VAX Text Processing Utility Manual: Part I

Order Number: AA–PBTMA–TE

June 1990

This manual describes the elements of the VAX Text Processing Utility (VAXTPU). It is intended as a reference manual for experienced programmers.

Revision/Update Information: This document supersedes the *VAX Text Processing Utility Manual* for VMS Version 5.2.

Software Version: VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop–VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	digital™

The following is a third-party trademark:

PostScript is a registered trademark of Adobe Systems Incorporated.

ZK4350

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

EXAMPLES

FIGURES

TABLES

Preface

Intended Audience

This manual is intended for experienced programmers who know at least one computer language. Some features of VAXTPU, for example, the callable interface and the built-in procedure FILE_PARSE, are intended for system programmers who have a good understanding of VMS system concepts. Relevant documents about the VMS operating system are listed under Associated Documents.

Document Structure

This manual consists of six expository chapters, a reference section, and seven appendixes. The six chapters discuss the following topics:

- Chapter 1 contains an overview of VAXTPU.
- Chapter 2 provides detailed information on VAXTPU data types.
- Chapter 3 discusses the lexical elements of VAXTPU. These include the character set, identifiers, variables, constants, and reserved words, such as VAXTPU language statements.
- Chapter 4 describes VAXTPU program development.
- Chapter 5 describes how to invoke VAXTPU.
- Chapter 6 discusses the VAXTPU screen manager and screen management issues.

The VAXTPU Reference Section (Chapter 7) provides detailed descriptions of the VAXTPU built-in procedures.

The seven appendixes are organized as follows:

- Appendix A contains sample procedures written in VAXTPU.
- Appendix B contains sample procedures written in DECwindows VAXTPU.
- Appendix C describes terminals supported by VAXTPU.
- Appendix D lists each VAXTPU message, its abbreviation, and its severity level.
- Appendix E contains the DEC Multinational Character Set.
- Appendix F lists the file types that VAXTPU supports.
- Appendix G discusses EVE$BUILD, a tool that enables you to layer applications onto EVE or build new VAXTPU applications.

Associated Documents

To learn how to use the Extensible VAX Editor (EVE), see the *Guide to VMS Text Processing*. For reference information on EVE commands, see *VMS EVE Reference Manual*.

The *VMS Utility Routines Manual* contains a chapter presenting the VAXTPU callable interface.

The *VMS System Messages and Recovery Procedures Reference Manual* contains the VAXTPU messages, as well as an explanation and suggested user action for each message. The messages are listed alphabetically by the abbreviation for the message text.

The *Overview of VMS Documentation* briefly describes all VMS system documentation, defining the intended audience for each manual and providing a synopsis of each manual's contents.

The *VMS DCL Dictionary* describes the VMS DCL commands that help you create, copy, and print files containing VAXTPU programs.

The *VMS System Services Volume* describes system services.

The *Introduction to VMS System Routines* and *VMS Utility Routines Manual* describe utility routines.

The *VMS Run-Time Library Routines Volume* describes routines of the run-time library.

The *VMS Record Management Services Manual* describes VMS RMS services.

Conventions

The following conventions are used in this document:

mouse	The term *mouse* is used to refer to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)
Return	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the Return key. (Note that the Return key is not usually shown in syntax statements or in all examples; however, assume that you must press the Return key after entering a command or responding to a prompt.)

CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
red ink	Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on. For online versions, user input is shown in **bold**.
. . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
{ }	Braces enclose a mandatory portion of the format of a built-in procedure or lexical element. When braces enclose a stacked list of items, you must choose one of the items. For example: { string / range }
[[ ]]	Double brackets in examples show an optional portion of the format of a built-in procedure or lexical element. When double brackets enclose an item or series of items, you can select one of the items. For example: [[string / range]]
[[, . . .]]	Double brackets enclosing a comma and horizontal ellipsis mean that you can repeat the preceding item one or more times, separating two or more items with commas. For example: `parameter [[,...]]`
[]	Delimits a case label. Single brackets do not indicate optional parameters in this manual.
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

UPPERCASE letters and special symbols	Uppercase letters and special symbols in syntax descriptions and sample procedures indicate VAXTPU reserved words and predeclared identifiers, and other user input that must be typed exactly as shown. For example:
	`PROCEDURE`
	`UNDERLINE`
	String constants are shown in lowercase to emphasize that they are strings. However, they, too, must be typed exactly as shown.

lowercase letters	Lowercase letters in syntax descriptions and sample procedures represent elements that you must replace according to the description in the text. For example, when a data type, such as buffer, is used in a syntax example, replace it with the variable name assigned to the data item when it was created. In the following assignment statement, *my_buffer_variable* is the variable name assigned to the buffer you are creating:
	
	`my_buffer_variable :=`
	`CREATE_BUFFER ('my_buf_name', 'my_file_name')`
	
	To specify a buffer as a parameter for a VAXTPU built-in procedure, use the variable for the buffer. For example, to erase the contents of the buffer created in the preceding statement, enter the following:
	
	`ERASE (my_buffer_variable)`

user_	Many of the sample procedures in this manual have the prefix *user_* as a part of the procedure name. Digital suggests that you replace the prefix *user* with your initials. This or some other convention helps to ensure that the variables and procedure names that you create do not conflict with either VAXTPU built-in procedure names, or the procedure names and variables of your editing interface.
filespec	Mnemonic for file specification.

VAXTPU Tutorial Section

1 Overview of the VAX Text Processing Utility

This chapter presents an overview of the VAX Text Processing Utility (VAXTPU). In particular, this chapter addresses the following questions:

- What is VAXTPU?
- What is DECwindows VAXTPU?
- What is EVE?
- What is the VAXTPU language?
- What hardware does VAXTPU support?
- How do I start using VAXTPU?
- How do I learn more about VAXTPU?

1.1 What Is VAXTPU?

VAXTPU is a high-performance, programmable, text processing utility. It is designed as a tool to aid application and system programmers in developing tools that manipulate text. Programmers, for example, can use VAXTPU to design an editor for a specific environment. The utility includes a high-level procedural language, a compiler, an interpreter, and an editing interface written in VAXTPU.

VAXTPU provides the following special features:

- Multiple buffers
- Multiple windows
- Multiple subprocesses
- Keystroke and buffer change journaling
- Text processing in batch mode
- Insert or overstrike text entry
- Free or bound cursor motion
- Learn sequences
- Pattern matching
- Key definition
- Procedural language
- Callable interface

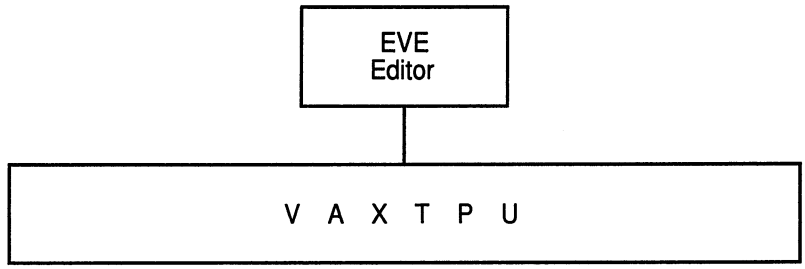
The editor or other application that you layer on top of VAXTPU becomes the interface between you and VAXTPU. You must either use the Extensible VAX Editor (EVE) or create your own interface to access VAXTPU.

Overview of the VAX Text Processing Utility

1.1 What Is VAXTPU?

You can think of VAXTPU as a base on which to layer text processing applications. The Extensible VAX Editor (EVE) is a good example of an application written in VAXTPU and layered on VAXTPU. See Figure 1–1.

Figure 1–1 VAXTPU as a Base for EVE



ZK-6545-GE

1.2 What Is DECwindows VAXTPU?

VAXTPU can display text in two environments: a character cell terminal, such as a VT320, or a bit-mapped workstation running the DECwindows windowing software.

DECwindows VAXTPU provides additional built-in procedures to interact with the DECwindows environment, including the ability to create and manipulate widgets, global selection, input focus, and the clipboard. For information about how to invoke the DECwindows version of VAXTPU, see Chapter 5. If you try to use the DECwindows features of VAXTPU on a character-cell terminal, VAXTPU returns an error.

Note that the windows referred to as DECwindows are not the same as VAXTPU windows. For more information about the difference between DECwindows windows and VAXTPU windows, see Chapter 4.

1.2.1 DECwindows VAXTPU and DECwindows Features

The DECwindows environment has a number of toolkits and libraries containing routines for creating and manipulating DECwindows interfaces. For example, DECwindows routines allow you to create and manipulate clipboard entries, global selections, and widgets. For an overview of the DECwindows libraries and toolkits, see the *VMS DECwindows Guide to Application Programming*.

DECwindows VAXTPU contains a number of built-in procedures that provide access to the routines in the DECwindows libraries and toolkits.

Using these DECwindows VAXTPU built-in procedures, you can create and manipulate various features of a DECwindows interface from within a VAXTPU program. For a list of the kinds of widgets you can create and manipulate using VAXTPU built-in procedures, see Chapter 4. In most cases, you use VAXTPU DECwindows built-in procedures without needing to know what DECwindows routine a given built-in procedure calls.

You cannot directly call DECwindows routines (such as XUI Toolkit or Xlib Toolkit routines) from within a program written in the VAXTPU language. To use a DECwindows routine in a VAXTPU program, you can use one or more of the following techniques:

- Use a VAXTPU built-in procedure that calls a DECwindows routine. Examples of such VAXTPU built-in procedures include the following:
  - CREATE_WIDGET
  - DELETE (WIDGET)
  - MANAGE_WIDGET
  - REALIZE_WIDGET
  - SEND_CLIENT_MESSAGE
  - SET (CLIENT_MESSAGE)
  - SET (DRM_HIERARCHY)
  - SET (ICON_NAME)
  - SET (ICON_PIXMAP)
  - SET (ICONIFY_PIXMAP)
  - SET (MAPPED_WHEN_MANAGED)
  - SET (WIDGET)
  - SET (WIDGET_CALL_DATA)
  - SET (WIDGET_CALLBACK)
  - UNMANAGE_WIDGET

  For more information about how to use the DECwindows built-ins in VAXTPU, see the individual built-in descriptions in the VAXTPU Reference Section. For more information about the types of widget resource values supported by VAXTPU, see Chapter 4.
- Using a compiled language that follows the VMS calling standard, write a function calling the desired XUI Toolkit routine. You can then use the built-in procedure CALL_USER in your VAXTPU program to invoke the program written in the non-VAXTPU language. For more information about using the built-in procedure CALL_USER, see the VAXTPU Reference Section.
- Using a compiled language that follows the VMS calling standard, write a program calling the desired XUI Toolkit routine. You can then invoke VAXTPU from the program using the VAXTPU callable interface. For more information about using the VAXTPU callable interface, see the *VMS Utility Routines Manual*.

The DECwindows version of VAXTPU does not provide access to all of the features of DECwindows. For example, there are no VAXTPU built-in procedures to handle floating-point numbers or to manipulate entities such as lines, curves, and fonts.

However, the DECwindows version of VAXTPU allows you to create a wide variety of widgets, to designate callback routines for those widgets, to fetch and set geometry and text-related resources of the widgets, and to perform other functions related to creating a DECwindows application. For example, the DECwindows EVE editor is a text processing interface created with DECwindows VAXTPU.

1.2.2 DECwindows VAXTPU and the DECwindows User Interface Language

You can use VAXTPU programs with DECwindows User Interface Language (UIL) files just as you would use programs in any other language with UIL files. For an example of a VAXTPU program and a UIL file designed to be used together, see the description of the CREATE_WIDGET built-in in the VAXTPU Reference Section. For more information about using UIL files in conjunction with programs written in other languages, see the *VMS DECwindows Guide to Application Programming*.

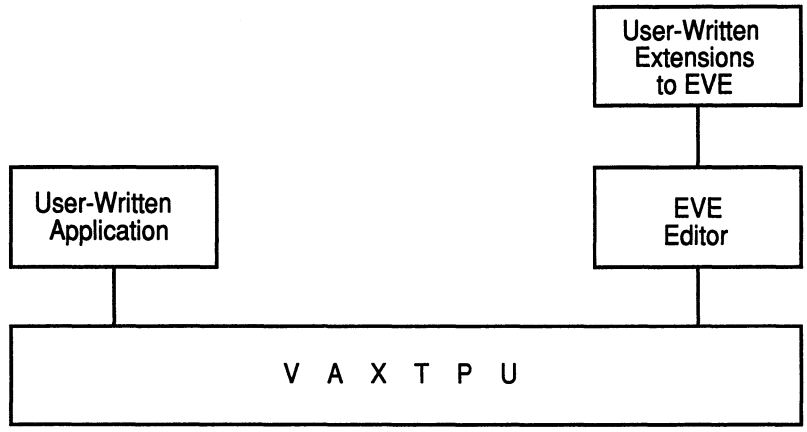
1.3 What Is EVE?

The Extensible VAX Editor (EVE) is the editor provided with VAXTPU. EVE is easy to learn and to use. Many of EVE's editing functions are accessed by pressing a single key on the EVE keypad. EVE is also a powerful and efficient editor, which makes it attractive to experienced users of text editors. The more advanced editing functions are accessible by entering commands on the EVE command line. Many of the special features of VAXTPU (such as multiple windows) are available with EVE commands. Other VAXTPU features can be accessed by entering VAXTPU statements from within EVE. EVE has both a character-cell and a DECwindows interface. To use EVE's DECwindows interface, you must be using a bit-mapped terminal or workstation.

EVE is a fully functional editor. However, it is designed to make customization easy. You can use either VAXTPU statements or EVE commands to tailor EVE to your editing style.

You can write extensions for EVE or you can write a completely separate interface for VAXTPU. See Figure 1–2.

Figure 1–2 VAXTPU as a Base for User-Written Interfaces



Extensions to EVE can be implemented with a VAXTPU command file (VAXTPU source code), with a VAXTPU section file (compiled VAXTPU code in binary form), or with an initialization file (commands in a format processed by the application layered on VAXTPU). Because a VAXTPU section file is already compiled, startup time for your editor or application is shorter using a section file than using a command file or an initialization file. For more information on using startup files, see Section 1.6.2.

To implement an editor or application that is entirely user written, use a section file. See Chapter 4 for information on VAXTPU command files, section files, and initialization files. See Appendix G for information on layering applications on VAXTPU.

For tutorial information on EVE, see the *Guide to VMS Text Processing*. For reference information on EVE commands, see the *VMS EVE Reference Manual*.

1.4 The VAXTPU Language

VAXTPU is a high-level, procedural programming language that allows you to perform text processing tasks. The VAXTPU language can be viewed as the most basic component of VAXTPU. To access the features of VAXTPU, write a program in the VAXTPU language and then use the utility to compile and execute the program. A program written in VAXTPU can be as simple as a single statement, or as complex as the section file that implements EVE.

The VAXTPU language is block structured and is easy to learn and use. VAXTPU language features include a large number of data types, relational operators, error interception, looping and case statements, and built-in procedures that simplify development or extension of an editor or application. Comments are indicated with a single comment character (!), so that you can document your procedures easily. There are

also capabilities for debugging procedures with user-written debugging programs.

1.4.1 VAXTPU Data Types

The VAXTPU language has an extensive set of data types. Data types are used to interpret the meaning of the contents of a variable. Unlike many languages, the VAXTPU language has no declarative statement to enforce which data type must be assigned to a variable. A variable in VAXTPU assumes a data type when it is used in an assignment statement. For example, the following statement assigns a string data type to the variable *this_var*:

```
this_var := 'This can be a string of your choice.';
```

The following statement assigns a window data type to the variable *x*. The window occupies 15 lines on the screen, starting at line 1, and the status line is off (not displayed).

```
x := CREATE_WINDOW (1, 15, OFF);
```

Many of the VAXTPU data types (for example, learn and pattern) are different from the data types usually found in programming languages. Following is a list of VAXTPU keywords used to specify data types:

- ARRAY — A structure for a collection of elements.
- BUFFER — A collection of text records. You can think of a buffer as an area in which to perform editing operations.
- INTEGER — An integer. The range of valid integer values in VAXTPU is –2,147,483,648 to 2,147,483,647.
- KEYWORD — A reserved word that has special meaning to the VAXTPU compiler.
- LEARN — A sequence of VAXTPU keystrokes.
- MARKER — A character position within a buffer. You can think of a marker as a placemark in a buffer.
- PATTERN — One or more sequences of characters. The pattern operators and the pattern built-in procedures return this data type as a result. Patterns are used with the built-in procedure SEARCH to locate specific text within a buffer.
- PROCESS — A VMS subprocess.
- PROGRAM — The compiled form of a sequence of VAXTPU executable statements.
- RANGE — All of the text that occurs between and including two markers.
- STRING — A character string.
- UNSPECIFIED — The initial state of a global variable after the code containing the variable declaration has been compiled.
- WINDOW — A subdivision of the screen. You can think of a window as an area in which to view a portion of the text in a buffer.

- WIDGET — A widget is a structure used as an interaction mechanism by which users give input to an application or receive messages from an application.

See Chapter 2 of this manual for a discussion of VAXTPU data types.

1.4.2 VAXTPU Language Declarations

VAXTPU language declarations include the following:

- Module declaration (MODULE/IDENT/ENDMODULE)
- Procedure declaration (PROCEDURE/ENDPROCEDURE)
- Constant declaration (CONSTANT)
- Global variable declaration (VARIABLE)
- Local variable declaration (LOCAL)

See Chapter 3 of this manual for a discussion of VAXTPU language declarations.

1.4.3 VAXTPU Language Statements

VAXTPU language statements include the following:

- Assignment statement (:=)
- Repetitive statement (LOOP/EXITIF/ENDLOOP)
- Conditional statement (IF/THEN/ELSE/ENDIF)
- Case statement (CASE/ENDCASE)
- Error statement (ON_ERROR/ENDON_ERROR)

See Chapter 3 of this manual for a discussion of VAXTPU language statements.

1.4.4 VAXTPU Built-In Procedures

The VAXTPU language has many built-in procedures that perform functions such as screen management, key definition, text manipulation, and program execution.

You can use built-in procedures to create your own procedures. You can also invoke built-in procedures from within EVE. See the VAXTPU Reference Section for a description of each of the VAXTPU built-in procedures.

1.4.5 User-Written Procedures

You can write your own procedures that combine VAXTPU language statements and calls to VAXTPU built-in procedures. VAXTPU procedures can return values and can be recursive. After you write a procedure and compile it, you use the procedure name to invoke it.

When writing a procedure, follow these guidelines:

- Start each procedure with the word PROCEDURE, followed by the procedure name of your choice.
- End each procedure with the word ENDPROCEDURE.
- Place a semicolon after each statement or built-in call if the statement or call is followed by another statement or call.

  Note that if the statement or call is *not* followed by another statement or call, the semicolon is not necessary.

Example 1–1 is a sample procedure that uses VAXTPU language statements (PROCEDURE/ENDPROCEDURE) and built-in procedures (POSITION, BEGINNING_OF, and CURRENT_BUFFER) to move the current character position to the beginning of the current buffer. The procedure displays a message with the MESSAGE built-in and obtains the name of the current buffer with the GET_INFO built-in.

Example 1–1 Sample User-Written Procedure

```
! This procedure moves the editing
! position to the top of the buffer

PROCEDURE user_top

   POSITION (BEGINNING_OF (CURRENT_BUFFER));
   MESSAGE ("Now in buffer" + GET_INFO (CURRENT_BUFFER, "name"));

ENDPROCEDURE;
```

Once you have compiled this procedure, you can invoke it with the name *user_top*. For information about writing procedures, see Chapter 3 and Chapter 4.

1.5 Terminals Supported by VAXTPU

VAXTPU runs on all VAX computers, and supports screen-oriented editing on the Digital VT300-, VT200-, and VT100-series terminals, as well as on other video display terminals that respond to the ANSI control functions.

One of the major goals in the design of VAXTPU is fast performance for screen-oriented editing. Optimum screen-oriented editing performance occurs when you run VAXTPU from VT300-series, VT220-series, and VT100-series terminals. Some video terminal hardware does not allow optimum VAXTPU performance. See Appendix C for a list of hardware characteristics that may adversely affect VAXTPU's performance.

Although you cannot use the screen-oriented features of VAXTPU on a VT52 terminal, on hardcopy terminals, or on foreign terminals that do not respond to ANSI control functions, you can run VAXTPU on these terminals with a line mode style of editing. For information on how to implement this style of editing, see the description of the /NODISPLAY qualifier in Chapter 5 and the sample line mode editor in Appendix A.

1.6 Invoking VAXTPU

To invoke VAXTPU from DCL, type the command EDIT/TPU, optionally followed by the name of your file. For example:

```
$  EDIT/TPU text_file.lis
```

This command opens TEXT_FILE.LIS for editing. Note that you can specify only one input file on the command line. You can include additional files from within VAXTPU later in your editing session with the built-in procedure READ_FILE or the EVE command GET FILE.

Digital suggests that you create a symbol like the following one to simplify invoking EVE:

```
$  EVE == "EDIT/TPU"
```

When you invoke VAXTPU with the preceding command, you are normally placed in EVE, the default editor. However, your system manager may have overridden this default.

1.6.1 Using EDIT/TPU Command Qualifiers

You can use qualifiers with the EDIT/TPU command. The qualifiers control such items as recovery from an interrupted session and the initialization files that set attributes of the application layered on VAXTPU. Qualifiers for the EDIT/TPU command are listed in Table 1–1.

