILL RING Command

Rotates the kill ring backward by the number of elements specified by the prefix.

Category

    :KILL-RING

Display Name Format

    Backward Kill Ring

Function Format

    BACKWARD-KILL-RING-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    Undefined

BACKWARD PAGE Command

Moves the point in the current buffer backward one page if prefix is NIL. If you specify an integer prefix argument, the point is moved backward (or forward, if prefix is negative) by the number of pages you indicated. A page delimiter is a character that has a "Page Delimiter" attribute value of 1.

Display Name Format

    Backward Page

Function Format

    BACKWARD-PAGE-COMMAND *prefix*

Arguments

prefix

    A fixnum specifying how many pages to move

ATTRIBUTE-NAME Function

Takes an attribute specifier as an argument and returns the display name of the attribute.

Format

ATTRIBUTE-NAME *attribute*

Arguments

attribute

An attribute specifier

Return Value

A string that is the display name of the attribute

BACKWARD CHARACTER Command

Moves the point in the current window back one character if the prefix argument is NIL. If you specify an integer prefix argument, the point is moved backward (or forward, if the prefix is negative) by the number of characters you indicated. An error is signaled if the point is at the beginning of a buffer.

Display Name Format

Backward Character

Function Format

BACKWARD-CHARACTER-COMMAND *prefix*

Arguments

prefix

A fixnum specifying how many characters to move

Return Value

The updated buffer point mark

BACKWARD KILL RING Command

Rotates the kill ring backward by the number of elements specified by the prefix.

Category

> :KILL-RING

Display Name Format

> Backward Kill Ring

Function Format

> BACKWARD-KILL-RING-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> Undefined

BACKWARD PAGE Command

Moves the point in the current buffer backward one page if prefix is NIL. If you specify an integer prefix argument, the point is moved backward (or forward, if prefix is negative) by the number of pages you indicated. A page delimiter is a character that has a "Page Delimiter" attribute value of 1.

Display Name Format

> Backward Page

Function Format

> BACKWARD-PAGE-COMMAND *prefix*

Arguments

prefix

> A fixnum specifying how many pages to move

Return Value

> The updated buffer point mark

BACKWARD SEARCH Command

Prompts for an argument string if the user does not supply one. The string is used as the pattern for a backward search. If the search is successful, the buffer point is moved to the beginning of the first matching string. If the user does not specify a string when prompted, the command takes the value of the Editor Variable "Last Search String." If the user specifies a prefix argument, n, this command looks for the nth occurrence of the pattern.

Display Name Format

> Backward Search

Function Format

> BACKWARD-SEARCH-COMMAND *prefix* &OPTIONAL *string*

Arguments

prefix

> The fixnum repeat count

string

> The string to search for. If you do not specify a string when prompted, string defaults to the value of the Editor variable "Last Search String."

Return Value

> The modified point

BACKWARD WORD Command

Moves the point back to the end of the preceding word. If you specify an integer prefix argument, the point is moved back the number of words you indicate. Words are delimited by characters having a "Word Delimiter" attribute value of 1.

Display Name Format

> Backward Word

Function Format

    BACKWARD-WORD-COMMAND *prefix*

Arguments

prefix

    A positive integer or NIL

Return Value

    The modified point

BEGINNING OF BUFFER Command

Moves the point to the beginning of the current buffer.

Display Name Format

    Beginning of Buffer

Function Format

    BEGINNING-OF-BUFFER-COMMAND *prefix*

Arguments

prefix

    Ignored

Return Value

    The modified point

BEGINNING OF LINE Command

Moves the point to the beginning of the current line. If you specify
an integer prefix argument, the point is moved down the number of
lines you indicated (or up, if the prefix is negative) and then to the
beginning of the new line.

Display Name Format

    Beginning of Line

Function Format

>   BEGINNING-OF-LINE-COMMAND *prefix*

Arguments

prefix

>   An integer or NIL

Return Value

>   The updated buffer point mark

BEGINNING OF OUTERMOST FORM Command

Moves the buffer point from inside a LISP form to the beginning of the outermost form surrounding it. If the point is in between two outer forms, it is moved to the beginning of the preceding one. If there is no preceding outer form, an Editor error is signaled. An outermost form is one whose opening parenthesis is in the leftmost column on the screen.

Display Name Format

>   Beginning of Outermost Form

Function Format

>   BEGINNING-OF-OUTERMOST-FORM-COMMAND *prefix*

Arguments

prefix

>   Ignored

Return Value

>   The updated buffer point

BEGINNING OF PARAGRAPH Command

Moves the specified mark to the beginning of the paragraph. The mark defaults to the current buffer point.

Display Name Format

>   Beginning of Paragraph

Function Format

>   BEGINNING-OF-PARAGRAPH-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

>   Ignored

mark

>   An Editor mark that defaults to the current buffer point

Return Value

>   The updated mark

BEGINNING OF WINDOW Command

Moves the cursor to the beginning of the current window.

Display Name Format

>   Beginning of Window

Function Format

>   BEGINNING-OF-WINDOW-COMMAND *prefix* &OPTIONAL *mark window*

Arguments

prefix

>   Ignored

mark

>   The mark to be placed at the beginning of the window. It
>   defaults to the current buffer point.

window

>   The window in which the mark is to be moved. It defaults to the
>   current window.

Function Format

> BEGINNING-OF-LINE-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> The updated buffer point mark

BEGINNING OF OUTERMOST FORM Command

Moves the buffer point from inside a LISP form to the beginning of the outermost form surrounding it. If the point is in between two outer forms, it is moved to the beginning of the preceding one. If there is no preceding outer form, an Editor error is signaled. An outermost form is one whose opening parenthesis is in the leftmost column on the screen.

Display Name Format

> Beginning of Outermost Form

Function Format

> BEGINNING-OF-OUTERMOST-FORM-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> The updated buffer point

BEGINNING OF PARAGRAPH Command

Moves the specified mark to the beginning of the paragraph. The mark defaults to the current buffer point.

Display Name Format

Beginning of Paragraph

Function Format

BEGINNING-OF-PARAGRAPH-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

Ignored

mark

An Editor mark that defaults to the current buffer point

Return Value

The updated mark

BEGINNING OF WINDOW Command

Moves the cursor to the beginning of the current window.

Display Name Format

Beginning of Window

Function Format

BEGINNING-OF-WINDOW-COMMAND *prefix* &OPTIONAL *mark window*

Arguments

prefix

Ignored

mark

The mark to be placed at the beginning of the window. It
defaults to the current buffer point.

window

The window in which the mark is to be moved. It defaults to the
current window.

Return Value

The current buffer point

BIND-ATTRIBUTE Function

Takes a defined Editor attribute as an argument and creates a binding
of that attribute in the specified context with the specified type and
value.

Format

```
BIND-ATTRIBUTE attribute &KEY :TYPE
                             :CONTEXT
                             :INITIAL-VALUE
```

Arguments

attribute

An Editor attribute specifier

:TYPE

A LISP type specifier. The default is (mod 2).

:CONTEXT

An Editor context specifier. The default is :GLOBAL.

:INITIAL-VALUE

The attribute value that all characters will initially have for
this attribute in this context. It must be of the type specified
by :TYPE. The default is 0.

Return Value

The attribute symbol

BIND COMMAND Command

Prompts the user for a command name, a key sequence, and a binding
context.   This command is useful for binding commands to keys without
leaving the context of the Editor. Completion and Alternatives are
available for the command name and for the style or buffer name
depending on the desired binding context. The key sequence must be
entered literally as the sequence of characters to bind. This
frequently requires the quoting of control characters.

53

Category

:GENERAL-PROMPTING

Display Name Format

Bind Command

Function Format

BIND-COMMAND-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The function associated with the command

BIND-COMMAND Function

Binds the specified key-sequence to the specified command in the specified context.

Format

BIND-COMMAND *command key-sequence* &OPTIONAL *context*

Arguments

command

An Editor command specifier command

key-sequence

A character or a sequence of characters. The key sequence cannot contain the characters CTRL/S or CTRL/Q. It should not contain the current cancel character (CTRL/C by default).

context

The context in which to bind the command. The argument context defaults to :GLOBAL.

Return Value

The function associated with the command

BIND-POINTER-COMMAND Function

Binds the specified action of the pointing device to the specified command in the specified context. The possible actions of the pointing device are a button transition (depressing or releasing) or a movement of the pointer cursor. The Editor invokes the bound command in response to a pointer action only when the pointer cursor is in the current window.

The :BUTTON-STATE parameter is used to indicate that one or more pointer buttons must be in a down state for the specified *pointer-action* to invoke the command. If the *pointer-action* argument is a button transition, then any value in the :BUTTON-STATE argument that corresponds to that button is ignored.

Format

        BIND-POINTER-COMMAND *command pointer-action* &KEY :CONTEXT
                                                          :BUTTON-STATE

Arguments

command

An Editor command specifier

pointer-action

A keyword, a button constant, or a list. The possible values are:

:MOVEMENT

The command is invoked by any movement of the pointer cursor within the current window. Cursor movement is defined as a movement across at least one character in any direction.

A button constant

The command is invoked by depressing the pointer button that corresponds to the constant. The constants are specified as UIS:POINTER-BUTTON-n, starting with UIS:POINTER-BUTTON-1 for the left-most button. Note that the symbols for button constants are located in the "UIS" package; see *VAX LISP/VMS Graphics Programming Guide* for more information.

A list whose CAR is a button constant

> If the CADR is non-NIL, the command is invoked when the pointer button corresponding to the CAR is depressed.

> If the CADR is NIL, the command is invoked when the pointer button corresponding to the CAR is released.

:CONTEXT *value*

> A context specifier. The default is :GLOBAL.

:BUTTON-STATE *value*

> A button constant or the LOGAND of two or more button constants. The button(s) indicated must be in a down state for the specified *pointer-action* to invoke the command.

> If a button transition is specified as the *pointer-action* argument, any value that corresponds to that button in the :BUTTON-STATE argument is ignored.

Return Value

> The function associated with the command

BIND-VARIABLE Function

Binds the specified Editor variable in the specified context. You get a warning if you attempt to bind a variable in a context in which it is already bound. The function specified in the :BIND-HOOK argument of DEFINE-EDITOR-VARIABLE is called and passed the symbol and the binding context.

Format

```
BIND-VARIABLE symbol &KEY :CONTEXT
                          :SET-VALUE-HOOK :SET-FUNCTION-HOOK
                          :INITIAL-VALUE :INITIAL-FUNCTION
```

Arguments

symbol

> An Editor variable specifier

:CONTEXT

> An Editor context specifier that defaults to :GLOBAL

:SET-VALUE-HOOK

> A function that is invoked whenever the value of the variable is
> set in the specified context. The function is called with three
> arguments -- the variable, the context of the variable, and the
> new value. It defaults to NIL.

:SET-FUNCTION-HOOK

> A function that is invoked whenever the function slot is changed
> in the specified context. The function is called with three
> arguments -- the variable, the context of the variable, and the
> new function. It defaults to NIL.

:INITIAL-VALUE

> The value given to the binding of the variable created in the
> specified context. It defaults to NIL.

:INITIAL-FUNCTION

> The function bound to the variable in the specified context.   It
> defaults to NIL.

Return Value

> The symbol that names the variable

BREAK-LINE Function

Breaks a line at the position pointed to by the specified mark.

Format

> BREAK-LINE *mark*

Arguments

mark

> A mark specifying the position at which a line is to be broken.
> If the mark is left-inserting, the mark is moved to the beginning
> of the new line. If the mark is right-inserting, the mark
> remains at the end of the original line.

Return Value

> The updated mark

BUFFER-CHECKPOINTED Function

Returns the pathname of the file where checkpoints of the specified buffer will be written, or NIL if the buffer is not being checkpointed. You can change either the file to which the buffer is checkpointed or make the buffer not checkpointed by using this form with SETF. When changing the value, you can set three possible values:

- NIL makes the buffer not checkpointed.

- A pathname writes the buffer to that file.

- T writes the buffer checkpoints to a file name the Editor creates from the name of the object being edited.

Format

BUFFER-CHECKPOINTED *buffer*

Arguments

buffer

An Editor buffer

Return Value

A pathname or NIL

BUFFER-CHECKPOINTED-TIME Function

Returns the universal time that the buffer was last checkpointed; or NIL, if it has not been checkpointed.

Format

BUFFER-CHECKPOINTED-TIME *buffer*

Arguments

buffer

An Editor buffer

Return Value

A value in universal time or NIL

BUFFER CREATION HOOK Editor Variable

Specifies a hook function that is called whenever a new buffer is created. The hook function is passed one argument -- the new buffer. The function is called after the complete buffer context is created, and in the context of the new buffer.

Display Name Format

> Buffer Creation Hook

Symbol Format

> BUFFER-CREATION-HOOK

BUFFER-CREATION-TIME Function

Returns the universal time at which the specified buffer was created. For information on universal time, see *COMMON LISP: The Language.*

Format

> BUFFER-CREATION-TIME *buffer*

Arguments

buffer

> The buffer for which the time is desired

Return Value

> The universal time at which the buffer was created

BUFFER DELETION HOOK Editor Variable

Specifies a hook function that is called just before a buffer is deleted. It is called in the context of the buffer to be deleted and before any alterations are made to the buffer. It is passed one argument -- the buffer to be deleted.

Display Name Format

> Buffer Deletion Hook

Symbol Format

> BUFFER-DELETION-HOOK

BUFFER-END Function

Changes the specified mark so that it points to the end of the buffer.

Format

BUFFER-END *mark* &OPTIONAL *buffer*

Arguments

mark

An Editor mark

buffer

An Editor buffer. This defaults to the buffer the mark is pointing into.

Return Value

The modified mark

BUFFER ENTRY HOOK Editor Variable

Specifies a hook function that is invoked whenever a different buffer becomes current. The function is called with one argument -- the new buffer -- and is evaluated in the context of the new buffer.

Display Name Format

Buffer Entry Hook

Symbol Format

BUFFER-ENTRY-HOOK

BUFFER EXIT HOOK Editor Variable

Specifies a hook function that is invoked whenever a different buffer becomes current. The function is called with one argument -- the old buffer -- and is evaluated in the context of the old buffer.

Display Name Format

Buffer Exit Hook

Symbol Format

    BUFFER-EXIT-HOOK

BUFFER-HIGHLIGHT-REGIONS Function

Returns a list of the highlight regions associated with the specified
buffer, or NIL if there are no such regions.

Format

    BUFFER-HIGHLIGHT-REGIONS *buffer*

Arguments

buffer

    An Editor buffer

Return Value

    A list of highlight regions or NIL

BUFFER-MAJOR-STYLE Function

Returns the major style associated with the specified buffer, or NIL
if there is none. You can use SETF with BUFFER-MAJOR-STYLE to change
the major style of a buffer. This action causes the "Major Style
Activation Hook" to be invoked.

Format

    BUFFER-MAJOR-STYLE *buffer*

Arguments

buffer

    An Editor buffer specifier

Return Value

    The major style of the buffer, or NIL

BUFFER-MINOR-STYLE-ACTIVE Function

Returns T if the specified style is active in the specified buffer. You can use SETF with BUFFER-MINOR-STYLE-ACTIVE to add minor styles to, or delete them from, a buffer. This action causes the "Minor Style Activation Hook" to be invoked.

Format

> BUFFER-MINOR-STYLE-ACTIVE *buffer style*

Arguments

buffer

> An Editor buffer specifier

style

> An Editor style specifier

Return Value

> T or NIL

BUFFER-MINOR-STYLE-LIST Function

Returns a list of the minor styles active in the specified buffer. The order of the styles is the same as the search order.

Format

> BUFFER-MINOR-STYLE-LIST *buffer*

Arguments

buffer

> An Editor buffer specifier

Return Value

> A list of the minor styles

BUFFER-MODIFIED-P Function

Is a predicate that returns T if the buffer has been modified and NIL if it has not. You can use SETF with BUFFER-MODIFIED-P to change the status of whether or not the buffer is considered to be modified.

Format

> BUFFER-MODIFIED-P *buffer*

Arguments

buffer

> An Editor buffer

Return Value

> T or NIL

BUFFER-NAME Function

Returns the name of the buffer you specify. You can use SETF with BUFFER-NAME to change the name of the buffer. This action causes the "Buffer Name Hook" to be invoked.

Format

> BUFFER-NAME *buffer*

Arguments

buffer

> An Editor buffer

Return Value

> The buffer name

BUFFER NAME HOOK Editor Variable

Is a hook function that is called with the buffer and the new name as arguments when the name of a buffer is changed.

Display Name Format

> Buffer Name Hook

Symbol Format

BUFFER-NAME-HOOK

BUFFER-OBJECT Function

Returns the object being edited in the buffer you specify. This object is a pathname in the case of a file, a symbol in the case of a LISP function or the value of a symbol, or the form of a generalized variable. You can use SETF with BUFFER-OBJECT to change the object being edited in the buffer. This action causes the "Buffer Object Hook" to be invoked.

Format

BUFFER-OBJECT *buffer*

Arguments

buffer

An Editor buffer

Return Value

The object being edited in the buffer

BUFFER OBJECT HOOK Editor Variable

Is a hook function that is called with the buffer and the new object as arguments when the object associated with a buffer is changed.

Display Name Format

Buffer Object Hook

Symbol Format

BUFFER-OBJECT-HOOK

BUFFER-PERMANENT Function

Is a predicate that returns T if the specified buffer is permanent (that is, if it cannot be deleted by the DELETE-BUFFER function). It returns NIL otherwise. You can change the permanent status of a buffer by using the SETF macro with this form.

Format

> BUFFER-PERMANENT *buffer*

Arguments

buffer

> An Editor buffer

Return Value

> T or NIL

BUFFER-POINT Function

Returns the point of the buffer you specify.

Format

> BUFFER-POINT *buffer*

Arguments

buffer

> An Editor buffer

Return Value

> A mark that is the point for the buffer

BUFFER-REGION Function

Returns the region of the buffer you specify.

Format

> BUFFER-REGION *buffer*

Arguments

buffer

> An Editor buffer

Return Value

> The buffer region

BUFFER RIGHT MARGIN Editor Variable

Can be set to an integer that specifies the last character position at which text can be inserted in each line by means of the commands "Self Insert" and "Quoted Insert". If more text is inserted than will fit on a line of this length, then the line is automatically broken at the last word break within the right margin. In the default Editor, this variable is bound globally and set to NIL.

Note that this variable does not affect the operation of other text-inserting commands, such as "EDT Paste" and "Yank".

Display Name Format

> Buffer Right Margin

Symbol Format

> BUFFER-RIGHT-MARGIN

BUFFER SELECT MARK Editor Variable

Is used by a number of commands that need to retain a special mark indicating a position in a buffer. It is bound to a mark by commands that select regions of a buffer. See also "Buffer Select Region".

Display Name Format

> Buffer Select Mark

Symbol Format

> BUFFER-SELECT-MARK

BUFFER SELECT REGION Editor Variable

Is bound by several commands to a "selected" region in a buffer. This variable is created as a local variable to each buffer. See also "Buffer Select Mark".

Display Name Format

> Buffer Select Region

Symbol Format

> BUFFER-SELECT-REGION

BUFFER-START Function

Changes mark so that it points to the beginning of the buffer.

Format

> BUFFER-START *mark* &OPTIONAL *buffer*

Arguments

mark

> An Editor mark

buffer

> A buffer. If no buffer is specified, the default is the buffer
> the mark points into.

Return Value

> The modified mark

BUFFER-TYPE Function

Returns a keyword indicating the type of object being edited in the
specified buffer.   This function returns NIL if there is no LISP
object or file associated with the buffer.

Format

> BUFFER-TYPE *buffer*

Arguments

buffer

> An Editor buffer

Return Value

One of the following keywords or NIL:

    :FILE - the object is a file
    :FUNCTION - the object is the function definition of a symbol
    :VALUE - the object is the value of a symbol
    :SETF-FORM - the object is a generalized variable acceptable to
    SETF

BUFFER-VARIABLES Function

Returns a list of Editor variables bound in the specified buffer.

Format

    BUFFER-VARIABLES *buffer*

Arguments

buffer

    An Editor buffer

Return Value

    A list of Editor variables (symbols)

BUFFER-WINDOWS Function

Returns a list of the windows that are associated with the specified
buffer. This list can include windows that are not visible.

Format

    BUFFER-WINDOWS *buffer*

Arguments

buffer

    An Editor buffer

Return Value

    A list of the windows that open into the buffer

BUFFER-WRITABLE Function

Returns T if modifications to the specified buffer can be written back as a new version of the file being edited or an update of the LISP object being edited, or NIL if they cannot. You can use SETF with BUFFER-WRITABLE to change the status of whether or not buffer modifications can be written.

Format

    BUFFER-WRITABLE *buffer*

Arguments

buffer

    An Editor buffer

Return Value

    T or NIL

BUFFER-WRITTEN-TIME Function

Returns the universal time that the buffer you specify was last "written" by the Write Current Buffer or Write Modified Buffers command, or NIL if the buffer has never been written. If the buffer is associated with a file, this function returns the time when the buffer contents were last written to the file. If the buffer is associated with a symbol or a SETF form, the time is the last time that the buffer contents were evaluated. (See *COMMON LISP: The Language* for a description of universal time.)

Format

    BUFFER-WRITTEN-TIME *buffer*

Arguments

buffer

    The buffer for which you want the time

Return Value

    Universal time that the buffer was last written, or NIL

BUFFERP Function

Is a predicate that returns T if its argument is a buffer.

Format

    BUFFERP *object*

Arguments

object

    Anything

Return Value

    T or NIL

CANCEL-CHARACTER Function

Returns the character that, if typed while in the Editor, causes the current action to be terminated. The initial value is #\^C. You can change the cancel character by using this form with SETF.

<div align="center">

NOTE

</div>

> The cancel character must be an ASCII control character whose character code is in the range 0 to 31. Also, it cannot be #\Return, #\Linefeed, #\Escape, #\^Q or #\^S.

Format

    CANCEL-CHARACTER

Arguments

    None

Return Value

    The current cancel character

CAPITALIZE REGION Command

Capitalizes all the words in the current select region.

Display Name Format

Capitalize Region

Function Format

CAPITALIZE-REGION-COMMAND *prefix* &OPTIONAL *region*

Arguments

prefix

Ignored

region

A region that defaults to the value of the "Buffer Select Region" Editor variable

Return Value

The modified region

CAPITALIZE WORD Command

Capitalizes the current word.

Display Name Format

Capitalize Word

Function Format

CAPITALIZE-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

Ignored

mark

An Editor mark that defaults to the current buffer point

Return Value

    A region containing the capitalized word

CATEGORY-COMMANDS Function

Returns a list of Editor commands that are cataloged under the specified category.

Format

    CATEGORY-COMMANDS *category*

Arguments

category

    A symbol used as a command category

Return Value

    A list of Editor command symbols

CENTER-WINDOW Function

Causes the display in the specified window to be adjusted so that the line pointed to by the specified mark is centered in the display window.

Format

    CENTER-WINDOW *window mark*

Arguments

window

    An Editor window

mark

    An Editor mark that must point into the same buffer the window is associated with

Return Value

    The mark

CHARACTER-ATTRIBUTE Function

Looks up and returns the value the specified attribute has for the character. You can use SETF with CHARACTER-ATTRIBUTE to modify the attributes of a character. Changing the value of a character attribute causes the "Character Attribute Hook" to be invoked.

Format

CHARACTER-ATTRIBUTE *attribute character* &OPTIONAL *context*

Arguments

attribute

An Editor attribute

character

The character whose attribute value you want

context

A context specifier. Defaults to the current context, unless the function is used with SETF in which case the context defaults to :GLOBAL.

Return Value

The attribute value for the specified character

CHARACTER ATTRIBUTE HOOK Editor Variable

Is a hook function that is called, with the attribute, character, context, and new value as arguments, just before the value of a character attribute is changed.

Display Name Format

Character Attribute Hook

Symbol Format

CHARACTER-ATTRIBUTE-HOOK

CHARACTER-OFFSET Function

Changes the specified mark so that it points *n* characters after its former position (or before its former position, if *n* is negative). If there are not *n* characters after the mark position (or before, if *n* is negative), mark is not modified, and NIL is returned.

Format

    CHARACTER-OFFSET *mark n*

Arguments

mark

    An Editor mark

n

    A fixnum

Return Value

    The modified mark or NIL

CHECKPOINT-BUFFER Function

Checkpoints the specified buffer to the specified file, which defaults to the checkpoint file previously specified for the buffer.

Format

    CHECKPOINT-BUFFER *buffer* &OPTIONAL *pathname*

Arguments

buffer

    An Editor buffer

pathname

    An optional pathname specifier that defaults to the checkpoint pathname specified earlier for the buffer

Return Value

Two values:

1. The truename of the checkpoint file written to, or NIL if no pathname existed. (For an explanation of the truename of a file, see the explanation of Pathname in *COMMON LISP: The Language.*)

2. The number of records written to the file.

CHECKPOINT-FREQUENCY Function

Returns an integer that gives the frequency at which file checkpointing is being performed. The frequency is measured in keystrokes, but only those that modify a buffer. If checkpointing has been disabled, the function returns NIL. The default frequency is 350. You can use SETF with CHECKPOINT-FREQUENCY to change the default value. To disable checkpointing, specify a value of NIL.

Format

> CHECKPOINT-FREQUENCY

Arguments

> None

Return Value

> The checkpointing frequency or NIL

CLEAR-INFORMATION-AREA Function

Clears the text in the Editor information area.

Format

> CLEAR-INFORMATION-AREA

Arguments

> None

Return Value

> None

CLOSE OUTERMOST FORM Command

Inserts at the mark the number of list-terminator characters needed to close the outermost LISP form. The mark defaults to the current buffer point. If the outermost form is already closed or if no outermost form is found, a message is displayed and no action occurs. (See "LISP Syntax" attribute, especially the value :LIST-TERMINATOR.)

Display Name Format

    Close Outermost Form

Function Format

    CLOSE-OUTERMOST-FORM-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

    Ignored

mark

    An Editor mark that defaults to the current buffer point

Return Value

    Undefined

COMMAND-CATEGORIES Function

Returns a list of the categories for the specified Editor command or NIL if there are no categories for this command. See Section 2.2.3.1 on categories in Chapter 2 of this manual.

Format

    COMMAND-CATEGORIES *command*

Arguments

command

    An Editor command specifier

Return Value

    A list of categories or NIL

COMMAND-NAME Function

Takes an Editor command specifier and returns the display name of the command.

Format

COMMAND-NAME *command*

Arguments

command

An Editor command specifier (display name or symbol)

Return Value

The display name of the command

COMPLETE-STRING Function

Searches through all strings in the specified Editor string table for those having the string argument as an initial substring. The searching is case insensitive.

Format

COMPLETE-STRING *string table*

Arguments

string

The character string to be searched for

table

An Editor string table

Return Value

Four values:

1. A string that is the maximum beginning portion common to all the strings found that match the string argument.

2. The value of a corresponding entry if a unique match was found; NIL otherwise.

3. T, if the second returned value is valid, or NIL, if the second returned value is not valid. (This allows NIL to be a valid value for a string table entry.)

4. T, if there are additional entries that start with the specified string. This is helpful to distinguish the case where a string is the key for a specific entry, and there are additional keys that begin with this string. For example, "Ed" and "Edit File" are both valid commands.

COPY-MARK Function

Returns a new mark pointing to the same position as the specified mark.

Format

    COPY-MARK *mark* &OPTIONAL *mark-type*

Arguments

mark

    An Editor mark

mark-type

    Either :LEFT-INSERTING, :RIGHT-INSERTING, or :TEMPORARY. The default is the type of the mark specified.

Return Value

    A new Editor mark

COPY-REGION Function

Takes an Editor region as an argument and returns a new disembodied region that contains a copy of the text of the region you specified. The new region does not share any lines with the original region.

Format

    COPY-REGION *region*

Arguments

region

    An Editor region

Return Value

A new Editor region

COUNT-REGION Function

Returns both the number of characters that are in the specified region and the number of lines that are in the region. A break between lines counts as a single character. The line count is always at least 1.

Format

COUNT-REGION *region*

Arguments

region

An Editor region

Return Value

Two values:

1. The number of characters in the region

2. The number of lines in the region

CURRENT-BUFFER Function

Returns the currently active Editor buffer. You can use SETF with CURRENT-BUFFER to change the buffer that is considered current. Changing the value of CURRENT-BUFFER causes the "Buffer Exit Hook" and the "Buffer Entry Hook" to be invoked.

Format

CURRENT-BUFFER

Arguments

None

Return Value

The current Editor buffer

CURRENT-BUFFER-POINT Function

Returns the mark that is the point for the current buffer. Calling this function is substantially faster than using the form (BUFFER-POINT (CURRENT-BUFFER)).

Format

    CURRENT-BUFFER-POINT

Arguments

    None

Return Value

    The buffer point of the current buffer

CURRENT-COMMAND-FUNCTION Variable

Is bound to the Editor command function currently being executed. The binding is established just before the function is called.

CURRENT-WINDOW Function

Returns the currently active Editor window. You can use SETF with CURRENT-WINDOW to change the window considered current. Changing the value of CURRENT-WINDOW causes the "Switch Window Hook" to be invoked. This change may also cause the value of (CURRENT-BUFFER) to be changed; if so, the "Buffer Exit Hook" and the "Buffer Entry Hook" are also invoked.

Format

    CURRENT-WINDOW

Arguments

    None

Return Value

    The current Editor window

CURRENT WINDOW POINTER PATTERN Editor Variable

Specifies a 16x16 bitmap that determines the pointer cursor pattern when the pointer is in the current Editor window. When set to NIL, the pointer cursor pattern is the VAXstation default (an arrow). See the functions SET-POINTER-PATTERN and MAKE-BITMAP in the *VAX LISP/VMS Graphics Programming Guide.*

Display Name Format

Current Window Pointer Pattern

Symbol Format

CURRENT-WINDOW-POINTER-PATTERN

CURRENT WINDOW POINTER PATTERN X Editor Variable

Specifies the horizontal coordinate of the active pixel of the bitmap specified by the Editor variable "Current Window Pointer Pattern". The possible values are an integer in the range 0-15 or NIL. See the function SET-POINTER-PATTERN in the *VAX LISP/VMS Graphics Programming Guide.*

Display Name Format

Current Window Pointer Pattern X

Symbol Format

CURRENT-WINDOW-POINTER-PATTERN-X

CURRENT WINDOW POINTER PATTERN Y Editor Variable

Specifies the vertical coordinate of the active pixel of the bitmap specified by the Editor variable "Current Window Pointer Pattern", relative to the lower edge of the bitmap. The possible values are an integer in the range 0-15 or NIL. See the function SET-POINTER-PATTERN in the *VAX LISP/VMS Graphics Programming Guide.*

Display Name Format

Current Window Pointer Pattern Y

Symbol Format

CURRENT-WINDOW-POINTER-PATTERN-Y

DEACTIVATE MINOR STYLE Command

Prompts the user for the name of a minor style active in the current buffer. It then deactivates that style in the current buffer. Alternatives and completion are available during the prompt. An Editor error is signaled if the style is not active.

Category

>:GENERAL-PROMPTING

Display Name Format

>Deactivate Minor Style

Function Format

>DEACTIVATE-MINOR-STYLE-COMMAND *prefix*

Arguments

prefix

>Ignored

Return Value

>The style that was deactivated

DEFAULT BUFFER VARIABLES Editor Variable

Is bound to a list of Editor variables that are to have local bindings in newly created buffers. Each element of the list is of the form:

>*variable-name* | *(variable-name initial-value initial-function)*

Each of the Editor variables listed is bound in the context of the new buffer. The initial value and function of this variable are NIL unless specified in a list element.

Display Name Format

>Default Buffer Variables

Symbol Format

>DEFAULT-BUFFER-VARIABLES

DEFAULT FILETYPE MINOR STYLES Editor Variable

Specifies an association list that maps file types to Editor styles. When a file is associated with a buffer (for example, in the "Edit File" command), the file type is looked up in this association list. If an entry is found, it must specify a style or list of styles that are to be activated as minor styles in the buffer.

The default is the list (("LSP" . "VAX LISP")). This means that a file type of LSP activates the "VAX LISP" minor style.

Display Name Format

Default Filetype Minor Styles

Symbol Format

DEFAULT-FILETYPE-MINOR-STYLES

DEFAULT LISP OBJECT MINOR STYLES Editor Variable

Specifies a list of minor styles that are to be activated in a buffer used to edit a LISP object. The default list is ("VAX LISP"), which means that the VAX LISP style is activated.

Display Name Format

Default LISP Object Styles

Symbol Format

DEFAULT-LISP-OBJECT-MINOR-STYLES

DEFAULT MAJOR STYLE Editor Variable

Is bound to the Editor style that is the default major style for a newly created buffer. The initial value is "EDT Emulation" style. You can use SETF with the VARIABLE-VALUE function to change this default. Changing the value of this variable changes the major style for the "Help" and "General Prompting" buffers. Any previously created buffers retain their original major styles. See

Display Name Format

Default Major Style

Symbol Format

        DEFAULT-MAJOR-STYLE

DEFAULT MINOR STYLES Editor Variable

Is bound to a list of Editor styles that are the default minor styles
for a newly created buffer. The initial value of this variable is
NIL. Any previously created buffers retain their original minor
styles.

Display Name Format

        Default Minor Styles

Symbol Format

        DEFAULT-MINOR-STYLES

DEFAULT SEARCH CASE Editor Variable

Is bound to a keyword that specifies whether differences in case are
to be ignored in searches. If the keyword is :CASE-SENSITIVE, the
search commands perform case-sensitive searches; if the keyword is
:CASE-INSENSITIVE, the search commands perform case-insensitive
searches. Initially, the value of this variable is :CASE-INSENSITIVE.

Display Name Format

        Default Search Case

Symbol Format

        DEFAULT SEARCH CASE

DEFAULT WINDOW LABEL Editor Variable

Is bound to a string, a function, or NIL that is used by MAKE-WINDOW
as the default window label. If it is a function, it must have one
argument (a window). If the value is the null string (""), the window
is bordered but has no label. If the value is NIL, the window is
unbordered.

Display Name Format

        Default Window Label

Symbol Format

DEFAULT-WINDOW-LABEL

DEFAULT WINDOW LABEL EDGE Editor Variable

Is bound to keyword that specifies which edge of a window the label text will be displayed on. The variable is globally bound to :BOTTOM.

Display Name Format

Default Window Label Edge

Symbol Format

DEFAULT-WINDOW-LABEL-EDGE

DEFAULT WINDOW LABEL OFFSET Editor Variable

Is bound to a positive integer or NIL. The value specifies the default offset value to be used for the label position of a newly created window. The global binding is NIL, which causes labels to be centered.

Display Name Format

Default Window Label Offset

Symbol Format

DEFAULT-WINDOW-LABEL-OFFSET

DEFAULT WINDOW LABEL RENDITION Editor Variable

Is bound to a keyword or a list of keywords that specify the default video rendition to be applied to the label of a newly created window. The keyword can be any of :NORMAL, :REVERSE, :UNDERLINE, :BOLD, or :BLINK. The global binding is :REVERSE.