Table 1–1 Qualifiers to the DCL Command EDIT/TPU

Qualifier	Default
/[NO]COMMAND[=command-file]	/COMMAND=TPU$COMMAND.TPU
/[NO]CREATE	/CREATE
/[NO]DEBUG[=debug-file]	/NODEBUG
/[NO]DISPLAY[= { CHARACTER_CELL | DECWINDOWS }]	/DISPLAY=CHARACTER_CELL
/[NO]INITIALIZATION[=init-file]	/INITIALIZATION=EVE$INIT.EVE
/[NO]INTERFACE[= { CHARACTER_CELL | DECWINDOWS }]	/INTERFACE=CHARACTER_CELL
/[NO]JOURNAL[=journal-file]	/JOURNAL
/[NO]MODIFY	/MODIFY

(continued on next page)

Table 1–1 (Cont.) Qualifiers to the DCL Command EDIT/TPU

Qualifier	Default
/[NO]OUTPUT[=output-file]	/OUTPUT=input-file
/[NO]READ_ONLY	/NOREAD_ONLY
/[NO]RECOVER	/NORECOVER
/[NO]SECTION[=section-file]	/SECTION
/START_POSITION=(row[,column])	/START_POSITION=(1,1)
/WORK[=work-file]	/WORK=TPU$WORK.TPU$WORK
/[NO]WRITE	/WRITE

For descriptions of the EDIT/TPU command qualifiers, see Chapter 5.

1.6.2 Using Startup Files

Command files and section files can create or customize a VAXTPU editor or application. Another kind of file, the initialization file, can customize EVE or other layered applications, using EVE or other application-specific commands, settings, and key bindings.

A command file is a file containing VAXTPU source code. A command file has the file type TPU. It is used with the VAXTPU qualifier /COMMAND=*filespec*. VAXTPU tries to read a command file unless you specify /NOCOMMAND. The default command file is the file called TPU$COMMAND.TPU in your current directory, if such a file exists. You can specify a different file by defining the logical name TPU$COMMAND.

A section file is the compiled form of VAXTPU source code. It is a binary file that has the default file type TPU$SECTION. It is used with the qualifier /SECTION=*filespec*. The default section file is TPU$SECTION.TPU$SECTION in the area SYS$SHARE. VMS is shipped with the systemwide logical name TPU$SECTION defined as EVE$SECTION. This definition causes the EVE editor to be invoked by default when you use the DCL command EDIT/TPU. You must specify a different section file (for example, /SECTION= my_section_file) or /NOSECTION if you do not want to use the EVE interface.

Note: When you invoke VAXTPU with the /NOSECTION qualifier, VAXTPU does not use any binary file to provide an interface. Even the RETURN and DELETE keys are not defined. Use /NOSECTION when you are running a standalone command file or when you are creating a new section file and do not want the procedures, variables, and definitions from an existing section file to be included. See Chapter 4 and Chapter 5 for more information on /NOSECTION.

An initialization file contains commands for a VAXTPU-based application. For example, an initialization file for EVE can contain commands defining keys or setting margins. Initialization files are extremely easy to create, but they cause VAXTPU to start up somewhat more slowly than section and command files do. To invoke an initialization file, use the qualifier

/INITIALIZATION. For more information on using initialization files, see the *Guide to VMS Text Processing* and Chapter 4.

You can use either a command file or a section file, or both, to customize or extend an existing interface. A command file is generally used for minor customization of an interface. Because startup time is faster with a section file, a section file is generally used when the customization is lengthy or complex, or when you are creating an interface that is not layered on an existing editor or application. You can use an initialization file only if your application supports the use of such a file.

The source files for EVE are in SYS$EXAMPLES. To see a list of the EVE source files, type the following at the DCL prompt:

```
$ DIRECTORY SYS$EXAMPLES:EVE$*.TPU
```

If you cannot find these files on your system, see your system manager.

Chapter 4 describes how to write and process command files and section files.

1.7 Using Journal Files

VAXTPU offers two ways to recover your work in case of a system failure:

- Keystroke journaling
- Buffer change journaling

In keystroke journaling, VAXTPU keeps track of each keystroke made by the user during a session, regardless of which buffer is in use when the user presses the key. If a system interruption occurs during a session, the user can use the /JOURNAL and /RECOVER qualifiers to reconstruct the work done during the session. For more information on recovery using a keystroke journal file, see Section 5.4.7 and the *VMS EVE Reference Manual.*

Buffer change journaling creates a separate journal file for each text buffer. The application can use the enhanced SET (JOURNALING) built-in to direct VAXTPU to establish and maintain a separate journal file for any buffer or buffers created during the session. The application programmer or user can also use the SET (JOURNALING) built-in to turn buffer change journaling off or on for a given buffer during a session.

In the buffer's journal file, VAXTPU keeps track of the following record attributes (and any changes made to them):

- Left margin setting
- Modifiability or unmodifiability
- Display value

The journal file also tracks:

- Characters inserted in and deleted from a record (including the location where the change took place)

- Records inserted in and deleted from a buffer (including the location where the change took place)

For more information on record attributes and display values, see the descriptions of the SET (RECORD_ATTRIBUTE) and SET (DISPLAY_VALUE) built-in procedures in the VAXTPU Reference Section.

Note that buffer change journaling does *not* keep a record of all keystrokes performed while editing a given buffer.

You can use both keystroke and buffer change journaling at the same time (except on DECwindows, where you can use *only* buffer change journaling). To turn on keystroke journaling, the application uses the JOURNAL_OPEN built-in.

The application layered on VAXTPU, not the VAXTPU engine, determines what kind of journaling is turned on and under what conditions. Table 1–2 shows the journaling behavior established by EVE, which is the VAXTPU default editor.

Table 1–2 Journaling Behavior Established by EVE

Qualifier	Effect on Keystroke Journaling	Effect on Buffer Change Journaling
None specified	Disabled	Enabled
/JOURNAL	Disabled	Enabled
/JOURNAL = *filename*	Enabled	Enabled
/NOJOURNAL	Disabled	Disabled. Note, however, that you can use SET (JOURNALING) to enable buffer change journaling even if /NOJOURNAL was specified.

To determine whether buffer change journaling is turned on, use a statement similar to the following:

```
status := GET_INFO (buffer_name, "journaling");
```

To determine the name of the keystroke journal file, use a statement similar to the following:

```
filename := GET_INFO (SYSTEM, "journal_file");
```

Caution: Journal files contain a record of *all* information being edited. Therefore, when editing files containing secure or confidential data, be sure to keep the journal files secure as well.

1.7.1 Buffer Change Journal File Naming Algorithm

By default, VAXTPU creates the buffer change journal file name by using the following algorithm:

1 Convert all characters in the buffer name that are not alphanumeric, dollar sign, underscore, or hyphen to underscores.

2 Truncate the resulting file name to 39 characters.

3 Add the file type .TPU$JOURNAL.

For example, a buffer named TEST.BAR has a default journal file name of TEST_BAR.TPU$JOURNAL.

VAXTPU puts all journal files in the directory defined by the logical name TPU$JOURNAL. By default, this logical is defined as SYS$SCRATCH. You can reassign this logical name. For example, if you want journal files written to the current default directory, define TPU$JOURNAL as [].

1.8 Learning More About VAXTPU

This manual is a reference volume for experienced programmers who want to program in VAXTPU. The manual assumes that you are familiar with programming concepts and VMS system concepts. Even though VAXTPU is a language that is easy to read and learn, you must study the language to use it successfully.

The suggested path for learning to use VAXTPU is to read the documentation describing EVE first if you are not familiar with that editor. The chapter describing the EVE interface in the *Guide to VMS Text Processing* contains tutorial material for new EVE users. It also contains material for more experienced users of text editors and explains how to use VAXTPU to extend the EVE interface.

When you are familiar with EVE, you may want to extend or customize it. Study the source code to see which procedures, variables, and key definitions the editor uses. Then write VAXTPU procedures to implement your extensions. Make sure that the VAXTPU procedures you write to customize or extend the editor do not conflict with procedures or variables that EVE uses.

When you have successfully compiled and executed the VAXTPU procedures shown in the *Guide to VMS Text Processing*, use this manual to learn more about the VAXTPU language. In this manual, Chapter 2, on VAXTPU data types; Chapter 3, on lexical elements of the VAXTPU language; and the VAXTPU Reference Section, on VAXTPU built-in procedures, describe the elements of the VAXTPU language. Chapter 5 tells you how to invoke VAXTPU with the procedures and programs you have developed.

To help you learn about the VAXTPU language, this manual contains many examples of VAXTPU procedures and programs. Every built-in procedure in the VAXTPU Reference Section has an example that is a simple, one-line VAXTPU statement using the built-in procedure. Many of the descriptions of the built-in procedures in the VAXTPU Reference Section also have a short sample procedure that uses the built-in procedure in an appropriate context. Appendix A contains longer sample procedures that perform useful editing tasks. These procedures are merely samples; adapt them for your own use. You must substitute an appropriate value for any item in lowercase in sample procedures and syntax examples.

1.8 Learning More About VAXTPU

Some system programmers may not want to follow the suggested path of learning about VAXTPU by studying and extending EVE. If you want to design your own VAXTPU-based editor or application rather than using EVE, see Chapter 4.

2 VAXTPU Data Types

A data type is a group of elements that "belong together;" the elements are all formed in the same way and are treated uniformly. The data type of a variable determines the operations that can be performed on it. The VAXTPU data types are represented by the following keywords:

- ARRAY
- BUFFER
- INTEGER
- KEYWORD
- LEARN
- MARKER
- PATTERN
- PROCESS
- PROGRAM
- RANGE
- STRING
- UNSPECIFIED
- WIDGET
- WINDOW

Data types are used to interpret the contents of a variable. Unlike many programming languages, VAXTPU permits any variable to have any type of data as a value. VAXTPU has no declaration statement to restrict the type of data that can be assigned to a variable. VAXTPU variables take on a data type when they are placed on the left-hand side of an assignment statement. The right-hand side of the assignment statement determines the data type of the variable.

Although you can construct variables freely, VAXTPU built-in procedures require that their parameters be of specific data types. Each built-in procedure can operate only on certain data types. Some built-in procedures return a value of a certain data type when they are executed. The following sections describe the VAXTPU data types.

2.1 Array

An array is a structure for storing and manipulating a group of elements. These elements can be of any data type. You create arrays with the built-in procedure CREATE_ARRAY. For example, the following statement creates the array *new_array*:

```
new_array := CREATE_ARRAY;
```

You can delete arrays with the built-in procedure DELETE.

When you create an array, you can optionally direct VAXTPU to allocate a specified number of integer-indexed array elements. VAXTPU processes this block of preallocated elements very quickly. You can direct VAXTPU to create such a block of elements only at the time you create the array. The following statement creates the array *int_array*, directs VAXTPU to allocate 10 sequential, integer-indexed elements to the array, and specifies that the lowest index value should be 1:

```
int_array := CREATE_ARRAY (10, 1);
```

Regardless of whether you specify a preallocated block of elements, you can always add array elements dynamically. Dynamically added elements can be of any data type except learn, pattern, program, or unspecified. You can mix the data types of indexes in an array.

In the following code fragment, the array *mix_array* is created and the integer 1 is stored in the array element indexed by the marker *mark1*.

```
mix_array := CREATE_ARRAY;
mark1 := MARK (NONE);
mix_array {mark1} := 1;
mix_array {"Kansas"} := "Toto";
```

You can index dynamic elements with integers, even if this means that the array ends up with more integer-indexed elements than you specified when you created the array. Note, however, that VAXTPU does not process dynamically added integer-indexed elements as quickly as it processes preallocated elements.

To refer to an array element, use the name of an existing array variable followed by the array index enclosed in braces { } or parentheses ( ). For example, if you had created an array and stored it in the variable *my_array*, the following would be valid element names:

```
my_array{2}
my_array("fred")
```

To create an element dynamically for an existing array, simply use the new element as the target of an assignment statement. For example, the following statement creates the element "*string1*" in the array *my_array* and assigns to the element the string "*Topeka*":

```
my_array{"string1"} := "Topeka";
```

In the following example, the first statement creates an integer-indexed array, *int_array*. The array has 10 elements; the first element starts at index 1. The second statement stores a string in the first integer-indexed element of the array. The third statement stores a buffer in the eighth

element of the array. The fourth statement adds an integer-indexed element dynamically. This new element contains a string.

```
int_array := CREATE_ARRAY (10, 1);
int_array {1} := "Store a string in the first element";
int_array {8} := CURRENT_BUFFER;
int_array {42} := "This is a dynamically created element.";
```

If you assign a value to an element that has not yet been created, then that element is dynamically created and both the index and the value are stored. Subsequent references to that element index return the stored value.

In most cases, if you reference an element that has not yet been created and you do not assign a value to the nonexistent element, VAXTPU does not create the element. VAXTPU simply returns the data type unspecified. However, if you reference a nonexistent element by passing the nonexistent element to a procedure, VAXTPU actually adds a new element to the array, giving the element the index you pass to the procedure. VAXTPU assigns to this new element the data type unspecified.

You can delete an element in the array by assigning the data type unspecified to the element. For example, the following statement deletes the element *my_array {"fred"}*:

```
my_array {"fred"} := TPU$K_UNSPECIFIED;
```

The following code fragment shows how you can find all the indexes in an array:

```
the_index := GET_INFO (the_array, "FIRST");

LOOP
   EXITIF the_index = TPU$K_UNSPECIFIED;
   .
   .
   .
   the_index := GET_INFO (the_array, "NEXT");
ENDLOOP;
```

Note: VAXTPU does not guarantee the order in which it will return the array indexes. Future versions of VAXTPU may return the indexes in a different order than the current version.

2.2 Buffer

A buffer is a work space for manipulating text. A buffer can be empty or it can contain text records. You can have multiple buffers. A value of the buffer data type is returned by the built-in procedures CREATE_BUFFER, CURRENT_BUFFER, and GET_INFO. CREATE_BUFFER is the only built-in procedure that creates a new buffer. CURRENT_BUFFER and GET_INFO return pointers to existing buffers.

The following statement makes the variable *my_buf* a variable of type buffer:

```
my_buf := CREATE_BUFFER ("my_buffer");
```

When you use a buffer as a parameter for VAXTPU built-in procedures, you must use as the parameter the variable to which you assigned the buffer. For example, if you want to erase the contents of the buffer created in the preceding statement, enter the following:

```
ERASE (my_buf);
```

In this statement, *my_buf* is the identifier for the variable *my_buf*. The string "my_buffer" is the name associated with the buffer. The distinction between the name of the buffer variable and the name of the buffer can be useful when developing an application layered on VAXTPU. For example, the application can manipulate a given buffer (such as the main buffer in EVE) using an internal buffer name such as *main_buffer*. However, the application can associate the name of the user's input file with the buffer, making it easier for the user to remember which buffer contains the contents of a given file.

If you want to delete the buffer itself, use the built-in procedure DELETE with the buffer variable as the parameter.

More than one buffer variable can represent the same buffer. The following statement causes both *my_buf* and *old_buf* to point to the same buffer:

```
old_buf := my_buf;
```

A buffer remains in VAXTPU's internal list of buffers even when there are no variables pointing to it. You can use the built-in procedure GET_INFO to retrieve buffers from VAXTPU's internal list.

Creating a buffer does not cause the information contained in the buffer to become visible on the screen. The buffer must be associated with a window that is mapped to the screen for the buffer contents to be visible. Editing can take place in a buffer even if the buffer is not mapped to a window on the screen.

The current buffer contains the active editing point. The editing point can be different from the cursor position, and often each is in a different location. When the current buffer is associated with a visible window (one that is mapped to the screen), the editing point and the cursor position are usually the same.

A line in a buffer can contain up to 960 characters. This limit is subject to change in future versions. If you try to create a line that is longer than 960 characters, VAXTPU truncates the inserted text and inserts only the amount that fills the line to 960 characters. If you try to read a file containing lines longer than 960 characters, VAXTPU truncates from such lines all characters after the 960th character.

A single buffer can be associated with 0 to 255 windows for editing purposes. It is often useful to have a buffer visible in two windows so that you can look at two separate parts of the same file. For example, you could display a set of declarations in one window and code that uses the declarations in another window. Edits made to a buffer show up in all windows to which that buffer is mapped and in which the editing point is visible.

2.3 Integer

VAXTPU uses the integer data type to represent numeric data. VAXTPU performs only integer arithmetic. The type integer consists of the whole number values ranging from –2,147,483,648 to 2,147,483,647. In VAXTPU, an integer constant is a sequence of decimal digits; no commas or decimal points are allowed.

The following example assigns a value of the integer data type to the variable *x*:

```
x := 12345;
```

VAXTPU also supports binary, octal, and hexadecimal integers. Binary integers are preceded by *%b* or *%B*, octal by *%o* or *%O*, and hexadecimal by *%x* or *%X*. Thus, all the following statements are acceptable:

```
x := %B10000;
x := %o20;
x := %X130;
x := 12345;
```

2.4 Keyword

Keywords are reserved words in VAXTPU that have special meaning to the compiler.

To see a list of all VAXTPU keywords, use the SHOW (KEYWORDS) built-in.

Keywords are used in the following ways:

- As parameters for VAXTPU built-in procedures (ALL, BLINK, PF2, and so forth). The first parameter of the built-in procedure SET is always a keyword (for instance, PAD, SCROLLING, STATUS_LINE).
- As values returned by VAXTPU built-in procedures, such as CURRENT_DIRECTION, KEY_NAME, LAST_KEY, READ_KEY, and GET_INFO. For example, the call GET_INFO (window, "status_video") has the following keywords as possible return values:
  - BLINK
  - BOLD
  - NONE
  - REVERSE
  - SPECIAL_GRAPHICS
  - UNDERLINE
- As pattern directives. The following keywords fall into this category:
  - ANCHOR
  - BUFFER_BEGIN
  - BUFFER_END

— LINE_BEGIN
— LINE_END
— PAGE_BREAK
— REMAIN
— UNANCHOR

These keywords are described in the VAXTPU Reference Section because they behave like built-in procedures.

- To specify the VAXTPU data types (BUFFER, MARKER, LEARN, and so forth).
- To report WARNING or ERROR status conditions (TPU$_BADMARGINS, TPU$_CREATEFAIL, TPU$_NOEOBSTR, and so forth).
- To pass the names of keys to VAXTPU procedures. See Table 2–1 for information on keywords used to refer to keys.

Table 2–1 shows the correspondence between keywords used as VAXTPU key names and the keys on the VT300, VT200, and VT100 series of keyboards. Note that it is not necessarily advisable to define a key or control sequence just because there is a VAXTPU keyword for the key or sequence. Also, because they are special to the VMS terminal driver, Digital recommends that you avoid defining the following control characters and function key:

- CTRL/C
- CTRL/O
- CTRL/Q
- CTRL/S
- CTRL/T
- CTRL/X
- CTRL/Y
- F6

Table 2–1 Keywords Used for Key Names

VAXTPU Key Name	VT300-Series, VT200-Series Key	VT100 Key
PF1	PF1	PF1
PF2	PF2	PF2
PF3	PF3	PF3
PF4	PF4	PF4
KP0, KP1, . . . , KP9	0, 1, . . . , 9	0, 1, . . . , 9

(continued on next page)

Table 2–1 (Cont.) Keywords Used for Key Names

VAXTPU Key Name	VT300-Series, VT200-Series Key	VT100 Key
PERIOD	.	.
COMMA	,	,
MINUS	–	–
ENTER	ENTER	ENTER
UP	Up arrow	Up arrow
DOWN	Down arrow	Down arrow
LEFT	Left arrow	Left arrow
RIGHT	Right arrow	Right arrow
E1	Find / E1	
E2	Insert Here / E2	
E3	Remove / E3	
E4	Select / E4	
E5	Prev Screen / E5	
E6	Next Screen / E6	
HELP	Help / F15	
DO	Do / F16	
F6, F7, . . . , F20	F6, F7, . . . , F20	
NUL_KEY	CTRL/SPACE	CTRL/SPACE
TAB_KEY	Tab	Tab
RET_KEY	RETURN	RETURN
DEL_KEY	<⌫	DELETE
LF_KEY	CTRL/J	Line Feed
BS_KEY	CTRL/H	Backspace
CTRL_A_KEY	CTRL/A[1]	CTRL/A[1]
CTRL_B_KEY	CTRL/B	CTRL/B
.	.	.
.	.	.
.	.	.
CTRL_Z_KEY	CTRL/Z	CTRL/Z

[1]CTRL/A means pressing the CTRL key simultaneously with the A key. *A* and *a* produce the same results.

2.5 Learn

A learn sequence is a collection of VAXTPU keystrokes. The built-in procedure LEARN_BEGIN causes VAXTPU to start collecting keystrokes and the built-in procedure LEARN_END stops the collection of keystrokes

and returns a value of the learn data type as a result. The following example assigns a learn data type to the variable *x*:

```
LEARN_BEGIN (EXACT);
      .
      .
      .
x := LEARN_END;
```

All keystrokes that you enter between the built-in procedures LEARN_BEGIN and LEARN_END are stored in the variable *x*. The keyword EXACT specifies that, when the learn sequence is replayed, the input (if any) for the built-in procedures READ_CHAR, READ_KEY, and READ_LINE (if used in the learn sequence) will be the same as the input entered when the learn sequence was created. If you specify NO_EXACT, a replay of a learn sequence containing keys which invoke the built-in procedures READ_LINE, READ_KEY, or READ_CHAR looks for new input. For information on replaying a learn sequence, see the descriptions of LEARN_BEGIN and LEARN_END in the VAXTPU Reference Section.

The execution of a learn sequence can be interrupted by the built-in LEARN_ABORT. For information on using LEARN_ABORT, see the description of LEARN_ABORT in the VAXTPU Reference Section. To enable your user-written VAXTPU procedures to work successfully with learn sequences, you must observe the following coding rules when you write procedures that you or someone else can bind to a key:

- The procedure should return true and false as needed to indicate whether execution of the procedure completed successfully.
- The procedure should invoke the LEARN_ABORT built-in in case of error.

These practices help prevent a learn sequence from finishing if the learn sequence calls the user-written procedure and the procedure is not executed successfully.

Note that a procedure that does not explicitly return a value returns 0 by default, thus aborting a learn sequence.

Note: Learn sequences do not include mouse input or characters inserted in a widget.

2.6 Marker

A marker is a reference point in a buffer. You can think of a marker as a "place mark." To create a marker, use the MARK built-in.

The following example assigns a value of the marker data type to the variable *x*:

```
x := MARK (NONE);
```

After this statement is executed, the variable *x* contains the character position where the editing point was located when the statement was executed. The editing point is the point in a buffer at which most editing operations are carried out. For more information on the editing point, see Chapter 6.

You can cause a marker to be displayed with varying video attributes (BLINK, BOLD, REVERSE, UNDERLINE). The keyword NONE in the preceding example specifies that the marker does not have any video attributes.

When you use the MARK built-in, VAXTPU puts the marker on the buffer's editing point. The editing point is not necessarily the same as the window's cursor position. See Chapter 6 for more information on the difference between the buffer's editing point and the window's cursor position.

A marker can be either **bound** or **free**. Free markers are useful for establishing place marks in locations that do not contain characters, such as locations before the beginning of a line, after the end of a line, in the white space created by a tab, or below the end of a buffer. By placing a free marker in such a location, you make it possible to establish the editing point at that location without inserting padding space characters that could complicate later operations such as FILL.

A marker is bound if there is a character in the position marked by the editing point at the time you create the marker. A bound marker is tied to the character on which it is created. If you move the character to which a marker is bound, the marker moves with the character. If you delete the character to which a marker is bound, VAXTPU binds the marker to the nearest character or to the end of the line if that is closer than any character.

To force the creation of a bound marker, use the MARK built-in with any of its parameters except FREE_CURSOR. This operation creates a bound marker even if the editing point is beyond the end of a line, before the beginning of a line, in the middle of a tab, or beyond the end of a buffer. To create a bound marker in a location where there is no character, VAXTPU fills the space between the marker and the nearest character with padding space characters.

A marker is usually free if all of the following conditions are true:

- You used MARK (FREE_CURSOR) to create the marker.
- There was no character in the position marked by the editing point at the time you created the marker.
- Nothing has happened to cause the marker to become bound.

The following paragraphs explain each of these conditions in more detail.

If you use the built-in MARK (FREE_CURSOR) and there is a character in the position marked by the editing point, the marker is bound even though you specify otherwise. Once a marker becomes bound, it remains bound throughout its existence. To determine whether a marker is bound, use the following GET_INFO call:

```
GET_INFO (marker_variable, "bound");
```

VAXTPU keeps track of the location of a free marker by measuring the distance between the marker and the character nearest to the marker. If you move the character from which VAXTPU measures distance to a free marker, the marker moves too. VAXTPU preserves a uniform

distance between the character and the marker. If you collapse white space containing one or more free markers (for example, if you delete a tab or use the APPEND_LINE built-in), VAXTPU preserves the markers and binds them to the nearest character.

If you use the POSITION built-in to establish the editing point at a free marker, the marker remains free and the editing point is also said to be *free*; that is, the editing point is not bound to a character. For more information on characteristics of the editing point, see Section 6.3. Some operations cause VAXTPU to fill the space between a free marker and the nearest character with padding space characters, thereby converting the free marker to a bound marker. For example, if you type text into the buffer when the editing point is detached, VAXTPU inserts padding space characters between the nearest character and the editing point. Using any of the following built-in procedures when the editing point is detached also causes VAXTPU to perform padding:

- APPEND_LINE
- COPY_TEXT
- CURRENT_CHARACTER
- CURRENT_LINE
- CURRENT_OFFSET
- ERASE_CHARACTER
- ERASE_LINE
- MOVE_HORIZONTAL
- MOVE_VERTICAL
- MOVE_TEXT
- SELECT
- SELECT_RANGE
- SPLIT_LINE

Example 2–1 shows how to suppress padding while using these built-ins. The example assumes that the editing point is free. The code in this example assigns the string representation of the current line to the variable *bat* without adding padding blanks to the buffer.

To remove a marker, use the built-in procedure DELETE with the marker as a parameter. For example, the following statement deletes the marker *mark1*:

```
DELETE (mark1);
```

You can also set all variables referring to the marker to refer to something else, for example, *tpu$k_unspecified* or *0*. The following statement sets the variable *mark1* to 0:

```
mark1 := 0;
```

Example 2–1 Suppressing the Addition of Padding Blanks

```
x := MARK (FREE_CURSOR);                    ! Places a marker at the
                                            ! detached editing point

POSITION (SEARCH_QUIETLY ("",FORWARD));     ! Moves the active editing
                                            ! point to the nearest
                                            ! text character

bat :=  CURRENT_LINE;                       ! Assigns the string
                                            ! representation of the
                                            ! current line to bat without
                                            ! adding padding blanks

POSITION (x);                               ! Returns the active editing
                                            ! point to the free marker
```

Note that if *mark1* were the only variable referring to a marker, that marker would be deleted upon execution of the previous statement.

The marker data type is returned by the built-in procedures MARK, SELECT, BEGINNING_OF, END_OF, and GET_INFO.

2.7 Pattern

A pattern is a structure that VAXTPU uses when it searches for text in a buffer. You can think of a pattern as a template that VAXTPU compares to the searched text, looking for a match between the pattern and the searched text. You can use a variable whose data type is the pattern data type when you specify the first parameter to the SEARCH and SEARCH_QUIETLY built-ins.

To create a pattern, use VAXTPU pattern operators (+, &, | , @) to connect any of the following:

- String constants
- String variables
- Pattern variables
- Calls to pattern built-in procedures
- The following keywords:
  - ANCHOR
  - BUFFER_BEGIN
  - BUFFER_END
  - LINE_BEGIN
  - LINE_END
  - PAGE_BREAK
  - REMAIN
  - UNANCHOR

- Parentheses (to enclose expressions)

Patterns can be simple or complex. A simple pattern can be composed of sets of strings connected by one of the pattern operators. The following example indicates that *pat1* matches either the string "abc" or the string "def":

```
pat1 := "abc" | "def";
```

Note that if you connect two strings with the + operator, the result is a string rather than a pattern. For example, the following statement gives *pat1* the string data type:

```
pat1 := "abc" + "def";
```

The SEARCH and SEARCH_QUIETLY built-ins accept such a string as a parameter.

A more complex pattern uses pattern built-in procedures and existing patterns to form a new pattern. The following example indicates that *pat2* matches the string "abc" followed by the longest string that contains any characters from the string "12345":

```
pat2 := "abc" + SPAN ("12345");
```

Pat2 matches the string "abc123" in the text string "xyzabc123def".

Following are additional examples of statements that create complex patterns:

```
pat1 := any( "abc" );
pat2 := line_begin + remain;
pat3 := "abc" | "xes";
pat4 := pat1 + "12";
pat5 := "xes" @ var1;
pat6 := "abc" & "123";
```

You can assign a pattern to a variable and then use the variable as a parameter for the built-in procedure SEARCH or SEARCH_QUIETLY. SEARCH or SEARCH_QUIETLY looks for the character sequences specified by the pattern that you use as a parameter. If SEARCH or SEARCH_QUIETLY finds a match for the pattern, the built-in returns a range containing the text that matches the pattern. The range can be assigned to a variable.

The following example uses strings and pattern operators to create a pattern that is stored in the variable *my_pat*. The variable is then used with the built-in procedure SEARCH or SEARCH_QUIETLY in a forward direction. If SEARCH or SEARCH_QUIETLY finds a match for *my_pat*, the range of matching text is stored in the variable *match_range*. The built-in procedure POSITION causes the editing point to move to the beginning of *match_range*.

```
my_pat := ("abc" | "def") + "::";
match_range := SEARCH (my_pat, FORWARD);
POSITION (match_range);
```

2.7.1 Pattern Built-In Procedures

The following built-in procedures return values of the pattern data type:

- ANY — Matches one or more characters. You specify a set of characters to be matched and an integer indicating how many of them to match. For example, the following statement creates a pattern that matches any two of the characters *h*, *i*, *j*, *k*, and *l*.

```
pat1 := ANY ("hijkl",2);
```

- ARB — Matches an arbitrary sequence of characters. You use ARB's parameter to specify the number of characters to be matched. For example, the following statement creates a pattern that matches the next five characters starting at the editing point:

```
pat1 := ARB (5);
```

- MATCH — Looks on the current line for the sequence of characters you specify. If VAXTPU locates the sequence in the searched text, MATCH returns a range starting at the editing point and ending at the last character of the sequence. For example, the following statement stores in *pat1* a pattern that matches a string of characters starting with the editing point up to and including the characters *abc*:

```
pat1 := MATCH ("abc");
```

- NOTANY — Matches one or more characters; you specify how many characters to match and which characters must not appear in the matched characters. For example, the following statement creates a pattern that matches the first character that is not an *X*, a *Y*, or a *Z*:

```
pat1 := NOTANY ("XYZ");
```

- SCAN — Matches any characters that are not specified in the parameter. SCAN matches as many characters as possible, and must match at least one character. Matching stops at the end of a line, when SCAN finds one of the excluded characters, or if matching the character would prevent the rest of the pattern from matching. For example, the following statement stores in *pat1* a pattern that matches the longest string of characters that does not contain *a*, *b*, or *c*:

```
pat1 := SCAN ("abc");
```

  Note that the keyword REVERSE modifies the behavior of SCAN in reverse searches. For more information, see the description of the SCAN built-in procedure in the VAXTPU Reference Section.

- SCANL — Same as above, except that SCANL does not stop at the end of a line. For example, the following statement creates a pattern that matches a sentence. It assumes that a sentence ends with a period (.), exclamation point (!), or question mark (?), and that a sentence starts with a capital letter. The matched text does not include the punctuation mark ending the sentence.

```
sentence_pattern := any ("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
                    + scanl (".!?");
```

Note that the keyword REVERSE modifies the behavior of SCANL in reverse searches. For more information, see the description of the SCANL built-in procedure in the VAXTPU Reference Section.

- SPAN — Matches as many characters as possible, all of which must be present in the text you pass as an argument. SPAN must match at least one character. SPAN stops matching when it reaches the end of a line, if it finds a character that was not specified, or if matching the character would prevent the rest of the pattern from matching. For example, the following statement creates a pattern that matches any sequence of numbers:

```
pat1 := span ("0123456789");
```

  Note that the keyword REVERSE modifies the behavior of SPAN in reverse searches. For more information, see the description of the SPAN built-in procedure in the VAXTPU Reference Section.

- SPANL — Same as above, except that SPANL does not stop at the end of a line. For example, the following statement stores a pattern in *pat1* that matches the longest sequence of numbers starting at the editing point and continuing to a nonnumeric character, or the end of the range or buffer:

```
pat2 := SPANL ("0123456789");
```

  Note that the keyword REVERSE modifies the behavior of SPANL in reverse searches. For more information, see the description of the SPANL built-in procedure in the VAXTPU Reference Section.

2.7.2 Keywords That Can Be Used to Build Patterns

The following keywords can be used as the first argument to the SEARCH or SEARCH_QUIETLY built-ins. They can also be used to form patterns in expressions using the pattern operators.

- ANCHOR — Directs SEARCH or SEARCH_QUIETLY to try to match the next pattern element at the current search location. Normally, when SEARCH or SEARCH_QUIETLY fails to find a match for a pattern, the built-in retries the search, moving the starting position one character forward or backward, depending upon the direction of the search. If ANCHOR appears as the first element of a complex pattern, the search does not move the starting position. If the pattern does not match starting in the original position, the search fails. For more information on using ANCHOR, see the description in the VAXTPU Reference Section.
- BUFFER_BEGIN — Matches the beginning of the buffer in which the search is executed.
- BUFFER_END — Matches the end of the buffer in which the search is executed.
- LINE_BEGIN — Matches the beginning of a line.
- LINE_END — Matches the end of a line.
- PAGE_BREAK — Matches the form feed or page break character.

- REMAIN — Matches the rest of the characters on the line.
- UNANCHOR — Allows the next pattern element to match anywhere at or after the current search location.

2.7.3 Pattern Operators

The following are the VAXTPU pattern operators:

- Concatenation operator (+)
- Link operator (&)
- Alternation operator (|)
- Partial pattern assignment operator (@)

The pattern operators are equal in VAXTPU's precedence of operators. For more information on the precedence of VAXTPU operators, see Chapter 3. Pattern operators associate from left to right. Thus, the following two VAXTPU statements are identical:

```
pat1 := a + b & c | d @ e;
pat1 := (((a + b) & c) | d) @ e;
```

In addition to the pattern operators, two relational operators, equal (=) and not equal (<>), can be used to compare patterns.

The following sections discuss the pattern operators.

2.7.3.1 + (Pattern Concatenation Operator)

The concatenation operator tells SEARCH or SEARCH_QUIETLY that text matching the right pattern element must immediately follow the text matching the left pattern element in order for the complete pattern to match. In other words, the concatenation operator specifies a search in which the right pattern element is anchored to the left. For example, the following pattern matches only if there is a line in the searched text that ends with the string *abc*.

```
pat1 := "abc" + line_end;
```

If SEARCH or SEARCH_QUIETLY finds such a line, the built-in returns a range containing the text *abc* and the end of the line.

Digital recommends that you use the concatenation operator rather than the link operator unless you specifically require the link operator.

2.7.3.2 & (Pattern Linking Operator)

The link operator (&) is very similar to the concatenation operator (+). Unlike the concatenation operator, the link operator does not necessarily cause an anchored search. If you define a pattern by specifying any pattern element, an ampersand (&), and a pattern or keyword variable, a search for each subpattern is not an anchored search.

If you link elements other than pattern variables, the search is an anchored search unless you specify otherwise. Strings, constants, and the results of built-in procedures are not pattern variables.

For example, suppose two subpattern variables are defined as follows:

```
p1 := "a" & ANY("012345678");
p2 := "c" & ARB (1);
```

Suppose you then define the following pattern variable:

```
pat_var := p1 & p2
```

Given this sequence of definitions, a search for *pat_var* succeeds if VAXTPU encounters the following string:

```
a5xcd
```

Because two pattern variables are linked, VAXTPU searches first for the text that matches *p1*, then unanchors the search, and then searches for the text that matches *p2*.

To specify an anchored search when the right-hand subpattern is a pattern or keyword variable, use a plus sign (+). You must use a plus sign (+) to anchor the search if the right-hand subpattern is a keyword variable. If the right-hand subpattern is a pattern variable, you can anchor the right-hand subpattern by using the ANCHOR keyword as the first element of that subpattern.

For example, if you have defined the following patterns:

```
p1 := LINE_BEGIN + "a";
p2 := "b" + LINE_END;
```

You anchor the search for *p2* by using (+), as follows:

```
pat_var := p1 + p2;
```

If you used an ampersand (&), you would unanchor the search for *p2*.

Alternatively, you can anchor the search for *p2* by defining *p2* as follows:

```
p2 := ANCHOR + "b" + LINE_END;
```

2.7.3.3 | (Pattern Alternation Operator)

The alternation operator (|) tells SEARCH or SEARCH_QUIETLY to match a sequence of characters if those characters match either of the pattern elements separated by the alternation operator. Thus, the following pattern matches either the string *abc* or the string *xes*:

```
pat1 := "abc" | "xes";
```

If the text being searched contains text that matches both alternatives, SEARCH or SEARCH_QUIETLY matches the earliest occurring match. If two matches start at the same character, SEARCH or SEARCH_QUIETLY matches the left element. For example, suppose you had the search text *abcd* and the following pattern definitions:

```
pat1 := "abc" | "bcd";
pat2 := "bcd" | "abc";
pat3 := "bc"  | "bcd";
pat4 := "bcd" | "bc";
```

Given these definitions and search text, a search for the patterns *pat1* and *pat2* would return a range containing the text *abc*. A search for the pattern *pat3* would return a range containing the text *bc*. Finally, a search for the pattern *pat4* would return a range containing the text *bcd*.

2.7.3.4 @ (Partial Pattern Assignment Operator)

The partial pattern assignment operator (@) tells SEARCH or SEARCH_QUIETLY to create a range that contains the text matching the pattern element to the left of the partial pattern assignment operator. When the search is completed, the variable to the right of the partial pattern assignment operator references the created range. If SEARCH or SEARCH_QUIETLY is given the search text *abcdefg* and the following pattern, it returns a range containing the text *abcdefg*.

```
pat1 := "abc" + (arb(2) @ var1) + remain;
```

SEARCH or SEARCH_QUIETLY also assigns to *var1* a range containing the text *de*.

If you assign to a variable a partial pattern that matches a position, rather than a character, the partial pattern variable is a range containing the character or line-end at the point in the file where the partial pattern was matched. For example, in any of the following patterns containing partial pattern assignments, the variable *partial_pattern_variable* contains the character or line-end at the point in the file where the partial pattern was matched:

- "" @ partial_pattern_variable
- ANCHOR @ partial_pattern_variable
- UNANCHOR @ partial_pattern_variable
- LINE_BEGIN @ partial_pattern_variable
- BUFFER_BEGIN @ partial_pattern_variable

Note that if you use one of the preceding patterns when the cursor is free (that is, in an area that does not contain text, such as the area after the end of a line) the variable *partial_pattern_variable* contains the line-end or character nearest to the cursor.

SEARCH or SEARCH_QUIETLY does partial pattern assignment only if the complete pattern matches. If the complete pattern matches, it makes assignments only to those variables paired with pattern elements that are used in the complete match. If a partial pattern assignment variable appears more than once in a pattern in places where it is legal for a partial pattern assignment to occur, the last occurrence in the pattern determines what range SEARCH assigns to the variable. For example, with the search text *abcdefg* and the following pattern, SEARCH or SEARCH_QUIETLY returns a range containing the text *abcde* and assigns a range containing the text *d* to the variable *var1*.

```
pat1 := "a" + ("b" @ var1) + "c" + ("d" @ var1)
        + ("e" | ("x" @ var1));
```

2.7.3.5 Relational Operators

The two relational operators, equal (=) and not equal (<>), can be used to compare patterns. Two patterns are equal if they are the same pattern, as *pat1* and *pat2* are in the following example:

```
pat1 := notany("abc", 2) + span("123");
pat2 := pat1;
```

Two patterns are also equal if they have the same internal representation. Patterns have the same internal representation only if they are built in exactly the same way. The order of the characters in the arguments to ANY, NOTANY, SCAN, SCANL, SPAN, and SPANL does not matter when you are comparing patterns returned by any of these built-ins. Other than this, almost any difference in the building of two patterns makes those patterns unequal. For example, suppose you defined the variable *this_pat* as follows:

```
this_pat := ANY ("abc");
```

Given this definition, the following patterns match the same text but are not equal:

```
pat1 := LINE_BEGIN + ANY ("abc");
pat2 := LINE_BEGIN + this_pat;
```

2.7.4 Pattern Compilation and Execution

When you execute a VAXTPU statement that contains a pattern expression, VAXTPU builds an internal representation of the pattern. VAXTPU uses the current contents of any buffers or ranges used as arguments to pattern built-ins in the pattern expression to build the internal representation. Later changes to those buffers and ranges do not affect the internal representation for the pattern. VAXTPU also uses the current values of any variables used in the pattern expression. Later changes to these variables do not affect the internal representation of the pattern. For example, suppose you wrote the following code fragment:

```
p1  := "abc";
p2  := "123";
pat := p1 & p2;
p1  := "xyz";
SEARCH (pat, FORWARD);
```

Given this code fragment, the search matches the string "*abc123*" because the variable *pat* is evaluated as it is built from *p1* and *p2* during the assignment statement.

2.7.5 Searching

The SEARCH and SEARCH_QUIETLY built-ins use the following algorithm to find a match for a pattern.

1 Put the internal marker that marks the search position at the starting position for the search. The starting position is determined as follows:

- If the user does not specify where to search, search the current buffer, starting at the editing point.

- If the user specifies a buffer or range where the search is to take place, start at the beginning or end of the buffer or range depending on the direction of the search.

2 Check whether the pattern matches text, starting at the current search position and extending toward the end of the searched buffer or range. If a range is being searched, the matched text cannot extend beyond the end of that range. If the pattern matches, return a range containing the matching text and stop searching.

3 If the previous step fails, move the search position one character forward or backward, depending upon the direction of the search. If this is impossible because the search position is at the end or beginning of the searched buffer or range, stop searching. If this step succeeds, repeat the previous step.

Note: This algorithm changes if you specify a reverse search for a pattern starting with SCAN, SPAN, SCANL, or SPANL. For more information, see the descriptions of these built-in procedures in the VAXTPU Reference Section.

2.7.6 Anchoring a Search

Anchoring a pattern forces SEARCH or SEARCH_QUIETLY to match the anchored part of the pattern to text starting at the current search position. If the anchored part of a pattern fails to match that text, SEARCH or SEARCH_QUIETLY stops searching.

Normally, all pattern elements other than the first pattern element of a pattern are anchored. This means that a pattern can match text starting at any point in the searched text but that once it starts matching, each pattern element must match the text immediately following the text that matched the previous pattern element.

To direct VAXTPU to stop searching if the characters starting at the editing point do not match the pattern, use the keyword ANCHOR as the first pattern element. For example, the following pattern matches only if the string *abc* occurs at the editing point:

```
pat1 := ANCHOR + "abc";
```

There are two ways to unanchor pattern elements in the midst of a pattern. The easiest is to concatenate or link the UNANCHOR keyword before the pattern element you want to unanchor. Thus, in the following pattern the pattern element *xyz* is unanchored:

```
pat1 := "abc" + UNANCHOR + "xyz";
```

This means that the pattern *pat1* matches any text beginning with the characters *abc* and ending with the characters *xyz*. It does not matter what or how many characters or line breaks appear between the two sets of characters. Of course, since SEARCH or SEARCH_QUIETLY matches the first *xyz* it finds, the text between the two sets of characters by definition does not contain the string *xyz*.

The second way to unanchor a pattern element is to use the special properties of the link operator (&). While the concatenation operator always anchors the right pattern element to the left, the link operator does so only if the right pattern element is not a pattern variable. If the link operator's right pattern element is a pattern variable, the link operator unanchors that pattern element. Thus, the pattern *pat2* defined by the following assignments matches any sequence of text beginning with the letter *a* and ending with a digit.

```
pat1 := ANY ("0123456789");
pat2 := "a" & pat1;
```

Any amount of text can occur between the *a* and the digit. *Pat2* matches the same text as the following pattern:

```
pat3 := "a" + UNANCHOR + ANY( "0123456789" );
```

The link operator unanchors a pattern variable regardless of what the left pattern element is. In particular, the following two patterns match the same text:

```
pat2 := "a" & pat1;
pat3 := "a" & ANCHOR & pat1;
```

If you are using pattern variables to form patterns and you wish those variables to be anchored, you have two choices: you can use the concatenation operator, or you can use the keyword ANCHOR as the first element of any pattern the pattern variables reference.