Display Name Format

Default Window Label Rendition

Symbol Format

DEFAULT-WINDOW-LABEL-RENDITION

DEFAULT WINDOW LINES WRAP Editor Variable

Is used to determine whether lines in a newly created window should wrap or truncate. A value of NIL indicates that lines should truncate; otherwise, lines wrap. The global binding is NIL.

Display Name Format

Default Window Lines Wrap

Symbol Format

DEFAULT-WINDOW-LINES-WRAP

DEFAULT WINDOW RENDITION Editor Variable

Is bound to a keyword or a list of keywords that specify the default video rendition to be applied to a newly created window. The keyword can be any of :NORMAL, :REVERSE, :UNDERLINE, :BOLD, or :BLINK. The global binding is :NORMAL.

Display Name Format

Default Window Rendition

Symbol Format

DEFAULT-WINDOW-RENDITION

DEFAULT WINDOW TRUNCATE CHAR Editor Variable

Is bound to a character that is used to indicate the truncation of a displayed line. This variable is globally bound to the #\> character.

Display Name Format

Default Window Truncate Char

Symbol Format

DEFAULT-WINDOW-TRUNCATE-CHAR

DEFAULT WINDOW TYPE Editor Variable

Is bound to a keyword that specifies the default type of a created window. Possible values are :ANCHORED or :FLOATING. The global binding is :ANCHORED.

Display Name Format

Default Window Type

Symbol Format

DEFAULT-WINDOW-TYPE

DEFAULT WINDOW WIDTH Editor Variable

Is bound to a value that is the default width of a newly created window. The global value of this variable is set to be the width of the screen. If the screen width is altered, the value of this variable is changed by the global Screen Modification Hook function.

Display Name Format

Default Window Width

Symbol Format

DEFAULT-WINDOW-WIDTH

DEFAULT WINDOW WRAP CHAR Editor Variable

Is bound to the default character that is used to indicate wrapping of text in a window. The variable is globally bound to #\<.

Display Name Format

Default Window Wrap Char

Symbol Format

DEFAULT-WINDOW-WRAP-CHAR

DEFINE-ATTRIBUTE Macro

Creates a new attribute having the specified name and documentation string. Note that the type of the attribute is not defined until it is bound.

Format

DEFINE-ATTRIBUTE *name* &OPTIONAL *documentation*

Arguments

name

The name of the attribute. This can be specified as either a symbol or a list of the form (symbol :DISPLAY-NAME string), where "string" is used as an alternate reference to this attribute.

documentation

A string that is the documentation text for this attribute

Return Value

The symbol of the attribute

DEFINE-COMMAND Macro

Creates a new Editor command by making a new LISP function from the specified arglist and forms.

As a rule, commands have names of the form NAME-OF-COMMAND-COMMAND and display names of the form "Name of Command." The command can be executed only in the Editor, either through a key binding or as the argument of the command "Execute Named Command" (bound to CTRL/Z globally or PF1 7 in EDT Emulation style). The created function can be called from any LISP code.

Format

DEFINE-COMMAND *name*
     arglist
     &OPTIONAL *command-documentation*
     &BODY *forms*

Arguments

name

> The symbol that will name the command. This can be specified as
> either a symbol or a list of the form
>
> > *(symbol {keyword-value-pair})*
>
> The acceptable keywords are:
>
> :DISPLAY-NAME *string*
>
> > The display name for the command, which will be entered in
> > the *EDITOR-COMMAND-NAMES* string table.
>
> :CATEGORY categories
>
> > The categories must be a symbol or a list of symbols that
> > are user-defined categories the command is cataloged under.
> > The list can be referenced using the COMMAND-CATEGORIES
> > command. A list of all commands belonging to a specific
> > category can be obtained with the CATEGORY-COMMANDS
> > function.

arglist

> The list of formal parameters of the command. This is identical
> to the argument list in DEFUN.   There must be at least one
> argument, however.

command-documentation

> An optional documentation string associated with the command.
> This string is associated with the command name and has a
> documentation type of EDITOR-COMMAND.

forms

> A list of forms that make up the body of the function executed
> when the command is invoked. These forms are identical to the
> forms given to DEFUN and can include a function documentation
> string and declarations.

Return Value

> The function associated with the command

DEFINE-EDITOR-VARIABLE Macro

Defines an Editor variable. The symbol is interned in the current package and proclaimed to be a special variable. This definition must appear prior to any bindings or other uses of the variable.

Format

```
DEFINE-EDITOR-VARIABLE name
                       &OPTIONAL documentation
                       &KEY :BIND-HOOK :UNBIND-HOOK
```

Arguments

name

> The name may be specified as either a symbol or a list of the form (symbol :DISPLAY-NAME string), where string is a user-defined name for the variable. The print name of the symbol and the display name (if supplied) are entered as a key into the *EDITOR-VARIABLE-NAMES* string table with the symbol placed into the data slot of the table.

documentation

> A string that is included in the documentation of the symbol with a documentation type of EDITOR-VARIABLE.

:BIND-HOOK

> A function that is invoked whenever the variable is bound in a context. The function is called with two arguments -- the symbol and the context in which the variable is being bound.

:UNBIND-HOOK

> A function that is called when the binding of the variable is removed in the context. The function is called with two arguments -- the variable and the context. It defaults to NIL.

Return Value

> The symbol of the Editor variable

DEFINE-KEYBOARD-MACRO Function

Causes the Editor to start remembering keystrokes as they are typed at the terminal until a call is made to the END-KEYBOARD-MACRO function. If an optional string is supplied, a keyboard macro is created and returned as if that string were a sequence of characters that had been

entered and remembered previously. The Editor does not remember
entered keystrokes if a string argument is supplied.

Format

    DEFINE-KEYBOARD-MACRO &OPTIONAL *string*

Arguments

string

> An optional string that will be used in place of a sequence of
> keystrokes

Return Value

> A function, when called, will execute the keyboard macro if a
> string argument is supplied, otherwise T

DELETE-AND-SAVE-REGION Function

Deletes the region and returns a copy of the region containing the
deleted text.

Format

    DELETE-AND-SAVE-REGION *region*

Arguments

region

> An Editor region

Return Value

> A copy of the region that was deleted

DELETE-BUFFER Function

Deletes the specified buffer. The calling of this function causes the
"Buffer Deletion Hook" to be invoked. If you delete the current
buffer and do not specify a value for new-current, the current buffer
is set by the same rules used in the NEXT-WINDOW function, provided
other buffers are displayed.

If none are displayed, the Editor makes an arbitrary choice. If there
are no other user-created buffers, the Editor returns to an initial

state as if it had been invoked originally by the typing of (ED) with no arguments.

Format

>DELETE-BUFFER *buffer* &OPTIONAL *new-current*

Arguments

buffer

>An Editor buffer

new-current

>An Editor buffer that becomes the new current buffer

Return Value

>The symbol naming the Editor buffer

DELETE-CHARACTERS Function

Deletes a specified number of characters after the specified mark (or before it, if the number is negative). If there are not enough characters after (or before) the mark, the buffer is not modified.

Format

>DELETE-CHARACTERS *mark* &OPTIONAL *n*

Arguments

mark

>An Editor mark

n

>A fixnum, which defaults to 1, specifying the number of characters to delete.

Return Value

>The number of characters deleted, or NIL if there were not *n* characters to delete.

DELETE CURRENT BUFFER Command

Deletes the current buffer. If the buffer is modified, the user is
asked whether to save the contents of the buffer. If another buffer
is visible on the screen, that buffer becomes the new current buffer.
If not, the Editor makes an arbitrary choice of another buffer to be
the new current buffer.

Display Name Format

     Delete Current Buffer

Function Format

     DELETE-CURRENT-BUFFER-COMMAND *prefix buffer new-current*

Arguments

prefix

     Ignored

buffer

     The buffer to delete. Default is the current buffer.

new-current

     The buffer that becomes the new current buffer. The default is
     as specified above.

Return Value

     T

DELETE LINE Command

Deletes lines or parts of lines, depending on the prefix argument and
the location of the current buffer point:

> • If the prefix is NIL, the command deletes between the current
> buffer point and the end of the line. If there are
> non-whitespace characters before the end of the line, the
> command deletes those characters and does not delete the
> newline character. If there are no characters or only
> whitespace characters before the end of the line, the command
> deletes to the end of the line, including the newline
> character.

● If the prefix is an integer, the command deletes the characters between the beginning of the line indicated by the prefix and the current buffer point. A prefix of 0 indicates the current line, 1 indicates the next line, -1 indicates the previous line, and so on.

Display Name Format

Delete Line

Function Format

DELETE-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

A disembodied region containing the deleted text

DELETE-MARK Function

Deletes the specified Editor mark. You use this function primarily to remove permanent marks when they are no longer needed. If the mark being deleted is the buffer point of a buffer, the window point of a window, the display beginning or end of a window, or a mark defining a buffer region, the results are unpredictable.

Format

DELETE-MARK *mark*

Arguments

mark

An Editor mark

Return Value

NIL

DELETE NAMED BUFFER Command

Deletes the specified Editor buffer and any windows associated with it. The appropriate hook functions are invoked. If the name is NIL, the user is prompted for a name.

Category

:GENERAL-PROMPTING

Display Name Format

Delete Named Buffer

Function Format

DELETE-NAMED-BUFFER-COMMAND *prefix* &OPTIONAL *name*

Arguments

prefix

Ignored

name

The name of the buffer to delete. It defaults to NIL.

Return Value

T

DELETE NEXT CHARACTER Command

Causes the character following the point in the current window to be deleted. If you specify an integer prefix argument, characters following the point are deleted in the amount indicated.

Display Name Format

Delete Next Character

Function Format

DELETE-NEXT-CHARACTER-COMMAND *prefix*

Arguments

prefix

> A positive integer or NIL

Return Value

> The number of characters deleted

DELETE NEXT WORD Command

Deletes the next word. If you supply an integer prefix argument, the command deletes as many words as you specify.

Display Name Format

> Delete Next Word

Function Format

> DELETE-NEXT-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> A positive integer or NIL

mark

> An Editor mark that defaults to the current buffer point

Return Value

> The region containing the word(s) deleted

DELETE PREVIOUS CHARACTER Command

Deletes the character preceding the point in the current window if the prefix argument is NIL. If you specify an integer prefix argument, characters preceding the point (or following it, if the prefix is negative) are deleted in the amount indicated.

Display Name Format

> Delete Previous Character

Function Format

DELETE-PREVIOUS-CHARACTER-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

Undefined

DELETE PREVIOUS WORD Command

Deletes the previous word. If you supply an integer prefix, the command deletes as many words as you specify.

Display Name Format

Delete Previous Word

Function Format

DELETE-PREVIOUS-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

A positive integer or NIL

mark

An Editor mark that defaults to the current buffer point

Return Value

The region containing the word(s) deleted

DELETE-REGION Function

Deletes the text in the specified region; the empty region remains.

Format

DELETE-REGION *region*

Arguments

region

> An Editor region

Return Value

> NIL

DELETE WHITESPACE Command

Deletes the whitespace characters following the current buffer point.

Display Name Format

> Delete Whitespace

Function Format

> DELETE-WHITESPACE-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> Ignored

mark

> An Editor mark that defaults to the current buffer point

Return Value

> NIL

DELETE-WINDOW Function

Deletes a window from the Editor. If the window is displayed, it is removed from the display and then deleted. If the window is the current window, a new current window is selected from any other currently displayed windows. If there are none, a new window is displayed from other available buffers. The functions in "Window Deletion Hook" are called prior to any alterations of the window.

Format

> DELETE-WINDOW *window*

Arguments

window

    An Editor window

Return Value

    T

DELETE WORD Command

Deletes the characters from the current point to the beginning of the next word. If you specify an integer prefix argument, n, characters from the current point to the end of the nth word are deleted.

Display Name Format

    Delete Word

Function Format

    DELETE-WORD-COMMAND *prefix*

Arguments

prefix

    A positive integer or NIL

Return Value

    Undefined

DESCRIBE Command

Displays in the Help buffer the documentation string of the specified object. If the type or name is NIL, the command prompts the user for it in the Editor prompting window.

Category

    :GENERAL-PROMPTING

Display Name Format

    Describe

Function Format

> DESCRIBE-OBJECT-COMMAND *prefix* &OPTIONAL *type name*

Arguments

prefix

> Ignored

type

> A named object type specifier that defaults to NIL -- ATTRIBUTE, COMMAND, BUFFER, STYLE, VARIABLE, KEYBOARD-MACRO, KEY-BINDING, or SYMBOL

name

> The symbol or display name of a named Editor object of the appropriate type

Return Value

> None

DESCRIBE-OBJECT-COMMAND Function

See "Describe" command.

DESCRIBE WORD Command

Does a LISP DESCRIBE operation on the word at the argument mark and displays the result in the "Help" buffer. The mark defaults to the current buffer point.

Display Name Format

> Describe Word

Function Format

> DESCRIBE-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> Ignored

mark

An Editor mark that defaults to the current buffer point

Return Value

Undefined

DESCRIBE WORD AT POINTER Command

Does a LISP DESCRIBE operation on the symbol indicated by the pointer. If the pointer indicates a character that is a list terminator, this command momentarily highlights the matching list-initiator character. If the list initiator is not visible in the window, the line containing it is displayed in the information area, and the matching list initiator is highlighted. (See "LISP Syntax" attribute, especially the values :CONSTITUENT, :LIST-TERMINATOR, and :LIST-INITIATOR.)

Display Name Format

Describe Word at Pointer

Function Format

DESCRIBE-WORD-AT-POINTER-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

Undefined

DOWNCASE REGION Command

Makes all alphabetic characters in the current buffer select region lower case.

Display Name Format

Downcase Region

Function Format

>   DOWNCASE-REGION-COMMAND *prefix* &OPTIONAL *region*

Arguments

prefix

>   Ignored

region

>   An Editor region that defaults to the buffer select region

Return Value

>   The region

DOWNCASE WORD Command

Makes all alphabetic characters in the current word lower case.

Display Name Format

>   Downcase Word

Function Format

>   DOWNCASE-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

>   Ignored

mark

>   An Editor mark that defaults to the current buffer point

Return Value

>   The region containing the word that was made all lower case, or NIL

ED Command

Performs the same actions as the ED function, but you can use it from within the Editor. If you do not specify an object or a type, the command prompts you for it.

Category

> :GENERAL-PROMPTING

Display Name Format

> Ed

Function Format

> EDIT-LISP-OBJECT-COMMAND *prefix* &OPTIONAL *object type*

Arguments

prefix

> Ignored

object

> Any object that is valid for the ED function

type

> Any keyword that is a valid value for the :TYPE keyword to the ED function -- that is, :FUNCTION or :VALUE. This is valid only if the object argument is a symbol.

Return Value

> Undefined

ED Function

Invokes the VAX LISP Editor. This function takes an optional x argument that specifies what is to be edited. The value can be a namestring, pathname, or symbol. In the VAX LISP implementation, the value can also be a list.

This function can be called recursively. This is helpful when implementing commands such as "Query Search Replace".

EDITOR OBJECT DESCRIPTIONS

Format

    ED &OPTIONAL x &KEY :TYPE :READ-ONLY

Arguments

x

> A namestring, pathname, symbol, or list that specifies what is to be edited.
>
> A namestring or pathname specifies a file. A symbol specifies a LISP symbol. If you supply a symbol argument, then you can also specify a :TYPE keyword argument, :FUNCTION or :VALUE, which tells the Editor whether you want to edit the symbol's function/macro definition or its value.
>
> If you specify a list, the list must be a generalized variable that can be specified in a call to the SETF macro. The list is evaluated, and it returns a value to edit. When you write the buffer containing the value, the Editor replaces the value of the generalized variable with the new value.

:TYPE *keyword*

> You can specify this argument only if the x argument is a symbol. The possible values are:
>
> :FUNCTION (the default)
>
>> The Editor is invoked to edit the function or macro definition associated with the specified symbol.
>
> :VALUE
>
>> The Editor is invoked to edit the specified symbol's value.

:READ-ONLY *value*

> You can specify this argument to indicate whether modifications to the buffer can be written back as a new version of the file being edited or an update of the LISP object being edited. (See description of BUFFER-WRITABLE.) The possible values are:
>
> NIL (the default)
>
>> Modifications can be written.
>
> Non-NIL
>
>> Modifications cannot be written.

Return Value

> None (by default). See RETURN-FROM-EDITOR for returning values
> from a call to ED.

EDIT FILE Command

Prompts for a file name if the user does not supply one. If there is
a buffer containing that file, the command switches you to it.
Otherwise, a new buffer is created whose name is the name of the file,
the named file is read into the buffer, and the command switches you
to that buffer. If there is a buffer with the same name as the file,
but that buffer's file is a different file (for example, same name but
in a different directory), the user is prompted for a new buffer name.
If the user does not supply a new buffer name, the old buffer is
reused.

Category

> :GENERAL-PROMPTING

Display Name Format

> Edit File

Function Format

> EDIT-FILE-COMMAND *prefix* &OPTIONAL *file*

Arguments

prefix

> Ignored

file

> A pathname, namestring, stream, or NIL.

Return Value

> The buffer

EDIT-LISP-OBJECT-COMMAND Function

See "Ed" command.

EDITOR-ATTRIBUTE-NAMES Variable

Specifies a string table that contains the names of all the defined
Editor attributes.

EDITOR-BUFFER-NAMES Variable

Specifies a string table that contains the names of all the existing
Editor buffers.

EDITOR-COMMAND-NAMES Variable

Specifies a string table that contains the names of commands currently
defined in the Editor.

EDITOR-DEFAULT-BUFFER Variable

Can be set to a buffer specifier. When the Editor has no other
windows to display, it will display a window into the specified
buffer. If the value of this variable is NIL (the default) or if the
specified buffer does not exist, then the Editor will display a window
into the buffer "Basic Introduction" when it has nothing else to show.

This variable is useful, for example, to set up a LISP "scratch pad"
buffer to be used when you are not editing files.

EDITOR ENTRY HOOK Editor Variable

Specifies a hook function that is to be called when the user invokes
the Editor. It is called after the Editor is initialized and just
before the processing of the arguments to the ED function. The
context searching order is the one that was in effect when an Editor
pause occurred, or :GLOBAL if the Editor is being called for the first
time.

Display Name Format

    Editor Entry Hook

Symbol Format

> EDITOR-ENTRY-HOOK

EDITOR-ERROR Function

Is the error-signaling mechanism for the Editor. This function creates an error message by applying FORMAT to the format string and the remaining arguments. It writes this message in the Information Area, calls the ATTENTION function, and returns to the Editor command level.

Format

> EDITOR-ERROR &OPTIONAL *format-string* &REST *args*

Arguments

format-string

> The control string for the FORMAT function

args

> Arguments to be passed to FORMAT

Return Value

> None

EDITOR-ERROR-WITH-HELP Function

Is similar to the EDITOR-ERROR function but can provide the user with additional information on the error that has occurred. This function applies FORMAT to the error string and the remaining arguments to obtain an error message that is displayed in the Information Area. It calls the ATTENTION function and returns to the Editor command loop. In addition, it applies FORMAT to the information string and the remaining arguments to obtain another string that is saved for later display to the user.

The initial error message should be brief and give the experienced user enough information to understand the problem. The information message can be tailored to the less experienced user or can be used to provide more extensive supplemental information on the nature of the error.

Format

EDITOR-ERROR-WITH-HELP *information-string error-string* &REST *args*

Arguments

information-string

A format string to be applied to the args to produce additional information

error-string

A format string to be applied to the args to produce a brief error message

args

Any arguments needed to supply error information

Return Value

None

EDITOR EXIT HOOK Editor Variable

Specifies a hook function that is invoked when you exit from the Editor. The function is called immediately after the execution of an "Exit" command. The context searching order is what was in effect at the start of the "Exit" command.

Display Name Format

Editor Exit Hook

Symbol Format

EDITOR-EXIT-HOOK

EDITOR-HELP-BUFFER Buffer

See "Help" buffer.

EDITOR INITIALIZATION HOOK Editor Variable

Can be set (using SETF with VARIABLE-FUNCTION) to a hook function of no arguments that is called whenever the Editor is initialized. The Editor is initialized the first time you call the ED function in a LISP session, and anytime you call ED after having exited the Editor.

Note that when this hook function is called, the Editor state is not fully established. In particular, CURRENT-BUFFER, CURRENT-BUFFER-POINT, and CURRENT-WINDOW return no values.

Also, no Editor style is active when this function is called. Therefore, only the global binding of the variable is meaningful.

This variable has no initial function definition.

Display Name Format

> Editor Initialization Hook

Symbol Format

> EDITOR-INITIALIZATION-HOOK

EDITOR-KEYBOARD-MACRO-NAMES Variable

Is a string table that contains the names of all named keyboard macros.

EDITOR-LISTEN Function

Returns T if there is another character immediately available from the terminal while you are using the Editor. This function returns NIL otherwise.

Format

> EDITOR-LISTEN

Arguments

> None

Return Value

> T or NIL

EDITOR PAUSE HOOK Editor Variable

Specifies a hook function that is invoked when you pause the Editor. The function is called immediately after the execution of a "Pause" command. The context searching order is what was in effect at the start of the "Pause" command.

Display Name Format

Editor Pause Hook

Symbol Format

EDITOR-PAUSE-HOOK

EDITOR-PROMPTING-BUFFER Buffer

See "General Prompting" buffer.

EDITOR-READ-CHAR Function

Returns the next character read from the terminal. You can call this function only when the Editor is active.

Format

EDITOR-READ-CHAR

Arguments

None

Return Value

A character

EDITOR-READ-CHAR-NOHANG Function

Returns the next character typed at the terminal if a character is immediately available. The function returns NIL otherwise.

Format

EDITOR-READ-CHAR-NOHANG

Arguments

> None

Return Value

> A character; or NIL, if none is available

EDITOR-RETAIN-SCREEN-STATE Variable

Specifies whether or not the state of the screen is to be retained when the you cause the Editor to pause. The default value is NIL, which means that the screen is erased when the Editor pauses and is restored to its previous state when the Editor is reentered.

When you are debugging new commands, however, it may be desirable for you to alter this behavior. When the variable is set to T, then the screen is not erased when the Editor pauses.

EDITOR-STYLE-NAMES Variable

Specifies a string table that contains the names of all the defined Editor styles.

EDITOR-UNREAD-CHAR Function

Unreads the last character read from the terminal while in the Editor. See *COMMON LISP: The Language* for more information about unreading.

Format

> EDITOR-UNREAD-CHAR *character*

Arguments

character

> The last character read from the terminal

Return Value

> NIL

EDITOR-VARIABLE-NAMES Variable

Specifies a string table that contains the names of all the defined Editor variables.

EDT APPEND Command

Deletes the current select region of text (the region defined from the current buffer point and the mark in the "Buffer Select Mark" variable) and stores the deleted region in the Editor variable "EDT Paste Buffer". The region is inserted at the end of the current contents, if any, of the paste buffer.

A select region is a region established by executing the command "EDT Select" (or "Set Select Region"). "EDT Append" can add to text that was previously deleted and stored by execution of either "EDT Cut" or "EDT Append".

Display Name Format

    EDT Append

Function Format

    EDT-APPEND-COMMAND *prefix*

Arguments

prefix

    Ignored

Return Value

    Undefined

EDT BACK TO START OF LINE Command

Moves the curent buffer point to the beginning of the current line. If the point is already at the beginning of a line, it is moved to the beginning of the previous line. If a positive integer prefix is supplied, the point is moved backward the number of lines indicated.

Display Name Format

    EDT Back to Start of Line

Function Format

    EDT-BACK-TO-START-OF-LINE-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The modified point

EDT BEGINNING OF LINE Command

Moves the point of the current buffer to the beginning of the next line if the prefix argument is NIL, and if "EDT Direction Mode" is :FORWARD. If "EDT Direction Mode" is :BACKWARD, the point is moved backward to the nearest beginning of a line.

If you specify a positive integer prefix argument, n, the point is moved to the nth beginning of a line in the direction indicated by "EDT Direction Mode." If you specify a negative integer prefix argument, n, the point is moved to the nth beginning of a line in the direction opposite to that indicated by EDT-Direction-Mode.

Display Name Format

    EDT Beginning of Line

Function Format

    EDT-BEGINNING-OF-LINE-COMMAND *prefix*

Arguments

prefix

    A fixnum or NIL

Return Value

    The modified point

EDT CHANGE CASE Command

Changes the case of any characters in the region specified by the "Buffer Select Region" Editor variable or the character at the current buffer point if no region is specified.

Display Name Format

> EDT Change Case

Function Format

> EDT-CHANGE-CASE-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> Undefined

EDT CUT Command

Deletes the current select region of text (the region defined from the current buffer point and the mark in the "Buffer Select Mark" variable) and stores the deleted region in the Editor variable "EDT Paste Buffer". The previous contents of the paste buffer are lost.

A select region is a region established by executing the command "EDT Select" (or "Set Select Mark").

Display Name Format

> EDT Cut

Function Format

> EDT-CUT-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

Undefined

EDT DELETE CHARACTER Command

Deletes the character at the position of the cursor (that is, at the position immediately following the point in the current buffer) if the prefix argument is NIL. If you specify a positive integer prefix argument, characters following the point are deleted in the amount indicated. The Editor variable "EDT Deleted Character" is set to the last character deleted.

Display Name Format

EDT Delete Character

Function Format

EDT-DELETE-CHARACTER-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The last character deleted

EDT DELETE LINE Command

Deletes the line of text starting at the current buffer point and extending to the beginning of the next line. If you specify an integer prefix argument, lines are deleted in the amount indicated (in a forward direction if integer is positive, in a backward direction if integer is negative). The Editor variable "EDT Deleted Line" is set to the last line deleted.

Display Name Format

EDT Delete Line

Function Format

EDT-DELETE-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

A region containing the last line deleted

EDT DELETE PREVIOUS CHARACTER Command

Deletes the character preceding the current buffer point. If a prefix argument is supplied, the command deletes the number of previous characters specified by the argument. The value of the "EDT Deleted Character" Editor variable is set to the last character deleted.

Display Name Format

EDT Delete Previous Character

Function Format

EDT-DELETE-PREVIOUS-CHARACTER-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The last character deleted

EDT DELETE PREVIOUS LINE Command

Deletes from the current buffer point back to the beginning of the current line. If the point was already at the beginning of a line, the command deletes back to the beginning of the previous line. If a prefix argument, *n*, is supplied, *n* lines are deleted. The value of the "EDT Deleted Line" Editor variable is set to a region containing the last line deleted.

Display Name Format

EDT Delete Previous Line

Function Format

> EDT-DELETE-PREVIOUS-LINE-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> A region containing the last deleted line

EDT DELETE PREVIOUS WORD Command

Deletes from the current buffer point back to the beginning of the current word. If the point was already at the beginning of a word, the command deletes back to the beginning of the previous word. If a prefix argument, n, is supplied, n words are deleted. The value of the "EDT Deleted Word" Editor variable is set to a region containing the last word deleted.

Display Name Format

> EDT Delete Previous Word

Function Format

> EDT-DELETE-PREVIOUS-WORD-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> A region containing the last deleted word

EDT DELETE TO END OF LINE Command

Deletes all characters found between the point of the current buffer and the end of the current line. If you specify a positive integer prefix argument, n, all characters found between the buffer point and the nth end-of-line following the point are deleted. If you specify a negative integer prefix argument, n, the sign is ignored and the

absolute value of the argument is used. The value of the Editor variable "EDT Deleted Line" is set to the last line deleted.

Display Name Format

EDT Delete to End of Line

Function Format

EDT-DELETE-TO-END-OF-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The line that was deleted

EDT DELETE WORD Command

Deletes all characters between the point of the current buffer and the next end of a word. If you specify a positive integer prefix argument, n, the characters between the current buffer point and the end of the nth following word are deleted. If you specify a negative integer prefix argument, the sign is ignored and the absolute value is used. The Editor variable "EDT Deleted Word" is set to the last word deleted.

Display Name Format

EDT Delete Word

Function Format

EDT-DELETE-WORD-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The deleted word

EDT DELETED CHARACTER Editor Variable

Is bound to the last character deleted by the "EDT Delete Character" command.

Display Name Format

EDT Deleted Character

Symbol Format

EDT-DELETED-CHARACTER

EDT DELETED LINE Editor Variable

Is bound to the region containing the last line deleted by the "EDT Delete Line" command or by the "EDT Delete to End of Line" command.

Display Name Format

EDT Deleted Line

Symbol Format

EDT-DELETED-LINE

EDT DELETED WORD Editor Variable

Is bound to the region containing the last word deleted by the "EDT Delete Word" command.

Display Name Format

EDT Deleted Word

Symbol Format

EDT-DELETED-WORD

EDT DESELECT Command

See "Unset Select Mark" command.

EDT DIRECTION MODE Editor Variable

Specifies the direction in which certain commands in "EDT Emulation" style are to operate. The possible values are :FORWARD and :BACKWARD.

Display Name Format

    EDT Direction Mode

Symbol Format

    EDT-DIRECTION-MODE

EDT EMULATION Style

Is the default major style for the VAX LISP Editor. This style is designed to imitate the basic keypad behavior of the VMS EDT Editor.

Display Name Format

    EDT Emulation

Symbol Format

    EDT-EMULATION

EDT END OF LINE Command

Moves the current buffer point to the next end-of-line if "EDT Direction Mode" is :FORWARD or to the previous end-of-line if "EDT Direction Mode" is :BACKWARD. If you specify an integer prefix argument, n, the point is moved to the nth end of a line in the direction indicated by "EDT Direction Mode"; if the integer is negative, the direction is opposite to that indicated by "EDT Direction Mode."

Display Name Format

    EDT End of Line

Function Format

    EDT-END-OF-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The modified point

EDT MOVE CHARACTER Command

Moves the point of the current buffer by one character if prefix is NIL. If the value of "EDT Direction Mode" is :FORWARD, the point is moved forward; if :BACKWARD, it is moved backward. If you specify an integer prefix argument, the point is moved in the direction of "EDT Direction Mode" by the number of characters indicated. If you specify a positive integer prefix argument, *n*, the point is moved *n* characters in the direction indicated by "EDT Direction Mode." If you specify a negative integer prefix argument, *n*, the point is moved *n* characters in the direction opposite to that indicated by "EDT Direction Mode."

Display Name Format

EDT Move Character

Function Format

EDT-MOVE-CHARACTER-COMMAND *prefix*

Arguments

prefix

A fixnum or NIL

Return Value

The modified point

EDT MOVE PAGE Command

Moves the point one page in the direction of "EDT Direction Mode" if the prefix argument is NIL. If you specify a positive integer prefix argument, the point is moved in the direction of "EDT Direction Mode" by the number of pages indicated; if the integer is negative, the direction is opposite to that of "EDT Direction Mode."

Display Name Format

    EDT Move Page

Function Format

    EDT-MOVE-PAGE-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The modified point

EDT MOVE WORD Command

Moves the current buffer point to the next or previous beginning of a word in the direction indicated by "EDT Direction Mode" if the prefix argument is NIL. If you specify a positive integer prefix argument, the point is moved in the direction of "EDT Direction Mode" by the number of words indicated; if the integer is negative, the direction is opposite to that of "EDT Direction Mode."

Display Name Format

    EDT Move Word

Function Format

    EDT-MOVE-WORD-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The current buffer point

EDT PASTE Command

Inserts the contents of the region bound to the Editor variable "EDT Paste Buffer" at the current buffer point.

Display Name Format

EDT Paste

Function Format

EDT-PASTE-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The inserted region

EDT PASTE AT POINTER Command

Moves the current buffer point to the position indicated by the pointer and then inserts at that location the region contained in the paste buffer. If the pointer is beyond the end of a line, the region is inserted at the end of that line. If the pointer is beyond the end of the buffer region, the paste region is inserted at the end of the buffer region.

Display Name Format

EDT Paste at Pointer

Function Format

EDT-PASTE-AT-POINTER-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The inserted region

EDT PASTE BUFFER Editor Variable

Is bound in the "EDT Emulation" style. The value of this variable is the region most recently deleted by the "EDT Cut" command.

Display Name Format

EDT Paste Buffer

Symbol Format

EDT-PASTE-BUFFER

EDT QUERY SEARCH Command

Prompts for a string to use as a pattern in a search. The search is forward if "EDT Direction Mode" is :FORWARD; backward, if "EDT Direction Mode" is :BACKWARD. The point is moved to the end of the first matching string if the search is forward, or to the beginning of the first matching string if the search is backward. If the prefix argument is an integer, n, the command searches for the nth occurrence of the string. If n is negative, the command searches in the direction opposite to the setting of "EDT Direction Mode."

Display Name Format

EDT Query Search

Function Format

EDT-QUERY-SEARCH-COMMAND *prefix* &OPTIONAL *string*

Arguments

prefix

An integer or NIL

string

The string to search for

Return Value

The modified point

EDT REPLACE Command

Deletes the current select region of text (the region defined from the current buffer point and the mark in the "Buffer Select Mark" variable) and replaces it with the region stored in the Editor variable "EDT Paste Buffer".

A select region is a region established by executing the command "EDT Select" (or "Set Select Region"). The replacement text is text placed in the paste buffer by means of either "EDT Cut" or "EDT Append".

Display Name Format

EDT Replace

Function Format

EDT-REPLACE-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

Undefined

EDT SCROLL WINDOW Command

Scrolls the current window in the direction indicated by "EDT Direction Mode" by a distance that is two-thirds the height of the window, if the prefix argument is NIL. If prefix is positive, the window is scrolled in the direction of "EDT Direction Mode" by a distance of prefix times half the height of the window; if prefix is negative, the window is scrolled in the direction opposite to the setting of "EDT Direction Mode" by a distance of prefix times half the height of the window.

Display Name Format

EDT Scroll Window

Function Format

EDT-SCROLL-WINDOW-COMMAND *prefix*

Arguments

prefix

> A fixnum or NIL

Return Value

> The new point

EDT SEARCH AGAIN Command

Searches for text that matches the search string saved in the "Last Search String" Editor variable. If "EDT Direction Mode" is :FORWARD, the direction of the search is forward; if "EDT Direction Mode" is :BACKWARD, the direction of the search is backward.

Display Name Format

> EDT Search Again

Function Format

> EDT-SEARCH-AGAIN-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> The modified point

EDT SELECT Command

See "Set Select Mark" command.

EDT SET DIRECTION BACKWARD Command

Sets the value of the Editor variable "EDT Direction Mode" to :BACKWARD. The prefix argument is ignored.

Display Name Format

> EDT Set Direction Backward

Function Format

> EDT-SET-DIRECTION-BACKWARD-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> :BACKWARD

EDT SET DIRECTION FORWARD Command

Sets the value of the Editor variable "EDT Direction Mode" to :FORWARD. The prefix argument is ignored.

Display Name Format

> EDT Set Direction Forward

Function Format

> EDT-SET-DIRECTION-FORWARD-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> :FORWARD

EDT SPECIAL INSERT Command

Inserts the character at the current buffer point whose character code is specified by the prefix argument. For example, to insert a DELETE character, you specify a prefix argument of 127. The character is inserted with no special interpretation by the Editor.