2.8 Process

In VAXTPU, a process is a VMS subprocess. The built-in procedure CREATE_PROCESS returns a value of the process data type.

VAXTPU processes have the same restrictions that VMS subprocesses have. Following are some of the restrictions:

- You cannot create more VAXTPU processes than your account subprocess quota allows.
- You cannot spawn a subprocess in an account that has the CAPTIVE flag set.
- Only VMS utilities that can perform I/O to a mailbox and that do simple reads and writes (for example, MAIL) can run in a VAXTPU process. Programs like FMS, EDT, PHONE, or any other program that takes full control of the screen do not work properly in a VAXTPU process. See the built-in procedure SPAWN for information on running these types of programs from VAXTPU.
- You do not see any prompts from the utility you are using. For example, in MAIL, you have to be aware of the sequence of prompts for sending a mail message because you do not see the prompts.

The following example assigns a value of the process data type to the variable *x*:

```
x := CREATE_PROCESS (main_buffer, "MAIL");
```

The first parameter specifies that the output from the subprocess is to be stored in MAIN_BUFFER. The string "MAIL" is the first command sent to the subprocess.

To pass subsequent commands to a subprocess, use the built-in procedure SEND, as follows:

```
SEND ("MAIL", x);
```

To pass the READ command to the Mail Utility, enter the following VAXTPU statement:

```
SEND ("READ", x);
```

The output from the READ command is stored in the buffer associated with the subprocess *x*. If the buffer associated with a subprocess is deleted, the subprocess is deleted as well.

2.9 Program

A program is the compiled form of a sequence of VAXTPU procedures and executable statements. The built-in procedures COMPILE and LOOKUP_KEY can optionally return a value of the program data type as a result. The following example assigns a value of the program data type to the variable *x*:

```
x := COMPILE (main_buffer);
```

MAIN_BUFFER must contain only VAXTPU declarations, executable statements, and comments. All declarations must come before any executable statements that are not included in the declarations. The declarations and statements are compiled and the resulting program is stored in the variable *x*.

2.10 Range

A range contains all the text between (and including) two markers. You can form a range with the built-in procedure CREATE_RANGE. A range is associated with characters within a buffer. If the characters within a range move, the range moves with them. If characters are added or deleted between two markers that delimit a range, the size of the range changes. If all the characters in a range are deleted, the range moves to the nearest character.

VAXTPU does not support ranges of zero length unless the range begins and ends at the end of a buffer. All other ranges contain at least one character (which could be a space character) or a line-end (if the range is created at the end of a line).

If you create a range by specifying a free marker as a parameter to the CREATE_RANGE built-in, VAXTPU creates a new marker and binds the marker to the text nearest to the free marker position. VAXTPU uses the new bound marker as the range delimiter. This operation does not cause insertion of padding spaces.

Deleting the markers used to create a range does not affect the range.

To convert the contents of a range to a string, use either the STR or the SUBSTR built-in.

To remove a range, use the built-in procedure DELETE with the range as a parameter. For example, the following statement deletes the range *range1*:

```
DELETE (range1);
```

You can also delete a range by removing all variable references to the range. To do this, set all variables referring to the range to some other value, such as 0. For example, the following statement sets the variable *range1* to 0:

```
range1 := 0;
```

Deleting a range does not remove the characters of the range from the buffer; it merely removes the range data structure. To remove the characters of a range, use the built-in procedure ERASE with the range as a parameter. For example, *ERASE (my_range)* removes all the characters in *my_range*, but it does not remove the range structure. Using the statement DELETE (*range_variable*) removes the range data structure, but does not affect the characters in the range.

The following built-in procedures, as well as the partial pattern assignment operator, all return values of the range data type:

- CHANGE_CASE
- CREATE_RANGE
- EDIT
- GET_INFO
- READ_GLOBAL_SELECT
- READ_CLIPBOARD
- SEARCH
- SEARCH_QUIETLY
- SELECT_RANGE
- TRANSLATE

For example, the following example assigns a value of the range data type to the variable *x*:

```
x := CREATE_RANGE (mark1, mark2, UNDERLINE);
```

You can specify the video attribute with which VAXTPU should display a range. The possible attributes are BLINK, BOLD, REVERSE, and UNDERLINE. The keyword UNDERLINE in the preceding example specifies that the characters in the range will be underlined when they appear on the screen. You cannot give more than one video attribute to a range. However, to apply multiple video attributes to a given set of characters, you can define more than one range containing those characters and give one video attribute to each range.

2.11 String

VAXTPU uses the string data type to represent character data. A value of the string data type can contain any of the elements of the DEC Multinational Character Set. To specify a string constant, enclose the value in quotation marks. In VAXTPU, you can use either the quotation mark (") or the apostrophe (') as the delimiter for a string. The following statements assign a value of the string data type to the variable *x*:

```
x := 'abcd';
x := "abcd";
```

To specify the quote character itself within a string, type the character twice if you are using the same quote character as the delimiter for the string. The following statements show how to quote an apostrophe and a quotation mark, respectively:

```
x := '''';          ! The value assigned to x is '.
x := """";          ! The value assigned to x is ".
```

If you use the alternate quote character as the delimiter for the string within which you want to specify a quote character, you do not have to type the character twice. The following statements show how to quote an apostrophe and a quotation mark, respectively, when the alternate quote character is used to delimit the string:

```
x := "'";           ! The value assigned to x is '.
x := '"';           ! The value assigned to x is ".
```

A null string is a string of length zero. You can assign a null string to the variable *x* in the following way:

```
x := '';
```

To create a string from the contents of a range, use the STR or the SUBSTR built-in. To create a string from the contents of a buffer, use the STR built-in.

The maximum length for a string is 65,535 characters. A restriction of the VAXTPU compiler is that a string constant (an open quotation mark, some characters, and a close quotation mark) must have both its opening and closing quotation marks on the same line. Note that while a string can be up to 65,535 characters long, a line in a VAXTPU buffer can only be 960 characters long. If you try to create a line that is longer than 960 characters, VAXTPU truncates the inserted text to the amount that fills the line to 960 characters.

Many VAXTPU built-in procedures return a value of the string data type. The built-in procedure ASCII, for example, returns a string for the ordinal value that you use as a parameter. The following statement returns the string "K" in the variable *my_char*:

```
my_char := ASCII (75);
```

To replicate a string, specify the string to be reproduced, then the multiplication operator (*), and then the number of times you want the string to be replicated. For example, the following VAXTPU statement inserts 10 underscores into the current buffer at the editing point:

```
COPY_TEXT ("_" * 10)
```

Note that the string to be replicated must be on the left-hand side of the operator. For example, the following VAXTPU statement produces an error:

```
COPY_TEXT (10 * "_")
```

To reduce a string, specify the string to be modified, then the subtraction operator (–), and then the substring to be removed. For example, the following table shows the effects of two string-reduction operations:

VAXTPU Statement	Result
COPY_TEXT ("FILENAME.MEM"–"FILE")	Inserts the string "NAME.MEM" into the current buffer at the editing point.
COPY_TEXT ("woolly"–"wool")	Inserts the string "ly" into the current buffer at the editing point.

2.12 Unspecified

An unspecified value is the initial value of a variable after it has been compiled (added to the VAXTPU symbol table). In the following example, the built-in procedure COMPILE creates the variable *x* and initially gives it the data type unspecified unless *x* has previously been declared as a global variable:

```
COMPILE ("x := 1");
```

An assignment statement that creates a variable must be executed before a data type is assigned to the variable. In the following example, when you use the built-in procedure EXECUTE to run the program that is stored in the variable *prog*, the variable *x* is assigned an integer value:

```
prog := COMPILE ("x := 1");
EXECUTE (prog);
```

To give a variable the data type unspecified, assign the predefined constant TPU$K_UNSPECIFIED to the variable:

```
prog := TPU$K_UNSPECIFIED;
```

2.13 Widget

The DECwindows version of VAXTPU provides the widget data type to support DECwindows widgets. The non-DECwindows version of VAXTPU does not support this data type.

A widget is an interaction mechanism by which users give input to an application or receive messages from an application. For more information about what a widget is, see the *VMS DECwindows Guide to Application Programming*.

You can use the equal operator (=) or the not-equal operator (<>) on widgets to determine whether they are equal (that is, whether they are the same widget instance), but you cannot use any other relational or arithmetic operators on them. For information about the difference between a class of widgets and a widget instance, see the *VMS DECwindows Guide to Application Programming*.

Once you have created a widget instance, VAXTPU does not delete the widget instance, even if there are no variables referencing it. To delete a widget, use the DELETE built-in.

DECwindows VAXTPU provides the same support for DECwindows gadgets that it provides for widgets. A gadget is a structure similar to a widget, but it is not associated with its own unique DECwindows window. Gadgets do not require as much memory to implement as widgets do. In most cases, you can use the same DECwindows VAXTPU built-ins on gadgets that you use on widgets. For more information about gadgets, see the *VMS DECwindows Guide to Application Programming*.

2.14 Window

A window is a portion of the screen used to display as much of the text in a buffer as will fit in the screen area. In EVE, the screen contains three windows by default: a large window for viewing the text in the user's editing buffer, and two one-line windows, for displaying commands and messages. In EVE or in a user-written interface, the screen can be subdivided to create more windows.

A variable of the window data type "contains" a window. The built-in procedures CREATE_WINDOW, CURRENT_WINDOW, and GET_INFO return a value of the window data type. CREATE_WINDOW is the only built-in procedure that creates a new window. The following example assigns a value of the window data type to the variable *x*:

```
x := CREATE_WINDOW (1, 12, OFF);
```

The first parameter specifies that the window starts at screen line number 1. The second parameter specifies that the window is 12 lines in length. The keyword OFF specifies that a status line is not to be displayed when the window is mapped to the screen.

2.14.1 Window Dimensions

Windows are defined in lines and columns. In EVE, all windows extend the full width of the screen or terminal emulator. In VAXTPU, you can set the window width to be narrower than the width of the screen or terminal emulator.

The allowable dimensions of a window often depend on whether the window has a status line, a horizontal scroll bar, or both. A status line occupies the last line of a window. By default, a status line contains information about the buffer and the file associated with the window. You can turn a status line on or off with the built-in SET (STATUS_LINE). A horizontal scroll bar is a one-line widget at the bottom of a window

that the user can use to shift the window to the right or left, controlling what text in the buffer can be seen through the window. You can turn a horizontal scroll bar on or off with the built-in SET (SCROLL_BAR). Lines on the screen are counted from one to the number of lines on the screen; lines in a window are counted from one to the number of lines in the window. Columns on the screen are counted from one to the physical width of the screen; columns in a window are counted from one to the number of columns in the window. The minimum length for a window is one line if you do not include a status line or horizontal scroll bar, two lines if you include either a status line or a horizontal scroll bar, and three lines if you include both a status line and scroll bar.

The maximum length of a window is the number of lines on your screen. For example, if your screen is 24 lines long, the maximum size for a single window is 24 lines. On the same size screen, you can have a maximum of 24 visible windows if you do not use status lines or horizontal scroll bars. If you use a status line and a horizontal scroll bar for each window, the maximum number of visible windows is eight.

2.14.2 Creating Windows

When you use a device that supports windows (see Appendix C for information on terminals that VAXTPU supports), you or the section file that initializes your application must create and map windows. In most instances, it is also advisable to map a buffer to the window. To map a buffer to a window, use the MAP built-in. If you do not associate a buffer with a window and map the window to the screen, the only items displayed on the screen are messages that are written to the screen at the cursor position.

The built-in procedure CREATE_WINDOW defines the size and location of a window and specifies whether a status line is to be displayed. CREATE_WINDOW also adds the window to VAXTPU's internal list of windows available for mapping. At creation, a window is marked as being *not visible* and *not mapped* and the following values for the window are calculated and stored:

- *Original_top* — Screen line number of the top of the window when it was created.
- *Original_bottom* — Screen line number of the bottom of the window when it was created (not including the status line).
- *Original_length* — Number of lines in the window (including the status line).

Later calls to ADJUST_WINDOW may change these values.

2.14.3 Window Values

When you create a window with the CREATE_WINDOW built-in procedure,VAXTPU saves the numbers of the screen lines that delimit the window in *original_top* and *original_bottom.* When you map a window to the screen with the built-in procedure MAP, the window becomes visible on the screen. If it is the only window on the screen, its *visible_top* and *visible_bottom* values are the same as its *original_top* and *original_bottom* values. You can display the *original* and the *visible* values with SHOW (WINDOWS) or retrieve them using the built-in procedure GET_INFO.

However, if there is already a window on the screen and you map another window over part of it, the values for the previous window's *visible_top*, *visible_bottom*, and *visible_length* are modified. The value for *visible_length* of the previous window is different from its *original_length* until the new window is removed from the screen. As long as the new window is on the screen and does not have another window mapped over it, its original top and bottom are the same as its visible top and bottom.

2.14.4 Mapping Windows

When you want a window and its associated buffer to be visible on the screen, use the built-in procedure MAP. Mapping a window to the screen has the following effects:

- The mapped window becomes the current window and the cursor is moved to the editing point in the buffer associated with the window.
- The buffer associated with the window becomes the current buffer.
- The window is marked as *visible* and *mapped.*
- The *visible_top*, *visible_bottom*, and *visible_length* of the window are calculated and stored. Initially, these values are the same as the *original* values that were calculated when the window was created. (See the last item in the following list.)

Mapping a window to the screen may have the following side effects:

- The newly mapped window may occlude other windows. This happens when the *original_top* or *original_bottom* line of the newly mapped window overlaps the boundaries of existing visible windows. Overlapping can cause some windows to be totally occluded or *not visible.* Note that occluded windows are still marked *mapped*; when the window that is covering them is unmapped, they may reappear on the screen without being explicitly remapped.
- If the newly mapped window divides a window into two parts, only the top part of the segmented window continues to be updated. The lower part of the segmented window is erased at the next window update.
- The *visible_top, visible_bottom,* and *visible_length* values of a window that is occluded change from their *original* values.

When a newly mapped window becomes the current window (the built-in procedures MAP, POSITION, and ADJUST_WINDOW cause this to happen), the cursor is placed in the current window. In addition to the active cursor position in the current window, there is a marker designating a cursor position in all other windows. The cursor position in a window other than the current window is the last location of the cursor when it was in the window. By maintaining a cursor position in all windows, VAXTPU allows you to edit in multiple locations of a single buffer if that buffer is associated with more than one window. For more information on the cursor position in a window, see Chapter 6 and the description of the POSITION built-in in the VAXTPU Reference Section.

2.14.5 Removing Windows

To remove a window from the screen, you can use either the built-in procedure UNMAP or the built-in procedure DELETE. UNMAP removes a window from the screen. However, the window is still in VAXTPU's internal list of windows. It is available to be remapped to the screen without being recreated. DELETE removes a window from the screen and also removes it from VAXTPU's list of windows. It is then no longer available for future mapping to the screen.

Unmapping or deleting a window has the following effects:

- The unmapped window is marked as *not visible* and *not mapped.*
- Another window becomes the current window and the cursor is moved to the last cursor position in that window.
- If other windows were occluded by the window you removed from the screen, text from the occluded windows reappears on the screen. The *visible_top, visible_bottom,* and *visible_length* values of the previously occluded windows are modified according to the lines that are returned to them when the occluding window is unmapped. When an occluding window is removed, the window or windows it occluded become visible again.

2.14.6 Screen Manager

The screen manager is the part of VAXTPU that controls the display of data on the screen. You can manipulate data without having it appear on a terminal screen (see Chapter 4). However, if you use the VAXTPU window capability to make your edits visible, the screen manager controls the screen.

In the main control loop of VAXTPU, the screen manager is not called to perform its duties until all commands bound to the last key pressed have finished executing and all input in the type-ahead buffer has been processed. Upon completion of all the commands, the screen manager updates every window to reflect the current state of the part of the buffer that is visible in the window. If you want to make the screen reflect changes to the buffer prior to the end of a procedure, use the built-in procedure UPDATE to force the updating of the window. Using UPDATE is recommended with built-in procedures such as CURRENT_COLUMN

that query VAXTPU for the current cursor position. To ensure that the cursor position returned is the correct location (up to the point of the most recently issued command), use UPDATE before using CURRENT_COLUMN or CURRENT_ROW.

2.14.7 Getting Information on Windows

There are two VAXTPU built-in procedures that return information about windows: GET_INFO and SHOW (WINDOW).

GET_INFO returns information that you can store in a variable. You can get information about the *visible* and *original* values of windows, as well as about other attributes that you have set up for your window environment. See the description of GET_INFO in the VAXTPU Reference Section.

SHOW (WINDOW) or SHOW (WINDOWS) puts information about windows in the SHOW_BUFFER. If you use an editor that has an INFO_WINDOW, you can display the SHOW_BUFFER information in the INFO_WINDOW.

2.14.8 Terminals That Do Not Support Windows

VAXTPU supports windows only for ANSI CRTs. (See Appendix C if you need more information about VAXTPU terminal support.) If the logical name SYS$INPUT points to an unsupported device, windows cannot be used. When you are working on an unsupported device, you must specify /NODISPLAY when you invoke VAXTPU, or the utility exits with an error condition. The qualifier /NODISPLAY informs VAXTPU that you do not expect the device from which you are issuing VAXTPU commands to support screen-oriented editing. See Chapter 4 and Chapter 5 for more information on the /NODISPLAY qualifier.

3 Lexical Elements of the VAXTPU Language

3.1 Overview

A VAXTPU program is composed of lexical elements. A lexical element may be an individual character, such as an arithmetic operator, or it may be a group of characters, such as an identifier. The basic unit of a lexical element is a character from the DEC Multinational Character Set. (See Appendix E for a complete list of the DEC multinational characters.) This chapter describes the following VAXTPU lexical elements:

- Character set (Section 3.2)
- Identifiers (Section 3.3)
- Variables (Section 3.4)
- Constants (Section 3.5)
- Operators (Section 3.6)
- Expressions (Section 3.7)
- Reserved words (Section 3.8)
- Lexical keywords (Section 3.9)

3.2 Character Set

The DEC Multinational Character Set is an 8-bit character set with 256 characters. Each character is assigned a decimal equivalent number ranging from 0 to 255. The first 128 characters in the set correspond to the American Standard Code for Information Interchange (ASCII) character set. The characters from 128 to 255 are extended control characters and supplemental multinational characters. The characters can be grouped into the following categories:

0–31	Nonprinting characters such as tab, line feed, carriage return, and bell
32	Space
33–64	Special characters such as the ampersand (&), question mark (?), equal sign (=), and the numbers 0 through 9
65–122	The uppercase and lowercase letters A through Z and a through z
123–126	Special characters such as the left brace ({) and the tilde (~)
127	Delete
128–159	Extended control characters
160	Reserved

161–191	Supplemental special graphics characters such as the copyright sign (©) and the degree sign (°)
192–254	The supplemental multinational uppercase and lowercase letters such as the Spanish Ñ and ñ
255	Reserved

The VAXTPU compiler does not distinguish between uppercase and lowercase characters except when they appear as part of a quoted string. For example, the word EDITOR has the same meaning when written in any of the following ways:

```
EDITOR
EDitOR
editor
```

The following, however, are quoted strings, and therefore represent different values:

```
"XYZ"
"xyz"
```

3.2.1 Entering Control Characters

There are two ways to enter control characters in VAXTPU:

1. Use the built-in procedure ASCII with the decimal value of the control character that you want to enter. For example, the following statement causes the escape character to be entered in the current buffer:

   ```
   COPY_TEXT (ASCII (27));
   ```

2. Use the special functions provided by EVE to enter control characters:

   - EVE provides a QUOTE command that is bound to CTRL/V to insert control characters in a buffer. For example, to use the quote command to insert an escape character in a buffer, follow these steps:

     a. Press CTRL/V.

     b. Press the ESCAPE key (on VT100-series terminals) or CTRL/[.

     The following example shows the previous steps:

     CTRL/V ESC

   - EVE's EDT-like keypad setting provides a SPECINS key sequence to insert control characters in a buffer. For example, take the following steps to enter a control character using the SPECINS key:

     a. Press the GOLD key.

     b. Enter the ASCII value of the special character that you want to insert in the buffer; in this case 27 (the escape character). (Use the keys on the keyboard, not the ones on the keypad.)

     c. Press the GOLD key again.

d. Press the SPECINS key on the EDT keypad.

The following example shows the previous steps:

GOLD 27 GOLD SPECINS

3.2.2 VAXTPU Symbols

Certain symbols have special meanings in VAXTPU. They can be used as statement delimiters, operators, or other syntactic elements. The VAXTPU symbols are listed in Table 3–1.

Table 3–1 VAXTPU Symbols

Name	Symbol	VAXTPU Function
Apostrophe	'	Delimits a string
Assignment operator	:=	Assigns a value to a variable
At sign	@	Partial pattern assignment operator
Left brace	{	Opens an array element index expression
Close parenthesis	)	Ends parameter list, expression, procedure call, argument list, or array element index
Comma	,	Separates parameters
Exclamation point	!	Begins comment
Dollar sign	$	Indicates a variable, constant, keyword, or procedure name that is reserved to Digital
Right brace	}	Closes array element index expression
Equal sign	=	Relational operator
Greater than sign	>	Relational operator
Greater than or equal to sign	>=	Relational operator
Slash	/	Integer division operator
Asterisk	*	Integer multiplication operator
Left bracket	[	Begins case label
Less than sign	<	Relational operator
Less than or equal to sign	<=	Relational operator
Minus sign	–	Subtraction operator
Not equal sign	<>	Relational operator
Vertical bar	|	Pattern alternation operator
Open parenthesis	(	Begins parameter list, expression, argument list, or array element index
Ampersand	&	Pattern linkage operator

(continued on next page)

Table 3–1 (Cont.) VAXTPU Symbols

Name	Symbol	VAXTPU Function
Plus sign	+	String concatenation operator, pattern concatenation operator, integer addition operator
Quotation mark	"	Delimits string
Right bracket	]	Ends case label
Semicolon	;	Separates language statements
Underscore	_	Separates words in identifiers

3.3 Identifiers

In VAXTPU, identifiers are used to name programs, procedures, keywords, and variables. An identifier is a combination of alphabetic characters, digits, dollar signs, and underscores, and it must conform to the following restrictions:

- An identifier cannot contain any spaces or symbols except the dollar sign and the underscore.
- Identifiers cannot be more than 132 characters long.