Display Name Format

EDT Special Insert

Function Format

EDT-SPECIAL-INSERT-COMMAND *prefix*

Arguments

prefix

The prefix argument is interpreted as the character code of a character to be inserted.

Return Value

The character

EDT SUBSTITUTE Command

Causes the text in the "EDT Pasté Buffer" Editor variable to replace a string just located in the text by the "EDT Query Search" or "EDT Search Again" command. After the text is replaced, the command searches for the next occurrence of the search string. If a prefix argument is supplied, the command is executed the number of times indicated.

Display Name Format

EDT Substitute

Function Format

EDT-SUBSTITUTE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

Undefined

EDT UNDELETE CHARACTER Command

Restores the character last deleted by the "EDT Delete Character" command (that is, the value of the Editor variable "EDT Deleted Character") to the position of the current buffer point if prefix is NIL. If you specify an integer prefix argument, the character is inserted the number of times indicated.

Display Name Format

EDT Undelete Character

Function Format

EDT-UNDELETE-CHARACTER-COMMAND *prefix*

Arguments

prefix

A positive integer or NIL

Return Value

The inserted character

EDT UNDELETE LINE Command

Restores the line last deleted by the "EDT Delete Line" or "EDT Delete to End of Line" command (that is, the value of the Editor variable "EDT Deleted Line") to the position of the current buffer point if prefix is NIL. If you specify an integer prefix argument, the line is inserted the number of times indicated.

Display Name Format

EDT Undelete Line

Function Format

EDT-UNDELETE-LINE-COMMAND *prefix*

Arguments

prefix

A positive integer or NIL

Return Value

The inserted region

EDT UNDELETE WORD Command

Restores the word last deleted by the "EDT Delete Word" command (that is, the value of the Editor variable "EDT Deleted Word") to the position of the current buffer point if prefix is NIL. If you specify an integer prefix argument, the word is inserted the number of times indicated.

Display Name Format

EDT Undelete Word

Function Format

EDT-UNDELETE-WORD-COMMAND *prefix*

Arguments

prefix

A positive integer or NIL

Return Value

The inserted region

EMACS Style

Is an Editor style designed to imitate the functions and key bindings of an EMACS-based editor.

Display Name Format

EMACS

Symbol Format

EMACS

EMACS BACKWARD SEARCH Command

Searches backward for the search string specified in the last command.
If the last command was not a searching command, the "EMACS Backward
Search" command prompts for a search string. If no prefix is
supplied, this command searches for the first occurrence of the search
string. For a prefix *n*, the command searches for the nth occurrence
of the string.

Category

> : EMACS-SEARCH

Display Name Format

> EMACS Backward Search

Function Format

> EMACS-BACKWARD-SEARCH-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> The updated current buffer point

EMACS FORWARD SEARCH Command

Searches forward for the search string specified in the last command.
If the last command was not a searching command, the "EMACS Forward
Search" command prompts for a search string. If no prefix is
supplied, this command searches for the first occurrence of the search
string. For a prefix *n*, the command searches for the nth occurrence
of the string.

Category

> : EMACS-SEARCH

Display Name Format

> EMACS Forward Search

Function Format

> EMACS-FORWARD-SEARCH-COMMAND *prefix* &OPTIONAL *string*

Arguments

prefix

> An integer or NIL

string

> A string

Return Value

> The updated current buffer point

EMPTY-LINE-P Function

Returns T if the specified mark points into a line having no characters; otherwise, the function returns NIL.

Format

> EMPTY-LINE-P *mark*

Arguments

mark

> An Editor mark

Return Value

> T or NIL

END KEYBOARD MACRO Command

Ends the keyboard macro started with the "Start Keyboard Macro" command. After this command is executed, the keyboard macro can be executed by means of the "Execute Keyboard Macro" command.

Display Name Format

> End Keyboard Macro

Function Format

> END-KEYBOARD-MACRO-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> A function that, if called, executes the keyboard macro

END-KEYBOARD-MACRO Function

Terminates the keyboard macro started with the START-KEYBOARD-MACRO function. This function returns a function that, when called, executes the keyboard macro.

Format

> END-KEYBOARD-MACRO

Arguments

> None

Return Value

> A function

END OF BUFFER Command

Moves the buffer point to the end of the current buffer.

Display Name Format

> End of Buffer

Function Format

> END-OF-BUFFER-COMMAND *prefix*

Arguments

prefix

 Ignored

Return Value

 The updated buffer point

END OF LINE Command

Moves the point to the end of the current line if the prefix argument
is NIL. If you specify an integer prefix argument, the point is moved
down the number of lines indicated (or up, if prefix is negative) and
then to the end of the line.

Display Name Format

 End of Line

Function Format

 END-OF-LINE-COMMAND *prefix*

Arguments

prefix

 An integer or NIL

Return Value

 The modified point

END-OF-LINE-P Function

Returns T if the specified mark points to the position immediately
following the last character on a line; otherwise, the function
returns NIL.

Format

 END-OF-LINE-P *mark*

Arguments

mark

>   An Editor mark

Return Value

>   T or NIL

END OF OUTERMOST FORM Command

Moves the buffer point from inside a LISP form to the end of the outermost form surrounding it. If the point is between two outer forms, it is moved to the end of the following one. An outermost form is one whose opening parenthesis is in the leftmost column on the screen.

Display Name Format

>   End of Outermost Form

Function Format

>   END-OF-OUTERMOST-FORM-COMMAND *prefix*

Arguments

prefix

>   Ignored

Return Value

>   The updated buffer point mark

END OF PARAGRAPH Command

Moves the mark to the end of the paragraph. If the mark is not supplied, it defaults to the current buffer point.

Display Name Format

>   End of Paragraph

Function Format

>   END-OF-PARAGRAPH-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> Ignored

mark

> An Editor mark that defaults to the current buffer point

Return Value

> The updated mark

END OF WINDOW Command

Moves the cursor to the end of the current window.

Display Name Format

> End of Window

Function Format

> END-OF-WINDOW-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> The updated current buffer point

ENQUEUE-EDITOR-COMMAND Function

Places the argument function onto the queue for later processing by the Editor. The function can be any LISP function; it will be called in the correct time relation to commands invoked from the keyboard and from the pointing device. The value of the keyword :ARGUMENTS can be a list of arguments to be passed to the argument function.

Format

> ENQUEUE-EDITOR-COMMAND *function* &KEY :ARGUMENTS

Arguments

function

> Any LISP function

:ARGUMENTS

> A list of arguments to be passed to the argument function. The default is NIL.

Return Value

> The argument function

Causes the text in the region bound to the "Buffer Select Region" variable to be evaluated as LISP code. The result of the evaluation is printed in the information area. The result is also bound to the "LISP Evaluation Result" variable.

Display Name Format

> Evaluate LISP Region

Function Format

> EVALUATE-LISP-REGION-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> A list of the values returned from the region evaluated

EXCHANGE POINT AND SELECT MARK Command

Moves the cursor to the location bound to the "Buffer Select Mark" variable in the current buffer, and sets the "Buffer Select Mark" variable to point to the old cursor position. The command function returns the updated buffer select mark or NIL if no mark was selected.

Display Name Format

> Exchange Point and Select Mark

Function Format

EXCHANGE-POINT-AND-SELECT-MARK-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The updated buffer select mark or NIL

EXECUTE KEYBOARD MACRO Command

Executes the most recently defined keyboard macro once if the prefix argument is NIL. If you specify an integer prefix argument, the command is executed the number of times indicated.

Display Name Format

Execute Keyboard Macro

Function Format

EXECUTE-KEYBOARD-MACRO-COMMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The value returned by the last function executed in the keyboard macro

EXECUTE NAMED COMMAND Command

Prompts the user for the name of an Editor command to execute if you do not specify one. The prefix argument is passed to the command you want executed.

Display Name Format

Execute Named Command

Function Format

> EXECUTE-NAMED-COMMAND-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> The value returned by the named command

EXIT Command

Returns control to LISP, and the Editor state is lost. If there are modified buffers, the Editor asks the user if the buffers should be saved. If the response is yes, the Editor executes the "Write Modified Buffers" command.

Display Name Format

> Exit

Function Format

> EXIT-EDITOR-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> Nothing

EXIT-EDITOR-COMMAND Function

See "Exit" command.

EXIT RECURSIVE EDIT Command

Causes the Editor to exit one level of calls to the ED function, returning no values from ED. You must use this command to return from a recursive call to ED. If invoked from the Editor's top level, the command has the same effect as "Pause Editor".

This command is commonly used in conjunction with the command "Query Search Replace".

Display Name Format

>   Exit Recursive Edit

Function Format

>   EXIT-RECURSIVE-EDIT-COMMAND *prefix*

Arguments

prefix

>   Ignored

Return Value

>   Undefined

FIND-AMBIGUOUS Function

Returns a list of those strings in the string table that begin with the specified string. The list is in alphabetical order. String comparisons are case insensitive.

Format

>   FIND-AMBIGUOUS *string string-table*

Arguments

string

>   A string to be compared with the string-table entries

string-table

>   An Editor string table

Return Value

An alphabetically ordered list of those strings in the string table that begin with the specified string, or NIL if none.

FIND-ATTRIBUTE Function

Returns the symbol that names the specified attribute if the argument is a defined attribute, or NIL otherwise.

Format

FIND-ATTRIBUTE *attribute*

Arguments

attribute

An attribute specifier

Return Value

A symbol or NIL

FIND-BUFFER Function

Returns the buffer if the argument is a buffer specifier, or NIL otherwise.

Format

FIND-BUFFER *buffer*

Arguments

buffer

An Editor buffer specifier

Return Value

An Editor buffer or NIL

FIND-COMMAND Function

Returns the associated function if the argument is a command specifier, or NIL otherwise.

Format

FIND-COMMAND *command*

Arguments

command

An Editor command specifier

Return Value

The function associated with the command or NIL

FIND-STYLE Function

Returns an Editor style if the argument is a style specifier, or NIL otherwise.

Format

FIND-STYLE *style*

Arguments

style

An Editor style specifier

Return Value

An Editor style or NIL

FIND-VARIABLE Function

Returns an Editor variable symbol if the argument is an Editor variable, the symbol that names an Editor variable specifier, or NIL otherwise.

Format

FIND-VARIABLE *variable*

Arguments

variable

    An Editor variable specifier

Return Value

    The symbol that names the Editor variable, or NIL

FIRST-LINE-P Function

Is a predicate that returns T if the mark points into the first line in a buffer or a disembodied region.

Format

    FIRST-LINE-P *mark*

Arguments

mark

    An Editor mark

Return Value

    T or NIL

FORWARD CHARACTER Command

Moves the point in the current window forward by one character if the prefix argument is NIL. If you specify an integer prefix argument, the point is moved forward by the number of characters indicated (or backward, if the prefix is negative).

Display Name Format

    Forward Character

Function Format

    FORWARD-CHARACTER-COMMAND *prefix*

Arguments

prefix

An integer specifying how many characters to move

Return Value

The updated buffer point mark

FORWARD KILL RING Command

Rotates the kill ring forward by the number of elements specified by
the prefix argument. The prefix defaults to 1.

Category

:KILL-RING

Display Name Format

Forward Kill Ring

Function Format

FORWARD-KILL-RING-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

Undefined

FORWARD PAGE Command

Moves the point forward one page.   A page is delimited by any
character having a "Page Delimiter" attribute value of 1. If you
specify an integer prefix argument, the point is moved forward the
number of pages indicated.

Display Name Format

Forward Page

Function Format

> FORWARD-PAGE-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> Updated buffer point

FORWARD SEARCH Command

Prompts the user for a string to use as a pattern for a forward search. The string defaults to the value of the Editor variable "Last Search String." The buffer point is moved to the end of the matching string. If the user specifies a prefix argument, *n*, the search occurs *n* times.

Display Name Format

> Forward Search

Function Format

> FORWARD-SEARCH-COMMAND *prefix* &OPTIONAL *string*

Arguments

prefix

> An integer or NIL

string

> A string that defaults to the value of the Editor variable "Last Search String"

Return Value

> A modified buffer point

FORWARD WORD Command

Moves the point forward to the beginning of the next word. A word is delimited by a character having a "Word Delimiter" attribute value of 1. If you specify an integer prefix argument, the point is moved forward the number of words indicated.

Display Name Format

Forward Word

Function Format

FORWARD-WORD-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

A modified buffer point

GENERAL PROMPTING Buffer

Is used by the general prompting facility to display prompts and obtain input from the user. A floating window is associated with this buffer. The following commands are locally bound in this buffer:

1. "Prompt Help"
2. "Prompt Read and Validate"
3. "Prompt Show Alternatives"
4. "Prompt Complete String"

Display Name Format

General Prompting

Symbol Format

EDITOR-PROMPTING-BUFFER

GET-BOUND-COMMAND-FUNCTION Function

Returns the function of the command currently bound to the specified key-sequence; or, if you specify a context, bound to the specified key-sequence in that context. If the specified sequence is ambiguous, the keyword :PREFIX is returned.

Format

GET-BOUND-COMMAND-FUNCTION *key-sequence* OPTIONAL *context*

Arguments

key-sequence

A character or vector of characters

context

An Editor context specifier that defaults to the current context

Return Value

A function, :PREFIX, or NIL

GET-POINTER-STATE Function

Returns an object that contains the state of the pointing device at some point in time. The pointer-state information includes:

- The text position (line and character position) indicated by the pointer cursor.

- The window position (column and row coordinates in a particular Editor window) indicated by the pointer cursor.

- The state (up or down) of each button on the pointing device. If a button was in transition (being depressed or released) at the point in time for which the pointer state is stored, the button state is the state of the buttons at the end of the transition.

- In some cases, a particular previous action of the pointing device (see below).

If called from within an Editor command *and* if that command was invoked by an action of the pointing device, GET-POINTER-STATE returns an object containing the pointer state at the time of the pointer action that invoked the command. In this case the action information in the pointer-state object is the action that invoked the currently

executing command (see BIND-POINTER-COMMAND for information on pointer actions). If the command was not invoked by a pointer action, GET-POINTER-STATE returns the current state of the pointing device.

If called from outside the active Editor environment, this function returns an object that contains the current state of the pointing device: text position, window position, and button state. In this case the action information in the pointer-state object is NIL. If the pointer cursor is outside the Editor's display area, GET-POINTER-STATE returns NIL.

GET-POINTER-STATE is useful in commands that perform different actions depending on some feature of the pointer state other than the particular pointer action that invoked them.

The information contained in the pointer-state object can be accessed by means of the functions POINTER-STATE-TEXT-POSITION, POINTER-STATE-WINDOW-POSITION, POINTER-STATE-BUTTONS, and POINTER-STATE-ACTION.

Format

    GET-POINTER-STATE

Arguments

    None

Return Value

    A pointer-state object or NIL

GET-STRING-TABLE-VALUE Function

Searches the specified string table for an entry whose key matches the string argument. You can use this function with SETF to modify the contents of the string table.

Format

    GET-STRING-TABLE-VALUE *string string-table*

Arguments

string

    A character string

string-table

    An Editor string table

Return Value

Two values:

1.   The first value is the entry found, or NIL.

2.   The second value is T if the first value is valid.

GROW WINDOW Command

Increases the height of the specified window (or current window, if none is specified) by one line. If the window is anchored, there must be at least one other window also being displayed on the screen. Other displayed windows that are anchored decrease in height proportionately, except any window having only one line. Floating windows can always grow or shrink within the range of one line to the height of the screen.

If the prefix is a positive integer, the window grows by the number of lines indicated. If the prefix is a negative integer, the window shrinks by the number of lines indicated.

Display Name Format

Grow Window

Function Format

GROW-WINDOW-COMMAND *prefix* OPTIONAL *window*

Arguments

prefix

An integer or NIL

window

An Editor window

Return Value

The new window height

HELP Buffer

Is used to display Help information. A floating window is associated with this buffer. The buffer is used by the Editor "Help" command and by the prompting facility. It can also be used by user-defined commands.

Display Name Format

Help

Symbol Format

EDITOR-HELP-BUFFER

HELP Command

Is used to supply assistance to the user while the Editor is being used. This command makes a window into the "Help" buffer visible. It inserts the text of the help-string argument, if supplied, or the text of the current value of the "Help Text" Editor variable. If both are NIL, the command signals an Editor error with the message "No Help Available."

Display Name Format

Help

Function Format

HELP-COMMAND *prefix* &OPTIONAL *help-string*

Arguments

prefix

Ignored

help-string

An optional string for use as the current help text

Return Value

None

HELP ON EDITOR ERROR Command

Displays the last message created by the EDITOR-ERROR-WITH-HELP function in a window into the "Help" buffer.

Display Name Format

Help on Editor Error

Function Format

HELP-ON-EDITOR-ERROR-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

None

HELP TEXT Editor Variable

Specifies the help text to be displayed by the "Help" command. The value of this variable must be a string, a function of no arguments that returns a string, or NIL. The global binding of this variable contains the default help text for the Editor.

Display Name Format

Help Text

Symbol Format

HELP-TEXT

HIGHLIGHT-REGION-P Function

Returns T if the argument is a highlight region and NIL otherwise.

Format

HIGHLIGHT-REGION-P *object*

Arguments

object

    Any Lisp object

Return Value

    T or NIL

ILLEGAL OPERATION Command

Signals an Editor error with the message "Illegal Operation." This command is used to disable a command locally within a style or buffer. For example, to disable "Self Insert" for a particular character, bind the character to "Illegal Operation."

Display Name Format

    Illegal Operation

Function Format

    ILLEGAL-OPERATION-COMMAND *prefix*

Arguments

prefix

    Ignored

Return Value

    None

INDENT LISP LINE Command

Adjusts the indentation of the current LISP source line so that it is indented appropriately in the program context.

Display Name Format

    Indent LISP Line

Function Format

    INDENT-LISP-LINE-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> Undefined

INDENT LISP REGION Command

Adjusts the indentation of the LISP text in the region in the Editor variable "Buffer Select Region," so that the text lines up appropriately in the program context.

Display Name Format

> Indent LISP Region

Function Format

> INDENT-LISP-REGION-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> None

INDENT OUTERMOST FORM Command

Determines the outermost LISP form that surrounds the current buffer point and indents each line in the form appropriately. An outermost form is one whose opening parenthesis is in the leftmost column on the screen.

Display Name Format

> Indent Outermost Form

Function Format

> INDENT-OUTERMOST-FORM-COMMAND *prefix*

Arguments

prefix

>   Ignored

Return Value

>   Undefined

INFORMATION-AREA-HEIGHT Function

Returns the number of lines occupied by the information area at the
bottom of the screen. You can use this function with SETF to modify
the number of lines, but you cannot set that number to less than 1.
When you alter the height of the information area, the heights of any
anchored windows are adjusted accordingly.

Format

>   INFORMATION-AREA-HEIGHT

Arguments

>   None

Return Value

>   The number of lines occupied by the information area

INFORMATION-AREA-OUTPUT-STREAM Variable

Is bound to an output stream that can be used to write to the
information area. Lines written to the information area are truncated
if they are longer than the screen is wide. A TERPRI executed to this
stream scrolls the lines in the information area.

INFORMATION AREA POINTER PATTERN Editor Variable

Specifies a 16x16 bitmap that determines the pointer cursor pattern
when the pointer is in the Editor's information area. When set to
NIL, the pointer cursor pattern is the VAXstation default (an arrow).
See the functions SET-POINTER-PATTERN and MAKE-BITMAP in the *VAX
LISP/VMS Graphics Programming Guide.*

Display Name Format

    Information Area Pointer Pattern

Symbol Format

    INFORMATION-AREA-POINTER-PATTERN

INFORMATION AREA POINTER PATTERN X Editor Variable

Specifies the horizontal coordinate of the active pixel of the bitmap specified by the Editor variable "Information Area Pointer Pattern". The possible values are an integer in the range 0-15 or NIL. See the function SET-POINTER-PATTERN in the *VAX LISP/VMS Graphics Programming Guide*.

Display Name Format

    Information Area Pointer Pattern X

Symbol Format

    INFORMATION-AREA-POINTER-PATTERN-X

INFORMATION AREA POINTER PATTERN Y Editor Variable

Specifies the vertical coordinate of the active pixel of the bitmap specified by the Editor variable "Information Area Pointer Pattern", relative to the lower edge of the bitmap. The possible values are an integer in the range 0-15 or NIL. See the function SET-POINTER-PATTERN in the *VAX LISP/VMS Graphics Programming Guide*.

Display Name Format

    Information Area Pointer Pattern Y

Symbol Format

    INFORMATION-AREA-POINTER-PATTERN-Y

INITIALIZE-EDITOR Function

Initializes the Editor without actually entering it. The screen management system and all standard Editor buffers are initialized. The function is automatically called when the Editor is first invoked. This function must be called prior to the use of any window creation

or manipulation functions. If, for example, you want to create Editor
windows in an initialization file, you must include a call to this
function in the initialization file.

Format

    INITIALIZE-EDITOR

Arguments

    None

Return Value

    T, if the Editor is actually initialized by this call; or NIL, if
    it had already been initialized.

INSERT BUFFER Command

Prompts the user for a buffer name if one is not supplied and inserts
the text of the buffer specified into the current buffer at the buffer
point. The point is left at the end of the inserted text.

Category

    :GENERAL-PROMPTING

Display Name Format

    Insert Buffer

Function Format

    INSERT-BUFFER-COMMAND *prefix* &OPTIONAL *name*

Arguments

prefix

    Ignored

name

    An Editor buffer

Return Value

    The modified point

INSERT-CHARACTER Function

Inserts the specified character at the position of the specified mark. If the character is a #\Newline, the line is broken into two lines.

Format

INSERT-CHARACTER *mark character*

Arguments

mark

An Editor mark

character

A character

Return Value

The character

INSERT CLOSE PAREN AND MATCH Command

Inserts the last character typed at the current buffer point. If the character is a list terminator, this command finds and momentarily highlights the matching list initiator. If the list initiator is not visible in the window, the line containing it is displayed in the information area, and the matching list initiator is highlighted. If there is no matching list initiator, an Editor error is signaled. (See "LISP Syntax" attribute, especially the values :LIST-TERMINATOR and LIST-INITIATOR.)

Display Name Format

Insert Close Paren and Match

Function Format

INSERT-CLOSE-PAREN-AND-MATCH-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

> NIL

INSERT FILE Command

Prompts the user for the name of a file if none is specified and inserts the file at the current buffer point. The point is left at the end of the inserted text.

Category

> :GENERAL-PROMPTING

Display Name Format

> Insert File

Function Format

> INSERT-FILE-COMMAND *prefix* &OPTIONAL *file-name*

Arguments

prefix

> Ignored

file-name

> A pathname, namestring, or stream

Return Value

> The updated buffer point

INSERT-FILE-AT-MARK Function

Inserts the specified file into a buffer at the position of the specified mark. This function checks to see if there is enough dynamic memory available to load the file and signals an Editor error if there is not. There is an implied #\Newline character at the end of the file but not at the beginning.

Format

> INSERT-FILE-AT-MARK *pathname mark*

Arguments

pathname

    A pathname, namestring, or stream

mark

    An Editor mark

Return Value

    The mark

INSERT-REGION Function

Inserts the specified region at the position of the specified mark. The region text is copied in the process.

Format

    INSERT-REGION *mark region*

Arguments

mark

    An Editor mark

region

    An Editor region

Return Value

    The region

INSERT-STRING Function

Inserts a string at the position of the specified mark. Embedded newline characters cause additional Editor lines to be inserted. The string is copied. The optional *start* and *end* arguments allow you to specify a substring to be inserted.

Format

    INSERT-STRING *mark string* &OPTIONAL *start end*

Arguments

mark

> An Editor mark

string

> A string

start

> An integer that is an index into the string.   The   default   is
> zero.

end

> An integer that is an index into the string.   The default is   the
> length of the string.

Return Value

> The string

INVOKE-HOOK Function

Searches the entire current context, in the reverse of the usual
search order, for occurrences of the variable (that is, the search
occurs in the order -- the global definition, the major style of the
current buffer, the minor styles in reverse order, and the local
variables of the current buffer). The function then applies to the
specified args arguments all the functions bound to each occurrence of
the specified Editor variable.

Format

> INVOKE-HOOK *name* &REST *args*

Arguments

name

> An Editor variable specifier

args

> The arguments to be passed to the hook function

Return Value

The value that the last hook function returns

KILL ENCLOSING LIST Command

Deletes the list that immediately encloses the argument mark and returns a disembodied region that contains the deleted text. The mark defaults to the current buffer point. If the mark is located within a symbol, the list that immediately encloses the symbol is deleted. The mark is left at the location where the deleted text appeared.

If a positive prefix argument *n* is specified, the next *n* enclosing lists are deleted and returned as a disembodied region. If a zero or negative prefix argument is specified, no action occurs and NIL is returned. If the list to be deleted cannot be determined because of missing text, no action occurs and NIL is returned.

Display Name Format

Kill Enclosing List

Function Format

KILL-ENCLOSING-LIST-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

An integer or NIL

mark

An Editor mark that defaults to the current buffer point

Return Value

A disembodied region or NIL

KILL LINE Command

Deletes the rest of the current line and adds it to the end of the current kill-ring region if the previous command was in the category :KILL-RING; or, creates a new kill-ring region. If you supply a positive integer prefix *n*, the command deletes the rest of the current line and *n-1* lines following the current line; the line following the last line deleted is appended to the beginning portion of the current

line. If you supply a negative integer prefix -*n*, the command deletes the portion of the current line preceding the point and *n*-1 lines preceding the current line; the rest of the current line is appended to the line preceding the first line deleted.

Category

:KILL-RING

Display Name Format

Kill Line

Function Format

KILL-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

Undefined

KILL NEXT FORM Command

Deletes the LISP form immediately after the mark at the current parenthesis nesting level and returns a disembodied region that contains the deleted text. The mark defaults to the current buffer point. If a positive prefix argument *n* is specified, then the next *n* LISP objects at the current parenthesis nesting level are deleted and returned as a disembodied region. If no next form is found within the innermost enclosing list or if a negative prefix argument is supplied, no action occurs and NIL is returned.

Display Name Format

Kill Next Form

Function Format

KILL-NEXT-FORM-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> A positive integer or NIL

mark

> An Editor mark that defaults to the current buffer point

Return Value

> A disembodied region or NIL

KILL PARAGRAPH Command

Deletes the rest of the current paragraph and adds it to the end of the current kill-ring region if the previous command was in the category :KILL-RING; or, creates a new kill-ring region. If a prefix argument *n* is supplied, the command deletes the rest of the current paragraph and the next *n-1* paragraphs.

Category

> :KILL-RING

Display Name Format

> Kill Paragraph

Function Format

> KILL-PARAGRAPH-COMMAND *prefix*

Arguments

prefix

> Positive integer or NIL

Return Value

> Undefined

KILL PREVIOUS FORM Command

Deletes the LISP form immediately before the mark at the current parenthesis nesting level and returns a disembodied region that contains the deleted text. The mark defaults to the current buffer point. If a positive prefix argument *n* is specified, then the previous *n* LISP objects at the current parenthesis nesting level are deleted and returned as a disembodied region. If no previous form is found within the innermost enclosing list or if a negative prefix argument is supplied, no action occurs and NIL is returned.

Display Name Format

Kill Previous Form

Function Format

KILL-PREVIOUS-FORM-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

A positive integer or NIL

mark

An Editor mark that defaults to the current buffer point

Return Value

A disembodied region or NIL

KILL REGION Command

Deletes a region and adds it to the end of the current kill-ring region if the previous command was in the category :KILL-RING; or, creates a new kill-ring region. If the region is not supplied, it defaults to the region between the buffer select mark and the current buffer point.

Category

:KILL-RING

Display Name Format

Kill Region

Function Format

> KILL-REGION-COMMAND *prefix* &OPTIONAL *region*

Arguments

prefix

> Ignored

region

> An Editor region that defaults to the buffer-select-region

Return Value

> Undefined

KILL REST OF LIST Command

Deletes the part of the list that immediately follows the argument mark and returns a disembodied region that contains the deleted text. The mark defaults to the current buffer point. If the mark is not in a list or if the list terminator cannot be found, no action occurs and NIL is returned.

Display Name Format

> Kill Rest of List

Function Format

> KILL-REST-OF-LIST-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> Ignored

mark

> An Editor mark that defaults to the current buffer point

Return Value

> A disembodied region or NIL

LAST-CHARACTER-TYPED Variable

Is bound to the last character typed by the user.

LAST-LINE-P Function

Is a predicate that returns T if the specified mark points to the last line in a buffer or a disembodied region.

Format

    LAST-LINE-P *mark*

Arguments

mark

    An Editor mark

Return Value

    T or NIL

LAST SEARCH DIRECTION Editor Variable

Indicates the direction of the most recent search by means of a keyword having the value of either :FORWARD or :BACKWARD.

Display Name Format

    Last Search Direction

Symbol Format

    LAST-SEARCH-DIRECTION

LAST SEARCH PATTERN Editor Variable

Specifies the search pattern that was last used with the search commands.

Display Name Format

    Last Search Pattern

Symbol Format

>   LAST-SEARCH-PATTERN

LAST SEARCH STRING Editor Variable

Specifies the string that was last used with the search commands.

Display Name Format

>   Last Search String

Symbol Format

>   LAST-SEARCH-STRING

LINE-BUFFER Function

Returns the buffer associated with the Editor line. This function returns NIL if the line is not associated with any buffer.

Format

>   LINE-BUFFER *line*

Arguments

line

>   An Editor line

Return Value

>   An Editor buffer or NIL

LINE-CHARACTER Function

Returns the character in the text of the specified line at the position indicated by the specified index (the first character is specified by 0, the second by 1, and so on). The function returns NIL if there is no character at that position. It returns the #\NEWLINE character if the specified position is at the end of the line. You can use this function with SETF to change the character at that position.

Format

    LINE-CHARACTER *line index*

Arguments

line

    An Editor line

index

    A fixnum

Return Value

    A character or NIL

LINE-END Function

Changes the specified mark so that it points to the end of the line.

Format

    LINE-END *mark* &OPTIONAL *line*

Arguments

mark

    An Editor mark

line

    An Editor line that defaults to the line that the mark points
    into

Return Value

    The modified mark

LINE-LENGTH Function

Returns an integer that is the number of characters contained in the
Editor line. The line-break is not included.

Format

    LINE-LENGTH *line*

Arguments

line

    An Editor line

Return Value

    An integer

LINE-NEXT Function

Returns the line following the specified line.

Format

    LINE-NEXT *line*

Arguments

line

    An Editor line

Return Value

    An Editor line, or NIL if there is no following line

LINE-OFFSET Function

Changes the specified mark so that it points *n* lines after the line it currently points into (or *n* lines before, if *n* is negative). The function attempts to have the mark in the new line point to the same position it pointed to in the old line. If you do not want the mark to point to that position, you can specify another position in the new line by supplying a value for the index argument.

If there are not enough characters in the new line for a specified or defaulted position to exist, the mark is positioned at the end of the line. If there are not enough lines after (or before) the mark to satisfy the *n* argument, the mark is not modified, and NIL is returned.

Format

    LINE-OFFSET *mark n* &OPTIONAL *index*

Arguments

mark

    An Editor mark

n

    A fixnum

index

    A fixnum that defaults to the character position of mark

Return Value

    The modified mark, or NIL

LINE-PREVIOUS Function

Returns the line preceding the specified line.

Format

    LINE-PREVIOUS *line*

Arguments

line

    An Editor line

Return Value

    A line, or NIL if there is no preceding line

LINE-START Function

Changes the specified mark so that it points to the beginning of
line.

Format

    LINE-START *mark* &OPTIONAL *line*

Arguments

mark

> An Editor mark

line

> An Editor line that defaults to the line that the mark points into

Return Value

> The modified mark

LINE-STRING Function

Returns a character string that is the text contained in the Editor line. You can use LINE-STRING with the SETF macro to modify the text contained in an Editor line. In particular, if you do any destructive operations on the string (for example, using the LISP NSTRING-UPCASE function to make a portion of the text uppercase), you must use SETF to have the change appropriately reflected.

Format

> LINE-STRING *line*

Arguments

line

> An Editor line

Return Value

> A string

LINE-TO-REGION Function

Returns a region consisting of either the specified line if the argument is a line or the line that the mark points into if the argument is a mark.

Format

> LINE-TO-REGION *line-or-mark*

Arguments

line-or-mark

    An Editor line or an Editor mark

Return Value

    A region containing the specified line

LINE TO TOP OF WINDOW Command

Moves the line that the buffer point points into, so that it is the first displayed line in the current window.

Display Name Format

    Line to Top of Window

Function Format

    LINE-TO-TOP-OF-WINDOW-COMMAND *prefix* &OPTIONAL *mark window*

Arguments

prefix

    Ignored

mark

    A mark pointing into the line to go to the top of the window. The default is the current buffer point.

window

    The window in which the specified line is to be the top line. The default is the current window.

Return Value

    None

LINEP Function

Is a predicate that returns T if object is an Editor line.

Format

    LINEP *object*

Arguments

object

    Any LISP object

Return Value

    T or NIL

LINES-RELATED-P Function

Returns T if the two specified lines are in either the same buffer or
the same disembodied region.

Format

    LINES-RELATED-P *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LINE/= Function

Returns T if line1 and line2 are not the same. The two lines do not
have to be in the same buffer.

Format

    LINE/= *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LINE< Function

Returns T if line1 precedes line2. The two lines must be in the same buffer.

Format

    LINE< *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LINE<= Function

Returns T if line1 precedes line2 or is the same line as line2. The two lines must be in the same buffer.

Format

    LINE<= *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LINE= Function

Returns T if line1 and line2 are the same line. The two lines do not
have to be in the same buffer.

Format

    LINE= *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LINE> Function

Returns T if line1 follows line2. The two lines must be in the same
buffer.

Format

    LINE> *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LINE > = Function

Returns T if line1 follows line2 or if they are the same line. The two lines must be in the same buffer.

Format

    LINE>= *line1* *line2*

Arguments

line1

    An Editor line

line2

    Another Editor line

Return Value

    T or NIL

LISP COMMENT COLUMN Editor Variable

Is bound in "VAX LISP" style to an integer that indicates the column in which a LISP comment begins on a line. This Editor variable is used by the indentation commands as well as by the "Move to LISP Comment" command. By default, LISP comments begin at column 49. You can change the LISP comment column by using the SETF macro.

Display Name Format

LISP Comment Column

Symbol Format

LISP-COMMENT-COLUMN

LISP EVALUATION RESULT Editor Variable

Is bound to a list of the values returned when a LISP region is evaluated in the Editor.

Display Name Format

LISP Evaluation Result

Symbol Format

LISP-EVALUATION-RESULT

LISP SYNTAX Attribute

Used in "VAX LISP" style to determine the structure of the LISP source being edited. The value of this attribute for a given character defines the role (if any) that character plays in the syntax of LISP. The table below lists the values this attribute can take and shows a sample character for each attribute value. You can modify the value of this attribute for a given character by using the SETF macro on a CHARACTER-ATTRIBUTE form.

Display Name Format

LISP Syntax

Symbol Format

LISP-SYNTAX

EDITOR OBJECT DESCRIPTIONS

Table 1: "LISP Syntax" Attribute Values

Value	Example of Character	Description
:LIST-INITIATOR	#\\(	A character that signifies the beginning of a list.
:LIST-TERMINATOR	#\\)	A character that signifies the end of a list.
:COMMENT-DELIMITER	#\\;	A character that signifies the beginning of a comment.
:STRING-DELIMITER	#\\"	A character that delimits the beginning and end of a string.
:SINGLE-ESCAPE	#\\\\	A character used as a single escape character. See *COMMON LISP: The Language* for more information on single escape characters.
:MULTIPLE-ESCAPE	#\\|	A character used as a multiple escape character. See *COMMON LISP: The Language* for more information on multiple escape characters.
:WORD-DELIMITER	#\\space	A character used to separate words.
:READ-MACRO	#\\'	A macro character that the READ function accepts.
:CONSTITUENT	#\\A	A character used as a constituent character. See *COMMON LISP: The Language* for more information on constituent characters.

You can use the LOCATE-ATTRIBUTE function in conjunction with the "LISP Syntax" attribute to find an instance of a particular LISP syntactic element. For example, the following function call finds the first occurrence of a string delimiter beyond the current buffer point:

```
(LOCATE-ATTRIBUTE (CURRENT-BUFFER-POINT)
                  "LISP Syntax"
                  :TEST #'(LAMBDA (X) (EQ X :STRING-DELIMITER)))
```

NOTE

Characters whose "LISP Syntax" attribute value is :LIST-INITIATOR, :LIST-TERMINATOR, :COMMENT-DELIMITER, :STRING-DELIMITER, and :READ-MACRO are also considered to delimit words, in addition to the characters whose value is :WORD-DELIMITER.

LIST BUFFERS Command

Causes a list of the current buffers to be displayed in the Help window.

Display Name Format

   List Buffers

Function Format

   LIST-BUFFERS-COMMAND *prefix*

Arguments

prefix

   Ignored

Return Value

   NIL

LIST KEY BINDINGS Command

Prompts the user for a context in which to search for key bindings. If none is specified, all the currently visible bindings are used. The command formats and displays a list that includes the key sequence, the associated command name, and the context of the binding. Key sequences that are bound to the "Self Insert" command are not included in the list.

Category

   :GENERAL-PROMPTING

Display Name Format

   List Key Bindings

Function Format

>   LIST-KEY-BINDINGS-COMMAND *prefix* &OPTIONAL *context*

Arguments

prefix

>   Ignored

context

>   The context in which to search for bindings. If the value of the
>   context argument is NIL, the user is prompted for a context; if
>   the value is T, the entire visible set of bindings is used.

Return Value

>   None

LOCATE-ATTRIBUTE Function

Locates a character in a region or string whose value for the
specified attribute satisfies the given test.

If the argument you specify is a mark, the function begins searching
at the mark and proceeds in the direction specified by the :DIRECTION
keyword (:FORWARD, the default, or :BACKWARD). The search continues
until a suitable character is found, or until there are no more
characters, or until the mark specified by the :LIMIT keyword is
reached.   If a character is found, the mark is updated to point to
that character. If no character is found, the function returns NIL.

If the argument you specify is a string, the string is searched
starting at the beginning of the string or at the position indicated
by :START if the :START keyword is specified. If a character is found
that satisfies the test, the function returns the position of the
character in the string. If no suitable character is found, the
function returns NIL.

Format

>   LOCATE-ATTRIBUTE *mark-or-string* *attribute* &KEY :TEST
>   :CONTEXT
>   :DIRECTION
>   :LIMIT
>   :START :END

Arguments

mark-or-string

Either a mark that specifies the starting position or a string to be searched

attribute

An attribute specifier

:TEST

A predicate function of one argument used to test the attribute value of each character. :TEST returns non-NIL if the argument that specifies the attribute value of the character matches a character during the search. The default is #'PLUSP, which is often used with a 0 or 1 value for a character attribute. The attribute value may be any LISP object. The test function must accept objects of the appropriate type.

:CONTEXT

An Editor context specifier for the character attribute. The default is the current context.

:DIRECTION

The direction to scan for a character satisfying the attribute test. Values can be :FORWARD or :BACKWARD. The default is :FORWARD.

:LIMIT

Used only when you specify a mark for the mark-or-string argument. :LIMIT must be a mark that points into the same buffer or disembodied region as the mark you specify for the mark-or-string argument. If no character is found that satisfies the test before the :LIMIT mark is reached, the search fails. (The character pointed to by :LIMIT is included in the search.) If the search direction is forward, the limit mark must be located after the starting mark; otherwise, the limit mark must be located before the starting mark.

:START

Used only when you specify a string for the mark-or-string argument. :START is an integer that specifies the character position in the string where the search begins. The default value for :START is 0. The :START argument is ignored if you specify a mark for the mark-or-string argument.

:END

> Used only when you specify a string for the mark-or-string
> argument. :END is an integer that specifies the character
> position in the string where the search ends. The default value
> for :END is the length of the string. The :END argument is
> ignored if you specify a mark for the mark-or-string argument.

Return Value

Three values:

1. The modified mark, if the argument for mark-or-string is a
    mark and the search is successful; or, if the mark-or-string
    argument specified is a string, an integer that represents
    the character's position in the string; or NIL if no
    character is found (the search is unsuccessful).

2. The character at the position of the mark.

3. The value of the attribute for that character.

LOCATE-PATTERN Function

Searches for a text string that matches the specified search pattern.
You create search patterns by means of the MAKE-SEARCH-PATTERN
function. The mark is changed so that it points to the beginning of
the located text.

Format

    LOCATE-PATTERN *mark search-pattern*

Arguments

mark

> An Editor mark

search-pattern

> An Editor search pattern

Return Value

> The number of characters matched, or NIL

MAJOR STYLE ACTIVATION HOOK Editor Variable

Specifies a hook function that is called whenever a major style is activated in a buffer. The function is called with two arguments, the style and the buffer.

Display Name Format

Major Style Activation Hook

Symbol Format

MAJOR-STYLE-ACTIVATION-HOOK

MAKE-BUFFER Function

Takes a name specifier and creates a buffer of the specified name. The calling of this function causes the "Buffer Creation Hook" to be invoked. If a buffer having the specified name already exists, it is returned.

Format

```
MAKE-BUFFER buffer-name &KEY :DOCUMENTATION
                             :MAJOR-STYLE :MINOR-STYLES
                             :VARIABLES
                             :OBJECT :TYPE
                             :PERMANENT
```

Arguments

buffer-name

The name for the new buffer. This can be specified as either a symbol or a list of a symbol with the keyword :DISPLAY-NAME and a string; that is,

name | (name :DISPLAY-NAME string)

:DOCUMENTATION

A string used as the documentation string for the buffer.

:MAJOR-STYLE

An Editor style that is to be the major style of the buffer. This defaults to the global value of the "Default Major Style" Editor variable.

:MINOR-STYLES

> A list of Editor styles that are to be the minor styles of the buffer. This defaults to the global value of the "Default Minor Styles" Editor buffer.

:VARIABLES

> A list of Editor variables that are to be bound in the buffer. This defaults to the global value of the "Default Buffer Variables" Editor variable.

:OBJECT

> The object that is to be edited in the buffer. This can be a pathname, a symbol, or a list that is a form acceptable to the SETF macro.

:TYPE

> The type of the object being edited. This can be specified only if the object is a symbol. The possible values are :FUNCTION (the default) and :VALUE.

:PERMANENT

> If non-NIL, the buffer is created as a permanent buffer. A permanent buffer cannot be deleted with DELETE-BUFFER, it remains in the Editor across a suspend/resume cycle, and it remains if you exit the Editor. The default is NIL.

Return Value

Two values:

1. An Editor buffer

2. T, if this is a new buffer; NIL, if the buffer already existed

MAKE-COMMAND Function

Is used to turn an existing LISP function into an Editor command. The name options to the MAKE-COMMAND function are the same as those for the DEFINE-COMMAND macro and can include a display name, and a category or list of categories. The supplied function must be a function of at least one argument. The prefix argument is passed when the command is executed as a function.

Format

MAKE-COMMAND *name function* &OPTIONAL *documentation*

Arguments

name

A command name specifier

function

A LISP function that is to become an Editor command

documentation

The documentation string that will be used to describe the Editor command (not the function documentation)

Return Value

The function

MAKE-EDITOR-STREAM-FROM-REGION Function

Takes an Editor region and returns an Editor input stream.

Format

MAKE-EDITOR-STREAM-FROM-REGION *region*

Arguments

region

An Editor region

Return Value

An input stream

MAKE-EDITOR-STREAM-TO-MARK Function

Returns an Editor output stream that causes all output to be inserted at the specified mark.

Format

```
MAKE-EDITOR-STREAM-TO-MARK mark
```

Arguments

mark

An Editor mark

Return Value

An output stream

MAKE-EMPTY-REGION Function

Returns a new disembodied Editor region with permanent marks pointing into a line with no characters. The starting mark is right-inserting, and the ending mark is left-inserting.

Format

```
MAKE-EMPTY-REGION
```

Arguments

None

Return Value

An Editor region

MAKE-HIGHLIGHT-REGION Function

Returns a new highlight region. Whenever any of the text in the region is visible in a window, the display of the text is given the specified video rendition. The rendition can be specified as a keyword or list of keywords. The possible values are :BOLD, :BLINK, :REVERSE, and :UNDERLINE.

When specifying highlight regions, you must be aware of the background rendition of the window where the region will be visible. For example, specifying reverse video when the window is already in reverse video will have no apparent effect.

The *set* and *complement* arguments let you adjust the rendition of the display so that you can achieve the desired rendition. The attributes specified in the *set* argument will always be turned on when visible.

The attributes specified by the *complement* argument will complement the existing display values, including any that were turned on by the *set* argument. So to turn off reverse video in the highlight region, you must specify :REVERSE in both the *set* and *complement* arguments.

The highlight region can also be used as a normal region by any Editor function that takes a region as an argument.

Format

MAKE-HIGHLIGHT-REGION *start end* &OPTIONAL *set complement*

Arguments

start

An Editor mark that indicates the beginning of the region

end

An Editor mark that indicates the end of the region

set

A keyword or list of keywords specifying the video renditions to be turned on in the display when visible. The default is NIL.

complement

A keyword or list of keywords specifying the video renditions to be complemented in the display when visible. The default is NIL.

MAKE-MARK Function

Returns a new mark that points to the specified line at the position specified by the index argument.

Format

MAKE-MARK *line index* &OPTIONAL *mark-type*

Arguments

line

An Editor line

index

An integer in the range of 0 to the length of the line

mark-type

> The type of the mark to create -- :LEFT-INSERTING, :RIGHT-INSERTING, or :TEMPORARY. The default is :TEMPORARY.

Return Value

> A new Editor mark

MAKE-REGION Function

Creates an Editor region that starts and ends at the specified marks. Both marks must be in the same buffer or disembodied region.

Format

> MAKE-REGION *start-mark end-mark*

Arguments

start-mark

> A mark for defining the beginning of a region

end-mark

> A mark for defining the end of a region

Return Value

> A new region

MAKE-RING Function

Creates a ring buffer of the size specified by the integer argument.

Format

> MAKE-RING *integer* &OPTIONAL *delete-function*

Arguments

integer

> A positive integer, the maximum size of the ring

delete-function

> A function that is called any time an item is deleted from the ring, either by RING-PUSH or by the application of SETF to RING-REF. The function is called with two arguments -- the item being deleted and the ring. The default is NIL.

Return Value

> The new ring

MAKE-SEARCH-PATTERN Function

Creates a new search pattern that you can use in subsequent searching operations.

Format

> MAKE-SEARCH-PATTERN *kind direction string* &OPTIONAL *reuse-pattern*

Arguments

kind

> A search pattern type, either :CASE-SENSITIVE or :CASE-INSENSITIVE

direction

> A direction to search in -- either :FORWARD or :BACKWARD

string

> The string to be searched for

reuse-pattern

> A previously computed search pattern that will be destructively modified to create the new pattern

Return Value

> A new search pattern

MAKE-STRING-TABLE Function

Returns a new string table having no entries

Format

```
MAKE-STRING-TABLE
```

Arguments

None

Return Value

An empty string table

MAKE-STYLE Macro

Creates a new Editor style. The style will have no attribute, variable, or command bindings. If there is already a style of the specified name, the new style (with no bindings) will replace the old one. Note that any bindings present in the old style are lost.

Format

```
MAKE-STYLE name &OPTIONAL documentation &KEY :ACTIVATION-HOOK
                                              :DEACTIVATION-HOOK
```

Arguments

name

A symbol that names the style or a list of a symbol, the keyword :DISPLAY-NAME, and a string that will become the style's display name

documentation

A documentation string for the style

:ACTIVATION-HOOK

A function that will be invoked whenever this style is activated in a buffer. The function is called with two arguments -- the style and the buffer that the style is activated in.

:DEACTIVATION-HOOK

A function that will be invoked whenever the style is made
inactive in a buffer. The function is called with two arguments
-- the style and the buffer that the style is activated in.

Return Value

The new style

MAKE-WINDOW Function

Takes a buffer or a mark and returns a new window. If the argument is
a mark, the window opens into the buffer that contains the mark, and
the display starts with the line that the mark points into. If the
argument is a buffer, the window opens into that buffer, and the
display starts with the line pointed to by the buffer point. The
window is not automatically displayed.

The calling of this function invokes the "Window Creation Hook".

Format

```
MAKE-WINDOW buffer-or-mark &KEY :HEIGHT :WIDTH
                               :DISPLAY-ROW :DISPLAY-COLUMN
                               :TYPE
                               :LINES-WRAP
                               :LABEL
```

Arguments

buffer-or-mark

An Editor buffer or mark

:HEIGHT

The number of rows to be contained in the window. The minimum
value is one. This value is significant only if the window type
is :FLOATING.

:WIDTH

The number of characters that can be displayed horizontally in a
window. The minimum value is two. The maximum value (and
default) is the width of the available display area. This value
has significance only if the window type is :FLOATING.

:DISPLAY-ROW

>   The screen row (*y* position) at which to start displaying the text
>   of the window. The top row is 1. This value has significance
>   only if the window type is :FLOATING.

:DISPLAY-COLUMN

>   The screen column (*x* position) at which to start displaying the
>   text of the window. The left-hand column is 1. This value has
>   significance only if the window type is :FLOATING.

:TYPE

>   The display type of the window. Can be either :ANCHORED or
>   :FLOATING. The default is the value of the Editor variable
>   "Default Window Type".

:LINES-WRAP

>   If T, specifies that displayed lines are continued on the next
>   line of the display if the length of the Editor line exceeds the
>   width of the window. The default is the value of the Editor
>   variable "Default Window Lines Wrap".

:LABEL

>   A string, a function that returns a string, or NIL. If the value
>   is a string or a function that returns a string, the string is
>   used as the label for the window. Only as much of the string as
>   will fit on the specified side will be displayed. An empty
>   string ("") means that the window is unlabeled. A value of NIL
>   means that the window is not bordered. The default is the value
>   of "Default Window Label."

Return Value

>   The new window

MAP-BINDINGS Function

Calls, with the following three arguments, the specified function for
each key sequence having a binding in the specified context:

1. The sequence of characters bound

2. The command function the sequence is bound to

3. The context specification in which the binding was found. If
    an optional context is specified, only the key bindings in

that context are mapped. If no context is provided, the map is done over all the currently visible bindings.

Format

MAP-BINDINGS *function* &OPTIONAL *context*

Arguments

function

A function of three arguments

context

An optional context specification. The default is NIL.

Return Value

NIL

MAP-BUFFERS Function

Applies the specified function to each buffer in the Editor along with any additional arguments supplied. The specified function must be a function of at least one argument. The first argument will always be an Editor buffer object.

Format

MAP-BUFFERS *function* &REST *args*

Arguments

function

A function to be called for each buffer.   The function must accept at least one argument, a buffer.

args

Any additional arguments that must be passed to the specified function on each call

Return Value

NIL

MAP-STRINGS Function

Calls the specified function for each entry in the specified string table. That function is called with two arguments -- the string that is the key of the entry and the value of the entry.

Format

        MAP-STRINGS *function table*

Arguments

function

        A function of two arguments

table

        A string table

Return Value

        NIL

MARK-CHARPOS Function

Returns the number of characters in the line of text preceding the specified mark.

Format

        MARK-CHARPOS *mark*

Arguments

mark

        An Editor mark

Return Value

        A nonnegative fixnum

MARK-COLUMN Function

Returns the column (position n) at which the specified mark would be displayed, based on the "Print Representation" attribute of each character, if the screen were wide enough. This number is often different from the result of MARK-CHARPOS because some characters take up more than one column when they are displayed. For example, tab characters usually are not displayed as single blank characters. The first character of a line is at display position 1.

Format

    MARK-COLUMN *mark*

Arguments

mark

    An Editor mark

Return Value

    A positive fixnum

MARK-LINE Function

Returns the line that the specified mark points into.

Format

    MARK-LINE *mark*

Arguments

mark

    An Editor mark

Return Value

    An Editor line

MARK-TYPE Function

Returns the type of the specified mark. You can use this function with SETF to change the type of a mark.

Format

    MARK-TYPE *mark*

Arguments

mark

    An Editor mark

Return Value

    The type of the specified mark -- :LEFT-INSERTING, :RIGHT-INSERTING, or :TEMPORARY.

MARK-VISIBLE-P Function

Returns T if the mark position lies within the text contained in the window; returns NIL if it is not.

Format

    MARK-VISIBLE-P *mark window*

Arguments

mark

    An Editor mark

window

    An Editor window

Return Value

    T or NIL

MARK-WINDOW-POSITION Function

Returns NIL if the specified mark position does not lie within the text contained in the window; returns multiple values of the column and row positions of the specified mark if it is visible. The upper left corner position of a window is specified as 1,1.

Format

    MARK-WINDOW-POSITION *mark window*

EDITOR OBJECT DESCRIPTIONS

Arguments

mark

> An Editor mark

window

> An Editor window

Return Value

> Either NIL or the column and row position that the mark is displayed at

MARKP Function

Returns T if the argument is a mark; returns NIL if it is not.

Format

> MARKP *object*

Arguments

object

> Any LISP object

Return Value

> T or NIL

MARK/= Function

Returns T if mark1 and mark2 point to different positions; returns NIL otherwise. The marks can point into different buffers.

Format

> MARK/= *mark1 mark2*

Arguments

mark1

> An Editor mark

mark2

 Another Editor mark

Return Value

 T or NIL

MARK< Function

Returns T if mark1 points to a character preceding mark2; returns NIL otherwise. An error occurs if the marks point to different buffers.

Format

 MARK< *mark1 mark2*

Arguments

mark1

 An Editor mark

mark2

 Another Editor mark

Return Value

 T or NIL

MARK<= Function

Returns T if mark1 points to a character preceding mark2, or if they point to the same position; returns NIL otherwise. An error occurs if the marks point to different buffers.

Format

 MARK<= *mark1 mark2*

Arguments

mark1

 An Editor mark

mark2

    Another Editor mark

Return Value

    T or NIL

MARK= Function

Returns T if mark1 and mark2 point to the same position; returns NIL otherwise. The marks can point into different buffers.

Format

    MARK= *mark1 mark2*

Arguments

mark1

    An Editor mark

mark2

    Another Editor mark

Return Value

    T or NIL

MARK> Function

Returns T if mark1 points to a character following mark2; returns NIL otherwise. An error occurs if the marks point to different buffers.

Format

    MARK> *mark1 mark2*

Arguments

mark1

    An Editor mark

mark2

> Another Editor mark

Return Value

> T or NIL

MARK> = Function

Returns T if mark1 points to a character following mark2, or if they point to the same position; returns NIL otherwise. An error occurs if the marks point to different buffers.

Format

> MARK>= *mark1 mark2*

Arguments

mark1

> An Editor mark

mark2

> Another Editor mark

Return Value

> T or NIL

MAYBE RESET SELECT AT POINTER Command

Removes a previously set select mark, and thus a select region, if the current buffer point, the buffer select mark, and the pointer all indicate the same text position. If any of these conditions is not met, this command takes no action.

Display Name Format

> Maybe Reset Select at Pointer

Function Format

> MAYBE-RESET-SELECT-AT-POINTER-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> Undefined

MINOR STYLE ACTIVATION HOOK Editor Variable

Specifies a hook function that is called whenever a minor style is activated in a buffer. The function is called with two arguments, the style and the buffer.

Display Name Format

> Minor Style Activation Hook

Symbol Format

> MINOR-STYLE-ACTIVATION-HOOK

MOVE-MARK Function

Changes mark1 so that it points to the same position as mark2. The marks do not have to point into the same buffer or disembodied region.

Format

> MOVE-MARK *mark1* *mark2*

Arguments

mark1

> An Editor mark

mark2

> Another Editor mark

Return Value

> The updated mark1

MOVE-MARK-AFTER Function

Changes the specified mark so that it points to the character following its current position. If mark points to the last character in the buffer, it is not modified, and NIL is returned.

Format

    MOVE-MARK-AFTER *mark*

Arguments

mark

    An Editor mark

Return Value

    The modified mark or NIL

MOVE-MARK-BEFORE Function

Changes the mark so that it points to the character preceding its current position. If the mark points to the first character in the buffer, it is not modified, and NIL is returned.

Format

    MOVE-MARK-BEFORE *mark*

Arguments

mark

    An Editor mark

Return Value

    The modified mark, or NIL

MOVE-MARK-TO-POSITION Function

Changes the specified mark so that it points into the specified line at the character position indicated by the specified integer index.

Format

    MOVE-MARK-TO-POSITION *mark index* &OPTIONAL *line*

Arguments

mark

    An Editor mark

index

    A nonnegative fixnum less than or equal to the length of the line

line

    An Editor line that defaults to the line that the mark points into

Return Value

    The modified mark

MOVE POINT AND SELECT REGION Command

Moves the current buffer point to the position indicated by the pointer. In addition, if the previous command executed was in the category :MOVE-TO-POINTER and there was no select region, this command sets a buffer select mark and establishes a select region before moving the buffer point.

Display Name Format

    Move Point and Select Region

Function Format

    MOVE-POINT-AND-SELECT-REGION-COMMAND *prefix*

Arguments

prefix

    Ignored

Return Value

    Undefined

MOVE POINT TO POINTER Command

Moves the current buffer point to the position indicated by the pointer. If the pointer is beyond the end of a line, the buffer point is moved to the end of that line. If the pointer is beyond the end of the buffer region, the buffer point is moved to the end of the buffer region.

Category

> :MOVE-TO-POINTER

Display Name Format

> Move Point to Pointer

Function Format

> MOVE-POINT-TO-POINTER-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> The modified buffer point

MOVE TO LISP COMMENT Command

Moves the cursor to the comment part of the current line. If there is a comment on the current line, the cursor is moved to the comment delimiter. If there is no comment delimiter on the current line, blanks are inserted between the end of any executable LISP code on the line and the LISP comment column, a comment delimiter and a space are inserted at the LISP comment column, and the cursor is moved to the end of the line. If the length of executable code in the line does not allow for a clear separation of the executable code from the comment, a number of spaces are inserted before the comment delimiter.

This command makes use of the "LISP Comment Column" Editor variable and should only be used if that variable is bound in the current context.

Display Name Format

> Move to LISP Comment

Function Format

    MOVE-TO-LISP-COMMENT-COMMAND *prefix*

Arguments

prefix

    Ignored

Return Value

    The new buffer point

MOVE-WINDOW Function

Moves the displayed position of a window so that the upper left corner of the text area is at the specified row and column. If the window is visible, the display is altered immediately. If the window is not currently visible, it appears at the specified position when it is next shown (unless it is an anchored window being treated automatically).

Format

    MOVE-WINDOW *window row column*

Arguments

window

    An Editor window

row

    The row where the text display of the window should appear. The top row of the screen is 1.

column

    The column where the text display of the window should appear. The left-hand column of the screen is 1.

Return Value

    The window

NEW LINE Command

Breaks a line at the current buffer point. The resulting position of the buffer point is the beginning of the new line. If you specify a prefix argument, *n*, the command creates *n* lines.

Display Name Format

New Line Command

Function Format

NEW-LINE-COMMAND *prefix*

Arguments

prefix

A positive fixnum or NIL

Return Value

The updated buffer point mark

NEW LISP LINE Command

Creates a new line beginning at a column appropriate to the current indentation for LISP code.   The point is moved to the position following the new line and indentation.

Display Name Format

New LISP Line

Function Format

NEW-LISP-LINE-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The new buffer point

NEXT-CHARACTER Function

Returns the character immediately following the position of the mark. If there is no following character, the function returns NIL. You can use this function with the SETF macro to change the character following the mark.

Format

    NEXT-CHARACTER *mark*

Arguments

mark

    An Editor mark

Return Value

    A character, or NIL if there is no following character

NEXT FORM Command

Moves the current buffer point forward by the number of forms specified with the prefix argument, within the current parenthesis nesting level. The current buffer point is moved to the location immediately following the specified number of forms, and the new buffer point is returned. If a negative prefix argument is specified, the current buffer point is moved backward past the specified number of forms.

If the end of the current buffer or an outermost form is found before the end of the specified number of forms is reached, the Editor displays a message and returns NIL, and the point is not moved. If there are fewer forms at the current nesting level than the number specified by the prefix argument, the point is placed immediately before the list terminator character of the innermost list that encloses the point, and NIL is returned.

Display Name Format

    Next Form

Function Format

    NEXT-FORM-COMMAND *prefix*

Arguments

prefix

Integer or NIL

Return Value

The new buffer point or NIL

NOTE

Do not try to execute the "Next Form" command when the buffer point is located within a string or a multiple escape sequence. The results of a "Next Form" command in these circumstances can be incorrect.

NEXT LINE Command

Moves the point down one line. The relative horizontal character position (not the displayed position) of the point in the old line is maintained unless the end of the new line is to the left of that position. In such a case, the point will be at the end of the new line. If you specify an integer prefix argument, the point is moved down the number of lines indicated (or up, if the prefix is negative).

Category

:LINE-MOTION

Display Name Format

Next Line

Function Format

NEXT-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The new buffer point

NEXT-LISP-FORM Function

Moves the mark supplied as an argument to a point immediately following the end of the next form at the parenthesis nesting level of the mark. The updated mark is returned. If the mark is located within a symbol, it is moved to the end of the symbol. If an outermost form is found before the end of the next form, the function returns :OUTERMOST-FORM and does not move the mark. If no objects are found at the parenthesis level of the mark, the function moves the mark to a point immediately before the end of the innermost enclosing list and returns :END-OF-LIST. If the end of the buffer is found before the end of the next form, the function returns :END-OF-BUFFER and does not move the mark.

Format

    NEXT-LISP-FORM *mark*

Arguments

mark

    An Editor mark

Return Value

    The updated mark; or :END-OF-LIST, :OUTERMOST-FORM, or :END-OF-BUFFER

NEXT PARAGRAPH Command

Moves the mark to the beginning of the next paragraph. A paragraph is delimited by a whitespace line (see WHITESPACE-LINE-P function). The mark defaults to the current buffer point. If a prefix argument is supplied, the command moves the mark forward that many paragraphs.

Display Name Format

    Next Paragraph

Function Format

    NEXT-PARAGRAPH-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

    An integer or NIL

mark

An Editor mark that defaults to the current buffer point.

Return Value

The updated mark

NEXT SCREEN Command

Scrolls the window down a distance equal to the height of the window if the prefix argument is NIL. If you specify an integer prefix argument, the window is scrolled down the number of lines indicated (or up, if prefix is negative).

Display Name Format

Next Screen

Function Format

NEXT-SCREEN-COMMAND *prefix* &OPTIONAL *window*

Arguments

prefix

An integer or NIL

window

An Editor window that defaults to the current window

Return Value

The new buffer point

NEXT WINDOW Command

Moves the cursor from the current window to the window below it; that is, the current window is redefined. The cursor is then located at the window point of the new current window. If you specify an integer prefix argument, the command is executed the number of times indicated. The command circulates through all displayed windows regardless of window type.

Display Name Format

> Next Window

Function Format

> NEXT-WINDOW-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> The new current window

NEXT-WINDOW Function

Returns a window that is the "next" displayed window in sequence from the current window. If the window type argument is T, this function selects the next window that has the type of the current window. If the function is at the end of the list of that type, it switches to the opposite type of window and continues through that list. If the window type argument is either :FLOATING or :ANCHORED, the selection of the next window is made from only that type of window. The function returns NIL if there is not a window of the appropriate type currently displayed.

The optional *count* argument tells the function how many times to look for a next window. The argument can be positive or negative. A zero argument returns the current window. Repeatedly setting the current window to the next window with a window type of T results in circulation through all displayed windows.

Format

> NEXT-WINDOW &OPTIONAL *window-type count*

Arguments

window-type

> The type of the next window desired. One of :FLOATING, :ANCHORED, or T. The default is T.

count

> An integer specifying the number of windows to advance. the
> default is one.

Return Value

> The next window; or NIL, if there are no windows of the specified
> type

NONCURRENT WINDOW POINTER PATTERN Editor Variable

Specifies a 16x16 bitmap that determines the pointer cursor pattern
when the pointer is in an Editor window other than the current window.
When set to NIL, the pointer cursor pattern is the VAXstation default
(an arrow). See the functions SET-POINTER-PATTERN and MAKE-BITMAP in
the *VAX LISP/VMS Graphics Programming Guide*.

Display Name Format

> Noncurrent Window Pointer Pattern

Symbol Format

> NONCURRENT-WINDOW-POINTER-PATTERN

NONCURRENT WINDOW POINTER PATTERN X Editor Variable

Specifies the horizontal coordinate of the active pixel of the bitmap
specified by the Editor variable "Noncurrent Window Pointer Pattern".
The possible values are an integer in the range 0-15 or NIL. See the
function SET-POINTER-PATTERN in the *VAX LISP/VMS Graphics Programming
Guide*.

Display Name Format

> Noncurrent Window Pointer Pattern X

Symbol Format

> NONCURRENT-WINDOW-POINTER-PATTERN-X

NONCURRENT WINDOW POINTER PATTERN Y Editor Variable

Specifies the vertical coordinate of the active pixel of the bitmap specified by the Editor variable "Noncurrent Window Pointer Pattern", relative to the lower edge of the bitmap. The possible values are an integer in the range 0-15 or NIL. See the function SET-POINTER-PATTERN in the *VAX LISP/VMS Graphics Programming Guide*.

Display Name Format

Noncurrent Window Pointer Pattern Y

Symbol Format

NONCURRENT-WINDOW-POINTER-PATTERN-Y

OPEN LINE Command

Breaks a line at the current buffer point. The resulting point position is the end of the old line.

Display Name Format

Open Line

Function Format

OPEN-LINE-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The new buffer point

PAGE DELIMITER Attribute

Has a value of 1 for characters that separate pages, and 0 for all other characters.

Display Name Format

Page Delimiter

Symbol Format

    PAGE-DELIMITER

PAGE NEXT WINDOW Command

Scrolls the next window forward the number of lines indicated by the prefix argument or (without a prefix argument) scrolls the window forward to the next page.

Display Name Format

    Page Next Window

Function Format

    PAGE-NEXT-WINDOW-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The window point of the next window

PAGE-OFFSET Function

Updates the specified mark so that it points to the position of the next page break character (a character having a "Page Delimiter" attribute value of 1). An optional count argument lets you specify the number of page breaks to be located forward in the buffer if count is positive, and backward in the buffer if count is negative.

Format

    PAGE-OFFSET *mark* &OPTIONAL *count*

Arguments

mark

    The mark that is to be updated

count

The number of page breaks that are to be located

Return Value

The updated mark

PAGE PREVIOUS WINDOW Command

Scrolls the previous window forward to the next page. If an integer prefix argument is supplied, it scrolls the window by that many rows.

Display Name Format

Page Previous Window

Function Format

PAGE-PREVIOUS-WINDOW-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

Undefined

PAUSE EDITOR Command

Returns control from the Editor to LISP at the point at which the Editor was called. The current Editor state is saved, and the Editor restarts in that state the next time you call the ED function. Any changes to values or functions of symbols while the control is with the Editor are not reflected in LISP unless buffers have been explicitly evaluated.

Display Name Format

Pause Editor

Function Format

PAUSE-EDITOR-COMMAND *prefix*

Arguments

prefix

   Ignored

Return Value

   None

POINTER-STATE-ACTION Function

Takes a pointer-state object and returns the pointer-action information that is contained in the object, or NIL if there is none. The pointer-action information can be:

- :MOVEMENT if the action was to move the pointer cursor

- A button constant if the action was to depress or release a button on the pointing device. (See BIND-POINTER-COMMAND for information on button constants.) In this case POINTER-STATE-ACTION also returns a second value: T if the action was to depress the button, or NIL if the action was to release the button.

The pointer-action value(s) in a pointer-state object define the pointer action, if any, that invoked a command which in turn called GET-POINTER-STATE. See GET-POINTER-STATE for further information.

Format

   POINTER-STATE-ACTION *pointer-state*

Arguments

pointer-state

   A pointer-state object (as returned by GET-POINTER-STATE)

Return Value

Multiple values:

1. The keyword :MOVEMENT or a button constant

2. If a button constant is returned, POINTER-STATE-ACTION also returns T or NIL.

POINTER-STATE-BUTTONS Function

Takes a pointer-state object and returns the button-state information that is contained in the object. The button-state information indicates, for each button on the supported pointing device, whether the button was up or down. See GET-POINTER-STATE for information on the time at which the button state is captured in a pointer-state object. If a button was in transition (being depressed or released) at the point in time for which the pointer state is stored, the button-state information is the state of the buttons at the end of the transition.

Format

> POINTER-STATE-BUTTONS *pointer-state*

Arguments

pointer-state

> A pointer-state object (as returned by GET-POINTER-STATE)

Return Value

> A fixnum representing the state of the buttons. See the description of UIS:GET-BUTTONS in the *VAX LISP/VMS Graphics Programming Guide* for more information.

POINTER-STATE-P Function

Takes a LISP object and returns T if that object is a pointer-state object, or NIL if it is not. See GET-POINTER-STATE for information on pointer-state objects.

Format

> POINTER-STATE-P *object*

Arguments

object

> Any LISP object

Return Value

> T or NIL

POINTER-STATE-TEXT-POSITION Function

Takes a pointer-state object and returns the line and the character position that are contained in the object. These values define the text position indicated by the pointer cursor at the time the pointer-state object was created. See GET-POINTER-STATE for information on the time at which the pointer state is captured in a pointer-state object.

Format

    POINTER-STATE-TEXT-POSITION *pointer-state*

Arguments

pointer-state

    A pointer-state object (as returned by GET-POINTER-STATE)

Return Value

Two values:

1. The line indicated by the pointer cursor, or NIL if the pointer cursor was not indicating a line

2. The character position indicated by the pointer cursor, or NIL if the pointer cursor was not indicating a character position

POINTER-STATE-WINDOW-POSITION Function

Takes a pointer-state object and returns an Editor window, along with integers that are the x and y coordinates of a display position in that window. These values define the window position indicated by the pointer cursor at the time the pointer-state object was created. See GET-POINTER-STATE for information on the time at which the pointer state is captured in a pointer-state object.