VAXTPU identifiers for built-in procedures, constants, keywords, and global variables are reserved words.

You can create your own identifiers to name programs, procedures, constants, and variables. Note that any symbol that is neither declared nor used as the target of an assignment statement is assumed to be an undefined procedure.

3.4 Variables

Variables are names given to VAXTPU storage locations that hold values. A variable name can be any valid VAXTPU identifier that is not a VAXTPU reserved word or the name of a VAXTPU procedure. You assign a value to a variable by using a valid identifier as the left-hand side of an assignment statement. Following is an example of a variable assignment:

```
new_buffer := CREATE_BUFFER ("new_buffer_name");
```

Digital suggests that you establish some convention for naming variables, so that you can distinguish your variables from the variables in the section file that you are using.

VAXTPU allows two kinds of variables: global and local. Global variables are in effect throughout a VAXTPU environment. Local variables are evaluated only within the procedure or unbound code in which they are declared. A variable is implicitly global unless you use the LOCAL declaration. You can also declare global variables with the VARIABLE declaration.

Example 3–1 Global and Local Variable Declarations

```
VARIABLE user_tab_char;

! Tab key procedure.  Always inserts a tab, even if current mode
! is overstrike.

PROCEDURE user_tab

LOCAL this_mode;          ! Local variable for current mode

this_mode := GET_INFO (CURRENT_BUFFER, "mode"); ! Save current mode
SET (INSERT, CURRENT_BUFFER);                   ! Set mode to insert
user_tab_char := ASCII (9);                     ! Define the tab char
COPY_TEXT (user_tab_char);                      ! Insert tab
SET (this_mode, CURRENT_BUFFER);                ! Reset original mode

ENDPROCEDURE;
```

Example 3–1 shows a global variable declaration and a procedure that contains a local variable declaration:

The global variable *user_tab_char* is assigned a value when the procedure *user_tab* is executing. Since the variable is a global variable, it could have been assigned a value outside the procedure *user_tab*.

The local variable *this_mode* has the value established in the procedure *user_tab* only when this procedure is executing. You can have a variable also named *this_mode* in another procedure. The two variables are not the same and may have different values. You can also have a global variable named *this_mode*. However, using *this_mode* as a global variable when you are also using it as a local variable is likely to confuse people who read your code. VAXTPU will return an informational message during compilation if a local variable has the same name as a global variable.

3.5 Constants

VAXTPU has three types of constants: integers, strings, and keywords.

Integer constants can be any integer value that is valid in VAXTPU. See Chapter 2 for more information on the integer data type.

String constants can be one character or a combination of characters delimited by apostrophes or quotation marks. See Chapter 2 for a complete description of how to quote strings in VAXTPU.

Keywords are reserved words that have special meaning to the VAXTPU compiler. See Chapter 2 for a complete description of keywords.

With the CONSTANT declaration you can associate a name with a constant expression. User-defined constants can be locally or globally defined.

A local constant is a constant declared within a procedure declaration. The scope of the constant is limited to the procedure in which it is defined.

A global constant is a constant declared outside a procedure. Once a global constant has been defined, it is set for the life of the VAXTPU session. You can reassign to a constant the same value it was assigned previously, but you cannot redefine a constant during a VAXTPU session.

See Section 3.8.4.10.3 for a complete description of the CONSTANT declaration.

Example 3–2 shows a global constant declaration and a procedure that contains a local constant declaration.

Example 3–2 Global and Local Constant Declarations

```
! Define some global constants.
CONSTANT
   user_bell := BELL,
   user_hello := "Hello",
   user_ten := 10;

! Hello world procedure.

PROCEDURE user_hello_world
CONSTANT
   world := "world";
MESSAGE (user_hello + " " + world);    ! Display "Hello world"
                                       ! in message area
ENDPROCEDURE;
```

3.6 Operators

VAXTPU uses symbols and characters as language operators. There are four types of operators:

- Arithmetic
- String
- Relational
- Pattern
- Logical

Table 3–2 lists the symbols and language elements that VAXTPU uses as operators.

Table 3–2 VAXTPU Operators

Type	Symbol	Description
Arithmetic	+	Addition, unary plus
	–	Subtraction, unary minus
	*	Multiplication
	/	Division
String	+	String concatenation

(continued on next page)

Table 3–2 (Cont.) VAXTPU Operators

Type	Symbol	Description
	-	String reduction
	*	String replication
Relational	<>	Not equal to
	=	Equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
Pattern	|	Pattern alternation
	@	Partial pattern assignment
	+	Pattern concatenation
	&	Pattern linkage
Logical	AND	Boolean AND
	NOT	Boolean NOT
	OR	Boolean OR
	XOR	Boolean exclusive OR

Note that you can use the + operator to concatenate strings. You can also use the relational operators to compare a string with a string, a marker with a marker, or a range with a range.

The precedence of the operators in an expression determines the order in which the operands are evaluated. Table 3–3 lists the order of precedence for VAXTPU operators. Operators of equal precedence are listed on the same line.

Table 3–3 Operator Precedence

Operator	Precedence
unary +, unary –	Highest
NOT	
*, /, AND	
@, &, +, –, |, OR, XOR	
=, <>, <, <=, >, >=	
:=	Lowest

Expressions enclosed in parentheses are evaluated first. You must use parentheses for correct evaluation of an expression that combines relational operators.

You can use parentheses in an expression to force a particular order for combining operands. For example:

```
Expression              Result

8 * 5 / 2 - 4             16
8 * 5 / (2 - 4)          -20
```

3.7 Expressions

An expression can be a constant, a variable, a procedure, or a combination of these separated by operators. Expressions can be used in a VAXTPU procedure where an identifier or constant is required. Expressions are frequently used within VAXTPU conditional language statements.

The data types of all elements of a VAXTPU expression must be the same. Note, however, the following exceptions to this rule:

- You can mix keywords, strings, and pattern variables in expressions used to create patterns.
- You can mix data types when using the not equal (<>) and equal (=) relational operators.
- You can mix strings and integers when doing string replication.

Except for these cases, VAXTPU does not perform implicit type conversions to allow for the mixing of data types within an expression. If you mix data types, VAXTPU issues an error message.

In the following example, the elements (J > 4) and (my_string = "this is my string") each evaluate to an integer type (odd integers are true; even integers are false) so that they can be used following the VAXTPU IF statement:

```
IF (J > 4) AND (my_string = "this is my string")
THEN
    .
    .
    .
```

With the exception of patterns and the relational operators, the result of an expression is the same data type as the elements that make up the expression. The following example shows a pattern expression that uses a string data type on the right-hand side of the expression. The pattern keywords LINE_BEGIN and REMAIN are used with the string constant "the" to create a pattern data type that is stored in the variable *pat1*:

```
pat1 := LINE_BEGIN + "the" + REMAIN;
```

Whenever possible, the VAXTPU compiler evaluates constant expressions at compile time. VAXTPU built-in procedures that can return a constant value given constant input are evaluated at compile time.

In the example below, the variable *fubar* has a single string assigned to it:

```
fubar := ASCII (27) + "[0m";
```

Caution: Do not assume that the VAXTPU compiler automatically evaluates an expression in left-to-right order. In future releases, the compiler may evaluate expressions of equal precedence in any order.

To avoid the need to rewrite code, you should write as if this compiler optimization were already implemented. If you need the compiler to evaluate an expression in a particular order, you should force the compiler to evaluate each operand in order before using the expression. To do so, use each operand in an assignment statement before using it in an expression. For example, suppose you want to use ROUTINE_1 and ROUTINE_2 in an expression. Suppose, too, that ROUTINE_1 must be evaluated first because it prompts for user input. To get this result, you could use the following code:

```
PARTIAL_1 := ROUTINE_1;
PARTIAL_2 := ROUTINE_2;
```

You could then use a statement in which the order of evaluation was important, such as the following:

```
IF PARTIAL_1 OR PARTIAL_2
   .
   .
   .
```

There are five types of VAXTPU expressions:

- Arithmetic
- Relational
- Pattern
- Boolean
- String

The following sections discuss each of these expression types.

3.7.1 Arithmetic Expressions

You can use any of the arithmetic operators (+, –, *, /) with integer data types to form arithmetic expressions. VAXTPU performs only integer arithmetic. The following are examples of valid VAXTPU expressions:

```
12 + 4              ! adds two integers
"abc" + "def"       ! concatenates two strings
```

The following is not a valid VAXTPU expression because it mixes data types:

```
"abc" + 12           ! you cannot mix data types
```

When performing integer division, VAXTPU truncates the remainder; it does not round. The following examples show the results of division operations:

```
Expression          Result
 39 / 10              3
-39 / 10             -3
```

3.7.2 Relational Expressions

A relational expression tests the relationship between items of the same data type and returns an integer result. If the relationship is true, the result is integer 1; if the relationship is false, the result is integer 0.

Use the following relational operators with any of the VAXTPU data types:

- Not equal operator (<>)
- Equal operator (=)

For example, the following code fragment tests whether *string1* starts with a letter that occurs later in the alphabet than the starting letter of *string2*:

```
string1 := "gastropod";
string2 := "arachnid";
IF string1 > string2
THEN
    MESSAGE ("Out of alphabetical order ");
ENDIF;
```

Use the following relational operators for comparisons of integers, strings, or markers:

- Greater than operator (>)
- Less than operator (<)
- Greater than or equal to operator (>=)
- Less than or equal to operator (<=)

When used with markers, these operators test whether one marker is closer to (or farther from) the top of the buffer than another marker. (If markers are in different buffers, they will return as false.) For example, the procedure in Example 3–3 uses relational operators to determine which half of the buffer the cursor is located in.

Example 3–3 A Procedure Using Relational Operators on Markers

```
PROCEDURE which_half

LOCAL  number_lines,
       saved_mark;

saved_mark := MARK (FREE_CURSOR);
POSITION (BEGINNING_OF (CURRENT_BUFFER));
number_lines := GET_INFO (current_buffer, "record_count");
IF number_lines = 0
THEN
    MESSAGE ("The current buffer is empty");
ELSE
    MOVE_VERTICAL (number_lines/2);
      IF MARK (FREE_CURSOR) = saved_mark
      THEN
          MESSAGE ("You are at the middle of the buffer");
      ELSE
          IF MARK (FREE_CURSOR) < saved_mark
          THEN
            MESSAGE ("You are in the second half of the buffer");
          ELSE
            MESSAGE ("You are in the first half of the buffer");
          ENDIF;
      ENDIF;
ENDIF;

ENDPROCEDURE;
```

3.7.3 Pattern Expressions

A pattern expression consists of the pattern operators (+, &, |, @) combined with string constants, string variables, pattern variables, pattern procedures, pattern keywords, or parentheses. The following are valid pattern expressions:

```
pat1 := LINE_BEGIN + SPAN ("0123456789") + ANY ("abc");

pat2 := LINE_END + ("end"|"begin");

pat3 := SCAN (';"!') + (NOTANY ("'") & LINE_END);
```

See Chapter 2 for more information on pattern expressions.

3.7.4 Boolean Expressions

VAXTPU performs bitwise logical operations on Boolean expressions. This means that the logical operation is performed on the individual bits of the operands to produce the individual bits of the result. In the example below, the value of *user_variable* is set to 3.

```
user_variable := 3 AND 7;
```

As another example, if *user_var* were %X7777 (30583), then you would use the following statement to set *user_var* to %x0077 (119):

```
user_var := user_var AND %XFF
```

A true value in VAXTPU is any odd integer; a false value is any even integer. Use the logical operators (AND, NOT, OR, XOR) to combine one or more expressions. VAXTPU evaluates Boolean expressions enclosed in parentheses before other elements. The following example shows the use of parentheses to ensure that the Boolean expression is evaluated correctly:

```
IF (x = 12) AND (y <> 40)
THEN
  .
  .
  .
ENDIF;
```

3.8 Reserved Words

Reserved words are words that are defined by VAXTPU and that have a special meaning for the compiler.

VAXTPU reserved words can be divided into the following categories:

- Keywords
- Built-in procedure names
- Predefined constants
- Language elements

The following sections describe the categories of reserved words.

3.8.1 Keywords

Keywords are a VAXTPU data type. They are reserved words that have special meaning to the compiler. VAXTPU keywords can be redefined by the user only in local declarations (local constants, local variables, and parameters in a parameter list). If you give a local constant, local variable, or parameter the same name as that of a keyword, the compiler issues a message notifying you that the local declaration or parameter temporarily supersedes the keyword. In such a circumstance, the keyword is said to be **occluded**. See Chapter 2 for more information on keywords.

3.8.2 Built-In Procedure Names

The VAXTPU language has many built-in procedures that perform functions such as screen management, key definition, text manipulation, and program execution. VAXTPU built-in procedures are reserved words that can be redefined by the user only in local declarations (local constants, local variables, and parameters in a parameter list). If you give a local constant, local variable, or parameter the same name as that of a built-in procedure, the compiler issues a message notifying you that the local declaration or parameter temporarily supersedes the built-in. In such a circumstance, the built-in is said to be **occluded**. See the VAXTPU Reference Section for a complete description of the VAXTPU built-in procedures.

3.8.3 Predefined Constants

The following is a list of predefined global constants that VAXTPU sets up. These constants cannot be redefined by the user.

- FALSE
- TPU$K_ALT_MODIFIED
- TPU$K_CTRL_MODIFIED
- TPU$K_HELP_MODIFIED
- TPU$K_MESSAGE_FACILITY
- TPU$K_MESSAGE_ID
- TPU$K_MESSAGE_SEVERITY
- TPU$K_MESSAGE_TEXT
- TPU$K_SEARCH_CASE
- TPU$K_SEARCH_DIACRITICAL
- TPU$K_SHIFT_MODIFIED
- TPU$K_UNSPECIFIED
- TRUE

3.8.4 Declarations and Statements

A VAXTPU program can consist of a sequence of declarations and statements. These declarations and statements control the action performed in a procedure or a program. The following reserved words are the language elements that when combined properly make up the declarations and statements of VAXTPU.

- Module declaration
  - MODULE
  - IDENT
  - ENDMODULE
- Procedure declaration
  - PROCEDURE
  - ENDPROCEDURE
- Repetitive statement
  - LOOP
  - EXITIF
  - ENDLOOP
- Conditional statement
  - IF
  - THEN

  - ELSE
  - ENDIF
- Case statement
  - CASE
  - FROM
  - TO
  - INRANGE
  - OUTRANGE
  - ENDCASE
- Error statement
  - ON_ERROR
  - ENDON_ERROR
- RETURN statement
- ABORT statement
- Miscellaneous declarations
  - EQUIVALENCE
  - LOCAL
  - CONSTANT
  - VARIABLE

GLOBAL, UNIVERSAL, BEGIN, and END are words reserved for future expansion of the VAXTPU language.

The VAXTPU declarations and statements are reserved words that cannot be redefined by the user. Any attempt to redefine these words results in a compilation error.

3.8.4.1 The Module Declaration

The MODULE/ENDMODULE declaration allows you to group a series of global CONSTANT declarations, VARIABLE declarations, PROCEDURE declarations, and executable statements as one entity. After you compile a module, the compiler will generate two procedures for you. One procedure returns the identification for the module and the other contains all the executable statements for the module. The procedure names generated by the compiler are *module-name_MODULE_IDENT* and *module-name_MODULE_INIT*, respectively.

Syntax

```
MODULE module-name IDENT string-literal
   [declarations]
   [ON_ERROR ... ENDON_ERROR]
   statement_1;
        .
        .
        .
   statement_n;
ENDMODULE
```

The declarations part of a module can include any number of global VARIABLE, CONSTANT, and PROCEDURE declarations.

The ON_ERROR/ENDON_ERROR block, if used, must appear after the declarations and before the VAXTPU statements that make up the body of the module. Statements that make up the body of a module must be separated with semicolons. For more information on error handlers, see Section 3.8.4.7.

In the following example, the two procedures that are created by the compiler are *user_mod_module_ident* and *user_mod_module_init*. *User_mod_module_ident* returns the string "v1.0". *User_mod_module_init* calls the routine *user_hello*.

```
MODULE user_mod IDENT "v1.0"

PROCEDURE user_hello
    MESSAGE ("Hello");
ENDPROCEDURE;

ON_ERROR
    MESSAGE ("Good-bye");
END_ON_ERROR;

user_hello;
ENDMODULE
```

3.8.4.2 The Procedure Declaration

The PROCEDURE/ENDPROCEDURE declaration delimits a series of VAXTPU statements so they can be called as a unit. The PROCEDURE /ENDPROCEDURE combination allows you to declare a procedure with a name so that you can call it from another procedure or from the command line of a VAXTPU editing interface. Once you have compiled a procedure, you can enter the procedure name as a statement in another procedure, or enter the procedure name after the *VAXTPU Statement:* prompt on the command line of EVE.

Syntax

```
PROCEDURE procedure-name [[ (parameter-list) ]]
   [[local-declarations]]
   [[ON_ERROR ... ENDON_ERROR]]
   statement_1;
   statement_2;
        .
        .
        .
   statement_n;
ENDPROCEDURE;
```

The local declarations part of a procedure can include any number of LOCAL and CONSTANT declarations.

The ON_ERROR/ENDON_ERROR block, if used, must appear after the declarations and before the VAXTPU statements that make up the body of the procedure. For more information on error handlers, see Section 3.8.4.7.

After the ON_ERROR/ENDON_ERROR block, you can use any kind of VAXTPU language statements in the body of a procedure except another ON_ERROR/ENDON_ERROR block. Statements that make up the body of a procedure must be separated with semicolons.

Example

```
PROCEDURE version
   MESSAGE ("This is Version 1-020");
ENDPROCEDURE;
```

This procedure writes the text "This is Version 1–020" in the message area.

3.8.4.2.1 Procedure Names

A procedure name can be any valid identifier that is not a VAXTPU reserved word. Digital suggests that you use a convention when naming your procedures. For instance, you might prefix procedure names with your initials. In this way, you can easily distinguish procedures that you write from other procedures such as the VAXTPU built-in procedures. For example, if John Smith writes a procedure that creates two windows, he might name his procedure *js_two_windows*. This helps ensure that his procedure name is a unique name. Most of the sample procedures in this manual have the prefix *user_* with procedure names. Digital suggests that you replace the prefix *user* with your initials.

3.8.4.2.2 Procedure Parameters

Using parameters with procedures is optional. If you use parameters, they can be input parameters, output parameters, or both. For example:

```
PROCEDURE user_input_output (a, b)
   a := a + 5;
   b := a;
ENDPROCEDURE;
```

In the preceding procedure, *a* is an input parameter. It is also an output parameter because it is modified by the procedure *input_output*. In the same procedure, *b* is an output parameter.

The scope of procedure parameters is limited to the procedure in which they are defined. The maximum number of parameters in a parameter list is 127. A procedure can declare its parameters as required or optional. Required parameters and optional parameters are separated by a semicolon. Parameters before the semicolon are required parameters; those after the semicolon are optional. If no semicolon is specified, then the parameters are required.

Syntax

PROCEDURE proc-name [([req-param [...]] [;opt-param [...]])]
.
.
.
ENDPROCEDURE;

A procedure parameter is a place holder or dummy identifier that is replaced by an actual value in the program that calls the procedure. The value that replaces a parameter is called an **argument**. Arguments can be expressions. There does not have to be any correlation between the names used for parameters and the values used for arguments. All arguments are passed by reference. Example 3–4 shows a simple procedure with parameters.

Example 3–4 Simple Procedure with Parameters

```
!This procedure adds two integers. The parameters, int1 and int2,
!are replaced by the actual values that the user supplies.
!The result of the addition is written to the message area.

PROCEDURE ADD (int1, int2)

   MESSAGE (STR (int1 + int2));

ENDPROCEDURE;
```

For example, call the procedure ADD and specify the values 5 and 6 as arguments, as follows:

```
ADD (5, 6);
```

The string "11" is written to the message buffer.

Any caller of a procedure must call it using all required parameters. The caller can also use optional parameters. If the required parameters are not present or the procedure is called with too many parameters (more than the sum of the required and optional parameters), then VAXTPU issues an error.

If a procedure is called with the required number of parameters, but with less than the maximum number of parameters, then the remaining parameters up to the maximum automatically become "null parameters." A null parameter is a modifiable parameter of data type unspecified. A

null parameter can be assigned a value and will become the value it is assigned, but the parameter's value is discarded when the procedure exits.

Null parameters can also be explicitly passed to a procedure. This is done by omitting a parameter when calling the procedure.

Example 3–5 shows a more complex procedure that uses optional parameters.

Example 3–5 Complex Procedure with Optional Parameters

```
CONSTANT
    user_warning       := 0,        ! Warning severity code
    user_success       := 1,        ! Success severity code
    user_error         := 2,        ! Error severity code
    user_informational := 3,        ! Informational severity code
    user_fatal         := 4;        ! Fatal severity code
!
! Output a message with fatal/error/warning flash.
!
PROCEDURE user_message (the_text; the_severity)

LOCAL flash_it;
!
! Only flash warning, error, or fatal messages.
!
CASE the_severity FROM user_warning TO user_fatal
    [user_warning, user_error, user_fatal] : flash_it := TRUE;

    [user_success, user_informational] : flash_it := FALSE;

    [OUTRANGE] : flash_it := FALSE;

ENDCASE;
!
! Output the message - flash it, if appropriate.
!
MESSAGE (the_text);
IF flash_it
THEN
    SLEEP ("0 00:00:00.3");
    MESSAGE ("");
    SLEEP ("0 00:00:00.3");
    MESSAGE (the_text);
ENDIF;

ENDPROCEDURE;
```

Caution: Do not assume that the VAXTPU compiler automatically evaluates parameters in the order in which you place them. In future releases of VAXTPU, the compiler may evaluate parameters in any order.