Format

    POINTER-STATE-WINDOW-POSITION *pointer-state*

Arguments

pointer-state

    A pointer-state object (as returned by GET-POINTER-STATE)

Return Value

Three values if the pointer cursor was indicating an Editor window at the time the pointer-state object was created:

1. The Editor window indicated by the pointer cursor

2. An integer that is the window column position indicated by the pointer cursor

3. An integer that is the window row position indicated by the pointer cursor

If the pointer cursor was not indicating an Editor window at the time the pointer-state object was created, POINTER-STATE-WINDOW-POSITION returns NIL.

POSITION-WINDOW-TO-MARK Function

Repositions the specified window within its associated buffer to the line that contains the specified mark. This line becomes the first line displayed in the window. The mark's character position is ignored.

The window's screen position is not affected. The window point of the window remains at the same text position if possible; otherwise it moves to a position within the window (usually the center).

This function replaces the operation, in previous releases of the Editor, of repositioning a window by moving its window display start mark.

Format

> POSITION-WINDOW-TO-MARK *window mark*

Arguments

window

> An Editor window

mark .

> An Editor mark

Return Value

> The window point of the window

PREFIX-ARGUMENT Function

Returns the current value of the prefix argument. You can set a new value for the prefix argument by using the SETF macro with this function. The new value can be either NIL or a fixnum. Setting the value causes that value to be passed as the prefix argument to the next command executed.

Format

    PREFIX-ARGUMENT

Arguments

    None

Return Value

    A fixnum or NIL

PREVIOUS-CHARACTER Function

Returns the character immediately preceding the position of the mark. If there is no previous character, the function returns NIL. This function can be used with the SETF macro to change the character preceding the mark.

Format

    PREVIOUS-CHARACTER mark

Arguments

mark

    An Editor mark

Return Value

    A character or NIL

PREVIOUS-COMMAND-FUNCTION Variable

Is bound to the last Editor command function invoked.

PREVIOUS FORM Command

Moves the current buffer point backward by the number of forms specified with the prefix argument, within the current parenthesis nesting level. The current buffer point is moved to the location immediately before the specified number of forms, and the new buffer point is returned. If a negative prefix argument is specified, the current buffer point is moved forward past the specified number of forms.

If the beginning of the current buffer or an outermost form is found before the beginning of the specified number of forms is reached, the Editor displays a message and returns NIL, and the point is not moved. If there are fewer forms at the current nesting level than the number specified by the prefix argument, the point is placed immediately before the list initiator character of the innermost list that encloses the point, and NIL is returned.

Display Name Format

Previous Form

Function Format

PREVIOUS-FORM-COMMAND *prefix*

Arguments

prefix

Integer or NIL

Return Value

The new buffer point or NIL

NOTE

Do not try to execute the "Previous Form" command when the buffer point is located within a string or a multiple escape sequence. The results of a "Previous Form" command in these circumstances are incorrect.

Also, when using unmatched multiple escape characters or unmatched string delimiter characters in a comment, you should include a backslash (\) before these characters. Otherwise, the "Previous Form" command may fail, because the comment delimiter will be interpreted as part of a string or multiple escape sequence.

PREVIOUS LINE Command

Moves the point of the current buffer to the previous line. The relative horizontal character position (not the displayed position) of the point in the old line is maintained unless the end of the new line is to the left of that position. In such a case, the point will be at the end of the new line.

If you specify an integer prefix argument, the point is moved up the number of times indicated (or down, if prefix is negative). If there is no previous line, the point is moved to the beginning of the first line.

Category

:LINE-MOTION

Display Name Format

Previous Line

Function Format

PREVIOUS-LINE-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The new buffer point

PREVIOUS-LISP-FORM Function

Moves the mark supplied as an argument to a point immediately preceding the beginning of the previous form at the parenthesis nesting level of the mark. The updated mark is returned. If the mark is located within a symbol, it is moved to the beginning of the symbol. If an outermost form is found before the beginning of the previous form, the function returns :OUTERMOST-FORM and does not move the mark. If no forms are found at the parenthesis level of the mark, the function moves the mark to the beginning of the innermost enclosing list and returns :BEGINNING-OF-LIST. If the beginning of the buffer is found before the beginning of the previous form, the function returns :BEGINNING-OF-BUFFER and does not move the mark. If the function detects an error due to an unmatched string delimiter or multiple escape character in a comment, the function returns :FAILURE and does not move the mark.

Format

    PREVIOUS-LISP-FORM *mark*

Arguments

mark

    An Editor mark

Return Value

    The updated mark; or :BEGINNING-OF-LIST, :OUTERMOST-FORM,
    :FAILURE, or :BEGINNING-OF-BUFFER

PREVIOUS PARAGRAPH Command

Moves the mark to the end of the previous paragraph. A paragraph is
delimited by a whitespace line (see WHITESPACE-LINE-P function). The
mark defaults to the current buffer point. If a prefix argument is
supplied, the command moves the mark backward that many paragraphs.

Display Name Format

    Previous Paragraph

Function Format

    PREVIOUS-PARAGRAPH-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

    An integer or NIL

mark

    An Editor mark that defaults to the current buffer point

Return Value

    The updated mark

PREVIOUS SCREEN Command

Scrolls the specified window (or the current window, if none is specified) up a distance equal to the height of the window. If you specify an integer prefix argument, the window is scrolled up the number of lines indicated (or down, if the prefix is negative).

Display Name Format

Previous Screen

Function Format

PREVIOUS-SCREEN-COMMAND *prefix* &OPTIONAL *window*

Arguments

prefix

An integer or NIL

window

An Editor window that defaults to the current window

Return Value

The new buffer point

PREVIOUS WINDOW Command

Moves the cursor from the current window to the window above it; that is, the current window is redefined. The cursor is then located at the window point of the new current window. If you specify an integer prefix argument, the command is executed the number of times indicated. The command circulates through all displayed windows regardless of window type.

Display Name Format

Previous Window

Function Format

PREVIOUS-WINDOW-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The new current window

PRINT REPRESENTATION Attribute

Determines how a character is displayed on the screen. If the value of this attribute is NIL, the character is given no special treatment. If the value is a string, the string is displayed as the character representation. If it is a vector, the current column is used as an index into the vector to obtain a string to display. Using a vector is useful for displaying characters whose print representation is column dependent (such as tabs).

If the value is a function, then that function is called with two arguments -- the current column and the character -- to obtain a string.

The Print Representation attribute cannot be bound in any context other than :GLOBAL.

Display Name Format

    Print Representation

Symbol Format

    PRINT-REPRESENTATION

PROMPT ALTERNATIVES Editor Variable

Is bound to the alternatives argument for the general prompt that is currently in progress.

Display Name Format

    Prompt Alternatives

Symbol Format

    PROMPT-ALTERNATIVES

PROMPT ALTERNATIVES ARGUMENTS Editor Variable

Is bound to the alternatives arguments for the general prompt that is currently in progress.

Display Name Format

Prompt Alternatives Arguments

Symbol Format

PROMPT-ALTERNATIVES-ARGUMENTS

PROMPT COMPLETE STRING Command

Is used by the PROMPT-FOR-INPUT function to complete user input to a prompt. The command uses the information supplied by the :COMPLETION and :COMPLETION-ARGUMENTS arguments of the PROMPT-FOR-INPUT function.

NOTE

This command is an integral part of the PROMPT-FOR-INPUT function and should not be used in any context other than that of the "General Prompting" buffer. It can be rebound in that context to any desired key sequence.

Display Name Format

Prompt Complete String

Function Format

PROMPT-COMPLETE-STRING-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

None

PROMPT COMPLETION Editor Variable

Is bound to the completion argument for the general prompt that is currently in progress.

Display Name Format

Prompt Completion

Symbol Format

PROMPT-COMPLETION

PROMPT COMPLETION ARGUMENTS Editor Variable

Is bound to the list of completion function arguments for the general prompt that is currently in progress.

Display Name Format

Prompt Completion Arguments

Symbol Format

PROMPT-COMPLETION-ARGUMENTS

PROMPT DEFAULT Editor Variable

Is bound to the default value for the general prompt that is currently in progress.

Display Name Format

Prompt Default

Symbol Format

PROMPT-DEFAULT

PROMPT ERROR MESSAGE Editor Variable

Is bound to the error message argument of the general prompt currently in progress.

Display Name Format

Prompt Error Message

Symbol Format

PROMPT-ERROR-MESSAGE

PROMPT ERROR MESSAGE ARGUMENTS Editor Variable

Is bound to the error message arguments for the general prompt that is
currently in progress.

Display Name Format

Prompt Error Message Arguments

Symbol Format

PROMPT-ERROR-MESSAGE-ARGUMENTS

PROMPT-FOR-INPUT Function

Prompts for input, invokes the *validation* function with the user's
input string as the argument, and returns the return value of the
validation function. If the user enters no input (a null string),
PROMPT-FOR-INPUT can either return a default value or prompt again for
input. If the user's input is invalid, PROMPT-FOR-INPUT signals an
error and awaits further input.

You can specify a prompting message and a value to be returned if the
user enters no input. You can also provide alternatives, completion,
and help to the user during the prompt.

Format

```
PROMPT-FOR-INPUT validation &KEY :PROMPT
                               :REQUIRED
                               :DEFAULT :DEFAULT-MESSAGE
                               :ALTERNATIVES
                               :ALTERNATIVES-ARGUMENTS
                               :COMPLETION :COMPLETION-ARGUMENTS
                               :HELP :HELP-ARGUMENTS
                               :ERROR-MESSAGE
                               :ERROR-MESSAGE-ARGUMENTS
                               :SAVE-WINDOW-STATE
```

Arguments

validation

> A function of one argument. This function operates on the user's input string and returns the value that will be returned by PROMPT-FOR-INPUT. An example of a validation function might be FIND-BUFFER, which returns the buffer specified by the string or NIL if there is no buffer with that display name.

> If the validation function returns NIL, the user's input is not valid. In this case, PROMPT-FOR-INPUT signals an error and awaits further input. NIL can be a valid value if the validation function returns multiple values of NIL and T.

:PROMPT

> A string or a function that returns a string. This argument specifies the prompting message. The default is "Enter input ".

:REQUIRED

> T or NIL. This argument specifies the action to be taken if the user enters no input (a null string) in response to the prompt. If T, PROMPT-FOR-INPUT prints "Input required" in the information area and awaits further input. If NIL (the default), PROMPT-FOR-INPUT returns the value of the :DEFAULT argument.

:DEFAULT

> This argument specifies the value to be returned by PROMPT-FOR-INPUT if the user enters no input and if the value of :REQUIRED is NIL. The default is NIL.

:DEFAULT-MESSAGE

> NIL, T, a string, or a function of one argument that returns a string. This argument specifies a message to be displayed in the information area at the start of the prompt. Its purpose is to inform the user of a default return value.

> If the argument is NIL (the default), no message is displayed. If T, the value of :DEFAULT is printed. If a string, the string is used as the *control-string* argument in a call to FORMAT, and the result is printed. The value of :DEFAULT is used as the *data* argument to FORMAT. If a function, it is passed the value of :DEFAULT and the string that the function returns is printed.

:ALTERNATIVES

> A string, a string table, or a function of at least one argument to be called if the user requests input alternatives. If the

argument is a function, it is passed the string the user has typed so far and any additional arguments supplied as :ALTERNATIVES-ARGUMENTS. The default is the string "No alternatives available."

:ALTERNATIVES-ARGUMENTS

A list of arguments for the :ALTERNATIVES function. The default is NIL.

:COMPLETION

NIL, a string, a string table, or a function of at least one argument to be called if the user requests input completion. If the argument is a function, it is passed the string the user has typed so far and any additional arguments supplied as :COMPLETION-ARGUMENTS. If the argument is NIL (the default) and the user requests input completion, an Editor error is signaled.

:COMPLETION-ARGUMENTS

A list of arguments for the :COMPLETION function. The default is NIL.

:HELP

NIL, a string, or a function to be called if the user requests help. The default is "No help available." If the value is a string, it is displayed in the information area; if the string contains more lines than will fit in the information area, it is displayed in the "Help" buffer. If the argument is a function, it is called with any arguments supplied as :HELP-ARGUMENTS.

:HELP-ARGUMENTS

A list of arguments for the :HELP function. The default is NIL.

:ERROR-MESSAGE

A string or a function that returns a string. This argument specifies the error message to be displayed if the user's input is invalid. If the argument is a function, it is called with any arguments supplied as :ERROR-MESSAGE-ARGUMENTS.

:ERROR-MESSAGE-ARGUMENTS

A list of arguments for the :ERROR-MESSAGE function. The default is NIL.

:SAVE-WINDOW-STATE

> NIL or non-NIL. Non-NIL specifies that the "General Prompting" buffer remains the current buffer when the prompt is completed. (This is helpful when writing commands that prompt for more than one value.) NIL (the default) specifies that the buffer that was current when the prompt was initiated is to become current again when the prompt is completed.

Return Value

> The value returned by the validation function or the :DEFAULT value

PROMPT HELP Command

Is used by the PROMPT-FOR-INPUT function to display help when the user is being prompted. The help information is taken from the :HELP and :HELP-ARGUMENTS arguments of PROMPT-FOR-INPUT.

NOTE

> This command is an integral part of the PROMPT-FOR-INPUT function and should not be used in any context other than that of the "General Prompting" buffer. It can be rebound in that context to any desired key sequence.

Display Name Format

> Prompt Help

Function Format

> PROMPT-HELP-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> None

PROMPT HELP Editor Variable

Is bound to the help argument for the general prompt that is currently in progress.

Display Name Format

    Prompt Help

Symbol Format

    PROMPT-HELP

PROMPT HELP ARGUMENTS Editor Variable

Is bound to the help function arguments for the general prompt that is currently in progress.

Display Name Format

    Prompt Help Arguments

Symbol Format

    PROMPT-HELP-ARGUMENTS

PROMPT HELP CALLED Editor Variable

Specifies whether or not a help function has been called during the general prompt that is currently in progress. If the value of this variable is non-NIL at the completion of a prompt, the displayed help window is removed from the display.

Display Name Format

    Prompt Help Called

Symbol Format

    PROMPT-HELP-CALLED

PROMPT READ AND VALIDATE Command

Is used by the PROMPT-FOR-INPUT function to obtain and validate the current user response to a prompt. The validation function is taken from the validation function argument of the PROMPT-FOR-INPUT function. If the validation function succeeds, the value is returned by the PROMPT-FOR-INPUT function. Otherwise, this command signals an Editor error and waits for the user to correct the problem.

NOTE

This command is an integral part of the PROMPT-FOR-INPUT function and should not be used in any context other than that of the "General Prompting" buffer. It can be rebound in that context to any desired key sequence.

Display Name Format

Prompt Read and Validate

Function Format

PROMPT-READ-AND-VALIDATE-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

The return value of the validation function

PROMPT RENDITION COMPLEMENT Editor Variable

Set to a keyword or a list of keywords that specifies the video rendition of prompting messages. The rendition specified is relative to the terminal rendition setting. The keywords are :NORMAL, :REVERSE, :BOLD, :UNDERLINE, and :BLINK. The default is :UNDERLINE.

Display Name Format

Prompt Rendition Complement

Symbol Format

    PROMPT-RENDITION-COMPLEMENT

PROMPT RENDITION SET Editor Variable

Set to a keyword or a list of keywords that specifies the video rendition of prompting messages. The rendition specified is absolute, rather than relative to the terminal rendition setting. The keywords are :NORMAL, :REVERSE, :BOLD, :UNDERLINE, and :BLINK. The default is :NORMAL.

Display Name Format

    Prompt Rendition Set

Symbol Format

    PROMPT-RENDITION-SET

PROMPT REQUIRED Editor Variable

Specifies whether an input value is required for the general prompt that is currently in progress.

Display Name Format

    Prompt Required

Symbol Name

    PROMPT-REQUIRED

PROMPT SCROLL HELP WINDOW Command

Scrolls the Help window while in another window. When the scrolling reaches the end of the "Help" buffer, the window is reset to the beginning of the Help buffer. The command is bound in the "General Prompting" buffer so that prompt help can be scrolled without having to leave the prompting window.

Display Name Format

    Prompt Scroll Help Window

Function Format

PROMPT-SCROLL-HELP-WINDOW-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

Undefined

PROMPT SHOW ALTERNATIVES Command

Is used by the PROMPT-FOR-INPUT function to supply the user with a list of alternatives based on current input. The information for this command is supplied with the :ALTERNATIVES and :ALTERNATIVES-ARGUMENTS arguments of the PROMPT-FOR-INPUT function.

NOTE

This command is an integral part of the PROMPT-FOR-INPUT function and should not be used in any context other than that of the "General Prompting" buffer. It can be rebound in that context to any desired key sequence.

Display Name Format

Prompt Show Alternatives

Function Format

PROMPT-SHOW-ALTERNATIVES-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

None

PROMPT START Editor Variable

Is bound to a right-inserting mark that points to the starting position of the user's input in the prompt buffer. This description applies to the general prompt that is currently in progress. The user's input is defined as the region between this mark and the buffer point of the "General Prompting" buffer.

Display Name Format

    Prompt Start

Symbol Format

    PROMPT-START

PROMPT VALIDATION Editor Variable

Is bound to the validation function for the general prompt that is currently in progress.

Display Name Format

    Prompt Validation

Symbol Format

    PROMPT-VALIDATION

PUSH-WINDOW Function

Makes the specified window visible on the screen without removing any other windows. If the type of the window is :FLOATING, the function has the same effect as the SHOW-WINDOW function. If the window is :ANCHORED, the window is added to the list of currently visible anchored windows, and its height and those of the other anchored windows are adjusted so as to make them all about the same height. See also SHOW-WINDOW, which might remove another anchored window to make room for the new one.

The optional arguments allow some control over the relative vertical positioning of an anchored window. If the companion argument is supplied, it must be another visible anchored window. The new window is placed on the screen just below the companion window. If the optional insert-above argument is T, the new window is inserted on the screen just above the position of the companion window.

Format

PUSH-WINDOW *window* &OPTIONAL *companion insert-above*

Arguments

window

An Editor window to display

companion

A currently visible anchored window or NIL

insert-above

If NIL, the new window will be below the companion; if not NIL, it will appear above the companion.

Return Value

The window

QUERY SEARCH REPLACE Command

Prompts the user for a string to search for and a second string to replace occurrences of the first one. Completion is available during both prompts. The completion command inserts the string last searched for or the last replacement string, as appropriate. Once these strings are established, the command repeatedly searches for occurrences of the first string. At each one, the command stops and asks the user to enter one of several options about how to proceed. The options are:

space   Replace this occurrence and find the next one.

S or s   Replace this occurrence and stay here. The purpose of this is to let you examine the results of the change and perhaps decide to continue, quit, or do a recursive edit.

.   Replace this occurrence and then quit.

!   Replace all the remaining occurrences without asking. At the end the Editor will put out a message telling how many occurrences were replaced.

N or n   Do not replace this occurrence but do find the next one.

CTRL/C (or the current cancel character)
        Do not replace this occurrence and do quit.

Q or q Do not replace this occurrence and do quit, returning to the point at which the search began.

R or r Enter a recursive edit. Exit the recursion with CTRL/C (or the current cancel character). A recursive edit is designed to let you do any editing you need to do and then return to your original place in the search/replace cycle.

?   Display an abbreviated version of this text.

Category

:GENERAL-PROMPTING

Display Name Format

Query Search Replace

Function Format

QUERY-SEARCH-REPLACE-COMMAND *prefix* &OPTIONAL *search-string replace-string*

Arguments

prefix

Ignored

search-string

The string to be replaced. If this argument is not supplied, the user is prompted for a string.

replace-string

The string to replace the search-string with. If this argument is not supplied, the user is prompted for a string.

Return Value

None

QUOTED INSERT Command

Causes the next character typed to be inserted in the current buffer without interpretation by the Editor. If you specify an integer prefix argument, the character is inserted the number of times indicated.

Display Name Format

Quoted Insert

Function Format

QUOTED-INSERT-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The character or string inserted

READ FILE Command

Replaces the current buffer contents by reading in a file. If a file is not specified, the command prompts for a file name.

Display Name Format

Read File

Function Format

READ-FILE-COMMAND *prefix* &OPTIONAL *pathname*

Arguments

prefix

Ignored

pathname

The pathname specifier or NIL

Return Value

The current buffer point

REDISPLAY SCREEN Command

Erases and redisplays everything on the screen.

Display Name Format

> Redisplay Screen

Function Format

> REDISPLAY-SCREEN-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> None

REDISPLAY-SCREEN Function

Erases and redisplays the entire screen. This function is used when the terminal display has been altered by broadcast or garbage collection messages.

Format

> REDISPLAY-SCREEN

Arguments

> None

Return Value

> None

REGION-END Function

Returns a mark that points to the end of the region. Altering the position of the end of a buffer region can lead to unpredictable results.

Format

> REGION-END *region*

Arguments

region

> An Editor region

Return Value

> The ending mark of the region

REGION-READ-POINT Function

Returns a mark that specifies the next character to be read from an Editor region input stream. (See description of MAKE-EDITOR-STREAM-FROM-REGION function.) The mark is a new mark unless the optional *mark* argument is supplied; if a mark is specified, that mark is destructively modified to point to the next character to be read from the stream. Altering the returned mark does not affect the operation of the stream in any way.

Format

> REGION-READ-POINT *stream* &OPTIONAL *mark*

Arguments

stream

> An Editor region input stream

mark

> An Editor mark

Return Value

> An Editor mark

REGION-START Function

Returns a mark that points to the beginning of the specified region. Altering the position of the beginning of a buffer region can lead to unpredictable results.

Format

    REGION-START *region*

Arguments

region

    An Editor region

Return Value

    The starting mark of the region

REGION-TO-STRING Function

Returns a string that contains the characters in the region. Line breaks in the region are interpreted as newline characters.

Format

    REGION-TO-STRING *region*

Arguments

region

    An Editor region

Return Value

    A simple string

REGIONP Function

Returns T if the argument is an Editor region, or NIL if it is not.

Format

    REGIONP *object*

Arguments

object

    Any LISP object

Return Value

T or NIL

REMOVE CURRENT WINDOW Command

Removes the current window from the screen. The window is not deleted, but is no longer visible. The new current window will be one chosen according to the rules for the NEXT-WINDOW function. If there are no other windows visible, the Editor returns to its initial state. See REMOVE-WINDOW and *EDITOR-DEFAULT-BUFFER*.

Display Name Format

Remove Current Window

Function Format

REMOVE-CURRENT-WINDOW-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

T

REMOVE-HIGHLIGHT-REGION Function

Alters destructively the specified highlight region object so that it no longer affects the video display characteristics of the text contained in the region. The text in the region is not affected by this operation. The highlight region object, however, is destroyed and cannot be reused.

Format

REMOVE-HIGHLIGHT-REGION *region*

Arguments

region

An Editor highlight region

Return Value

    T

REMOVE OTHER WINDOWS Command

Removes all windows but the current window. The appropriate hook functions are invoked. The windows are not deleted.

Display Name Format

    Remove Other Windows

Function Format

    REMOVE-OTHER-WINDOWS *prefix*

Arguments

prefix

    Ignored

Return Value

    T

REMOVE-STRING-TABLE-ENTRY Function

Removes an entry having the specified key from the specified string table. This function is also a predicate that returns T if there was an entry for the specified key, and NIL if there was not.

Format

    REMOVE-STRING-TABLE-ENTRY *key-string string-table*

Arguments

key-string

    The string that is the key of the entry to remove

string-table

    The string table from which to remove the entry

Return Value

T or NIL

REMOVE-WINDOW Function

Removes the specified window from the display area. This function does not delete the window.

If the window being removed is the current window, the *new-current* argument can be used to specify the window that is to become current. If no value is specified, the NEXT-WINDOW function is called to select a new current window. If there are no other windows visible, the screen is restored to an initial state. (See NOTE below.)

The *resize-remainder* parameter in earlier versions of the Editor is obsolete and any value supplied is ignored. If the window being removed is an anchored window, the sizes of other visible anchored windows are always adjusted to fill the available display area.

Format

REMOVE-WINDOW *window* &OPTIONAL *resize-remainder new-current*

Arguments

window

A visible Editor window

resize-remainder

Obsolete. Any value supplied is ignored.

new-current

An Editor window. It need not be currently visible. The default is a visible window selected by the NEXT-WINDOW function.

Return Value

T if the window was displayed and has been removed from the display, or NIL if the window was not displayed.

NOTE

The REMOVE-WINDOW function will not remove the window that is associated with the buffer specified by *EDITOR-DEFAULT-BUFFER*. This is the window that

245

appears when you call the Editor without specifying a string, pathname, symbol, or list, and it normally appears only when the Editor has no other window to display. Displaying any other window will cover this window. If the value of *EDITOR-DEFAULT-BUFFER* is NIL, a window to the buffer "Basic Introduction" is shown when the Editor has nothing else to display.

REPLACE-PATTERN Function

Replaces *n* occurrences of the text matched by the search-pattern with the string replacement. The search starts at the specified mark. If *n* is NIL, all occurrences of the search-pattern following the mark are replaced.

Format

REPLACE-PATTERN *mark search-pattern replacement* &OPTIONAL *n*

Arguments

mark

An Editor mark

search-pattern

An Editor search pattern previously computed with MAKE-SEARCH-PATTERN

replacement

A string that will replace the old pattern in the text

n

A fixnum or NIL

Return Value

The number of occurrences replaced

RETURN-FROM-EDITOR Macro

Causes the ED function to return the value or values returned by the *result* form. If ED has been called recursively (for instance, by a command within the Editor), RETURN-FROM-EDITOR returns a result from the innermost call to ED.

This macro is useful for returning results from a recursive call to the Editor, as is done in the function PROMPT-FOR-INPUT.

Format

RETURN-FROM-EDITOR &OPTIONAL *result*

Arguments

result

A form that defaults to NIL.

Return Value

Not applicable

REVERSE-INVOKE-HOOK Function

Calls all the hook functions in the specified hook variable and passes the specified arguments. The order of invocation of the hook functions is the same as the normal context searching order. See also INVOKE-HOOK.

Format

REVERSE-INVOKE-HOOK *hook-variable* &REST *args*

Arguments

hook-variable

An Editor variable specifier

args

Any additional arguments that may need to be passed to the hook functions

Return Value

Undefined

RING-LENGTH Function

Returns two integers. The first is the number of slots used in the ring; the second is the maximum number of slots in the ring.

Format

    RING-LENGTH *ring*

Arguments

ring

    An Editor ring

Return Value

Two values:

1. The number of slots used in the ring

2. The maximum number of slots in the ring

RING-POP Function

Deletes the object at the zero position of the ring and returns it. The ring delete-function is not called. This decreases the current length of the ring by 1.

Format

    RING-POP *ring*

Arguments

ring

    An Editor ring

Return Value

    The object at the current position of the ring

RING-PUSH Function

Pushes the object onto the ring, deleting the oldest element if the ring is full. The ring delete-function is called if an object is deleted. This function is called with two arguments -- the object being deleted and the ring.

Format

RING-PUSH *ring object*

Arguments

ring

An Editor ring

object

Any LISP object

Return Value

The object that was pushed

RING-REF Function

Returns an element of the specified ring as specified by an integer index. You can specify any integer. A negative number is the number of slots backward from the end. If the absolute value of the integer is greater than the size of the ring, the integer is taken modulo the size of the ring. This function can be used with the SETF macro to replace an element of a ring. When replacing an element, the ring delete-function is called with two arguments -- the entry being replaced and the ring.

Format

RING-REF *ring* &OPTIONAL *index*

Arguments

ring

An Editor ring

index

An integer specifying the element of the ring to be returned. The default is 0.

Return Value

Two values:

1.   The specified object in the ring

2.   The positive index number of the referenced ring slot modulo
     the length of the ring

RING-ROTATE Function

Rotates a ring forward if the offset is positive, or backward if the
offset is negative.   For example, with an offset of +1, the second
element would become the first; with an offset of -1, the last element
would become the first.

Format

   RING-ROTATE *ring offset*

Arguments

ring

   An Editor ring

offset

   An integer

Return Value

   The object at the new zero position in the ring

RINGP Function

Returns T if the argument is an Editor ring; otherwise, returns NIL.

Format

   RINGP *object*

Arguments

object

   Any LISP object

Return Value

    T or NIL

SAME-LINE-P Function

Returns T if mark1 and mark2 point into the same line; returns NIL
otherwise.

Format

    SAME-LINE-P *mark1 mark2*

Arguments

mark1

    An Editor mark

mark2

    Another Editor mark

Return Value

    T or NIL

SCREEN-HEIGHT Function

Returns the current available height of the display device (screen).
This number can be less than the height of the physical device. It is
the height used by the Editor as the maximum displayable height. This
value can be changed by using the SETF macro. The value returned by
SCREEN-HEIGHT can be less than the specified value if the physical
device cannot accommodate the specified new height. Any anchored
windows will be adjusted to fit the new height.

Format

    SCREEN-HEIGHT

Arguments

    None

Return Value

    The current screen height

SCREEN MODIFICATION HOOK Editor Variable

Is a hook variable that is called whenever the screen height or width is changed, after all screen and window modifications have been made.

Display Name Format

    Screen Modification Hook

Symbol Format

    SCREEN-MODIFICATION-HOOK

SCREEN-WIDTH Function

Returns the current available width of the display device (screen). This number can be less than the width of the physical device. It is the width used by the Editor as the maximum displayable width. This value can be changed by using the SETF macro. The value returned by SCREEN-WIDTH can be less than the specified value if the physical device cannot accommodate the specified new width. Any anchored windows will be adjusted to fit the new width.

Format

    SCREEN-WIDTH

Arguments

    None

Return Value

    The current screen width

SCROLL-WINDOW Function

Scrolls the specified window by a certain number of lines. If the count is positive, the window scrolls down through the text, making the lines appear to be moving upward on the screen. The window is scrolled up through the buffer if the count is negative. The buffer point stays at the same position whenever possible; otherwise, it is centered on the screen.

Format

    SCROLL-WINDOW *window count*

Arguments

window

    An Editor window

count

    An integer

Return Value

    The buffer point of the window

SCROLL WINDOW DOWN Command

Scrolls the indicated or current window down (moves the text up) the number of lines indicated by the prefix.

Display Name Format

    Scroll Window Down

Function Format

    SCROLL-WINDOW-DOWN-COMMAND *prefix* &OPTIONAL *window*

Arguments

prefix

    An integer or NIL

window

    An Editor window that defaults to the current window

Return Value

    The buffer point of the window

SCROLL WINDOW UP Command

Scrolls the indicated or current window up (moves the text down) the number of lines indicated by the prefix argument.

Display Name Format

Scroll Window Up

Function Format

SCROLL-WINDOW-UP-COMMAND *prefix* &OPTIONAL *window*

Arguments

prefix

An integer or NIL

window

An Editor window that defaults to the current window

Return Value

The buffer point of the window

SELECT BUFFER Command

Makes the specified buffer the current buffer. If the buffer is not specified, the function prompts for a buffer name. If the buffer does not exist, a new buffer is created with the name you enter in response to the prompt.

Category

:GENERAL-PROMPTING

Display Name Format

Select Buffer

Function Format

SELECT-BUFFER-COMMAND *prefix* &OPTIONAL *buffer*

Arguments

prefix

Ignored

buffer

An Editor buffer

Return Value

The new current buffer

SELECT ENCLOSING FORM AT POINTER Command

Creates a select region that encompasses the LISP form indicated by the pointer. If the pointer is indicating a symbol, the region contains the symbol; if the pointer is indicating a list initiator or a list terminator, the region contains the list. If the command is invoked repeatedly, the select region expands to include that number of forms enclosing the one indicated by the pointer, stopping when it reaches an outermost form.

Display Name Format

Select Enclosing Form at Pointer

Function Format

SELECT-ENCLOSING-FORM-AT-POINTER-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

Undefined

SELECT OUTERMOST FORM Command

Creates and returns a region containing the outermost list (a list with its opening parenthesis in the leftmost screen column) that encloses the buffer point of the current buffer. If there is no outermost list enclosing the buffer point, the command selects the outermost list following the point if there is one, and otherwise selects the preceding outermost list. The command moves the point to the left parenthesis of the appropriate list and creates a mark at the right parenthesis of the same list. The created mark is bound to the "Buffer Select Mark" Editor variable. The region is bound to the "Buffer Select Region" Editor variable.

Display Name Format

Select Outermost Form

Function Format

SELECT-OUTERMOST-FORM-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

An Editor region containing the form

SELECT REGION RENDITION COMPLEMENT Editor Variable

Set to a keyword or a list of keywords that specifies the video rendition of an Editor select region. The rendition specified is relative to the rendition of the window where the region is displayed. The keywords are :NORMAL, :BOLD, :BLINK, :REVERSE, and :UNDERLINE. The value is set to NIL in the global context, and to :REVERSE in "EDT Emulation" and "EMACS" styles.

The values correspond to the possible values of the *complement* argument to MAKE-HIGHLIGHT-REGION. See also "Select Region Rendition Set" and "Buffer Select Region".

Display Name Format

Select Region Rendition Complement

Symbol Format

SELECT-REGION-RENDITION-COMPLEMENT

SELECT REGION RENDITION SET Editor Variable

Set to a keyword or a list of keywords that specifies the video rendition of an Editor select region. The rendition specified is absolute, rather than relative to the rendition of the window where the region is displayed. The keywords are :NORMAL, :BOLD, :BLINK, :REVERSE, and :UNDERLINE. The value is set globally to NIL.

The values correspond to the possible values of the *set* argument to MAKE-HIGHLIGHT-REGION. See also "Select Region Rendition Complement" and "Buffer Select Region".

EDITOR OBJECT DESCRIPTIONS

Display Name Format

    Select Region Rendition Set

Symbol Format

    SELECT-REGION-RENDITION-SET

SELF INSERT Command

Causes the last character typed to be inserted in the current buffer as text. If the prefix is an integer, the character is inserted the number of times indicated. This command is useful only when bound to keyboard characters that are to be inserted as ordinary text. All graphic characters are self-inserting.

Display Name Format

    Self Insert

Function Format

    SELF-INSERT-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The character or string of repeated characters

SET SCREEN HEIGHT Command

Prompts for a height if no prefix argument is supplied. The command sets the height of the screen to the number of rows specified.

Display Name Format

    Set Screen Height

Function Format

    SET-SCREEN-HEIGHT-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The new screen height

SET SCREEN WIDTH Command

Prompts for a width if no prefix argument is supplied. The command sets the width of the screen to the number of columns specified.

Display Name Format

Set Screen Width

Function Format

SET-SCREEN-WIDTH-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

The new screen width

NOTE

If you set the screen width of a terminal that does not have the Advanced Video Option to greater than 80, the screen height is limited to 12 lines. Therefore, you must also set the height of the screen to 12.

SET SELECT MARK Command

Selects and highlights a region of text for other commands to operate upon. This command sets the value of the Editor variable "Buffer Select Mark" to a mark that indicates the same position as the current buffer point. It then makes a highlight region between the select mark and the buffer point and sets the value of the Editor variable

"Buffer Select Region" to that region. The next command you execute that requires a select region will use the current value of "Buffer Select Region".