To avoid the need to rewrite code, you should write as if this compiler optimization were already implemented. If you need the compiler to evaluate parameters in a particular order, you should force the compiler to evaluate each parameter in order before calling the procedure. To do so, use each parameter in an assignment statement before calling the procedure. For example, suppose you want to call a procedure whose parameter list includes PARAM_1 and PARAM_2. Suppose, too, that

PARAM_1 must be evaluated first. To get this result, you could use the following code:

```
partial_1 := param_1;
partial_2 := param_2;
my_procedure (partial_1, partial_2);
```

3.8.4.2.3 Procedures That Return a Result

Procedures that return a result are called **function procedures**. Example 3–6 shows a procedure that returns a true (1) or false (0) value.

Note: All VAXTPU procedures return a result. If they do not do so explicitly, VAXTPU returns 0.

Example 3–6 Procedure That Returns a Result

```
PROCEDURE user_on_end_of_line !test if at eol, return true or false

IF CURRENT_OFFSET = LENGTH (CURRENT_LINE)    ! we are on eol
THEN
    user_on_end_of_line := 1                   ! return true
ELSE
    user_on_end_of_line := 0                   ! return false
ENDIF;

ENDPROCEDURE;
```

Another way of assigning a value of 1 or 0 to a procedure is to use the VAXTPU language statement RETURN followed by a value. See Example 3–13.

You can use a procedure that returns a result as a part of a conditional statement to test for certain conditions. Example 3–7 shows the procedure in Example 3–6 within another procedure.

Example 3–7 Procedure Within Another Procedure

```
PROCEDURE user_nested_procedure
   .
   .
   .
IF user_on_end_of_line = 1              ! at the eol mark
THEN
    MESSAGE ("Cursor is at the end of the line")
ELSE
    MESSAGE ("Cursor is not at the end of the line")
ENDIF;
   .
   .
   .
ENDPROCEDURE;
```

3.8.4.2.4 Recursive Procedures

Procedures that call themselves are called **recursive procedures**. Example 3–8 shows a procedure named *user_reverse* that displays a list of responses to the built-in procedure READ_LINE in reverse order. Notice that there is a call to the procedure *user_reverse* within the procedure body.

Example 3–8 Recursive Procedure

```
PROCEDURE user_reverse
LOCAL temp_string;

temp_string := READ_LINE("input>");
                            ! Read a response

IF temp_string <> " "       ! Quit if nothing entered
                            ! but the RETURN key.
THEN
    user_reverse            ! Call user_reverse recursively
ELSE
    RETURN                  ! All done, go to display lines
ENDIF;
MESSAGE (temp_string);      ! Display lines typed in reverse order
                            ! in the message window
ENDPROCEDURE;
```

3.8.4.2.5 Local Variables

The use of local variables in procedures is optional. If you use local variables, they hold the values that you assign them only in the procedure in which you declare them. The maximum number of local variables that you can use is 255. Local variables are initialized to 0.

Syntax

LOCAL variable-name [,...];

Note that if you declare a local variable in a procedure and, in the same procedure, use the EXECUTE built-in to assign a value to a variable with the same name as the local variable, the result of the EXECUTE built-in has no effect on the local variable. For example, consider the following code fragment:

```
PROCEDURE test
   LOCAL x;
   EXECUTE ("x := 3");
   MOVE_VERTICAL (x);
ENDPROCEDURE;
```

In this fragment, when the compiler evaluates the string "x := 3", the compiler assumes *x* is a global variable. The compiler creates a global variable *x* (if none exists) and assigns the value 3 to the variable. When the built-in MOVE_VERTICAL uses the local variable *x*, the local variable has the value 0 and the MOVE_VERTICAL built-in has no effect.

Note that local variables may also be declared in unbound code. See Section 3.8.4.10.2.

3.8.4.2.6 Constants

The use of constants in procedures is optional. The scope of a constant declared within a procedure is limited to the procedure in which it is defined. See Section 3.8.4.10.3 for more information on the CONSTANT declaration.

Syntax

CONSTANT constant-name := compile-time-constant-expression [,...];

3.8.4.2.7 ON_ERROR Statements

The use of ON_ERROR statements in procedures is optional. If you use an ON_ERROR statement, you must place it at the top of the procedure just after any LOCAL and CONSTANT declarations. The ON_ERROR statement specifies the action or actions to be taken if an ERROR or WARNING status is returned. See Section 3.8.4.7 for more information on ON_ERROR statements.

3.8.4.3 The Assignment Statement

The assignment statement assigns a value to a variable. In so doing, it associates the variable with the appropriate data type.

Syntax

```
identifier := expression;
```

Note that the assignment operator is a combination of two characters, a colon and an equal sign (:=). Do not confuse this operator with the equal sign (=), which is a relational operator that checks for equality.

VAXTPU does not do any type checking on the data type being stored. Any data type may be stored in any variable.

Example

```
x := "abc";
```

This assignment statement stores the string "*abc*" in variable *x*.

3.8.4.4 The Repetitive Statement

The LOOP/ENDLOOP statements specify the repetitive execution of a statement or statements until the condition specified by EXITIF is met.

Syntax

```
LOOP
   statement_1;
   statement_2;
    .
    .
    .
   EXITIF expression;
   statement_n;
ENDLOOP;
```

The EXITIF statement is the mechanism for exiting from a loop. You can place the EXITIF statement anywhere inside a LOOP/ENDLOOP combination. You can also use the EXITIF statement as many times as you like. When the EXITIF statement is true, it causes a branch to the statement following the ENDLOOP statement.

The syntax of the EXITIF statement is as follows:

```
EXITIF expression;
```

Note that the expression is optional—without it, EXITIF always exits the loop.

Any VAXTPU language statement except an ON_ERROR statement can appear inside a LOOP/ENDLOOP combination.

Example

```
LOOP
   EXITIF CURRENT_OFFSET = 0;
   temp_string := CURRENT_CHARACTER;
   EXITIF (temp_string <> " ") AND
          (temp_string <> ASCII(9));
   MOVE_HORIZONTAL (-1);
   temp_length := temp_length + 1;
ENDLOOP;
```

This procedure uses the EXITIF statement twice. Each expression following an EXITIF statement defines a condition that causes an exit from the loop. The statements in the loop are repeated until one of the EXITIF conditions is met.

3.8.4.5 The Conditional Statement

The IF/THEN statement causes the execution of a statement or group of statements, depending on the value of a Boolean expression. If the expression is true, the statement is executed. Otherwise, program control passes to the statement following the IF/THEN statement.

The optional ELSE clause provides an alternative group of statements for execution. The ELSE clause is executed if the test condition specified by IF/THEN is false.

The ENDIF statement specifies the end of a conditional statement.

Syntax

```
IF expression
THEN
    statement_1;
        .
        .
        .
    statement_n
[ELSE
    alternate-statement_1;
        .
        .
        .
    alternate-statement_n;]
ENDIF;
```

You can use any VAXTPU language statements except ON_ERROR statements in a THEN or ELSE clause.

Example

```
PROCEDURE set_direct

MESSAGE ("Press PF3 or PF4 to indicate direction");
temp_char := READ_KEY;
IF temp_char = KP5
THEN
    SET (REVERSE, CURRENT_BUFFER);
ELSE
    IF temp_char = KP4
    THEN
        SET (FORWARD, CURRENT_BUFFER);
    ENDIF;
ENDIF;

ENDPROCEDURE;
```

In this example, nested IF/THEN/ELSE statements test whether a buffer direction should be forward or reverse.

Caution: Do not assume that the VAXTPU compiler automatically evaluates all parts of an IF statement. In future releases, the compiler may evaluate only as much of an IF statement as needed to determine if the statement is true or false. For example, if two clauses of an IF statement are joined with an AND operator and one clause is false, the compiler in future releases may not evaluate the other clause because the condition will be false in any case. Similarly, if two clauses of an IF statement are joined with an OR operator and the one clause is true, the compiler may not evaluate the other clause.

To avoid the need to rewrite code, you should write as if this compiler optimization were already implemented. If you need the compiler to evaluate all clauses of a conditional statement, you should force the compiler to evaluate each clause before using the conditional statement. To do so, use each clause in an assignment statement before using it in a conditional statement. For example, suppose you want the compiler to evaluate both CLAUSE_1 and CLAUSE_2 in a conditional statement. To get this result, you could use the following code:

```
relation_1 := clause_1;
relation_2 := clause_2;
IF relation_1 AND relation_2
THEN
  .
  .
  .
ENDIF;
```

3.8.4.6 The Case Statement

The CASE statement is a selection control structure that allows you to list several alternate actions and choose one of them to be executed at run time. In a CASE statement constant values, or case labels, are associated with the possible executable statements or actions to be performed. The CASE statement then executes the statement or statements labeled with a value that matches the value of the case selector.

Syntax

```
CASE case-selector [[FROM
lower-constant-expr, TO upper-constant-expr]]
      [constant-expr_1 [[,...]] : statement [[,...]];
      [constant-expr_2 [[,...]] : statement [[,...]];
            .
            .
            .
      [constant-expr_n [[,...]] : statement [[,...]];

    [[INRANGE] : statement [[,...]] ;]

    [[OUTRANGE] : statement [[,...]] ;]
ENDCASE;
```

Note that the single brackets are not optional for case constants. Example 3–9 shows how to use the CASE statement in a procedure.

CASE constant expressions must evaluate at compile time to either a keyword, a string constant, or an integer constant. All constant expressions in the CASE statement must be of the same data type. There are two special case constants in VAXTPU: INRANGE and OUTRANGE. INRANGE matches anything that falls within the case range that does not have a case label associated with it. OUTRANGE matches anything that falls outside the case range. These special case constants are optional.

FROM and TO clauses of a CASE statement are not required. Note that if FROM and TO clauses are not specified, INRANGE and OUTRANGE labels refer to data between the minimum and maximum specified labels.

Example 3–9 shows a sample procedure that uses the CASE statement.

Example 3–9 Procedure Using the CASE Statement

```
PROCEDURE grades

answers := READ_LINE ("Enter number of correct answers:",5);
answers := INT (answers);

CASE answers FROM 0 TO 10
                   [10] : score := "A+";
                    [9] : score := "A";
                    [8] : score := "B";
                    [7] : score := "C";
                    [6] : score := "D";
           [0,1,2,3,4,5] : score := "F";
              [OUTRANGE] : score := "Invalid entry.";
ENDCASE;

MESSAGE (score);

ENDPROCEDURE;
```

This CASE statement compares the value of the constant selector *answers* to the case labels (the numbers 0 through 10). If the value of *answers* is any of the numbers from 0 through 10, the statement to the right of that number is executed. If the value of *answers* is outside the range of 0

through 10, the statement to the right of [OUTRANGE] is executed. The value of *score* is written in the message area after the execution of the CASE statement.

3.8.4.7 Error Handling

A block of code starting with ON_ERROR and ending with ENDON_ERROR defines the actions that are to be taken when a procedure fails to execute successfully. Such a block of code is called an *error handler*. An error handler is an optional part of a VAXTPU procedure or program. An error handler traps WARNING and ERROR status values. (See SET (INFORMATIONAL) and SET (SUCCESS) in the VAXTPU Reference Section for information on handling informational and success status values.)

It is good programming practice to put an error handler in all but the simplest procedures. However, if you omit the error handler, VAXTPU's default error handling behavior is as follows:

- If the user presses CTRL/C, VAXTPU places an error message in the message buffer, exits normally from all currently active procedures (in their reverse calling order), and returns to the "wait for next key" loop.
- If an error or warning is generated during a CALL_USER routine, ERROR is set to the keyword representing the failure status of the routine, ERROR_LINE is set to the line number of the error, and ERROR_TEXT is set to the message associated with the error or warning. VAXTPU places the message in the message buffer, then resumes execution at the statement after the statement that generated the error or warning.
- For other errors and warnings, ERROR is set to the keyword representing the error or warning, ERROR_LINE is set to the line number of the error, and ERROR_TEXT is set to the message associated with the error or warning. VAXTPU places the message in the message buffer, then resumes execution at the statement after the statement that generated the error or warning.

In a procedure, the error handler must be placed at the beginning of a procedure; after the procedure parameter list, the LOCAL or CONSTANT declarations, if present, and before the body of the procedure. In a program, the ON_ERROR language statements must be placed after all the global declarations (PROCEDURE, CONSTANT, and VARIABLE) and before any executable statements. Error statements can contain any VAXTPU language statements except other ON_ERROR statements.

There are three VAXTPU lexical elements that are useful in an error handler: ERROR, ERROR_LINE, and ERROR_TEXT.

ERROR returns a keyword for the error or warning. The VAXTPU Reference Section includes information on the possible error and warning keywords that can be returned by each built-in procedure. (See Appendix D for an alphabetized list of all the possible return statuses for VAXTPU and their severity levels. The *VMS System Messages and Recovery Procedures Reference Manual* includes all the possible return statuses for VAXTPU as well as the appropriate explanations and suggested user actions.)

ERROR_LINE returns the line number at which the error or warning occurs. If a procedure was compiled from a buffer or range, ERROR_LINE returns the line number within the buffer. (This may be different from the line number within the procedure.) If the procedure was compiled from a string, ERROR_LINE returns 1.

ERROR_TEXT returns the text of the error or warning, exactly as VAXTPU would display it in the message buffer, with all parameters filled in.

After the execution of an error statement, you can choose where to resume execution of a program. The options are the following:

- ABORT — This language statement causes an exit back to the VAXTPU "wait for next key" loop.
- RETURN — This language statement stops the execution of the procedure in which the error occurred but continues execution of the rest of the program.

If you do not specify ABORT or RETURN, the default is to continue executing the program from the point at which the error occurred.

VAXTPU provides two forms of error handler, procedural and case style.

3.8.4.7.1 Procedural Error Handlers

If a WARNING status is trapped by an ON_ERROR statement, the warning message is suppressed. However, if an ERROR status is trapped, the message is displayed. The ON_ERROR trap allows you to do additional error handling after the VAXTPU message is displayed.

Syntax

```
ON_ERROR
    statement_1;
    statement_2;
        .
        .
        .
    statement_n;
ENDON_ERROR;
```

Example 3–10 shows error statements at the beginning of a procedure. These statements return control to the caller if the input on the command line of an interface is not correct. Any warning or error status returned by a statement in the body of the procedure causes the error statements to be executed.

Example 3–10 Procedure Using the ON_ERROR Statement

```
!
! Gold 7 emulation (command line processing)
!
PROCEDURE command_line

LOCAL
   line_read, x;

ON_ERROR
    MESSAGE ("Unrecognized command: " + line_read);
    RETURN;
ENDON_ERROR;
!
! Get the command(s) to execute
!
line_read := READ_LINE ("VAXTPU Statement: "); ! get line from user
!
! compile them

!
IF line_read <> ""
THEN
    x := COMPILE (line_read);
ELSE
    RETURN
ENDIF;
!
! execute
!
IF x <> 0
THEN
    EXECUTE (x);
ENDIF;

ENDPROCEDURE;
```

The effects of a procedural error handler are as follows:

- If the user presses CTRL/C, VAXTPU places an error message in the message buffer, exits normally from all currently active procedures (in their reverse calling order), and returns to the "wait for next key" loop.
- If an error or warning is generated during a CALL_USER routine, ERROR is set to a keyword representing the failure status of the routine, ERROR_LINE is set to the line number of the error, and ERROR_TEXT is set to a warning or error message that is placed in the message buffer. Finally, VAXTPU runs the error handler code.
- For other warnings and errors, ERROR is set to a keyword representing the error or warning, ERROR_LINE is set to the line number of the error, and ERROR_TEXT is set to the error or warning message associated with the keyword. VAXTPU places error messages in the message buffer but suppresses the display of warning messages. Finally, VAXTPU runs the error handler code.

If an error or warning is generated during execution of a procedural error handler, VAXTPU behaves as follows:

- If the user presses CTRL/C during the error handler, VAXTPU puts an error message in the message buffer, exits normally from all currently active procedures (in their reverse calling order), and returns to the "wait for next key" loop.
- For other errors and warnings, the appropriate error or warning message is written to the message buffer. VAXTPU resumes execution at the next statement after the statement that generated the error.

3.8.4.7.2 Case-Style Error Handlers

Case-style error handlers provide a number of advantages over procedural error handlers. Case-style error handlers allow you to do the following:

- Suppress the automatic display of both WARNING and ERROR status messages
- Trap the TPU$_CONTROLC status
- Write clearer code

Syntax

```
ON_ERROR
    [condition_1]: statement_1;...
    [condition_2]: statement_2;...
          .
          .
          .
    [condition_n]: statement_n;
ENDON_ERROR;
```

You can use the [OTHERWISE] selector alone in an error handler as a shortcut. For example, the following two error handlers have the same effect:

```
! This error handler uses [OTHERWISE] alone as a shortcut.

ON_ERROR
[OTHERWISE] : ;
ENDON_ERROR

! This error handler has the same effect as using
! [OTHERWISE] alone.

ON_ERROR
[OTHERWISE] :
   LEARN_ABORT;
   RETURN (FALSE);
ENDON_ERROR;
```

Example 3–11 from the EVE editor shows a procedure with a case-style error handler.

Example 3–11 Procedure with a Case-Style Error Handler

```
PROCEDURE eve$learn_abort

ON_ERROR
    [TPU$_CONTROLC]:
       MESSAGE (ERROR_TEXT);
       RETURN (LEARN_ABORT);
ENDON_ERROR;

IF LEARN_ABORT
THEN
    eve$message (EVE$_LEARNABORT);
    RETURN (TRUE);
ELSE
    RETURN (FALSE);
ENDIF;

ENDPROCEDURE;
```

If a program or procedure has a case-style error handler, VAXTPU handles errors and warnings as follows:

- If the user presses CTRL/C, VAXTPU determines whether the error handler contains a selector labeled TPU$_CONTROLC. If so, VAXTPU sets ERROR to TPU$_CONTROLC, ERROR_LINE to the line that VAXTPU was executing when CTRL/C was pressed, and ERROR_TEXT to the message associated with TPU$_CONTROLC. VAXTPU then executes the statements associated with the selector. If there is no TPU$_CONTROLC selector, VAXTPU exits from the error handler and looks for a TPU$_CONTROLC selector in the procedures or program (if any) in which the current procedure is nested. If no TPU$_CONTROLC selector is found in the containing procedures or program, VAXTPU places the message associated with TPU$_CONTROLC in the message buffer.
- If an error or warning is generated during a CALL_USER routine, ERROR is set to a keyword representing the failure status of the routine, ERROR_LINE is set to the line number of the error, and ERROR_TEXT is set to the warning or error message associated with the keyword. VAXTPU then processes the error handler that trapped the CALL_USER error in the same way that VAXTPU processes normal case-style error handlers as described below.
- For other warnings and errors, ERROR is set to a keyword representing the error or warning, ERROR_LINE is set to the line number of the error, and ERROR_TEXT is set to the error or warning message associated with the keyword.

  The way a case-style error handler processes an error or warning depends on how the error handler traps the error. There are three possible ways, as follows:

  – The error handler can trap the error using a selector that matches the error exactly (that is, using a selector other than OTHERWISE).

- The error handler can trap the error using the OTHERWISE selector.
- The error handler can completely fail to trap the error.

The following discussion explains how a case-style error handler processes an error or warning in each of these circumstances.

If the error or warning is trapped by a selector other than OTHERWISE, VAXTPU does not place the error or warning message in the message buffer unless the error handler code instructs it to do so. In this case, after setting ERROR, ERROR_LINE, and ERROR_TEXT, VAXTPU executes the code associated with the selector. If the code does not return to the calling procedure or program, VAXTPU checks whether one of the selectors associated with the code just executed is TPU$_CONTROLC or OTHERWISE. If so, VAXTPU performs the equivalent of the following sequence:

```
special_error_symbol := 0;
LEARN_ABORT;
RETURN (FALSE);
```

If not, the error handler terminates and VAXTPU resumes execution at the next statement after the statement that generated the error or warning.

For more information on the special error symbol in VAXTPU, see the description of the built-in SET (SPECIAL_ERROR_SYMBOL) in the VAXTPU Reference Section.

If the error or warning is trapped by the OTHERWISE selector, VAXTPU writes the associated error or warning message in the message buffer. Next, VAXTPU executes the code associated with the OTHERWISE selector. If the code does not return to the calling procedure or program, VAXTPU performs the equivalent of the following sequence:

```
special_error_symbol := 0;
LEARN_ABORT;
RETURN (FALSE);
```

If the error or warning is not trapped by any selector, VAXTPU writes the associated error or warning message in the message buffer. Next, VAXTPU performs the equivalent of the following sequence:

```
special_error_symbol := 0;
LEARN_ABORT;
RETURN (FALSE);
```

If an error or warning is generated during execution of a case-style error handler, VAXTPU behaves as follows:

- If the user presses CTRL/C during the error handler, VAXTPU sets ERROR to TPU$_CONTROLC, ERROR_LINE to the line being executed when CTRL/C was pressed, and ERROR_TEXT to the message associated with TPU$_CONTROLC.

If one of the case selectors in the error handler is TPU$_CONTROLC, VAXTPU executes the code associated with the selector. If the code does not return to the calling procedure or program, VAXTPU performs the equivalent of the following sequence:

```
special_error_symbol := 0;
LEARN_ABORT;
RETURN (FALSE);
```

If none of the selectors is TPU$_CONTROLC, then VAXTPU exits from the error handler and looks for a TPU$_CONTROLC selector in the procedures or program (if any) in which the current procedure is nested. If VAXTPU does not find a TPU$_CONTROLC selector in the containing procedures or program, VAXTPU places the message associated with TPU$_CONTROLC in the message buffer.