You can control the video rendition of the select region with the Editor variables "Select Region Rendition Set" and "Select Region Rendition Complement".

Display Name Format

> Set Select Mark

or

> EDT Select

Function Format

> SET-SELECT-MARK-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> Undefined

SHOW-MARK Function

Highlights the position of the specified mark within the specified window for a certain length of time. Time is in units of seconds, and defaults to 0.5. The function terminates before the number of seconds specified in the time argument elapses if any input is typed on the terminal. If the mark's position is not visible on the terminal, SHOW-MARK returns NIL; otherwise, it returns T.

Format

> SHOW-MARK *mark window* &OPTIONAL *time*

Arguments

mark

> An Editor mark

window

    An Editor window

time

    A positive number indicating the number of seconds the mark will be highlighted. The default is 0.5.

Return Value

    T or NIL

SHOW TIME Command

Displays the current time and date in the information area.

Display Name Format

    Show Time

Function Format

    SHOW-TIME-COMMAND *prefix*

Arguments

prefix

    Ignored

Return Value

    Undefined

SHOW-WINDOW Function

Makes a window visible on the screen. The behavior of this function differs according to whether its argument is an anchored window or a floating window.

1. If the window is a floating window, it is placed at the screen row and column specified when the window was created unless this placement is overridden by an explicit specification of a *row* or *column* argument. This window will obscure any anchored or floating windows in its area. Its new row and column are remembered so .that the window will always return to that spot unless moved or reshown with different *row* or *column* arguments.

EDITOR OBJECT DESCRIPTIONS

2. If the window is a floating window that is already displayed but obscured by another floating window, this function places the specified window "on top of" the obscuring one(s).

3. If the window is an anchored window and it is already on the screen, no action occurs.

4. If the window is an anchored window and it is not on the screen, then any *row* or *column* argument is ignored and:

    a. If there is no other anchored window on the screen, the height of the window is set to the maximum allowable on the screen and it is made visible.

    b. If the number of anchored windows already on the screen is greater than zero but less than the value of the Editor variable "Anchored Window Show Limit", then the heights of the new and existing windows are adjusted so that all will have about equal space on the screen. The new window is made visible below the old.

    c. If the number of anchored windows already on the screen is greater than or equal to the value of the Editor variable "Anchored Window Show Limit", then the least recently used window is removed from the screen. The height of the new window is adjusted to fit the height of the window being removed, and the new window is made visible in the same position as the one being removed.

Format

   SHOW-WINDOW *window* &OPTIONAL *row column*

Arguments

window

   An Editor window

row

   An integer that specifies the screen row where the topmost line of text of the window is to appear. The top row of the screen is row 1.

column

   An integer that specifies the screen column where the leftmost text of the window is to appear. The left column of the screen is column 1.

Return Value

    The window

SHRINK WINDOW Command

Causes the current window to decrease in height by one line. If the
window is an anchored window, the heights of other anchored windows
are increased. If the prefix is a positive integer, the window
shrinks by the number of lines indicated. If the prefix is negative,
the window grows by the number of lines indicated.

Display Name Format

    Shrink Window

Function Format

    SHRINK-WINDOW-COMMAND *prefix*

Arguments

prefix

    An integer or NIL

Return Value

    The new window height

SIMPLE-PROMPT-FOR-INPUT Function

Prompts for input in the prompting window and returns the user's input
as a string.   The optional *prompt* argument specifies a prompting
message. If the user enters a null string, the function returns the
value of the optional *default* argument.

Format

    SIMPLE-PROMPT-FOR-INPUT &OPTIONAL *prompt default*

Arguments

prompt

    A string. The default is a null string.

default

> A value to be returned if the user enters a null string. The default is a null string.

Return Value

> A string as entered by the user or the value of the *default* argument

SPLIT WINDOW Command

Creates and returns a new Editor window by duplicating the current window and displaying both. If the original window is anchored, the heights of all anchored windows (including the new one) are adjusted.

Display Name Format

> Split Window

Function Format

> SPLIT-WINDOW-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> New window

START KEYBOARD MACRO Command

Starts an Editor keyboard macro. Each keystroke entered following this command is remembered, and all commands are executed. The keyboard macro can be ended with END-KEYBOARD-MACRO-COMMAND.

Display Name Format

> Start Keyboard Macro

Function Format

> START-KEYBOARD-MACRO-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> None

START NAMED KEYBOARD MACRO Command

Prompts the user for a name under which to catalog a new keyboard macro. Each keystroke entered following this command is remembered, as in a normal keyboard macro (see "Start Keyboard Macro" description). When the macro is completed (by "End Keyboard Macro"), it becomes the new current keyboard macro. It is also cataloged as a new named command by the system and can be treated just as any other named command. Its name is also entered in the *EDITOR-KEYBOARD-MACRO-NAMES* string table.

Category

> :GENERAL-PROMPTING

Display Name Format

> Start Named Keyboard Macro

Function Format

> START-NAMED-KEYBOARD-MACRO-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> NIL

START-OF-LINE-P Function

Is a predicate that returns T if the specified mark points to the beginning of a line and NIL otherwise.

Format

    START-OF-LINE-P *mark*

Arguments

mark

    An Editor mark

Return Value

    T or NIL

STRING-TABLE-P Function

Returns T if the argument is an Editor string table; NIL if it is not.

Format

    STRING-TABLE-P *object*

Arguments

object

    Any LISP object

Return Value

    T or NIL

STRING-TO-REGION Function

Returns a disembodied region containing the characters in the specified string.

Format

    STRING-TO-REGION *string*

Arguments

string

    A string

Return Value

    A new region

STYLE-NAME Function

Takes an Editor style specifier and returns the display name of the style.

Format

    STYLE-NAME *style*

Arguments

style

    An Editor style specifier

Return Value

    The string that is the display name of the style

STYLE-VARIABLES Function

Returns a list of symbols representing the Editor variables bound in a specified style.

Format

    STYLE-VARIABLES *style*

Arguments

style

    An Editor style

Return Value

    A list of symbols that name the Editor variables bound in the style

STYLEP Function

Returns T if the argument is an Editor Style, and NIL if it is not.

Format

> STYLEP *object*

Arguments

object

> Any LISP object

Return Value

> T or NIL

SUPPLY EMACS PREFIX Command

Sets the repetition count to four times its former value and returns the new count. That is, if the current prefix value is 1, this command sets the value to 4 if executed once, to 16 if executed twice, and so on.

Category

> :EMACS-PREFIX

Display Name Format

> Supply EMACS Prefix

Function Format

> SUPPLY-EMACS-PREFIX-COMMAND *prefix*

Arguments

prefix

> An integer or NIL

Return Value

> The repetition count

SUPPLY PREFIX ARGUMENT Command

Prompts the user for an integer and uses the response as a prefix argument for the next command invoked. The user terminates the prompt by pressing the RETURN key. If a prefix argument is supplied for this command, it multiplies the number entered as the response to the prompt.

Display Name Format

Supply Prefix Argument

Function Format

SUPPLY-PREFIX-ARGUMENT-COMMAND *prefix*

Arguments

prefix

The prefix argument for this command is an integer or NIL. It should not be confused with the prefix integer that this command returns for the subsequent command.

Return Value

The prefix integer for the next command invoked

SWITCH WINDOW HOOK Editor Variable

Is a hook function that is called with the new window as an argument before the value of (CURRENT-WINDOW) changes. If the change of (CURRENT-WINDOW) causes the value of (CURRENT-BUFFER) to change, the hooks "Buffer Entry Hook" and "Buffer Exit Hook" are also invoked.

Display Name Format

Switch Window Hook

Symbol Format

SWITCH-WINDOW-HOOK

TARGET COLUMN Editor Variable

Maintains the screen column for commands having the :LINE-MOVEMENT category (the "Previous Line" and "Next Line" commands, bound to the up arrow and down arrow, respectively). When one of these commands is entered, it checks the category of the previous command. If the previous command was not in the :LINE-MOVEMENT category, the current command sets the "Target Column" variable to the current column before moving the cursor. If the previous command was a :LINE-MOVEMENT command, the current command uses the value of the "Target Column" variable to position the cursor. This allows a series of :LINE-MOVEMENT commands to return the cursor to the original column after traversing one or more short lines.

Display Name Format

    Target Column

Symbol Format

    TARGET-COLUMN

TEXT OVERSTRIKE MODE Editor Variable

When set to T, causes characters inserted by means of "Self Insert" and "Quoted Insert" to replace any characters (except newline characters) previously located at the same positions. Text inserted at a newline character is inserted at the end of the same line (that is, the newline character is moved to the right). When this variable is set to NIL, newly inserted characters appear between previous characters. In the default Editor, this variable is bound globally and set to NIL.

Note that this variable does not affect the operation of other text-inserting commands, such as "EDT Paste" and "Yank".

Display Name Format

    Text Overstrike Mode

Symbol Format

    TEXT-OVERSTRIKE-MODE

TRANSPOSE PREVIOUS CHARACTERS Command

Transposes the pair of characters before the cursor (the current buffer point).

Display Name Format

>   Transpose Previous Characters

Function Format

>   TRANSPOSE-PREVIOUS-CHARACTERS-COMMAND *prefix*

Arguments

prefix

>   Ignored

Return Value

>   Undefined

TRANSPOSE PREVIOUS WORDS Command

Transposes the pair of words at and before the cursor (the current buffer point).

Display Name Format

>   Transpose Previous Words

Function Format

>   TRANSPOSE-PREVIOUS-WORDS-COMMAND *prefix*

Arguments

prefix

>   Ignored

Return Value

>   Undefined

UNBIND-ATTRIBUTE Function

Unbinds the specified attribute from the specified context. The unbind hook function defined for the attribute is called with two arguments -- the attribute and the context.

Format

> UNBIND-ATTRIBUTE *attribute* &OPTIONAL *context*

Arguments

attribute

> An attribute specifier

context

> The context from which to unbind the attribute. The default is :GLOBAL.

Return Value

> NIL

UNBIND-COMMAND Function

Deletes the binding of a key sequence to a command in the specified context.

Format

> UNBIND-COMMAND *key-sequence* &OPTIONAL *context*

Arguments

key-sequence

> A sequence of characters

context

> An Editor context specifier that defaults to :GLOBAL

Return Value

> The function that was bound, or NIL if no binding was found

271

UNBIND-POINTER-COMMAND Function

Deletes the binding of a pointer action to a command in the specified context.

Format

UNBIND-POINTER-COMMAND *pointer-action* &OPTIONAL *context*

Arguments

pointer-action

A keyword, a button constant, or a list that specifies an action of a supported pointing device. See BIND-POINTER-COMMAND for the possible values.

context

A context specifier. The default is :GLOBAL.

Return Value

The function associated with the command that was bound, or NIL if no binding was found.

UNBIND-VARIABLE Function

Unbinds the specified Editor variable from the specified context. The unbind hook function defined for the variable is called with two arguments -- the variable and the context.

Format

UNBIND-VARIABLE *variable* &OPTIONAL *context*

Arguments

variable

An Editor variable specifier

context

The context from which to unbind the variable. The default is :GLOBAL.

Return Value

NIL

UNDO PREVIOUS YANK Command

Deletes the previously yanked region but does not push this region onto the kill ring. More generally, this command deletes a region without pushing it onto the kill ring. The region is either the currently selected region (the region associated with the Buffer Select Region editor variable) or, if no currently selected region exists, a region defined by the buffer select mark and the current buffer mark.

Display Name Format

> Undo Previous Yank

Function Format

> UNDO-PREVIOUS-YANK-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> NIL

UNSET SELECT MARK Command

Deletes the select mark and removes the select region in the current buffer. That is, it cancels the action of the command "Set Select Mark" (or "EDT Select") by setting the value of the Editor variables "Buffer Select Mark" and "Buffer Select Region" to NIL.

Any text contained in the select region is not affected.

Display Name Format

> Unset Select Mark

or

> EDT Deselect

Function Format

> UNSET-SELECT-MARK-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> Undefined

UPCASE REGION Command

Makes the alphabetic characters in the region supplied as an argument all upper case. If no argument is supplied, the command uses the current select region.

Display Name Format

> Upcase Region

Function Format

> UPCASE-REGION-COMMAND *prefix* &OPTIONAL *region*

Arguments

prefix

> Ignored

region

> An Editor region that defaults to the buffer select region

Return Value

> Undefined

UPCASE WORD Command

Makes the alphabetic characters in the word around the specified mark all upper case. The mark defaults to the current buffer point.

Display Name Format

> Upcase Word

Function Format

    UPCASE-WORD-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

    Ignored

mark

    An Editor mark that defaults to the current buffer point

Return Value

    A region containing the word

UPDATE-DISPLAY Function

Updates any Editor windows that have changed and turns off batching of screen updates. This function does not detect messages issued by VMS (such as operator messages) and therefore might not erase them.

Format

    UPDATE-DISPLAY

Arguments

    None

Return Value

    None

UPDATE-WINDOW-LABEL Function

Updates the label of a window and returns the new label as a string. This function is useful when you must force the window to change at times other than when the display manger needs to change it.

Format

    UPDATE-WINDOW-LABEL *window*

Arguments

window

>   An Editor window

Return Value

>   The new label for the window

VARIABLE-BOUNDP Function

Returns T if the specified Editor variable has a value, and NIL if it does not.

Format

>   VARIABLE-BOUNDP *editor-variable* &OPTIONAL *context*

Arguments

editor-variable

>   An Editor variable specifier

context

>   An optional context specifier. Defaults to the current context.

Return Value

>   T or NIL

VARIABLE-FBOUNDP Function

Returns T if the specified Editor variable has a function definition, and NIL if it does not.

Format

>   VARIABLE-FBOUNDP *editor-variable* &OPTIONAL *context*

Arguments

editor-variable

>   An Editor variable specifier

context

An optional context specifier. Defaults to the current context.

Return Value

T or NIL

VARIABLE-FUNCTION Function

Returns the function definition of the Editor variable in the specifed context. An error is signaled if the argument symbol is not a defined Editor variable in the specified context.

You can use this function with the SETF macro to change the function definition of an Editor variable. If the function definition of an Editor variable is set, all the set hook functions associated with that variable are called.

Format

VARIABLE-FUNCTION *variable* &OPTIONAL *context*

Arguments

variable

An Editor variable specifier

context

A context specifier that defaults to the current context

Return Value

The function definition of the Editor variable

VARIABLE-NAME Function

Returns the display name of the specified Editor variable.

Format

VARIABLE-NAME *variable*

Arguments

variable

An Editor variable specifier

Return Value

The display name of the variable

VARIABLE-VALUE Function

Returns the value of the specified Editor variable in the specified context. An error is signaled if the argument symbol is not a defined Editor variable in the specified context. You can use this function with the SETF macro to change the value of a symbol. If the variable value of an Editor variable is set, all the set hook functions associated with that variable are called.

Format

VARIABLE-VALUE *variable* &OPTIONAL *context*

Arguments

variable

An Editor variable specifier

context

A context specifier that defaults to the current context

Return Value

The value of the Editor variable

VAX LISP Style

Is the default minor style for editing any LISP objects or for editing files with an extension of LSP.

Display Name Format

VAX LISP

Symbol Format

VAX-LISP

VIEW FILE Command

Prompts the user for the name of a file, if one is not supplied, and reads that file into a read-only buffer. A window into the buffer is created and becomes the new current window. If a buffer exists with that file, that buffer becomes the current one and is set to be read-only. An Editor error is signaled if any attempt to modify the buffer occurs.

Category

:GENERAL-PROMPTING

Display Name Format

View File

Function Format

VIEW-FILE-COMMAND *prefix* &OPTIONAL *file*

Arguments

prefix

Ignored

file

A pathname, namestring, or stream

Return Value

The new buffer

VISIBLE-WINDOWS Function

Returns a list of the windows currently visible on the screen. A window is considered "visible" if it has been displayed and not removed; thus, a window that is completely hidden by another window is still considered visible.

Format

        VISIBLE-WINDOWS

Arguments

        None

Return Value

        A list of the windows currently visible on the screen

WHAT CURSOR POSITION Command

Displays the following information about the current buffer point in
the information area:

        X = column
        Y = row
        L = line number (% of total)
        C = character number (% of total)
        W = window start-line number; window end-line number
        CH = char-code of current character

Display Name Format

        What Cursor Position

Function Format

        WHAT-CURSOR-POSITION-COMMAND *prefix*

Arguments

prefix

        Ignored

Return Value

        NIL

WHITESPACE Attribute

Has a value of 1 for whitespace characters and 0 for all other
characters.

Display Name Format

    Whitespace

Symbol Format

    WHITESPACE

WHITESPACE-AFTER-P Function

Is a predicate that returns **T** if all the characters following mark on the line have a "Whitespace" attribute value of 1; otherwise, it returns NIL.

Format

    WHITESPACE-AFTER-P *mark*

Arguments

mark

    An Editor mark

Return Value

    T or NIL

WHITESPACE-BEFORE-P Function

Is a predicate that returns **T** if all the characters preceding mark on the line have a "Whitespace" attribute of 1 or if the line is empty; otherwise, it returns NIL.

Format

    WHITESPACE-BEFORE-P *mark*

Arguments

mark

    An Editor mark

Return Value

    T or NIL

WHITESPACE-BETWEEN-P Function

Is a predicate that returns T if all the characters between the two marks have a "Whitespace" attribute value of 1. A mark at the end of a line precedes the newline character; a mark at the beginning of a line follows the newline character. It is an error for the marks to point into different buffers or disembodied regions.

Format

    WHITESPACE-BETWEEN-P *start-mark end-mark*

Arguments

start-mark

    An Editor mark pointing to the character that should start the scan

end-mark

    An Editor mark pointing to the character that should end the whitespace scan. The character at the end is not included in the scan.

Return Value

    T or NIL

WHITESPACE-LINE-P Function

Is a predicate that returns T if every character in the line has a "Whitespace" attribute of 1 (or if the line is empty); otherwise, it returns NIL.

Format

    WHITESPACE-LINE-P *line*

Arguments

line

    An Editor line

Return Value

    T or NIL

WINDOW-BUFFER Function

Returns the buffer associated with a window. You can use this function as a place indicator with the SETF macro to change the buffer associated with a window. Changing the value of WINDOW-BUFFER causes the "Window Buffer Hook" to be invoked.

Format

    WINDOW-BUFFER *window*

Arguments

window

    An Editor window

Return Value

    An Editor buffer

WINDOW BUFFER HOOK Editor Variable

Is a hook function that is called with the window and new buffer as arguments whenever a window is to be associated with a different buffer.

Display Name Format

    Window Buffer Hook

Symbol Format

    WINDOW-BUFFER-HOOK

WINDOW CREATION HOOK Editor Variable

Is a hook function that is called with a new window as an argument whenever a new window is created.

Display Name Format

    Window Creation Hook

Symbol Format

    WINDOW-CREATION-HOOK

WINDOW-CREATION-TIME Function

Returns the universal time at which the specified window was created. For information on universal time, see *COMMON LISP: The Language.*

Format

    WINDOW-CREATION-TIME *window*

Arguments

window

    An Editor window

Return Value

    The universal time at which the window was created

WINDOW DELETION HOOK Editor Variable

Is a hook function that is called with a window as an argument before it is deleted.

Display Name Format

    Window Deletion Hook

Symbol Format

    WINDOW-DELETION-HOOK

WINDOW-DISPLAY-COLUMN Function

Returns the physical screen column that the first text character of the specified window is displayed in. Columns are numbered beginning with 1. This function is not a place form acceptable to the SETF macro.

Format

    WINDOW-DISPLAY-COLUMN *window*

Arguments

window

    The window whose display column is to be returned

Return Value

   An integer specifying the column

WINDOW-DISPLAY-END Function

Returns a mark that points to the position just after the last position displayed in the window. Altering the position of this mark can have unpredictable results.

Format

   WINDOW-DISPLAY-END *window*

Arguments

window

   An Editor window

Return Value

   An Editor mark

WINDOW-DISPLAY-ROW Function

Returns the physical screen row that the first text character of the specified window is displayed in. Rows are numbered beginning with 1. This function is not a place form acceptable to the SETF macro.

Format

   WINDOW-DISPLAY-ROW *window*

Arguments

window

   The window whose display row is to be returned

Return Value

   An integer specifying the row

WINDOW-DISPLAY-START Function

This function returns a mark that points to the first position displayed in the window. This mark must always point to the beginning of a line (that is, its character position must be 0).

Format

    WINDOW-DISPLAY-START *window*

Arguments

window

    An Editor window

Return Value

    A mark

WINDOW-HEIGHT Function

Returns the height of the window as an integer. You can use this function as a place indicator to the SETF macro to change the height of a window. Changing the value of WINDOW-HEIGHT causes the "Window Modification Hook" to be invoked.

Format

    WINDOW-HEIGHT *window*

Arguments

window

    An Editor window

Return Value

    An integer

WINDOW-LABEL Function

Returns either a string to be used as the window label or a function used to create the the label string for a window. You can use this function with the SETF macro to change the label of a window.

Format

    WINDOW-LABEL *window*

Arguments

window

    An Editor window

Return Value

    A string, a function, or NIL

WINDOW-LABEL-EDGE Function

Returns the edge of the window that the label is on. The value can be :TOP, :BOTTOM, :LEFT, or :RIGHT. The default is :BOTTOM. This corresponds to the :LABEL-EDGE option of MAKE-WINDOW. You can use this function with the SETF macro to change the edge of the window that the label is on.

Format

    WINDOW-LABEL-EDGE *window*

Arguments

window

    An Editor window

Return Value

    The keyword indicating the edge the label is on

WINDOW-LABEL-OFFSET Function

Returns a nonnegative integer or NIL. If NIL, the label is centered on the specified side. If a number, the beginning of the label is offset by the number of characters from the start of the specified side. You can use this function with the SETF macro to change the offset of the label.

Format

    WINDOW-LABEL-OFFSET *window*

Arguments

window

>   An Editor window

Return Value

>   A positive integer or NIL

WINDOW-LABEL-RENDITION Function

Returns a keyword or a list of keywords specifying the video rendition for a window's label. The keywords are :NORMAL, :BLINK, :BOLD, :REVERSE, and :UNDERLINE. This function is acceptable as a place form to SETF. The new value can be a single keyword or a list of keywords.

Format

>   WINDOW-LABEL-RENDITION *window*

Arguments

window

>   The window whose label's video rendition is desired

Return Value

>   A list of keywords as described above

WINDOW-LINES-WRAP-P Function

Returns T if lines that are longer than the window is wide are wrapped, or NIL if they are truncated. This function is acceptable as a place form to the SETF macro to make lines truncated or wrapped in a window.

Format

>   WINDOW-LINES-WRAP-P *window*

Arguments

window

>   An Editor window

Return Value

> T or NIL

WINDOW MODIFICATION HOOK Editor Variable

Is a hook function that is called with the modified window as an argument whenever the height, type, or width of the window changes. It is called at the completion of the modification.

Display Name Format

> Window Modification Hook

Symbol Format

> WINDOW-MODIFICATION-HOOK

WINDOW-POINT Function

Returns a mark that retains the buffer point for a specified window. You can use this mark to alter the display for a window that is not the current window.

Format

> WINDOW-POINT *window*

Arguments

window

> An Editor window

Return Value

> An Editor mark

WINDOW-RENDITION Function

Returns a list of keywords specifying the video rendition for an entire window. The keywords are :NORMAL, :BLINK, :BOLD, :REVERSE, and :UNDERLINE. This function is acceptable as a place form to SETF. The new value can be a single keyword or a list of keywords.

Format

    WINDOW-RENDITION *window*

Arguments

window

    An Editor window

Return Value

    A list of keywords

WINDOW-TRUNCATE-CHAR Function

Returns the character used to indicate that a line is truncated. The default character is >. This function can be used as a place indicator with the SETF macro to change the truncation indicator character. Changing this character causes the window image to be recomputed if WINDOW-LINES-WRAP-P is NIL.

Format

    WINDOW-TRUNCATE-CHAR *window*

Arguments

window

    An Editor window

Return Value

    A character

WINDOW-TYPE Function

Returns a keyword indicating the type of the window. You can change the type of a window by using this form with SETF.

Format

    WINDOW-TYPE *window*

Arguments

window

    An Editor window

Return Value

    :FLOATING or :ANCHORED

WINDOW-WIDTH Function

Returns the width of the window as an integer. This function can be used with the SETF macro to change the width of a window. Changing the value of WINDOW-WIDTH causes the "Window Modification Hook" to be invoked.

Format

    WINDOW-WIDTH *window*

Arguments

window

    An Editor window

Return Value

    The width of the window

WINDOW-WRAP-CHAR Function

Returns the character used to indicate that the lines wrapped. The default character is <. This function can be used as a place indicator with the SETF macro to change the wrap indicator character. Changing this character causes the window image to be recomputed if WINDOW-LINES-WRAP-P is T.

Format

    WINDOW-WRAP-CHAR *window*

Arguments

window

    An Editor window

Return Value

A character

WINDOWP Function

Returns T if its argument is an Editor window, and NIL if it is not.

Format

WINDOWP *object*

Arguments

object

Any LISP object

Return Value

T or NIL

WITH-INPUT-FROM-REGION Macro

Makes an input stream from region and evaluates the forms as an implicit PROGN with the stream bound to the argument var. On exit from the macro, the stream is closed.

Format

WITH-INPUT-FROM-REGION *(var region)* {*declaration*}* {*form*}*

Arguments

var

The variable var is bound to the input stream.

region

An Editor region

Return Value

The value of the last evaluated form

WITH-MARK Macro

Evaluates the forms of the body with the variables bound to copies of the specified marks. The copied marks are deleted upon exit from the form.

Format

    WITH-MARK ({(var mark [type])}*) form

Arguments

(var mark [type])

Each variable is bound to a copy of the mark. The new mark will be of type :TEMPORARY unless otherwise specified by the type.

forms

A list of forms that are evaluated as an implicit progn

Return Value

The value of the last evaluated form

WITH-OUTPUT-TO-MARK Macro

Creates an output stream to mark and evaluates the forms as an implicit progn with the stream bound to the argument var. On exit from the macro, the stream is closed.

Format

    WITH-OUTPUT-TO-MARK (var mark) {declaration}* {form}*

Arguments

var

The variable that will be bound to the output stream

mark

An Editor mark where output from the stream will be inserted

Return Value

The value of the last evaluated form

WITH-SCREEN-UPDATE Macro

Used to batch any changes made to the screen until all the specified forms have completed. This form is especially useful when an Editor command makes a large number of changes to the screen, such as removing and showing several windows as part of one command. No alterations will appear on the screen until the specified forms complete, at which time the screen will change to reflect the final configuration.

Prompts written to the screen (by using either of the prompting functions or writing to the information area) will not appear on the screen while inside this macro.

You may not change the screen height or width while inside this macro.

Format

WITH-SCREEN-UPDATE &REST {*form*}*

Arguments

form

One or more forms to be evaluated before the screen update occurs

Return Value

The value of the last form executed

WORD DELIMITER Attribute

Has a value of 1 for characters that separate words, and 0 for all other characters.

Display Name Format

Word Delimiter

Symbol Format

WORD-DELIMITER

WORD-OFFSET Function

Updates a mark so that it points to the next word -- that is, to the next non-word-delimiting character beyond the next word-delimiting character. A word-delimiting character is a character having a "Word Delimiter" attribute value of 1. the count value specifies the number of word breaks that are to be located, going forward if positive and backward if negative.

Format

WORD-OFFSET *mark count*

Arguments

mark

The mark that is to be updated

count

The number of word breaks that are to be located

Return Value

The updated mark

WRITE CURRENT BUFFER Command

Writes the current buffer or the buffer specified as the optional argument. If the buffer is associated with a file, the resulting file is one with the same specification and the highest version number. The associated checkpoint file, if there is one, is deleted. If the buffer was created from a LISP object, the buffer contents are read (and evaluated, if the contents are a LISP function) to produce a new object. For example, if the buffer contained a function definition, the definition is changed.

Display Name Format

Write Current Buffer

Function Format

WRITE-CURRENT-BUFFER-COMMAND *prefix* &OPTIONAL *buffer*

Arguments

prefix

Ignored

buffer

An Editor buffer that defaults to the current buffer

Return Value

The pathname of the file that the buffer was written to; or the
value read from the buffer

WRITE-FILE-FROM-REGION Function

Writes the specified region to a specified file. The region can begin
or end in the middle of a line. Only the text in the region is
written to the file. Each line in the region corresponds to a record
in the file.

Format

WRITE-FILE-FROM-REGION *pathname region*

Arguments

pathname

A pathname or namestring

region

An Editor region

Return Value

Two values:

1. The truename of the file written. (For an explanation of the
    truename of a file, see *Common LISP: The Language*).

2. The count of the number of records written to the file.

WRITE MODIFIED BUFFERS Command

Performs the same operations as "Write Current Buffer" for each buffer containing an object (that is, a file or a LISP object) being edited.

Display Name Format

Write Modified Buffers

Function Format

WRITE-MODIFIED-BUFFERS-COMMAND *prefix*

Arguments

prefix

Ignored

Return Value

NIL

WRITE NAMED FILE Command

Prompts for a file name if it is not supplied.   The   function   writes the current Editor buffer to the file.

Category

:GENERAL-PROMPTING

Display Name Format

Write Named File

Function Format

WRITE-NAMED-FILE-COMMAND *prefix* &OPTIONAL *filename*

Arguments

prefix

Ignored

filename

A pathname, namestring, string, or stream

Return Value

The pathname to which the buffer was written

YANK Command

Copies the current region of the kill ring into the indicated buffer at the argument mark. The mark defaults to the current buffer point. If an integer prefix argument is supplied, that many copies of the kill-ring region are inserted.

Display Name Format

Yank

Function Format

YANK-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

An integer or NIL

mark

An Editor mark that defaults to the current buffer point

Return Value

NIL

YANK AT POINTER Command

Moves the current buffer point to the position indicated by the pointer and then inserts at that location the first region saved on the kill ring. If the pointer is beyond the end of a line, the region is inserted at the end of that line. If the pointer is beyond the end of the buffer region, the kill region is inserted at the end of the buffer region.

Display Name Format

Yank at Pointer

Function Format

> YANK-AT-POINTER-COMMAND *prefix*

Arguments

prefix

> Ignored

Return Value

> The current buffer point

YANK PREVIOUS Command

Rotates the kill ring forward, and copies the new current kill-ring region into the buffer at the argument mark. The mark defaults to the current buffer point. The prefix defaults to 1 and specifies how many copies are to be inserted (not how much to rotate the ring).

Display Name Format

> Yank Previous

Function Format

> YANK-PREVIOUS-COMMAND *prefix* &OPTIONAL *mark*

Arguments

prefix

> An integer or NIL

mark

> An Editor mark that defaults to the current buffer point

Return Value

> NIL

YANK REPLACE PREVIOUS Command

Deletes the previously yanked region, rotates the kill ring forward, and copies the new current kill-ring region into the current buffer at the current point. The prefix defaults to 1 and specifies how many copies are to be inserted.

Display Name Format

Yank Replace Previous

Function Format

YANK-REPLACE-PREVIOUS-COMMAND *prefix*

Arguments

prefix

An integer or NIL

Return Value

None

APPENDIXES

APPENDIX A

EDITOR OBJECTS BY CATEGORY

This appendix lists the Editor objects -- functions, variables, commands, and so on -- that pertain to each of the categories listed below. The categories are the major data types, subsystems, and utilities provided with the Editor.

```
ATTRIBUTES
ATTRIBUTES PROVIDED WITH VAX LISP
BUFFERS
BUFFERS PROVIDED WITH VAX LISP
COMMANDS
COMMANDS PROVIDED WITH VAX LISP
DISPLAY
EDITOR VARIABLES
EDITOR VARIABLES PROVIDED WITH VAX LISP
ERROR SIGNALING AND DEBUGGING
FILES
HELP
HOOKS
HOOK VARIABLES PROVIDED WITH VAX LISP
INVOKING AND EXITING THE EDITOR
KILL RING
LINES
LISP SYNTAX
MARKS
MISCELLANEOUS
POINTING DEVICE
PROMPTING AND TERMINAL INPUT
REGIONS
RINGS
SEARCHING
STRING TABLES
STRING TABLES PROVIDED WITH VAX LISP
STYLES
STYLES PROVIDED WITH VAX LISP
STYLE BINDINGS, "EDT EMULATION" STYLE
```

STYLE BINDINGS, "EMACS" STYLE
STYLE BINDINGS, "VAX LISP" STYLE
TEXT OPERATIONS
WINDOWS

ATTRIBUTES

```
ATTRIBUTE-NAME function
BIND-ATTRIBUTE function
CHARACTER-ATTRIBUTE function
"Character Attribute Hook" Editor variable
DEFINE-ATTRIBUTE macro
*EDITOR-ATTRIBUTE-NAMES* variable
FIND-ATTRIBUTE function
LOCATE-ATTRIBUTE function
UNBIND-ATTRIBUTE function
WHITESPACE-AFTER-P function
WHITESPACE-BEFORE-P function
WHITESPACE-BETWEEN-P function
WHITESPACE-LINE-P function
```

ATTRIBUTES PROVIDED WITH VAX LISP

```
"LISP Syntax"
"Page Delimiter"
"Print Representation"
"Whitespace"
"Word Delimiter"
```

BUFFERS

```
"Beginning of Buffer" command
BUFFER-CHECKPOINTED function
BUFFER-CHECKPOINTED-TIME function
"Buffer Creation Hook" Editor variable
BUFFER-CREATION-TIME function
"Buffer Deletion Hook" Editor variable
"Buffer Entry Hook" Editor variable
"Buffer Exit Hook" Editor variable
BUFFER-HIGHLIGHT-REGIONS function
BUFFER-MAJOR-STYLE function
BUFFER-MINOR-STYLE-ACTIVE function
BUFFER-MINOR-STYLE-LIST function
BUFFER-MODIFIED-P function
BUFFER-NAME function
"Buffer Name Hook" Editor variable
BUFFER-OBJECT function
"Buffer Object Hook" Editor variable
BUFFER-PERMANENT function
BUFFER-POINT function
BUFFER-REGION function
BUFFER-TYPE function
BUFFER-VARIABLES function
BUFFER-WINDOWS function
BUFFER-WRITABLE function
```

BUFFERS (cont.)

BUFFER-WRITTEN-TIME function
BUFFERP function
CHECKPOINT-BUFFER function
CURRENT-BUFFER function
CURRENT-BUFFER-POINT function
"Default Buffer Variables" Editor variable
"Default Major Style" Editor variable
"Default Minor Styles" Editor variable
DELETE-BUFFER function
"Delete Current Buffer" command
"Delete Named Buffer" command
EDITOR-BUFFER-NAMES variable
EDITOR-DEFAULT-BUFFER variable
"End of Buffer" command
FIND-BUFFER function
"List Buffers" command
MAKE-BUFFER function
MAP-BUFFERS function
"Maybe Reset Select at Pointer" command
"Move Point and Select Region" command
"Move Point to Pointer" command
"Select Buffer" command
"Set Select Mark" command
"Unset Select Mark" command

BUFFERS PROVIDED WITH VAX LISP

"General Prompting" buffer
"Help" buffer

COMMANDS

"Bind Command" command
BIND-COMMAND function
BIND-POINTER-COMMAND function
CATEGORY-COMMANDS function
COMMAND-CATEGORIES function
COMMAND-NAME function
CURRENT-COMMAND-FUNCTION variable
DEFINE-COMMAND macro
DEFINE-KEYBOARD-MACRO function
EDITOR-COMMAND-NAMES variable
EDITOR-KEYBOARD-MACRO-NAMES variable
END-KEYBOARD-MACRO function
ENQUEUE-EDITOR-COMMAND function
"Execute Keyboard Macro" command
"Execute Named Command" command
FIND-COMMAND function

COMMANDS (cont.)

GET-BOUND-COMMAND-FUNCTION function
"List Key Bindings" command
MAP-BINDINGS function
MAKE-COMMAND function
PREVIOUS-COMMAND-FUNCTION variable
"Start Keyboard Macro" command
"Start Named Keyboard Macro" command
"Supply EMACS Prefix" command
"Supply Prefix Argument" command
UNBIND-COMMAND function
UNBIND-POINTER-COMMAND function

COMMANDS PROVIDED WITH VAX LISP

"Activate Minor Style"
"Apropos"
"Apropos Word"
"Backward Character"
"Backward Kill Ring"
"Backward Page"
"Backward Search"
"Backward Word"
"Beginning of Buffer"
"Beginning of Line"
"Beginning of Outermost Form"
"Beginning of Paragraph"
"Beginning of Window"
"Bind Command"
"Capitalize Region"
"Capitalize Word"
"Close Outermost Form"
"Deactivate Minor Style"
"Delete Current Buffer"
"Delete Line"
"Delete Named Buffer"
"Delete Next Character"
"Delete Next Word"
"Delete Previous Character"
"Delete Previous Word"
"Delete Whitespace"
"Delete Word"
"Describe"
"Describe Word"
"Describe Word at Pointer"
"Downcase Region"
"Downcase Word"
"Ed"
"Edit File"
"EDT Append"

COMMANDS PROVIDED WITH VAX LISP (cont.)

"EDT Back to Start of Line"
"EDT Beginning of Line"
"EDT Capitalize"
"EDT Cut"
"EDT Delete Character"
"EDT Delete Line"
"EDT Delete Previous Character"
"EDT Delete Previous Line"
"EDT Delete Previous Word"
"EDT Delete to End of Line"
"EDT Delete Word"
"EDT Deselect"
"EDT End of Line"
"EDT Move Character"
"EDT Move Page"
"EDT Move Word"
"EDT Paste"
"EDT Paste at Pointer"
"EDT Query Search"
"EDT Replace"
"EDT Scroll Window"
"EDT Search Again"
"EDT Select"
"EDT Set Direction Advance"
"EDT Set Direction Reverse"
"EDT Special Insert"
"EDT Substitute"
"EDT Undelete Character"
"EDT Undelete Line"
"EDT Undelete Word"
"EMACS Backward Search"
"EMACS Forward Search"
"End Keyboard Macro"
"End of Buffer"
"End of Line"
"End of Outermost Form"
"End of Paragraph"
"End of Window"
"Evaluate LISP Region"
"Exchange Point and Select Mark"
"Execute Keyboard Macro"
"Execute Named Command"
"Exit"
"Exit Recursive Edit"
"Forward Character"
"Forward Kill Ring"
"Forward Page"
"Forward Search"
"Forward Word"
"Grow Window"

COMMANDS PROVIDED WITH VAX LISP (cont.)

"Help"
"Help on Editor Error"
"Illegal Operation"
"Indent LISP Line"
"Indent LISP Region"
"Indent Outermost Form"
"Insert Buffer"
"Insert Close Paren and Match"
"Insert File"
"Kill Enclosing List"
"Kill Line"
"Kill Next Form"
"Kill Paragraph"
"Kill Previous Form"
"Kill Region"
"Kill Rest of List"
"Line to Top of Window"
"List Buffers"
"List Key Bindings"
"Maybe Reset Select at Pointer"
"Move Point and Select Region"
"Move Point to Pointer"
"Move to Lisp Comment"
"New Line"
"New LISP Line"
"Next Form"
"Next Line"
"Next Paragraph"
"Next Screen"
"Next Window"
"Open Line"
"Page Next Window"
"Page Previous Window"
"Pause"
"Previous Form"
"Previous Line"
"Previous Paragraph"
"Previous Screen"
"Previous Window"
"Prompt Complete String"
"Prompt Help"
"Prompt Read and Validate"
"Prompt Scroll Help Window"
"Prompt Show Alternatives"
"Query Search Replace"
"Quoted Insert"
"Read File"
"Redisplay Screen"
"Remove Current Window"
"Remove Other Windows"

COMMANDS PROVIDED WITH VAX LISP (cont.)

"Scroll Window Down"
"Scroll Window Up"
"Select Buffer"
"Select Enclosing Form at Pointer"
"Select Outermost Form"
"Self Insert"
"Set Select Mark"
"Show Time"
"Shrink Window"
"Split Window"
"Start Keyboard Macro"
"Start Named Keyboard Macro"
"Supply EMACS Prefix"
"Supply Prefix Argument"
"Transpose Previous Characters"
"Transpose Previous Words"
"Undo Previous Yank"
"Unset Select Mark"
"Upcase Region"
"Upcase Word"
"View File"
"What Cursor Position"
"Write Current Buffer"
"Write Modified Buffers"
"Write Named File"
"Yank"
"Yank at Pointer"
"Yank Previous"
"Yank Replace Previous"

DISPLAY

CLEAR-INFORMATION-AREA function
EDITOR-RETAIN-SCREEN-STATE variable
INFORMATION-AREA-HEIGHT function
INFORMATION-AREA-OUTPUT-STREAM variable
"Redisplay Screen" command
REDISPLAY-SCREEN function
SCREEN-HEIGHT function
"Screen Modification Hook" Editor variable
SCREEN-WIDTH function
"Set Screen Height" command
"Set Screen Width" command
UPDATE-DISPLAY function
WITH-SCREEN-UPDATE macro

EDITOR VARIABLES

```
BIND-VARIABLE function
DEFINE-EDITOR-VARIABLE macro
*EDITOR-VARIABLE-NAMES* variable
FIND-VARIABLE function
UNBIND-VARIABLE function
VARIABLE-BOUNDP function
VARIABLE-FBOUNDP function
VARIABLE-FUNCTION function
VARIABLE-NAME function
VARIABLE-VALUE function
```

EDITOR VARIABLES PROVIDED WITH VAX LISP

```
"Anchored Window Show Limit"
"Buffer Creation Hook"
"Buffer Deletion Hook"
"Buffer Entry Hook"
"Buffer Exit Hook"
"Buffer Name Hook"
"Buffer Object Hook"
"Buffer Right Margin"
"Buffer Select Mark"
"Buffer Select Region"
"Character Attribute Hook"
"Current Window Pointer Pattern"
"Current Window Pointer Pattern X"
"Current Window Pointer Pattern Y"
"Default Buffer Variables"
"Default Filetype Minor Styles"
"Default Major Style"
"Default Minor Styles"
"Default Search Case"
"Default LISP Object Minor Styles"
"Default Window Label"
"Default Window Label Edge"
"Default Window Label Offset"
"Default Window Label Rendition"
"Default Window Lines Wrap"
"Default Window Rendition"
"Default Window Truncate Char"
"Default Window Type"
"Default Window Width"
"Default Window Wrap Char"
"Editor Entry Hook"
"Editor Exit Hook"
"Editor Initialization Hook"
"Editor Pause Hook"
"EDT Deleted Character"
"EDT Deleted Line"
```

EDITOR VARIABLES PROVIDED WITH VAX LISP (cont.)

"EDT Deleted Word"
"EDT Direction Mode"
"EDT Paste Buffer"
"Help Text"
"Information Area Pointer Pattern"
"Information Area Pointer Pattern X"
"Information Area Pointer Pattern Y"
"Last Search Direction"
"Last Search Pattern"
"Last Search String"
"LISP Comment Column"
"LISP Evaluation Result"
"Major Style Activation Hook"
"Minor Style Activation Hook"
"Noncurrent Window Pointer Pattern"
"Noncurrent Window Pointer Pattern X"
"Noncurrent Window Pointer Pattern Y"
"Prompt Alternatives"
"Prompt Alternatives Arguments"
"Prompt Completion"
"Prompt Completion Arguments"
"Prompt Default"
"Prompt Error Message"
"Prompt Error Message Arguments"
"Prompt Help"
"Prompt Help Arguments"
"Prompt Help Called"
"Prompt Rendition Complement"
"Prompt Rendition Set"
"Prompt Required"
"Prompt Start"
"Prompt Validation"
"Screen Modification Hook"
"Select Region Rendition Complement"
"Select Region Rendition Set"
"Switch Window Hook"
"Target Column"
"Text Overstrike Mode"
"Window Buffer Hook"
"Window Creation Hook"
"Window Deletion Hook"
"Window Modification Hook"

ERROR SIGNALING AND DEBUGGING

ATTENTION function
EDITOR-ERROR function
EDITOR-ERROR-WITH-HELP function
EDITOR-RETAIN-SCREEN-STATE variable
"Illegal Operation" command

FILES

```
BUFFER-CHECKPOINTED function
CHECKPOINT-FREQUENCY function
"Edit File" command
"Insert File" command
INSERT-FILE-AT-MARK function
"Read File" command
"View File" command
"Write Current Buffer" command
WRITE-FILE-FROM-REGION function
"Write Modified Buffers" command
"Write Named File" command
```

HELP

```
"Apropos" command
APROPOS-STRING-TABLE function
"Apropos Word" command
"Apropos Word" command
"Describe" command
"Describe Word" command
"Describe Word at Pointer" command
"Editor Help" command
EDITOR-HELP-BUFFER buffer
"Help" buffer
"Help" command
"Help on Editor Error" command
"Help Text" Editor variable
"Prompt Complete String" command
"Prompt Help" command
"Prompt Scroll Help Window" command
"Prompt Show Alternatives" command
```

HOOKS

```
INVOKE-HOOK function
REVERSE-INVOKE-HOOK function
```

HOOK VARIABLES PROVIDED WITH VAX LISP

```
"Buffer Creation Hook"
"Buffer Deletion Hook"
"Buffer Entry Hook"
"Buffer Exit Hook"
"Buffer Name Hook"
"Buffer Object Hook"
"Character Attribute Hook"
"Editor Entry Hook"
```

HOOK VARIABLES PROVIDED WITH VAX LISP (cont.)

```
"Editor Exit Hook"
"Editor Initialization Hook"
"Editor Pause Hook"
"Major Style Activation Hook"
"Minor Style Activation Hook"
"Screen Modification Hook"
"Switch Window Hook"
"Window Buffer Hook"
"Window Creation Hook"
"Window Deletion Hook"
"Window Modification Hook"
```

INVOKING AND EXITING THE EDITOR

```
"Ed" command
ED function
"Editor Entry Hook" Editor variable
"Editor Exit Hook" Editor variable
"Editor Initialization Hook" Editor variable
"Editor Pause Hook" Editor variable
"Exit Editor" command
"Exit Recursive Edit" command
INITIALIZE-EDITOR function
"Pause Editor" command
RETURN-FROM-EDITOR macro
```

KILL RING

```
"Backward Kill Ring" command
"Forward Kill Ring" command
"Kill Line" command
"Kill Paragraph" command
"Kill Region" command
"Undo Previous Yank" command
"Yank" command
"Yank at Pointer" command
"Yank Previous" command
"Yank Replace Previous" command
```

LINES

```
"Beginning of Line" command
BREAK-LINE function
"Delete Line" command
EMPTY-LINE-P function
"End of Line" command
END-OF-LINE-P function
```

LINES (cont.)

"Kill Line" command
LINE/= function
LINE< function
LINE<= function
LINE= function
LINE> function
LINE>= function
LINE-BUFFER function
LINE-CHARACTER function
LINE-END function
LINE-LENGTH function
LINE-NEXT function
LINE-OFFSET function
LINE-PREVIOUS function
LINE-START function
LINE-STRING function
LINE-TO-REGION function
"Line to Top of Window" command
LINEP function
LINES-RELATED-P function
"New Line" command
"New LISP Line" command
"Next Line" command
"Open Line" command
"Previous Line" command

LISP SYNTAX

"Beginning of Outermost Form" command
"Close Outermost Form" command
"Delete Whitespace" command
"Describe Word" command
"Describe Word at Pointer" command
"End of Outermost Form" command
"Evaluate LISP Region" command
"Indent LISP Line" command
"Indent LISP Region" command
"Indent Outermost Form" command
"Insert Close Paren and Match" command
"LISP Comment Column" Editor variable
"LISP Evaluation Result" Editor variable
"LISP Syntax" attribute
"Kill Enclosing List" command
"Kill Next Form" command
"Kill Previous Form" command
"Kill Rest of List" command
"Move to LISP Comment" command
"New LISP Line" command
"Next Form" command

LISP SYNTAX (cont.)

NEXT-LISP-FORM function
"Previous Form" command
PREVIOUS-LISP-FORM function
"Select Enclosing Form at Pointer" command
"Select Outermost Form" command

MARKS

"Beginning of Paragraph" command
"Beginning of Window" command
BUFFER-END function
BUFFER-POINT function
"Buffer Select Mark" Editor variable
BUFFER-START function
CHARACTER-OFFSET function
COPY-MARK function
CURRENT-BUFFER-POINT function
DELETE-MARK function
END-OF-LINE-P function
"End of Paragraph" command
"End of Window" command
"Exchange Point and Select Mark" command
FIRST-LINE-P function
LAST-LINE-P function
LINE-OFFSET function
MAKE-EDITOR-STREAM-TO-MARK function
MAKE-MARK function
MARK-VISIBLE-P
MARK/= function
MARK< function
MARK<= function
MARK= function
MARK> function
MARK>= function
MARK-CHARPOS function
MARK-COLUMN function
MARK-LINE function
MARK-TYPE function
MARK-WINDOW-POSITION function
MARKP function
MOVE-MARK function
MOVE-MARK-AFTER function
MOVE-MARK-BEFORE function
MOVE-MARK-TO-POSITION function
"Move to LISP Comment" command
NEXT-CHARACTER function
"Next Form" command
"Next Line" command
"Next Paragraph" command

MARKS (cont.)

"Next Screen" command
"Next Window" command
"Page Next Window" command
"Page Previous Window" command
PAGE-OFFSET function
PREVIOUS-CHARACTER function
"Previous Form" command
"Previous Line" command
"Previous Paragraph" command
"Previous Screen" command
"Previous Window" command
REGION-END function
REGION-START function
SAME-LINE-P function
"Scroll Window Down" command
"Scroll Window Up" command
SHOW-MARK function
START-OF-LINE-P function
"What Cursor Position" command
WINDOW-POINT function
WITH-MARK macro
WITH-OUTPUT-TO-MARK macro
WORD-OFFSET function

MISCELLANEOUS

CANCEL-CHARACTER function
"Show Time" command

POINTING DEVICE

BIND-POINTER-COMMAND function
"Current Window Pointer Pattern" Editor variable
"Current Window Pointer Pattern X" Editor variable
"Current Window Pointer Pattern Y" Editor variable
"Describe Word at Pointer" command
"EDT Paste at Pointer" command
GET-POINTER-STATE function
"Information Area Pointer Pattern" Editor variable
"Information Area Pointer Pattern X" Editor variable
"Information Area Pointer Pattern Y" Editor variable
"Maybe Reset Select at Pointer" command
"Move Point and Select Region" command
"Move Point to Pointer" command
"Noncurrent Window Pointer Pattern" Editor variable
"Noncurrent Window Pointer Pattern X" Editor variable
"Noncurrent Window Pointer Pattern Y" Editor variable
POINTER-STATE-ACTION function

POINTING DEVICE (cont.)

POINTER-STATE-BUTTONS function
POINTER-STATE-P function
POINTER-STATE-TEXT-POSITION function
POINTER-STATE-WINDOW-POSITION function
"Select Enclosing Form at Pointer" command
UNBIND-POINTER-COMMAND function
"Yank at Pointer" command

PROMPTING AND TERMINAL INPUT

EDITOR-LISTEN function
EDITOR-PROMPTING-BUFFER buffer
EDITOR-READ-CHAR function
EDITOR-READ-CHAR-NOHANG function
EDITOR-UNREAD-CHAR function
"General Prompting" buffer
INFORMATION-AREA-OUTPUT-STREAM variable
LAST-CHARACTER-TYPED variable
"Prompt Alternatives" Editor variable
"Prompt Alternatives Arguments" Editor variable
"Prompt Complete String" command
"Prompt Completion" Editor variable
"Prompt Completion Arguments" Editor variable
"Prompt Default" Editor variable
"Prompt Error Message" Editor variable
"Prompt Error Message Arguments" Editor variable
PROMPT-FOR-INPUT function
"Prompt Help" command
"Prompt Help" Editor variable
"Prompt Help Arguments" Editor variable
"Prompt Help Called" Editor variable
"Prompt Read and Validate" command
"Prompt Rendition Complement" Editor variable
"Prompt Rendition Set" Editor variable
"Prompt Required" Editor variable
"Prompt Scroll Help Window" command
"Prompt Show Alternatives" command
"Prompt Start" Editor variable
"Prompt Validation" Editor variable
SIMPLE-PROMPT-FOR-INPUT function

REGIONS

"Buffer Select Region" Editor variable
COPY-REGION function
COUNT-REGION function
DELETE-AND-SAVE-REGION function
DELETE-REGION function

REGIONS (cont.)

HIGHLIGHT-REGION-P function
MAKE-EDITOR-STREAM-FROM-REGION function
MAKE-EMPTY-REGION function
MAKE-HIGHLIGHT-REGION function
MAKE-REGION function
"Maybe Reset Select at Pointer" command
"Move Point and Select Region" command
REGION-END function
REGION-READ-POINT function
REGION-START function
REGION-TO-STRING function
REGIONP function
REMOVE-HIGHLIGHT-REGION function
"Select Region Rendition Complement" Editor variable
"Select Region Rendition Set" Editor variable
"Set Select Mark" command
STRING-TO-REGION function
"Unset Select Mark" command
WITH-INPUT-FROM-REGION macro

RINGS

MAKE-RING function
RING-LENGTH function
RING-POP function
RING-PUSH function
RING-REF function
RINGP function
ROTATE-RING function

SEARCHING

"Backward Search" command
"Default Search Case" Editor variable
"EMACS Backward Search" command
"EMACS Forward Search" command
"Forward Search" command
"Last Search Direction" Editor variable
"Last Search Pattern" Editor variable
"Last Search String" Editor variable
LOCATE-PATTERN function
MAKE-SEARCH-PATTERN function
"Query Search Replace" command
REPLACE-PATTERN function

STRING TABLES

```
APROPOS-STRING-TABLE function
COMPLETE-STRING function
FIND-AMBIGUOUS function
GET-STRING-TABLE-VALUE function
MAKE-STRING-TABLE function
MAP-STRINGS function
REMOVE-STRING-TABLE-ENTRY function
STRING-TABLE-P function
```

STRING TABLES PROVIDED WITH VAX LISP

```
*EDITOR-ATTRIBUTE-NAMES*
*EDITOR-STYLE-NAMES*
*EDITOR-KEYBOARD-MACRO-NAMES*
*EDITOR-COMMAND-NAMES*
*EDITOR-BUFFER-NAMES*
*EDITOR-VARIABLE-NAMES*
```

STYLES

```
"Activate Minor Style" command
BUFFER-MAJOR-STYLE function
BUFFER-MINOR-STYLE-ACTIVE function
BUFFER-MINOR-STYLE-LIST function
"Deactivate Minor Style" command
"Default Filetype Minor Styles" Editor variable
"Default LISP Object Minor Styles" Editor variable
"Default Major Style" Editor variable
"Default Minor Styles" Editor variable
*EDITOR-STYLE-NAMES* variable
"EDT Emulation" style
FIND-STYLE function
"Major Style Activation Hook" Editor variable
MAKE-STYLE macro
"Minor Style Activation Hook" Editor variable
STYLE-NAME function
STYLEP function
STYLE-VARIABLES function
"VAX LISP" style
```

STYLES PROVIDED WITH VAX LISP

```
"EDT Emulation" style
"EMACS" style
"VAX LISP" style
```

STYLE BINDINGS, "EDT EMULATION" STYLE

"Default Window Label" Editor variable
"EDT Append" command
"EDT Back to Start of Line" command
"EDT Beginning of Line" command
"EDT Change Case" command
"EDT Cut" command
"EDT Delete Character" command
"EDT Delete Line" command
"EDT Delete Previous Character" command
"EDT Delete Previous Line" command
"EDT Delete Previous Word" command
"EDT Delete to End of Line" command
"EDT Delete Word" command
"EDT Deleted Character" Editor variable
"EDT Deleted Line" Editor variable
"EDT Deleted Word" Editor variable
"EDT Deselect" command
"EDT Direction Mode" Editor variable
"EDT End of Line" command
"EDT Move Character" command
"EDT Move Page" command
"EDT Move Word" command
"EDT Paste" command
"EDT Paste at Pointer" command
"EDT Paste Buffer" Editor variable
"EDT Query Search" command
"EDT Replace" command
"EDT Scroll Window" command
"EDT Search Again" command
"EDT Select" command
"EDT Set Direction Backward" command
"EDT Set Direction Forward" command
"EDT Special Insert" command
"EDT Substitute" command
"EDT Undelete Character" command
"EDT Undelete Line" command
"EDT Undelete Word" command
"Select Region Rendition Complement" Editor variable
"Select Region Rendition Set" Editor variable
"Word Delimiter" Editor attribute

STYLE BINDINGS, "EMACS" STYLE

"Apropos Word" command
"Backward Character" command
"EMACS Backward Search" command
"Backward Word" command
"Beginning of Buffer" command
"Beginning of Line" command

STYLE BINDINGS, "EMACS" STYLE (cont.)

"Beginning of Paragraph" command
"Beginning of Window" command
"Capitalize Word" command
"Default Window Label" Editor variable
"Delete Current Buffer" command
"Delete Next Character" command
"Delete Previous Character" command
"Delete Previous Word" command
"Delete Next Word" command
"Delete Whitespace" command
"Describe Word" command
"Downcase Word" command
"Ed" command
"Edit File" command
"End of Buffer" command
"End of Line" command
"End of Paragraph" command
"End of Window" command
"Exchange Point and Select Mark" command
"Execute Keyboard Macro" command
"Execute Named Command" command
"Exit Recursive Edit" command
"Forward Character" command
"EMACS Forward Search" command
"Forward Word" command
"Grow Window" command
"Insert File" command
"Kill Line" command
"Kill Paragraph" command
"Kill Region" command
"Line to Top of Window" command
"List Buffers" command
"Next Line" command
"Next Paragraph" command
"Next Screen" command
"Next Window" command
"New Line" command
"Open Line" command
"Page Next Window" command
"Pause Editor" command
"Previous Line" command
"Previous Paragraph" command
"Previous Screen" command
"Previous Window" command
"Query Search Replace" command
"Quoted Insert" command
"Read File" command
"Redisplay Screen" command
"Remove Current Window" command
"Remove Other Windows" command

STYLE BINDINGS, "EMACS" STYLE (cont.)

"Scroll Window Down" command
"Scroll Window Up" command
"Select Buffer" command
"Select Region Rendition Complement" Editor variable
"Select Region Rendition Set" Editor variable
"Set Select Mark" command
"Unset Select Mark" command
"Show Time" command
"Shrink Window" command
"Split Window" command
"Supply EMACS Prefix" command
"Supply Prefix Argument" command
"Transpose Previous Characters" command
"Transpose Previous Words" command
"Undo Previous Yank" command
"Upcase Word" command
"View File" command
"What Cursor Position" command
"Write Current Buffer" command
"Write Modified Buffers" command
"Write Named File" command
"Yank" command
"Yank at Pointer" command
"Yank Previous" command
"Yank Replace Previous" command

STYLE BINDINGS, "VAX LISP" STYLE

"Beginning of Outermost Form" command
"Close Outermost Form" command
"Describe Word at Pointer" command
"End of Outermost Form" command
"Evaluate LISP Region" command
"Indent LISP Line" command
"Indent Outermost Form" command
"Insert Close Paren and Match" command
"LISP Comment Column" Editor variable
"LISP Evaluation Result" Editor variable
"LISP Syntax" attribute
"Move to LISP Comment" command
"New LISP Line" command
"Next Form" command
"Previous Form" command
"Select Enclosing Form at Pointer" command
"Select Outermost Form" command
"Word Delimiter" attribute

EDITOR OBJECTS BY CATEGORY

TEXT OPERATIONS

"Backward Character" command
"Backward Page" command
"Backward Word" command
"Beginning of Paragraph" command
"Beginning of Window" command
"Buffer Right Margin" Editor variable
"Capitalize Region" command
"Capitalize Word" command
DELETE-AND-SAVE-REGION function
DELETE-CHARACTERS function
"Delete Next Character" command
"Delete Next Word" command
"Delete Previous Character" command
"Delete Previous Word" command
DELETE-REGION function
"Delete Word" command
"Downcase Region" command
"Downcase Word" command
"End of Paragraph" command
"Forward Character" command
"Forward Page" command
"Forward Word" command
"Insert Buffer" command
INSERT-CHARACTER function
"Insert File" command
INSERT-FILE-AT-MARK function
INSERT-REGION function
INSERT-STRING function
"Kill Line" command
"Kill Paragraph" command
"Kill Region" command
NEXT-CHARACTER function
"Next Paragraph" command
PREVIOUS-CHARACTER function
"Previous Paragraph" command
"Quoted Insert" command
"Self Insert" command
"Text Overstrike Mode" Editor variable
"Transpose Previous Characters" command
"Transpose Previous Words" command
"Undo Previous Yank" command
"Upcase Region" command
"Upcase Word" command
"Yank" command
"Yank at Pointer" command
"Yank Previous" command
"Yank Replace Previous" command

EDITOR OBJECTS BY CATEGORY

WINDOWS

ALTER-WINDOW-HEIGHT function
"Anchored Window Show Limit" Editor variable
"Beginning of Window" command
BUFFER-HIGHLIGHT-REGIONS function
BUFFER-WINDOWS function
CENTER-WINDOW function
CURRENT-WINDOW function
"Default Window Label" Editor variable
"Default Window Label Edge" Editor variable
"Default Window Label Offset" Editor variable
"Default Window Label Rendition" Editor variable
"Default Window Lines Wrap" Editor variable
"Default Window Rendition" Editor variable
"Default Window Truncate Char" Editor variable
"Default Window Type" Editor variable
"Default Window Width" Editor variable
"Default Window Wrap Char" Editor variable
DELETE-WINDOW function
"EDT Scroll Window" command
"End of Window" command
"Grow Window" command
HIGHLIGHT-REGION-P function
"Line to Top of Window" command
MAKE-HIGHLIGHT-REGION function
MARK-WINDOW-POSITION function
MAKE-WINDOW function
MOVE-WINDOW function
"Next Screen" command
"Next Window" command
NEXT-WINDOW function
"Page Next Window" command
"Page Previous Window" command
"Previous Screen" command
"Previous Window" command
"Prompt Scroll Help window" command
PUSH-WINDOW function
"Remove Current Window" command
REMOVE-HIGHLIGHT-REGION function
"Remove Other Windows" command
REMOVE-WINDOW function
SCROLL-WINDOW function
"Scroll Window Down" command
"Scroll Window Up" command
SHOW-WINDOW function
"Shrink Window" command
"Split Window" command
"Switch Window Hook" Editor variable
UPDATE-DISPLAY function
UPDATE-WINDOW-LABEL function
VISIBLE-WINDOWS function

WINDOWS (cont.)

WINDOW-BUFFER function
"Window Buffer Hook" Editor variable
"Window Creation Hook" Editor variable
WINDOW-CREATION-TIME function
"Window Deletion Hook" Editor variable
WINDOW-DISPLAY-COLUMN function
WINDOW-DISPLAY-END function
WINDOW-DISPLAY-ROW function
WINDOW-DISPLAY-START function
WINDOW-HEIGHT function
WINDOW-LABEL function
WINDOW-LABEL-EDGE function
WINDOW-LABEL-OFFSET function
WINDOW-LABEL-RENDITION function
WINDOW-LINES-WRAP-P function
"Window Modification Hook" Editor variable
WINDOW-POINT function
WINDOW-RENDITION function
WINDOW-TRUNCATE-CHAR function
WINDOW-TYPE function
WINDOW-WIDTH function
WINDOW-WRAP-CHAR function
WINDOWP function

APPENDIX B

EDITOR COMMANDS AND BINDINGS

This appendix lists and briefly describes all the commands supplied with the VAX LISP Editor. Key bindings, pointer bindings, and binding contexts are also listed where applicable.

This appendix is identical to Table C.1 in Appendix C of the *VAX LISP/VMS User's Guide*.

Table B-1: Editor Commands and Key Bindings

Name	Binding(s)[1]	Description
Activate Minor Style	None	Prompts for the name of a minor style and then activates that style as a minor style in the current buffer
Apropos	None	Prompts for a string, then displays the names of objects of a specified type containing that string
Apropos Word	(:STYLE "VAX LISP") ESCAPE ?	Displays the result of evaluating the APROPOS function with the word at the cursor location as the argument
Backward Character	:GLOBAL ← (:STYLE "EMACS") CTRL/B	Moves the cursor backward one character, or by the number of characters specified by the prefix argument
Backward Kill Ring	None	Rotates the kill ring backward by one element, or by the number of elements specified by the prefix argument
Backward Page	None	Moves the cursor to the previous page break, or to the preceding page break specified by the prefix argument
Backward Search	None	Prompts for a search string, then moves the cursor to the beginning of the first preceding occurrence of that string, or to the preceding occurrence specified by the prefix argument
Backward Word	(:STYLE "EMACS") ESCAPE b	Moves the cursor to the end of the previous word, or to the end of the preceding word specified by the prefix argument
Beginning of Buffer	(:STYLE "EDT Emulation") PF1 5 (:STYLE "EMACS") ESCAPE <	Moves the cursor to the beginning of the buffer
Beginning of Line	(:STYLE "EMACS") CTRL/A	Moves the cursor to the beginning of the current line, or to the beginning of the following line specified by the prefix argument
Beginning of Outermost Form	(:STYLE "VAX LISP") CTRL/X <	Moves the cursor to the beginning of the outermost form currently containing it, or, if the cursor is not currently contained in a form, to the beginning of the preceding outermost form
Beginning of Paragraph	(:STYLE "EMACS") ESCAPE A	Moves the cursor to the beginning of the current paragraph
Beginning of Window	(:STYLE "EMACS") ESCAPE ,	Moves the cursor to the top of the current window
Bind Command	None	Prompts for a command name, a key sequence to bind to the command, and a context in which to bind the key sequence, then binds the key sequence to the command
Capitalize Region	None	Capitalizes the first letter of each word in the current select region

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 ⬚ Numeric keypad keys. → ↑ Arrow keys. ○↓● 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.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Capitalize Word	(:STYLE "EMACS") [ESCAPE] c	Capitalizes the first letter of the word at the cursor location
Close Outermost Form	(:STYLE "VAX LISP") [ESCAPE]]	Completes the outermost LISP form by inserting close-parentheses characters at the cursor position
Deactivate Minor Style	None	Prompts for the name of a minor style, then deactivates that minor style in the current buffer
Delete Current Buffer	(:STYLE "EMACS") [CTRL/X] [CTRL/D]	Deletes the current buffer; for modified buffers, asks if the contents of the buffer should first be saved
Delete Line	None	Deletes everything between the cursor and the end of the current line, or to the end of the following line specified by the prefix argument
Delete Named Buffer	None	Prompts for the name of a buffer, then deletes that buffer; if the buffer is modified, asks if the contents of the buffer should first be saved
Delete Next Character	(:STYLE "EMACS") [CTRL/D]	Deletes the character following the cursor, or the number of following characters specified by the prefix argument
Delete Next Word	(:STYLE "EMACS") [ESCAPE] d	Deletes everything from the cursor position to the end of the current word, or the number of following words specified by the prefix argument
Delete Previous Character	:GLOBAL [DELETE] (:STYLE "EMACS") [DELETE]	Deletes the character preceding the cursor position, or the number of preceding characters specified by the prefix argument
Delete Previous Word	(:STYLE "EMACS") [ESCAPE] [DELETE]	Deletes everything from the cursor position to the beginning of the current word, or the number of preceding words specified by the prefix argument
Delete Whitespace	(:STYLE "EMACS") [ESCAPE] [CTRL/D]	Deletes whitespace characters following the cursor location up to the next nonwhitespace character
Delete Word	None	Deletes everything from the cursor position to the beginning of the next word, including whitespace, or deletes the number of following words specified by the prefix argument
Describe	None	Prompts for the name and type of an object, then displays a description of that object
Describe Word	(:STYLE "VAX LISP") [CTRL/?]	Calls the DESCRIBE function with the word at the cursor position as the argument
Describe Word at Pointer[2]	(:STYLE "VAX LISP") [oo↓]	Calls the DESCRIBE function with the word at the pointer position as the argument
Downcase Region	None	Makes all alphabetic characters in the current select region lower case
Downcase Word	(:STYLE "EMACS") [ESCAPE] l	Makes all alphabetic characters in the word at the cursor position lower case

---

[1] [ ] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [,] Numeric keypad keys. [→] [↑] Arrow keys. [o↓●]Pointer button transition: o button up; ● button held down; ↓ button pressed; ↑ button released; [o●o]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Ed	(:STYLE "EMACS") CTRL/X CTRL/E	Prompts for a LISP object to edit and, if the object is a symbol, whether to edit its function definition or its value
Edit File	(:STYLE "EMACS") CTRL/X CTRL/V	Prompts for the specification of a file to edit; completion and alternatives are available during your response to the prompt
EDT Append	(:STYLE "EDT Emulation") 9	Appends the current select region to the contents of the paste buffer
EDT Back to Start of Line	(:STYLE "EDT Emulation") CTRL/H and BACKSPACE[4] and F12[3]	Moves the cursor to the beginning of the current line, or to the beginning of the previous line if the cursor is already at the beginning of a line; or moves back the number of lines specified by the prefix argument
EDT Beginning of Line	(:STYLE "EDT Emulation") 0	Moves the cursor to the beginning of the next line, if the current direction is forward, or to the beginning of the current or previous line, if the current direction is backward; moves by the number of lines specified by the prefix argument
EDT Change Case	(:STYLE "EDT Emulation") PF1 1	Changes the case (lower to upper and vice versa) of all characters in the select region, or, if no select region is defined, of the character at the cursor position
EDT Cut	(:STYLE "EDT Emulation") 6 and REMOVE [3] and o↓o	Deletes the current select region and replaces the contents of the paste buffer with it
EDT Delete Character	(:STYLE "EDT Emulation") ,	Deletes the character at the cursor position and stores it in the deleted character area; deletes the number of characters specified by the prefix argument.
EDT Delete Line	(:STYLE "EDT Emulation") PF4	Deletes from the cursor position to the beginning of the next line and stores the deleted line in the deleted line area; deletes the number of lines specified by the prefix argument
EDT Delete Previous Character	(:STYLE "EDT Emulation") DELETE	Deletes the character preceding the cursor and stores it in the deleted character area; deletes the number of characters specified by the prefix argument
EDT Delete Previous Line	(:STYLE "EDT Emulation") CTRL/U	Deletes from the cursor position to the beginning of the current line or, if the cursor is at the beginning of a line, to the beginning of the previous line; stores the result in the deleted line area; deletes the number of lines specified by the prefix argument
EDT Delete Previous Word	(:STYLE "EDT Emulation") CTRL/J and LINEFEED[4] and F13[3]	Deletes from the cursor position to the beginning of the current word. or, if the cursor is between words, to the beginning of the previous word; stores the result in the deleted word area; deletes the number of lines specified by the prefix argument

[1]☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 ☐ Numeric keypad keys. → ↑ Arrow keys. o↓● Pointer button transition: o button up; ● button held down; ↓ button pressed; ↑ button released; o●o → pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3]Key available only on LK201 keyboard.