- If the error is not due to the user pressing CTRL/C, the error message is written to the message buffer and VAXTPU performs the equivalent of the following sequence:

```
special_error_symbol := 0;
LEARN_ABORT;
RETURN (FALSE);
```

In a procedure with a case-style error handler, an ABORT statement produces the same effect as the sequence CTRL/C, with one exception. An ABORT statement in the TPU$_CONTROLC clause of a case-style error handler does not reinvoke the TPU$_CONTROLC clause, as is the case when CTRL/C is pressed while TPU$_CONTROLC is executing. Instead, an ABORT statement causes VAXTPU to exit from the error handler and look for a TPU$_CONTROLC selector in the procedures or program (if any) in which the current procedure is nested. If VAXTPU does not find a TPU$_CONTROLC selector in the containing procedures or program, VAXTPU places the message associated with TPU$_CONTROLC in the message buffer.

3.8.4.7.3 CTRL/C Handling

The ability to trap a CTRL/C in your VAXTPU program is both powerful and dangerous. When a user presses CTRL/C, the user usually wants the application that is running to prompt for a new command. The ability to trap the CTRL/C is intended to allow a procedure to clean up and exit gracefully, not to thwart the user.

3.8.4.8 The RETURN Statement

This statement causes a return to the procedure that called the current procedure or program. The return is to the statement following the statement that called the current procedure or program. You can specify an expression after the RETURN statement and the value of this expression is passed to the calling procedure.

Syntax

RETURN expression;

Note that the expression is optional; if it is missing, VAXTPU supplies a 0. Also, the return statement itself is optional. That is, if VAXTPU reaches the endprocedure of a procedure before encountering a return statement, it will return 0.

Example 3–12 shows a sample procedure in which a value is returned to the calling procedure.

Example 3–12 Procedure That Returns a Value

```
PROCEDURE user_get_shift_key

LOCAL key_to_shift;  ! Keyword for key pressed after shift key

SET (SHIFT_KEY, LAST_KEY);
key_to_shift := KEY_NAME (READ_KEY, SHIFT_KEY);
RETURN key_to_shift;

ENDPROCEDURE;
```

In addition to using RETURN to pass a value, you can use a 1 (true) or a 0 (false) with the RETURN statement to indicate the status of a procedure. Example 3–13 shows this usage of the RETURN statement.

Example 3–13 Procedure Returning a Status

```
PROCEDURE user_at_end_of_line

! This procedure returns a 1 (true) if user is at the end of a
! line, or a 0 (false) if the current character is not at the
! end of a line

ON_ERROR
! Suppress warning message
   RETURN (1);
ENDON_ERROR;

IF CURRENT_OFFSET = LENGTH (CURRENT_LINE)
THEN
    RETURN (1);
ELSE
    RETURN (0);
ENDIF;

ENDPROCEDURE;
```

The RETURN statement is often used in the ON_ERROR section of a procedure to specify a return to the calling procedure if an error occurs in the current procedure. Example 3–14 uses the RETURN statement in an ON_ERROR section.

Example 3–14 Using RETURN in an ON_ERROR Section

```
! Attach to the parent process.  Used when EVE is spawned
! from DCL and run in a subprocess ("kept VAXTPU"). The
! ATTACH command can be used for more flexible process control.

PROCEDURE eve_attach

ON_ERROR
    IF ERROR = TPU$_NOPARENT
    THEN
        MESSAGE ("Not running VAXTPU in a subprocess");
        RETURN;
    ENDIF;
ENDON_ERROR;

ATTACH;

ENDPROCEDURE;
```

3.8.4.9 The ABORT Statement

The ABORT statement stops any executing procedures and causes VAXTPU to wait for the next keystroke. ABORT is commonly used in error handlers. For additional information on using ABORT in error handlers, see Section 3.8.4.7.

Syntax

ABORT

Example 3–15 shows a simple error handler containing an ABORT statement.

Example 3–15 Simple Error Handler

```
ON_ERROR
   MESSAGE ("Aborting procedure because of error.");
   ABORT;
ENDON_ERROR;
```

3.8.4.10 Miscellaneous Declarations

This section describes the VAXTPU language declarations EQUIVALENCE, LOCAL, CONSTANT, and VARIABLE.

3.8.4.10.1 EQUIVALENCE Statement

The EQUIVALENCE statement lets you create synonyms. Equivalences work only when both the *real_name* and the *synonym_name* are defined at the same time. You cannot save a section file containing the *real_name* and then later use that section file to extend code which uses an EQUIVALENCE of the saved name. To avoid problems, include all EQUIVALENCE statements in the same compilation unit where the *real_name* is defined. Alternatively, the equivalences can reside in different compilation units, but all of the compilation units must be used when building the section file from scratch. If you use a base section file that you extend interactively, you cannot make equivalences to procedures or variables defined in the base section file.

Syntax

```
EQUIVALENCE synonym_name1 = real_name1,
                            synonym_name2 = real_name2, ...;
```

Elements of the EQUIVALENCE Statement

real_name
A user-defined global variable or procedure name. If *real_name* is undefined, VAXTPU defines it as an ambiguous name. This ambiguous name can become a variable or procedure later.

synonym_name
A name to be defined as a synonym for the *real_name*.

3.8.4.10.2 LOCAL

This declaration is used to identify certain variables as local variables rather than global variables. All variables are considered to be global variables unless you explicitly use the LOCAL statement to identify them as local variables. The LOCAL declaration in a procedure is optional. It must be specified after the PROCEDURE statement and before any ON_ERROR statement. LOCAL declarations and CONSTANT declarations can be intermixed.

The maximum number of local variables you can declare in a procedure is 255. Local variables are initialized to 0.

Syntax

```
LOCAL
    variable-name [[,...]];
```

Local variables may also be declared in unbound code. Such variables are accessible only within that unbound code.

Unbound code can occur in the following places:

- Module initialization code. This occurs after all procedure declarations within a module but before the ENDMODULE statement.
- Executable code. This occurs after all module and procedure declarations in a file but before the end of file.

Example

The following example shows a complete compilation unit. This unit contains a module named *mmm* that in turn, contains a procedure *bat* and some initialization code *mmm_module_init*, a procedure *bar* defined outside the module, and some unbound code at the end of the file. In each of these sections of code, a local variable *x* is defined. The variable is displayed using the MESSAGE built-in.

```
MODULE mmm IDENT "mmm"

PROCEDURE bat;      ! Declare procedure "bat" in module "mmm"

LOCAL
    x;  ! "x" is local to procedure "bat"

    x := "Within procedure bat, within module mmm";
MESSAGE (x);

ENDPROCEDURE; ! End procedure "bat"
```

```
LOCAL
        x;                          ! "x" is local to
                                    ! procedure "mmm_module_init"

x := "Starting or ending the module init code";
MESSAGE (x);
bat;
MESSAGE (x);

ENDMODULE;              ! End module "mmm"

PROCEDURE bar           ! Declare procedure "bar"

LOCAL
        x;                          ! "x" is local to procedure "bar"

x := "In procedure bar, which is outside all modules";
MESSAGE (x);

ENDPROCEDURE;           ! End procedure "bar"

LOCAL
        x;                          ! "x" is local to the unbound code...

x := "Starting or ending the unbound, non-init code";
MESSAGE (x);
mmm_module_init;
bat;
bar;
MESSAGE (x);
EXIT;
```

If this code is included in a file TEMP.TPU, the following DCL command demonstrates the scope of the various local variables:

```
$  EDIT/TPU/NOSECTION/NOINITIALIZE/NODISPLAY/COMMAND=temp.tpu
42 lines read from file TEMP.TPU;1
Starting or ending the unbound, non-init code
Starting or ending the module init code
Within procedure bat, within module mmm
Starting or ending the module init code
Within procedure bat, within module mmm
In procedure bar, which is outside all modules
Starting or ending the unbound, non-init code
```

3.8.4.10.3 CONSTANT

This declaration is used to associate a name with certain constant expressions. The constant expression must evaluate at compile time to a keyword, a string, an integer, or an unspecified constant value. The maximum length of a string constant allowed in a constant declaration is about 4000 characters in length. VAXTPU sets up some predefined global constants. See Section 3.8.3 for a list of predefined constants.

Constants can either be globally or locally defined. Global constants are constants declared outside procedure declarations. Once a global constant has been defined, it is set for the life of the VAXTPU session. An attempt to redefine a constant will succeed only if the constant value is the same.

Local constants are constants declared within a procedure. A local CONSTANT declaration must be specified after the PROCEDURE statement and before any ON_ERROR statement. LOCAL statements and CONSTANT statements can be intermixed.

Syntax

CONSTANT
    constant-name := compile-time-constant-expression [[,...]];

3.8.4.10.4 VARIABLE

This declaration is used to identify certain variables as global variables. Any symbols that are neither declared nor used as the target of an assignment statement before being referenced by VAXTPU are assumed to be undefined procedures. The VARIABLE declaration must be used outside a procedure declaration. Global variables are initialized to unspecified.

Syntax

```
VARIABLE
    variable-name [[,...]];
```

3.9 Lexical Keywords

The following two sections explain the VAXTPU lexical keywords and how to use them for:

- Conditional compiling
- Specifying the radix of numeric constants

3.9.1 Conditional Compilation

The following lexical keywords control what code is compiled under different conditions:

- %IF
- %IFDEF
- %THEN
- %ELSE
- %ENDIF

Syntax

Conditional compilation lexical keywords are used in a manner similar to ordinary IF/THEN/ELSE/ENDIF statements. The syntax is as follows:

```
%IFDEF variable_or_proc_name %THEN ... [%ELSE ...] %ENDIF
```

or

```
%IF boolean_expression %THEN ... [%ELSE ...]  %ENDIF
```

Description

If you use the %IFDEF structure, specify *variable_or_proc_name* as the name of a VAXTPU procedure or variable. IFDEF is a statement that says: "if a variable or procedure with this name is defined." If the name is defined, the compiler compiles the code marked by %THEN. If the name is not defined, the compiler compiles the code marked by %ELSE.

If you use the %IF structure, specify *boolean_expression* as either a numeric constant or a defined global variable whose value is an integer. Any odd value is true and any even value is false. If the variable or constant contains a value that is odd, the compiler compiles the code marked by %THEN. If the variable or constant contains a value that is even, the compiler compiles the code marked by %ELSE.

You do not have to put conditional compilation lexical keywords at the beginning of a line. You can nest conditional statements to a depth of 2**32-1.

Example

```
ON_ERROR
    [TPU$_CREATEFAIL]:
%IF eve$x_option_decwindows
%THEN
      IF eve$x_decwindows_active
      THEN
          eve$popup_message (MESSAGE_TEXT (EVE$_CANTCREADCL, 1));
      ELSE
          eve$message (EVE$_CANTCREADCL);
      ENDIF;
%ELSE
      eve$message (EVE$_CANTCREADCL);
%ENDIF
      eve$learn_abort;
      RETURN (FALSE);
   [OTHERWISE]:
ENDON_ERROR;
```

This ON_ERROR procedure determines whether a popup message widget or a simple message is used, depending on whether the code is being compiled by a DECwindows version of VAXTPU.

3.9.2 Specifying the Radix of Numeric Constants

You can specify constants with binary, octal, hexadecimal, and decimal radices.

To specify a numeric constant in binary, precede the number with %B. The number can consist only of digits 0 and 1.

To specify a numeric constant in octal, precede the number with %O. The number can consist only of digits 0 through 7.

To specify a numeric constant in hexadecimal, precede the number with %X. The number can consist of digits 0–9 and A–F.

There is no radix specifier for decimal. Any numeric constant without an explicit radix specifier is assumed to be decimal. The radix specifier may be in uppercase or lowercase.

Examples

The following are examples of correct numeric constants:

```
!
! Many different ways of saying the same thing.
!
CONSTANT binary_constant := %b11111;
CONSTANT octal_constant := %o37;
CONSTANT decimal_constant := 31;
CONSTANT hex_constant := %x1f;
!
! Compile time expressions work, too.
!
CONSTANT negative_value := -%x1f;
CONSTANT strange_zero := hex_constant - %x1f;
```

Invalid constructs for numeric constants return the error level message TPU$_UNKLEXICAL, "Unknown lexical element" during compilation. The following examples are not valid:

```
constant bad_binary := %b123;       ! only 0's and 1's are legal.
constant bad_hex := %x10abg;        ! 'g' is illegal digit.
constant not_a_radix := %z0123;     ! No such radix.
```

4 VAXTPU Program Development

Previous sections have described the lexical elements of the VAXTPU language, such as data types, language statements, expressions, built-in procedures, and so on. This section describes how to combine these elements in VAXTPU programs. VAXTPU programs can be used to perform editing tasks, to customize or extend an existing application, or to implement your own application layered on VAXTPU.

For information on calling VAXTPU from a program written in another programming language, see the *VMS Utility Routines Manual.*

Before you start writing programs to customize or extend an existing application, be very familiar with the VAXTPU source code that creates the editor or application that you want to change. For example, if you use the Extensible VAX Editor (EVE) and you want to change the size of the main window, you must know and use the procedure name that EVE uses for that window. (If you were changing the main window, you would use the procedure name *eve$main_window*. Many of the EVE variables and procedure names begin with *eve$*.)

The sample procedures and syntax examples in this book use uppercase letters for items that you can enter exactly as shown. VAXTPU reserved words, such as built-in procedures, keywords, and language statements, are shown in uppercase. Lowercase items in a syntax example or sample procedure indicate that you must provide an appropriate substitute for that item.

This section discusses the following topics:

- Creating VAXTPU programs
- Creating DECwindows VAXTPU programs
- Writing code compatible with DECwindows EVE
- Compiling VAXTPU programs
- Executing VAXTPU programs
- Using VAXTPU startup files
- Debugging VAXTPU programs

4.1 Creating VAXTPU Programs

When you write a VAXTPU program, keep the following pointers in mind:

- You can use EVE or some other editor to enter or change the source code of a program in the VAXTPU language.
- A program can be a single executable statement or a collection of executable statements.

- You can use executable statements either within procedures or outside procedures. You must place all procedure declarations before any executable statements that are not in procedures.
- You can enter VAXTPU statements from within EVE by using the EVE command TPU. For more information on using this command, see the command description in the *VMS EVE Reference Manual* or see the *Guide to VMS Text Processing*.

4.1.1 Simple Programs

The following statement is an example of a simple program:

```
SHOW (SUMMARY);
```

The preceding statement, entered after the appropriate prompt from your editor, causes VAXTPU to execute the program associated with the SHOW (SUMMARY) statement. If you use EVE with a user-written command file, your screen may display text similar to Example 4–1:

Example 4–1 SHOW (SUMMARY) Display

```
 VAXTPU V2.6 1990-06-03 03:31

Journal file: LCLD$:[DOC.SRC]GET_INFO.TJL;1
Section file: TPU$SECTION
Section file was image activated
Timer Message:         working

 20 System buffers and 7 User buffers
 3768 calls to LIB$GET_VM, 360 calls to LIB$FREE_VM, 831528 bytes still allocated
```

4.1.2 Complex Programs

When writing complex VAXTPU programs, avoid the following practices:

- Creating very large procedures
- Creating a very large number of procedures
- Including large numbers of executable statements that are not within procedures

These practices, if carried to extremes, can cause the parser stack to overflow.

The VAXTPU parser currently allows a maximum stack depth of 1000 syntax tree nodes. When the parser first encounters a VAXTPU statement, the parser assigns each token in the statement to a syntax tree node. For example, the statement "a := 1" contains three tokens, each of which occupies a syntax tree node. After the parser parses this statement, only the assignment statement remains on the stack of nodes. The *a* and the *1* are subtrees to the assignment syntax tree node.

The most common cause of stack overflow, which is signaled by the status TPU$_STACKOVER, is creating one or more large procedures whose statements occupy too many syntax tree nodes. To make your program manageable by the parser, break the large procedures into smaller ones.

Other possible reasons for a TPU$_STACKOVER condition are that you have too many statements that are not in procedures, or that you have too many small procedures. If you have too many small procedures, you must either consolidate them or break them into separate files.

To see an example of a complex VAXTPU program, you can examine the source files that implement EVE. The EVE source code files, located in SYS$EXAMPLES:EVE$*.*, contain many procedure declarations and executable statements specifying EVE's screen layout and display. These files also contain key definitions specifying which editing operations are performed when you press certain keys on the keyboard. You can examine these files to learn the programming techniques that were used to create EVE.

See Section 4.6 for information on using a command file or section file to create or customize an application layered on VAXTPU. See Appendix G for information on using the EVE$BUILD module to layer applications on top of EVE.

4.1.3 Program Syntax

The rules for writing VAXTPU programs are very simple. You must use a semicolon to separate each executable statement from other statements. In a program, you must place all procedure declarations before any executable statements that are not part of a procedure declaration. For information on VAXTPU data types, see Chapter 2. For information on VAXTPU language elements, see Chapter 3. Example 4–2 shows the correct syntax for a VAXTPU program.

Example 4–2 Syntax of a VAXTPU Program

```
PROCEDURE
    .
    .
    .
ENDPROCEDURE

PROCEDURE;
    .
    .
    .
ENDPROCEDURE;
    .
    .
    .
PROCEDURE
    .
    .
    .
ENDPROCEDURE;
```

(continued on next page)

Example 4–2 (Cont.) Syntax of a VAXTPU Program

```
statement 1;
statement 2;
    .
    .
    .
statement n;
```

A variety of syntactically correct VAXTPU programs is shown in Example 4–3.

Example 4–3 Sample VAXTPU Programs

```
! Program 1
! This program consists of a single VAXTPU built-in procedure.

   SHOW (KEYWORDS);

! Program 2
! This program consists of an assignment statement that
! gives a value to the variable video_attribute

   video_attribute := UNDERLINE;

! Program 3
! This program consists of the VAXTPU LOOP statement (with
! a condition for exiting) and the VAXTPU built-in procedure ERASE_LINE.

  x := 0; LOOP x :=x+1; EXITIF x > 100; ERASE_LINE; ENDLOOP;

! Program 4
! This program consists of a single procedure that makes
! VAXTPU quit the editing session.

   PROCEDURE user_quit
      QUIT;          ! do VAXTPU quit operation
   ENDPROCEDURE;

! Program 5
! This program is a collection of procedures that
! makes VAXTPU accept "e", "ex", or "exi" as
! the command for a VAXTPU exit operation.

   PROCEDURE e
      EXIT;          ! do VAXTPU exit operation
   ENDPROCEDURE;

   PROCEDURE ex
      EXIT;
   ENDPROCEDURE;

   PROCEDURE exi
      EXIT;
   ENDPROCEDURE;
```

4.2 Programming in DECwindows VAXTPU

This section provides information about programming with DECwindows VAXTPU.

4.2.1 Widgets Supported by DECwindows VAXTPU

DECwindows VAXTPU enables you to create widgets from within VAXTPU programs by using the CREATE_WIDGET built-in. For information about how to use widgets to create a DECwindows text processing interface, see the *XUI Style Guide* and the *VMS DECwindows Guide to Application Programming*. For information about the characteristics of specific widgets, see the *VMS DECwindows Toolkit Routines Reference Manual*.

Using the CREATE_WIDGET built-in, you can create the following widgets in VAXTPU:

- Caution_box
- Dialog_box
- File_selection
- Label
- List_box
- Main_window
- Menu_bar
- Popup_attached_db
- Popup_dialog_box
- Popup_menu
- Pulldown_entry
- Pulldown_menu
- Push_button
- Scroll_bar (vertical and horizontal)
- Separator
- Simple_text
- Toggle_button

4.2.2 Input Focus Support in DECwindows VAXTPU

In VMS DECwindows, at most one of the applications on the screen can have the **input focus**; that is, can accept user input from the keyboard. For more information about the input focus, see the *XUI Style Guide*.

DECwindows VAXTPU automatically grabs the input focus whenever the user causes an unmodified M1DOWN event (that is, an event not modified by SHIFT, CTRL, or other modifying key) while the pointer cursor is in either of the following locations:

- VAXTPU's main window widget
- VAXTPU's title bar

When DECwindows VAXTPU grabs input focus or when an application layered on VAXTPU requests input focus, DECwindows assigns the input focus to VAXTPU only if and when it is possible to do so. Therefore, your application should use the GET_INFO (SCREEN, "input_focus") built-in to test whether it actually has the input focus before performing any operation that requires the input focus.

Digital recommends that you use *only* a DECwindows section file with DECwindows VAXTPU. (Note that all versions of EVE shipped with VMS V5.1 or later are compatible with DECwindows and are suitable for building DECwindows section files.) However, if you do not follow this recommendation, VAXTPU's automatic grabbing of the input focus allows your layered application to interact with other DECwindows applications.

4.2.3 Global Selection Support in DECwindows VAXTPU

Global selection in VMS DECwindows is a means of preserving information selected by the user so the user's selection, or data about the user's selection, can be passed between DECwindows applications. Each DECwindows application can own one or more global selections.

4.2.3.1 Difference Between Global Selection and Clipboard

A global selection differs from the clipboard in that the global selection changes dynamically as the user changes the select range, while the contents of the clipboard remain unchanged until the user uses a command (such as EVE's STORE TEXT command) that sends new information to the clipboard. Note that by default EVE does not use the clipboard.

4.2.3.2 Handling of Multiple Global Selections

At any particular time, a global selection is owned by at most one DECwindows application; a global selection can also be unowned. A DECwindows application can own more than one global selection at the same time. For example, an application layered on VAXTPU can own both the primary and secondary global selections. The DECwindows server determines which application currently owns which global selection. Information about a global selection property may be stored in different formats, but the format of a particular property must be the same for all DECwindows applications. VAXTPU directly accepts information that is stored in integer or string format. VAXTPU handles information in other formats by describing the information in an array. For more information about this array, see the descriptions of the built-ins GET_GLOBAL_SELECT and WRITE_GLOBAL_SELECT in the VAXTPU Reference Section.