[4]Key available only on VT100 terminal.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
EDT Delete to End of Line	(:STYLE "EDT Emulation") PF1 2	Deletes from the cursor position to the end of the current line or, if the cursor is at the end of a line, to the end of the next line; stores the result in the deleted line area; deletes the number of lines specified by the prefix argument
EDT Delete Word	(:STYLE "EDT Emulation") -	Deletes from the cursor position to the beginning of the next word; stores the result in the deleted word area; deletes the number of words specified by the prefix argument
EDT Deselect	(:STYLE "EDT Emulation") PF1 .	Cancels the current select region; equivalent to "Unset Select Mark"
EDT End of Line	(:STYLE "EDT Emulation") 2	Moves the cursor to the end of the current, next, or previous line, depending on starting cursor position and current direction; moves by the number of lines specified by the prefix argument
EDT Move Character	(:STYLE "EDT Emulation") 3	Moves the cursor forward or backward by one character, according to the current direction; moves the number of characters specified by the prefix argument
EDT Move Page	(:STYLE "EDT Emulation") 7	Moves the cursor to the preceding or following page break, depending on the current direction; moves the number of pages specified by the prefix argument
EDT Move Word	(:STYLE "EDT Emulation") 1	Moves the cursor to the beginning of the next, current, or preceding word, depending on current direction and cursor starting position; moves the number of words specified by the prefix argument
EDT Paste	(:STYLE "EDT Emulation") PF1 6 and INSERT HERE[3]	Inserts the contents of the paste buffer at the cursor location
EDT Paste at Pointer[2]	(:STYLE "EDT Emulation") ●↓○	Inserts the contents of the paste buffer at the pointer cursor location
EDT Query Search	(:STYLE "EDT Emulation") PF1 PF3 and FIND[3]	Prompts for a search string and moves the cursor to the following or preceding occurrence of the string, depending on the current direction; moves to the occurrence specified by the prefix argument
EDT Replace	(:STYLE "EDT Emulation") PF1 9	Replaces the current select region with the contents of the paste buffer
EDT Scroll Window	(:STYLE "EDT Emulation") 8	Scrolls the window in the direction indicated by the current direction
EDT Search Again	(:STYLE "EDT Emulation") PF3	Searches for the next or previous occurrence of the search string that was last entered, according to the current direction

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 . Numeric keypad keys. → ↑ Arrow keys. ○↓● Pointer button transition: ○ button up; ● button held down; ↓ button pressed; ↑ button released; ○●○→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

[3] Key available only on LK201 keyboard.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
EDT Select	(:STYLE "EDT Emulation") [.] and [SELECT][3]	Places a mark at the cursor position to indicate one end of a select region; equivalent to "Set Select Mark"
EDT Set Direction Backward	(:STYLE "EDT Emulation") [5]	Sets the current direction to backward
EDT Set Direction Forward	(:STYLE "EDT Emulation") [4]	Sets the current direction to forward
EDT Special Insert	(:STYLE "EDT Emulation") [PF1] [3]	Inserts the character whose ASCII code is specified by the prefix argument at the cursor position
EDT Substitute	(:STYLE "EDT Emulation") [PF1] [ENTER]	If the cursor is located at the beginning of the current search string, replaces the search string with the contents of the paste buffer, then finds the next occurrence of the search string
EDT Undelete Character	(:STYLE "EDT Emulation") [PF1] [,]	Inserts the contents of the deleted character area at the cursor location
EDT Undelete Line	(:STYLE "EDT Emulation") [PF1] [PF4]	Inserts the contents of the deleted line area at the cursor location
EDT Undelete Word	(:STYLE "EDT Emulation") [PF1] [-]	Inserts the contents of the deleted word area at the cursor location
EMACS Backward Search	(:STYLE "EMACS") [CTRL/R]	Searches backward for the first occurrence of the search string specified in the previous command; prompts for a search string if the previous command was not a searching command; searches for the occurrence of the search string specified by the prefix argument
EMACS Forward Search	(:STYLE "EMACS") [CTRL/\]	Searches forward for the first occurrence of the search string specified in the previous command; prompts for a search string if the previous command was not a searching command; searches for the occurrence of the search string specified by the prefix argument
End Keyboard Macro	:GLOBAL [CTRL/X])	Ends the collection of keystrokes for a keyboard macro
End of Buffer	(:STYLE "EDT Emulation") [PF1] [4]	Moves the cursor to the end of the buffer
	(:STYLE "EMACS") [ESCAPE] >	
End of Line	(:STYLE "EMACS") [CTRL/E]	Moves the cursor to the end of the current line, or forward by the number of lines specified by the prefix argument and then to the end of the line
End of Outermost Form	(:STYLE "VAX LISP") [CTRL/X] >	Moves the cursor to the outermost form currently surrounding the cursor, or, if the cursor is between outermost forms, to the end of the following outermost form

[1] [ ] indicates nonprinting characters or pointer activity. [CTRL/D] Hold down [CTRL] while typing letter. [PF1] [0] [.] Numeric keypad keys. [→] [↑] Arrow keys. [○↓●] Pointer button transition: ○button up; ●button held down; ↓button pressed; ↑button released; [○●○]→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
End of Paragraph	(:STYLE "EMACS") ESCAPE e	Moves the cursor to the end of the current paragraph
End of Window	(:STYLE "EMACS") ESCAPE .	Moves the cursor to the end of the text in the current window
Evaluate LISP Region	(:STYLE "VAX LISP") CTRL/X CTRL/A	Evaluates the select region as LISP code; displays the result of the evaluation in the information area
Exchange Point and Select Mark	(:STYLE "EMACS") CTRL/X CTRL/X	Moves the cursor to the other end of the current select region, and the mark delimiting the select region to the old cursor position; in other words, preserves the select region but places the cursor at the other end of it
Execute Keyboard Macro	:GLOBAL CTRL/X CTRL/E (:STYLE "EMACS") CTRL/X e	Executes the current keyboard macro once, or the number of times specified by the prefix argument
Execute Named Command	:GLOBAL CTRL/Z and DO[3] (:STYLE "EDT Emulation") PF1 7 (:STYLE "EMACS") ESCAPE x	Prompts for the name of a command to execute; input completion and alternatives are available during your response to the prompt
Exit Editor	None	Returns control to the LISP interpreter, discarding the current Editor state; asks if modified buffers should first be saved
Exit Recursive Edit	(:STYLE "EMACS") ESCAPE CTRL/G	Terminates a recursive edit, or pauses the Editor if not doing a recursive edit
Forward Character	:GLOBAL → (:STYLE "EMACS") CTRL/F	Moves the cursor forward one character
Forward Kill Ring	None	Rotates the kill ring forward by one element, or by the number of elements specified by the prefix argument
Forward Page	None	Moves the cursor to the next page break, or to the following page break specified by the prefix argument
Forward Search	None	Prompts for a search string, then moves the cursor forward to the end of the first occurrence of the string; moves the cursor to the occurrence of the string specified by the prefix argument
Forward Word	(:STYLE "EMACS") ESCAPE f	Moves the cursor to the beginning of the next word, or the beginning of the word specified by the prefix argument
Grow Window	(:STYLE "EMACS") CTRL/X z	Increases the height of the current window by one row, or by the number of rows specified by the prefix argument
Help	:GLOBAL PF2 and HELP[3]	Displays help on your current situation

---

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 ☐ Numeric keypad keys. → ↑ Arrow keys. o↓● Pointer button transition: o button up; ● button held down; ↓ button pressed; ↑ button released; o●o → pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Kill Previous Form	None	Deletes the LISP form immediately preceding the cursor and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted form; deletes the number of preceding forms specified by the prefix argument within the current parentheses nesting level
Kill Region	(:STYLE "EMACS") CTRL/W and o↓o	Deletes the current select region and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted region
Kill Rest of List	None	Deletes the rest of the enclosing list and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted list fragment
Line to Top of Window	(:STYLE "EMACS") ESCAPE !	Moves the line containing the cursor to the top of the window
List Buffers	(:STYLE "EMACS") CTRL/X CTRL/B	Displays a list of all buffers
List Key Bindings	None	Displays a list of all visible key bindings or of all keys bound in a specified context
Maybe Reset Select at Pointer[2]	:GLOBAL ↑oo	If the pointer cursor has not moved, cancels the select region that was started with ↓oo; if the pointer cursor has moved since ↓oo, does nothing
Move Point and Select Region[2]	:GLOBAL ●oo→	Moves the text cursor with the pointer cursor, marking a select region
Move Point to Pointer[2]	:GLOBAL ●oo	Moves the text cursor to the pointer cursor
Move to LISP Comment	(:STYLE "VAX LISP") CTRL/X ;	If there is no comment on the current line, moves the cursor to the comment column and inserts a semicolon and space; if there is a comment, moves the cursor to the comment
New Line	:GLOBAL RETURN (:BUFFER "General Prompting") LINEFEED (:STYLE "EMACS") RETURN	Breaks a line at the cursor position, leaving the cursor at the start of the new line
New LISP Line	(:STYLE "VAX LISP") LINEFEED	Breaks a line at the cursor position, and indents the new line by the appropriate amount in the context of the program
Next Form	(:STYLE "VAX LISP") CTRL/X .	Moves the cursor to the end of the next form, or to the end of the following form specified by the prefix argument; does not move outside the current parentheses nesting level

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0
↓ Numeric keypad keys. → ↑ Arrow keys. o↓● Pointer button transition: o button up; ● button held down; ↓ button pressed; ↑ button released; o●o→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Help on Editor Error	:GLOBAL CTRL/X ?	Displays information on the last Editor error that occurred
Illegal Operation	None	Signals an Editor error; use to disable a key binding locally within a style or buffer
Indent LISP Line	(:STYLE "VAX LISP") TAB	Adjusts the current LISP line so that it is indented properly in the context of the program
Indent LISP Region	None	Adjusts the indentation of the LISP code in the current select region
Indent Outermost Form	(:STYLE "VAX LISP") CTRL/X TAB	Indents each line in the outermost LISP form containing the cursor
Insert Buffer	None	Prompts for a buffer name, then inserts the contents of the specified buffer at the cursor location
Insert Close Paren and Match	(:STYLE "VAX LISP"))	Inserts a close-parenthesis character at the cursor location and highlights the corresponding open-parenthesis character
Insert File	(:STYLE "EMACS") CTRL/X CTRL/I	Prompts for a file specification, then inserts the contents of the file at the cursor location; input completion and alternatives are available during your response to the prompt
Kill Enclosing List	None	Deletes the LISP list that encloses the cursor and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted list; deletes the number of enclosing lists specified by the prefix argument
Kill Line	(:STYLE "EMACS") CTRL/K	Deletes the rest of the current line and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted line; deletes the number of lines specified by the prefix argument
Kill Next Form	None	Deletes the LISP form immediately following the cursor and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted form; deletes the number of following forms specified by the prefix argument within the current parentheses nesting level
Kill Paragraph	(:STYLE "EMACS") ESCAPE k	Deletes the rest of the current paragraph and adds it to the current kill-ring region if the previous command was a kill-ring command, or creates a new kill-ring region to hold the deleted paragraph; deletes the number of paragraphs specified by the prefix argument

---

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 ⏺ Numeric keypad keys. → ↑ Arrow keys. ⊙↓⬤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.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Next Line	:GLOBAL ⬇ (:STYLE "EMACS") CTRL/N	Moves the cursor to the next line or down by the number of lines specified by the prefix argument, keeping the cursor in the same column if possible
Next Paragraph	(:STYLE "EMACS") ESCAPE n	Moves the cursor to the beginning of the next paragraph, or to the following paragraph specified by the prefix argument
Next Screen	:GLOBAL NEXT SCREEN[3]	Moves the window down in the buffer by one screenful, or by as many screenfuls as are specified by the prefix argument
Next Window	:GLOBAL CTRL/X CTRL/N (:STYLE "EMACS") CTRL/X p	Selects another window on the screen to be the current window; eventually circulates through all the windows on the screen
Open Line	(:STYLE "EDT Emulation") PF1 0 (:STYLE "EMACS") CTRL/O	Breaks a line at the cursor location, leaving the cursor at the end of the old line
Page Next Window	(:STYLE "EMACS") ESCAPE CTRL/V	Scrolls the next window on the screen down one page; or, if a prefix argument is supplied, scrolls the next window by that many rows
Pause Editor	:GLOBAL CTRL/X CTRL/Z (:STYLE "EMACS") CTRL/G	Saves the Editor state and returns control to the LISP interpreter
Previous Form	(:STYLE "VAX LISP") CTRL/X ,	Moves the cursor to the beginning of the previous form, or to the beginning of the preceding form specified by the prefix argument; does not move outside the current parentheses nesting level
Previous Line	:GLOBAL ⬆ (:STYLE "EMACS") CTRL/P	Moves the cursor to the previous line, or up by the number of lines specified by the prefix argument; keeps the cursor in the same column if possible
Previous Paragraph	(:STYLE "EMACS") ESCAPE p	Moves the cursor to the end of the previous paragraph, or to the end of the preceding paragraph specified by the prefix argument
Previous Screen	:GLOBAL PREV SCREEN[3] (:STYLE "EMACS") ESCAPE v	Moves the cursor up in the buffer by one screenful, or by as many screenfuls as are specified by the prefix argument
Previous Window	(:STYLE "EMACS") CTRL/X n	Makes another window on the screen into the current window; eventually circulates through all windows on the screen
Prompt Complete String	(:BUFFER "General Prompting") CTRL/	Attempts to complete your response to a prompt, based on what you have typed already and the choices available in the situation
Prompt Help	(:BUFFER "General Prompting") PF2	Displays information on whatever is being prompted for

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 ⬚ Numeric keypad keys. → ⬆ Arrow keys. o⬇● Pointer button transition: o button up; ● button held down; ⬇ button pressed; ⬆ button released; o●○→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[3] Key available only on LK201 keyboard.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Prompt Read and Validate	(:BUFFER "General Prompting") RETURN and ENTER	Used to terminate prompt input
Prompt Scroll Help Window	(:BUFFER "General Prompting") CTRL/V	Scrolls the Help window down while another buffer is current; supplied to allow you to scroll the Help window while responding to a prompt
Prompt Show Alternatives	(:BUFFER "General Prompting") PF1 PF2	Displays a list of alternatives that can be entered in response to the current prompt, based on what you have typed already
Query Search Replace	(:STYLE "EMACS") ESCAPE q	Prompts for a search string and a replacement; offers a number of options at each replacement opportunity
Quoted Insert	:GLOBAL CTRL/X \ (: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 by the number of rows specified by the prefix argument
Scroll Window Up	(:STYLE "EMACS") ESCAPE z	Scrolls the current window up in the buffer by one row, or by the number of rows specified by the prefix argument
Select Buffer	(:STYLE "EMACS") CTRL/X b	Prompts for a buffer name, then makes that buffer the current buffer; creates a new buffer if necessary
Select Enclosing Form at Pointer	(:STYLE "VAX LISP") ↓oo	Places the form enclosing the cursor in a select region; if the cursor is already in a select region, expands the region to the next outermost form
Select Outermost Form	(:STYLE "VAX LISP") CTRL/X CTRL/	Makes the outermost LISP form containing the cursor into a select region
Self Insert	:GLOBAL All graphic characters	Inserts the last character typed at the cursor location
Set Screen Height	None	Sets the screen height to the number of rows specified by the prefix argument; prompts for height if no prefix argument is defined

---

[1] ☐ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 ☐ Numeric keypad keys. → ↑ Arrow keys. o↓● Pointer button transition: o button up; ● button held down; ↓ button pressed; ↑ button released; o●o → pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
Set Screen Width	None	Sets the screen width to the number of columns specified by the prefix argument; prompts for the width if no prefix argument is defined
Set Select Mark	(:STYLE "EDT Emulation") `.`	Places a mark at the cursor position to indicate one end of a select region
	(:STYLE "EMACS") `CTRL/`	
Show Time	(:STYLE "EMACS") `CTRL/X` `CTRL/T`	Displays the time and date in the information area
Shrink Window	(:STYLE "EMACS") `CTRL/X` `CTRL/Z`	Shrinks the current window by one row, or by the number of rows specified by the prefix argument
Split Window	(:STYLE "EMACS") `CTRL/X` 2	Splits the current window into two identical windows
Start Keyboard Macro	:GLOBAL `CTRL/X` (	Starts collecting keystrokes for a keyboard macro, replacing any unnamed keyboard macro that already exists
Start Named Keyboard Macro	None	Prompts for a name, then starts collecting keystrokes for a keyboard macro; the resulting keyboard macro is catalogued under the name you give and can be treated as a command
Supply EMACS Prefix	(:STYLE "EMACS") `CTRL/U`	Sets the prefix argument to four, if no prefix argument was defined, or to four times its former value, if a prefix argument was defined
Supply Prefix Argument	(:STYLE "EDT Emulation") `PF1` `PF1`	Prompts for a prefix argument; if a prefix argument is already defined, multiplies it by the number you enter
	(:STYLE "EMACS") `ESCAPE` `CTRL/U`	
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` `.`	Cancels the current select region
	(:STYLE "EMACS") `ESCAPE` `CTRL/`	
Upcase Region	None	Changes all alphabetic characters in the current select region to upper case
Upcase Word	(:STYLE "EMACS") `ESCAPE` u	Changes all alphabetic characters in the word at the cursor location to upper case
View File	(:STYLE "EMACS") `CTRL/X` `CTRL/F`	Prompts for a file specification, then reads the specified file into a read-only buffer

[1] `☐` indicates nonprinting characters or pointer activity. `CTRL/D` Hold down `CTRL` while typing letter. `PF1` `0` `.` Numeric keypad keys. `→` `↑` Arrow keys. `○↓●` 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.

Table B-1 (cont.)

Name	Binding(s)[1]	Description
What Cursor Position	(:STYLE "EMACS") CTRL/X =	Displays information about the cursor location
Write Current Buffer	(:STYLE "EMACS") CTRL/X s	Writes out the current buffer; creates a new file version, or updates the LISP symbol whose function or value slot is being edited
Write Modified Buffers	(:STYLE "EMACS") CTRL/X CTRL/M	Performs the "Write Current Buffer" operation for each buffer that has been modified
Write Named File	(:STYLE "EMACS") CTRL/X CTRL/W	Prompts for a file specification, then creates a file having that specification from the contents of the current buffer
Yank	(:STYLE "EMACS") CTRL/Y	Inserts the current kill-ring region at the cursor location; inserts as many copies as are specified by the prefix argument
Yank at Pointer[2]	(:STYLE "EMACS") ●↓○	Inserts the current kill-ring region at the pointer cursor location
Yank Previous	(:STYLE "EMACS") ESCAPE y	Rotates the kill ring forward, then inserts the new current kill-ring region at the cursor location; inserts as many copies as are specified by the prefix argument
Yank Replace Previous	(:STYLE "EMACS") ESCAPE CTRL/Y	Deletes the previously yanked region, rotates the kill ring forward, and inserts the new current kill-ring region at the cursor location; inserts as many copies as are specified by the prefix argument

---

[1] □ indicates nonprinting characters or pointer activity. CTRL/D Hold down CTRL while typing letter. PF1 0 , Numeric keypad keys. → ↑ Arrow keys. ○↓● Pointer button transition: ○button up; ●button held down; ↓button pressed; ↑button released; ○●○→ pointer movement with buttons in specified state. Pointer buttons invoke command only when pointer cursor is in the current window.

[2] Available only on VAXstation.

B-13

APPENDIX C

BOUND KEYS AND KEY SEQUENCES

This appendix lists all the keys and key sequences that are bound to commands in the VAX LISP Editor, along with the context in which each binding occurs.

If a key or key sequence is bound in more than one context, all but one of the bindings are "shadowed" or inaccessible. The Editor searches through the contexts in the following order and accepts the first binding it encounters:

- Current buffer

- Minor styles active in the current buffer, beginning with the most recently activated

- Major style active in the current buffer

- Global Editor context

This appendix is identical to Table C.2 in Appendix C of the *VAX LISP/VMS User's Guide*.

Table C-1: Editor Key Bindings

Key(s)	Context and Command
Single Keys	
CTRL/SPACE	(:BUFFER "General Prompting") Prompt Complete String
	(:STYLE "EMACS") Set Select Mark
CTRL/A	(:STYLE "EMACS") Beginning of Line
CTRL/B	(:STYLE "EMACS") Backward Character

BOUND KEYS AND KEY SEQUENCES

Table C-1 (cont.)

Key(s)	Context and Command
CTRL/D	(:STYLE "EMACS") Delete Next Character
CTRL/E	(:STYLE "EMACS") End of Line
CTRL/F	(:STYLE "EMACS") Forward Character
CTRL/G	(:STYLE "EMACS") Pause Editor
CTRL/H or BACKSPACE	(:STYLE "EDT Emulation") EDT Back to Start of Line
TAB or CTRL/I	(:STYLE "VAX LISP") Indent LISP Line
CTRL/J or LINEFEED	(:BUFFER "General Prompting") New Line (:STYLE "VAX LISP") New LISP Line (:STYLE "EDT Emulation") EDT Delete Previous Word
CTRL/K	(:STYLE "EMACS") Kill Line
CTRL/L	(:STYLE "EMACS") Redisplay Screen
RETURN or CTRL/M	(:BUFFER "General Prompting") Prompt Read and Validate (:STYLE "EMACS") New Line :GLOBAL New Line
CTRL/N	(:STYLE "EMACS") Next Line
CTRL/O	(:STYLE "EMACS") Open Line
CTRL/P	(:STYLE "EMACS") Previous Line
CTRL/R	(:STYLE "EMACS") Backward Search
CTRL/T	(:STYLE "EMACS") Transpose Previous Characters
CTRL/U	(:STYLE "EMACS") Supply EMACS Prefix (:STYLE "EDT Emulation") EDT Delete Previous Line
CTRL/V	(:BUFFER "General Prompting") Prompt Scroll Help Window (:STYLE "EMACS") Next Screen
CTRL/W	(:STYLE "EMACS") Kill Region (:STYLE "EDT Emulation") Redisplay Screen
CTRL/Y	(:STYLE "EMACS") Yank

Table C-1 (cont.)

Key(s)	Context and Command
CTRL/Z	(:STYLE "EMACS") Scroll Window Down :GLOBAL Execute Named Command
CTRL/∧	(:STYLE "EMACS") EMACS Forward Search
CTRL/?	(:STYLE "VAX LISP") Describe Word
DELETE or <X]	(:STYLE "EMACS") Delete Previous Character (:STYLE "EDT Emulation") Delete Previous Character :GLOBAL Delete Previous Character
)	(:STYLE "VAX LISP") Insert Close Paren and Match
keypad 0	(:STYLE "EDT Emulation") EDT Beginning of Line
keypad 1	(:STYLE "EDT Emulation") EDT Move Word
keypad 2	(:STYLE "EDT Emulation") EDT End of Line
keypad 3	(:STYLE "EDT Emulation") EDT Move Character
keypad 4	(:STYLE "EDT Emulation") EDT Set Direction Forward
keypad 5	(:STYLE "EDT Emulation") EDT Set Direction Backward
keypad 6	(:STYLE "EDT Emulation") EDT Cut
keypad 7	(:STYLE "EDT Emulation") EDT Move Page
keypad 8	(:STYLE "EDT Emulation") EDT Scroll Window
keypad 9	(:STYLE "EDT Emulation") EDT Append
keypad .	(:STYLE "EDT Emulation") Set Select Mark
keypad ENTER	(:BUFFER "General Prompting") Prompt Read and Validate
keypad ,	(:STYLE "EDT Emulation") EDT Delete Character
keypad -	(:STYLE "EDT Emulation") EDT Delete Word
keypad PF2	(:BUFFER "General Prompting") Prompt Help (:STYLE "EDT Emulation") Help :GLOBAL Help
keypad PF3	(:STYLE "EDT Emulation") EDT Search Again

Table C-1 (cont.)

Key(s)	Context and Command
keypad PF4	(:STYLE "EDT Emulation") EDT Delete Line
Up arrow	:GLOBAL Previous Line
Down arrow	:GLOBAL Next Line
Right arrow	:GLOBAL Forward Character
Left arrow	:GLOBAL Backward Character
All graphic characters	:GLOBAL Self Insert

Single Keys -- LK201 Keyboard Only

F12	(:STYLE "EDT Emulation") EDT Back to Start of Line
F13	(:STYLE "EDT Emulation") EDT Delete Previous Word
HELP	:GLOBAL Help
DO	:GLOBAL Execute Named Command
FIND	(:STYLE "EDT Emulation") EDT Query Search
INSERT HERE	(:STYLE "EDT Emulation") EDT Paste
REMOVE	(:STYLE "EDT Emulation") EDT Cut
SELECT	(:STYLE "EDT Emulation") EDT Select
PREV SCREEN	:GLOBAL Previous Screen
NEXT SCREEN	:GLOBAL Next Screen

Two-Key Sequences Starting with CTRL/X

CTRL/X CTRL/SPACE	(:STYLE "VAX LISP") Select Outermost Form
CTRL/X CTRL/A	(:STYLE "VAX LISP") Evaluate LISP Region
CTRL/X CTRL/B	(:STYLE "EMACS") List Buffers
CTRL/X CTRL/D	(:STYLE "EMACS") Delete Current Buffer
CTRL/X CTRL/E	(:STYLE "EMACS") Ed :GLOBAL Execute Keyboard Macro

Table C-1 (cont.)

Key(s)	Context and Command
CTRL/X CTRL/F	(:STYLE "EMACS") View File
CTRL/X TAB or CTRL/X CTRL/I	(:STYLE "EMACS") Insert File (:STYLE "VAX LISP") Indent Outermost Form
CTRL/X RETURN or CTRL/X CTRL/M	(:STYLE "EMACS") Write Modified Buffers
CTRL/X CTRL/N	:GLOBAL Next Window
CTRL/X CTRL/R	(:STYLE "EMACS") Read File :GLOBAL Remove Current Window
CTRL/X CTRL/T	(:STYLE "EMACS") Show Time
CTRL/X CTRL/V	(:STYLE "EMACS") Edit File
CTRL/X CTRL/W	(:STYLE "EMACS") Write Named File
CTRL/X CTRL/X	(:STYLE "EMACS") Exchange Point and Select Mark
CTRL/X CTRL/Z	(:STYLE "EMACS") Shrink Window :GLOBAL Pause Editor
CTRL/X (	:GLOBAL Start Keyboard Macro
CTRL/X)	:GLOBAL End Keyboard Macro
CTRL/X ,	(:STYLE "VAX LISP") Previous Form
CTRL/X .	(:STYLE "VAX LISP") Next Form
CTRL/X 1	(:STYLE "EMACS") Remove Other Windows
CTRL/X 2	(:STYLE "EMACS") Split Window
CTRL/X ;	(:STYLE "VAX LISP") Move to LISP Comment
CTRL/X <	(:STYLE "VAX LISP") Beginning of Outermost Form
CTRL/X >	(:STYLE "VAX LISP") End of Outermost Form
CTRL/X =	(:STYLE "EMACS") What Cursor Position
CTRL/X ?	:GLOBAL Help on Editor Error

BOUND KEYS AND KEY SEQUENCES

Table C-1 (cont.)

Key(s)	Context and Command
CTRL/X \	:GLOBAL Quoted Insert
CTRL/X b	(:STYLE "EMACS") Select Buffer
CTRL/X d	(:STYLE "EMACS") Remove Current Window
CTRL/X e	:GLOBAL Execute Keyboard Macro
CTRL/X n	(:STYLE "EMACS") Previous Window
CTRL/X p	(:STYLE "EMACS") Next Window
CTRL/X q	(:STYLE "EMACS") Quoted Insert
CTRL/X s	(:STYLE "EMACS") Write Current Buffer
CTRL/X z	(:STYLE "EMACS") Grow Window

Two-Key Sequences Starting with ESCAPE

Key(s)	Context and Command
ESCAPE CTRL/SPACE	(:STYLE "EMACS") Unset Select Mark
ESCAPE CTRL/D	(:STYLE "EMACS") Delete Whitespace
ESCAPE CTRL/G	(:STYLE "EMACS") Exit Recursive Edit
ESCAPE CTRL/U	(:STYLE "EMACS") Supply Prefix Argument
ESCAPE CTRL/V	(:STYLE "EMACS") Page Next Window
ESCAPE CTRL/W	(:STYLE "EMACS") Undo Previous Yank
ESCAPE CTRL/Y	(:STYLE "EMACS") Yank Previous Replace
ESCAPE !	(:STYLE "EMACS") Line to Top of Window
ESCAPE ,	(:STYLE "EMACS") Beginning of Window
ESCAPE .	(:STYLE "EMACS") End of Window
ESCAPE <	(:STYLE "EMACS") Beginning of Buffer
ESCAPE >	(:STYLE "EMACS") End of Buffer
ESCAPE ?	(:STYLE "VAX LISP") Apropos Word
ESCAPE]	(:STYLE "VAX LISP") Close Outermost Form

Table C-1 (cont.)

Key(s)	Context and Command
ESCAPE a	(:STYLE "EMACS") Beginning of Paragraph
ESCAPE b	(:STYLE "EMACS") Backward Word
ESCAPE c	(:STYLE "EMACS") Capitalize Word
ESCAPE d	(:STYLE "EMACS") Delete Next Word
ESCAPE e	(:STYLE "EMACS") End of Paragraph
ESCAPE f	(:STYLE "EMACS") Forward Word
ESCAPE k	(:STYLE "EMACS") Kill Paragraph
ESCAPE l	(:STYLE "EMACS") Downcase Word
ESCAPE n	(:STYLE "EMACS") Next Paragraph
ESCAPE p	(:STYLE "EMACS") Previous Paragraph
ESCAPE q	(:STYLE "EMACS") Query Search Replace
ESCAPE t	(:STYLE "EMACS") Transpose Previous Words
ESCAPE u	(:STYLE "EMACS") Upcase Word
ESCAPE v	(:STYLE "EMACS") Previous Screen
ESCAPE x	(:STYLE "EMACS") Execute Named Command
ESCAPE y	(:STYLE "EMACS") Yank Previous
ESCAPE z	(:STYLE "EMACS") Scroll Window Up
ESCAPE DELETE or ESCAPE <X]	(:STYLE "EMACS") Delete Previous Word

Two-Key Sequences Starting with Keypad PF1

keypad PF1 0	(:STYLE "EDT Emulation") Open Line
keypad PF1 1	(:STYLE "EDT Emulation") EDT Change Case
keypad PF1 2	(:STYLE "EDT Emulation") EDT Delete to End of Line
keypad PF1 3	(:STYLE "EDT Emulation") EDT Special Insert

Table C-1 (cont.)

Key(s)	Context and Command
keypad PF1 4	(:STYLE "EDT Emulation") End of Buffer
keypad PF1 5	(:STYLE "EDT Emulation") Beginning of Buffer
keypad PF1 6	(:STYLE "EDT Emulation") EDT Paste
keypad PF1 7	(:STYLE "EDT Emulation") Execute Named Command
keypad PF1 9	(:STYLE "EDT Emulation") EDT Replace
keypad PF1 .	(:STYLE "EDT Emulation") Unset Select Mark
keypad PF1 ENTER	(:STYLE "EDT Emulation") EDT Substitute
keypad PF1 ,	(:STYLE "EDT Emulation") EDT Undelete Character
keypad PF1 -	(:STYLE "EDT Emulation") EDT Undelete Word
keypad PF1 PF1	(:STYLE "EDT Emulation") Supply Prefix Argument
keypad PF1 PF3	(:BUFFER "General Prompting") Prompt Show Alternatives
keypad PF1 PF3	(:STYLE "EDT Emulation") EDT Query Search
keypad PF1 PF4	(:STYLE "EDT Emulation") EDT Undelete Line

APPENDIX D

FUNCTION KEYS AND KEYPAD KEYS

This appendix provides information needed to specify the function keys and keypad keys on DIGITAL keyboards in LISP code. The table below lists the actual character sequence generated by each function key and keypad key.

You can include these character sequences in a LISP sequence (vector, list, or string) and pass the LISP sequence as the *key-sequence* argument in a call to BIND-COMMAND.

This list is identical to Table 3-4 of the *VAX LISP/VMS User's Guide*.

Table D-1: 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

Table D-1 (cont.)

Key	Characters Generated
Arrow Keys (LK201 and VT100)	

Up Arrow	#\ESCAPE #\[#\A
Down Arrow	#\ESCAPE #\[#\B
Right Arrow	#\ESCAPE #\[#\C
Left Arrow	#\ESCAPE #\[#\D

Function, HELP, and DO keys (LK201)

F6	#\ESCAPE #\[#\1 #\7 #\~
F7	#\ESCAPE #\[#\1 #\8 #\~
F8	#\ESCAPE #\[#\1 #\9 #\~
F9	#\ESCAPE #\[#\2 #\0 #\~
F10	#\ESCAPE #\[#\2 #\1 #\~
F11	#\ESCAPE #\[#\2 #\3 #\~
F12	#\ESCAPE #\[#\2 #\4 #\~
F13	#\ESCAPE #\[#\2 #\5 #\~
F14	#\ESCAPE #\[#\2 #\6 #\~
HELP (F15)	#\ESCAPE #\[#\2 #\8 #\~
DO (F16)	#\ESCAPE #\[#\2 #\9 #\~
F17	#\ESCAPE #\[#\3 #\1 #\~
F18	#\ESCAPE #\[#\3 #\2 #\~
F19	#\ESCAPE #\[#\3 #\3 #\~
F20	#\ESCAPE #\[#\3 #\4 #\~

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 #\~

INDEX

INDEX

This index covers Parts I, II, and III, and Appendix A. Page numbers appear in the following forms:

c-n   (for example, 2-13)   Refers to a chapter and page in Part I

n   (for example, 25)   Refers to a page in Part II or III

A-n   (for example, A-22)   Refers to a page in Appendix A

Note that the pages in Parts II and III are numbered sequentially.

INDEX

INDEX

INDEX

INDEX

INDEX