Global selections are identified in VAXTPU either as strings or keywords. While DECwindows provides for many global selections, applications conforming to the *XUI Style Guide* are concerned with only two selections, the **primary** and **secondary** selections. VAXTPU provides a pair of keywords (PRIMARY and SECONDARY) to refer to these selections. VAXTPU also provides built-in procedures that allow layered applications to manipulate global selection information.

You can refer to other global selections by specifying a string instead of the keywords PRIMARY and SECONDARY. For example, if your application has a global selection whose name is *auxiliary*, specify the selection using the string "*auxiliary*". Note that selection names are case sensitive; the string "*auxiliary*" does not refer to the same global selection as the string "*AUXILIARY*".

4.2.3.3 Relation of Global Selection to Input Focus in DECwindows VAXTPU

An application that conforms to the *XUI Style Guide* requests ownership of the primary global selection in its input focus grab procedure. Regardless of whether the application conforms, when VAXTPU obtains the input focus, it automatically grabs the primary global selection if it is not already the owner. An application cannot prevent VAXTPU from attempting to assert ownership of the primary global selection when VAXTPU receives the input focus. If VAXTPU obtains the primary selection by grabbing ownership itself, VAXTPU automatically executes the application's global selection grab routine if one is present. If you are writing an application that conforms to the *XUI Style Guide* and you find that VAXTPU has had to grab ownership of the primary selection itself and execute the global select grab routine, your application may have a design problem.

4.2.3.4 DECwindows VAXTPU's Response to Requests for Information About the Global Selection

VAXTPU provides a three-level hierarchy for responding to requests from another application for information about the current selection. Applications layered on VAXTPU may specify a routine that responds to requests for information about global selections either for the entire application or for one or more buffers in the application. When VAXTPU receives a request for information, it checks whether there is a routine for the current buffer that responds to information about global selections. If no buffer-specific routine is available, VAXTPU checks for an application-wide routine. If no application-wide routine is available, VAXTPU attempts to respond to the request itself, but it can only respond to a limited number of requests. It provides information about the primary selection and provides information about the file name, font, line number, and text. VAXTPU responds to all other requests with a message that no information is available. Note that VAXTPU itself does not send requests for information about the global selection to other DECwindows applications. VAXTPU applications may do so using the various built-ins.

VAXTPU's responses to requests for information about the primary selection are as follows:

"FILE_NAME"	VAXTPU responds with the string returned by the built-in procedure GET_INFO (CURRENT_BUFFER, "file_name").
"FONT"	VAXTPU responds with the string returned by the built-in procedure GET_INFO (SYSTEM, "default_font").
"LINE_NUMBER"	VAXTPU responds with the value of type span containing the record number where the select range starts and the record number where the select range ends.
"TEXT" or "STRING"	VAXTPU responds with the text of the select range as a string, with each line break represented by a line feed.

Digital recommends that you not use a non-DECwindows section file with DECwindows VAXTPU. However, if you do not follow this recommendation, VAXTPU's automatic grabbing of the primary global selection allows your layered application to interact with other DECwindows applications. If an application requests information about the primary global selection while VAXTPU owns the selection, VAXTPU attempts to respond to the request if the application cannot do so. If VAXTPU responds to the request by sending the text of a buffer or range, VAXTPU converts the buffer or range to a string, converts line breaks to line feeds, and inserts padding blanks before text to fill any unoccupied space between the margins. If neither the application nor VAXTPU can respond to the request, VAXTPU informs DECwindows that the requested information is not available.

VAXTPU does not automatically grab the secondary selection. Layered applications are responsible for handling this selection.

4.2.4 Using Callbacks in DECwindows VAXTPU

This section presents background information on the DECwindows concept of **callbacks** and explains how DECwindows VAXTPU implements this concept.

4.2.4.1 Background on DECwindows Callbacks

A **callback** is a mechanism used by a DECwindows widget to notify an application that the widget has been modified in some way. DECwindows applications have one or more **callback routines** that define what the application does in response to the callback.

For more information about the use of callbacks and callback routines in DECwindows programs, see the *VMS DECwindows Guide to Application Programming*.

Callbacks can pass values known as **closures**, which are strings or integers whose function depends on the application you are writing. Note that closures are referred to as *tags* in DECwindows documentation. For more information about what closures are and how to use them, see Section 4.2.5.

4.2.4.2 Understanding the Difference Between VAXTPU's Internally-Defined Callback Routines and a Layered Application's Callback Routines

VAXTPU implements the DECwindows concept of callback routines by providing internally-defined routines that deliver the information obtained from a widget's callback to a layered application. These routines are referred to as "internally-defined VAXTPU callback routines."

Note that when a widget calls back to VAXTPU, VAXTPU packages the callback information, adds the information to its input queue, and returns to the widget. VAXTPU may not process the callback packet on its input queue until some time later. As a result, the information about the widget that VAXTPU gets from the callback may not match the information returned by the built-in GET_INFO (widget_variable, "widget_info").

When VAXTPU processes the callback packet, it executes the program or learn sequence that was associated with the widget, using the CREATE_WIDGET built-in or the SET (WIDGET_CALLBACK) built-in. This program or learn sequence controls what the application does in response to the callback information passed by the VAXTPU callback routines. An application's callback routines are referred to as "application-level callback action routines."

The following subsections present information on internally-defined VAXTPU callback routines first, and then present information on application-level callback action routines.

4.2.4.3 Using Internally-Defined VAXTPU Callback Routines with UIL

VAXTPU declares two internally-defined callback routines to the X Resources Manager (XRM) to handle incoming callbacks and dispatch them to the layered application:

- TPU$WIDGET_INTEGER_CALLBACK — Use this routine as the callback routine for all callbacks that have an integer closure.
- TPU$WIDGET_STRING_CALLBACK — Use this routine as the callback routine for all callbacks that have a string closure.

Note that although DECwindows allows you to specify a different callback routine for each reason that a widget can call back, DECwindows VAXTPU does not support this capability. Instead, it provides only the two callback routines mentioned.

Use these callback routines only if you are specifying a widget's callback resources in a User Interface Language (UIL) file. When a widget is part of an X Resource Manager hierarchy, do not include callback resource names or values in the array you pass to SET (WIDGET). Instead, specify one of the two internally defined callback routines in the UIL file.

4.2.4.4 Using Internally-Defined VAXTPU Callback Routines with Widgets Not Defined by UIL

Although the SET (WIDGET) built-in allows you to specify values for various resources of a widget, there are restrictions on specifying values for callback resources of widgets. When a widget is not part of an XUI Resource Manager hierarchy, specify the names of the callback resources in the array you pass to SET (WIDGET), and specify 0 as the value of each such callback resource. VAXTPU automatically substitutes its common callback entry point for the 0 value. Note that a widget calls back only for those reasons specified in the widget's argument list. If a reason is omitted from the list, the corresponding event does not cause a callback.

4.2.4.5 Using Application-Level Callback Action Routines

When VAXTPU receives a widget callback, it identifies and executes the layered application procedure or learn sequence that has been designated as the callback action routine. You can designate a procedure or learn sequence as a callback action routine either when the widget is created, using the built-in CREATE_WIDGET, or at some later time, using the built-in SET (WIDGET_CALLBACK). Note that when you specify an application-level callback program or learn sequence with CREATE_WIDGET or SET (WIDGET_CALLBACK), all widgets in the same X Resource Manager hierarchy have the same callback program or learn sequence. Therefore, the callback program or learn sequence must have a mechanism for handling all possible callback reasons.

4.2.4.6 Callable Interface-Level Callback Routines

If you are layering an application on VAXTPU or on EVE, you can specify callable interface-level callback routines only if you are specifying a widget's callback resources in a User Interface Language (UIL) file.

Callbacks can pass values known as **closures**, which are strings or integers whose function depends on the application you are writing. Note that DECwindows documentation refers to closures as **tags**. For more information about what closures are and how to use them, see Section 4.2.5.

You use the VAXTPU callable interface routine TPU$WIDGET_INTEGER_CALLBACK as the callback routine for all callbacks that have an integer closure and the VAXTPU routine TPU$WIDGET_STRING_CALLBACK for all callbacks that have a string closure.

Although the SET (WIDGET) built-in allows you to specify values for various resources of a widget, there are restrictions on specifying values for callback resources of widgets. When a widget is part of an XUI Resource Manager hierarchy, do not include callback resource names or values in the array you pass to SET (WIDGET). Instead, specify the callback routine in the UIL file. When a widget is not part of an X Resource Manager hierarchy, specify the names of the callback resources in the array you pass to SET (WIDGET), and specify 0 as the value of each such callback resource. VAXTPU automatically substitutes its common callback entry point for the 0 value. Note that a widget calls back only for those reasons specified in the widget's argument list. If a reason is omitted from the list, the corresponding event does not cause a callback.

4.2.5 Using Closures in DECwindows VAXTPU

DECwindows allows you to specify a closure value for a widget. Note that DECwindows documentation refers to closures as **tags**. DECwindows does not define what a closure value is; a closure is simply a value that DECwindows understands how to recognize and manipulate so that a DECwindows application programmer can use the value if needed in the application. For general information about using closures in DECwindows, see the *VMS DECwindows Guide to Application Programming*.

When a widget calls back to the DECwindows application, the callback parameters include the closure value assigned to the widget. DECwindows allows the application to define the significance and possible values of the closure.

VAXTPU supports closure values of type string and integer. Closure values are optional for widgets used by applications layered on VAXTPU. If you do not specify a closure value, the built-in GET_INFO (WIDGET, "callback_parameters", array) returns unspecified in the "*closure*" array element. If you create a widget without using a UIL file, the built-in GET_INFO (WIDGET, "callback_parameters", array) returns the closure you specified as a parameter to CREATE_WIDGET. If you create a widget using a UIL file, the built-in GET_INFO (WIDGET, "callback_parameters", array) returns the closure value (if any) defined in the XUI Resource Manager. If none is defined, the built-in returns unspecified.

VAXTPU leaves it to the layered application to use the closure in any way the application programmer wishes. VAXTPU passes through to the application any closure value received as part of a callback.

DECwindows EVE provides an example of how an application can use closure values. DECwindows EVE assigns a unique closure value to every widget instance that can be created during an EVE editing session. Each closure value corresponds to something that EVE must do in response to the activation of that particular widget. When an event causes VAXTPU to execute EVE's main callback program, the built-in GET_INFO (WIDGET, "callback_parameters", array) returns the widget activated, the reason code (the reason the widget is calling back), and the closure associated with the particular widget instance. EVE's main callback program contains an array that is indexed with values identical to the widget closure values. Each array element contains a pointer to the EVE code to be executed in response to the corresponding widget's callback. EVE's callback program uses the closure value to locate the appropriate array index so the correct EVE routine can be executed in response to the callback.

If your layered application does not use EVE's callback program, then its callback program or learn sequence must have a mechanism for determining which widget is calling back and which application code should be executed as a result.

4.2.6 Specifying Values for Widget Resources in DECwindows VAXTPU

This section discusses techniques for specifying values for widget resources.

4.2.6.1 VAXTPU Data Types for Specifying Resource Values

VAXTPU supports the following data types with which to specify values for widget resources:

- String
- Array of strings
- Integer

VAXTPU converts the value you specify into the data type appropriate for the widget resource you are setting. Table 4–1 shows the relationship between VAXTPU data types for widget resources and DECwindows data types for widget resources.

Table 4–1 Correspondence Between VAXTPU Data Types and DECwindows Argument Data Types

DECwindows Argument Data Type	VAXTPU Data Type
Array of strings	Array of strings
Boolean	Integer
Callback	Integer (0)
Compound string	String
Compound string table	Array of strings
Dimension	Integer
Integer	Integer
Position	Integer
Short	Integer
String	String
Unsigned character	Integer

VAXTPU does not support setting values for resources (such as pixmap, color map, font, icon, widget, and so on) whose data types are not listed in this table.

When you pass an array specifying values for a widget's resources using CREATE_WIDGET or SET (WIDGET), VAXTPU verifies that each array index is a string corresponding to a valid resource name for the specified widget. VAXTPU also verifies that the data type of the value you specify is valid for the specified resource.

4.2.6.2 Specifying a List as a Resource Value

List box and file selection widgets manipulate lists. For example, the file selection widget manipulates a list of files. The widget resource that stores such a list is specified to VAXTPU using an array.

To handle an array that passes a list to a widget, DECwindows must know how many elements the array contains. For example, if you, the application programmer, set the value of the "items" resource of a list box widget to point to a given array, DECwindows does not handle the array successfully unless the list box widget's "itemsCount" resource contains the number of elements in the array.

However, you do not necessarily know how many elements the array has at a given moment. To help you pass arrays, VAXTPU has a convention for referring to widget resources. If you follow the convention, VAXTPU will handle the resource that stores the number of array elements. The following paragraphs discuss the naming convention in more detail.

When you use the VAXTPU built-in procedure SET (WIDGET) to pass a list to a widget, specify both the list name and the list count resource in the same array index, separated by a line feed (ASCII (10)). The array element should be the array that is to be passed. For example, to specify the "items" resource to the list box widget, use code similar to the following:

```
line_feed := ASCII (10);
resource_array {"items" + line_feed + "itemsCount"}:=list_array;
```

The line-feed character, ASCII (10), is a delimiter separating two resource names.

VAXTPU automatically generates two resource entries. The first is the array of strings specifying the data to the list box for the "items" resource. The second is the count of elements in the array for the "itemsCount" resource.

To get resource values from a widget, use the following statement:

```
GET_INFO (widget, "WIDGET_INFO", array)
```

The indices of the array parameter are strings or string constants naming the resources whose values you want. (The initial values in the array are unimportant.) The GET_INFO statement directs VAXTPU to fetch the specified resource values of the specified widget and put the values in the array.

For list box widgets or file selection widgets, one element of the array receives another array containing the list manipulated by the widget. The indices of this array are of type integer. The lowest index has the value 0, and each subsequent index is incremented by 1. The contents of the array elements are of type string.

When you create the index of the element that receives the widget's list, you must observe the naming convention so that VAXTPU can handle both the list itself and the resource value specifying the length of the list. Give the index the following format:

```
items<line-feed>items_count
```

For example, if you used GET_INFO (widget, "WIDGET_INFO", array) to get resource values from a list box widget, you could specify the index for the element storing the widget's list as follows:

```
"items" + ASCII(10) + "itemsCount"
```

Note that the element for the widget's list does not actually contain an array until after execution of the GET_INFO statement. When VAXTPU encounters the GET_INFO statement, it parses the indices of the specified array. When VAXTPU parses the index of the element for the widget's list, it fetches both the list itself and the length of the list. Using the resource specifying the length, VAXTPU creates an array of the correct size to hold the widget's list.

See Section B.1 for sample uses of DECwindows VAXTPU built-ins.

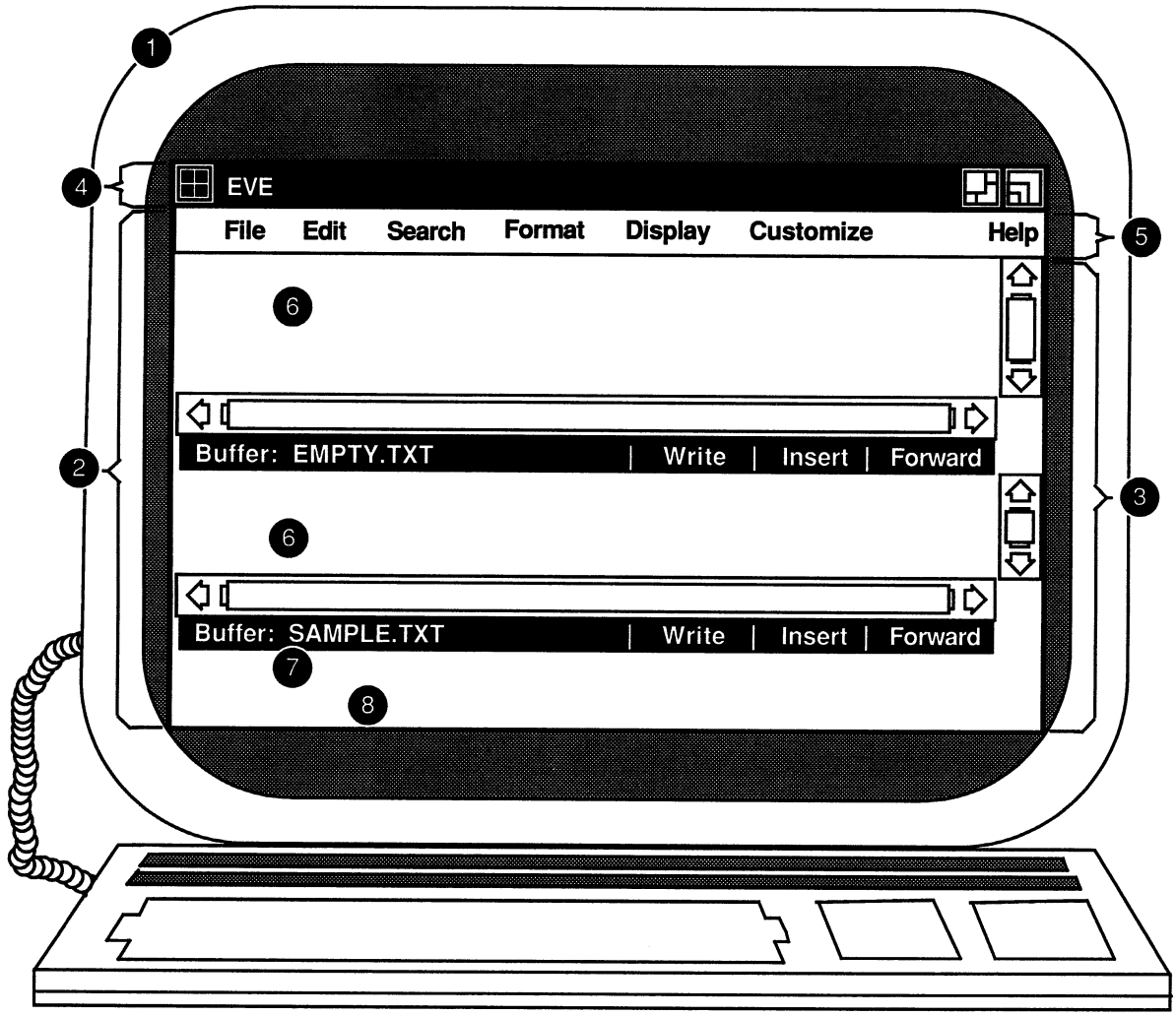
4.3 Writing Code Compatible with DECwindows EVE

This section provides information useful for programmers who extend DECwindows EVE or layer applications on DECwindows EVE.

4.3.1 Screen Objects in Applications Layered on DECwindows VAXTPU

Figure 4–1 and its accompanying text show the nomenclature for the screen objects used in EVE and, optionally, in other applications layered on VAXTPU.

Figure 4–1 Nomenclature of DECwindows VAXTPU Screen Objects



ZK–0239A–GE

Key to Figure 4–1

1 Display—In VAXTPU, the term **display** refers to the physical display device on which screen objects are visible.

2 Main window widget—This widget is created by VAXTPU, not by the layered application. Although the main window widget is not visible as a separate entity, it is the ancestor of all of EVE's visible widgets. The VAXTPU SCREEN keyword, when used as a parameter to a widget-related built-in, refers to the main window widget.

VAXTPU's main window widget is associated with a DECwindows window. Both DECwindows and VAXTPU have objects called "windows." VAXTPU windows have much the same function as DECwindows windows, but VAXTPU windows operate within a more limited scope.

A DECwindows window is a viewport enabling a DECwindows application to make visible some text and graphics. For example, a DECwindows window can be used as a viewport onto a widget. A DECwindows window is mapped to an area on a physical display device. For more information about DECwindows windows, see the *VMS DECwindows Guide to Application Programming*.

A VAXTPU window is a viewport onto a VAXTPU buffer. EVE windows always have the same width as the VAXTPU screen. For more information about the VAXTPU screen, see item 3 in this key. You can map a VAXTPU window only within an area of the physical display device occupied by a VAXTPU screen. For more information about mapping VAXTPU windows, see Chapter 6.

3 VAXTPU screen—This widget is created by VAXTPU, not by the layered application. When you use the SCREEN keyword as a parameter to a built-in unrelated to widgets, the keyword refers to the VAXTPU screen. In non-DECwindows VAXTPU, the phrase "VAXTPU screen" means all the area visible on the physical terminal screen.

4 Title bar—The title bar for EVE (or any other application layered on VAXTPU) is created by DECwindows, not by VAXTPU or the layered application.

5 Menu bar—The EVE menu bar widget is created by EVE, not by VAXTPU. You can optionally create a menu bar widget in any application layered on VAXTPU. If you do so, make the menu bar widget a child of the VAXTPU main window widget.

6 EVE user window—This window is created by EVE and is mapped to a buffer. It is a VAXTPU window, not a widget. Other applications layered on VAXTPU should create one or more user windows in which to display the results of the user's actions.

7 EVE command window—This window is created by EVE. It is a VAXTPU window, not a widget. Other applications layered on VAXTPU can optionally create a command window.

8 EVE message window—This window is created by EVE. It is a VAXTPU window, not a widget. Other applications layered on VAXTPU can optionally create a message window.

4.3.2 Select Ranges in DECwindows EVE

This section is intended for programmers extending EVE or layering an application on EVE.

EVE can use only one type of selection at a time. There are four possible types of selection: dynamic selection, static selection, found range selection, and DECwindows primary or secondary global selection. The ways in which these selections differ are explained in the following sections.

EVE has a routine called EVE$SELECTION that returns the current selection, regardless of whether the selection is dynamic, static, formed from a found range, or the primary global selection. It is possible to use the VAXTPU built-in SELECT_RANGE to obtain the current selection if the selection is a dynamic selection. However, Digital recommends that you use EVE$SELECTION to obtain the current selection, because this routine returns the current selection regardless of how it was created. To see how the EVE$SELECTION routine works and what parameters it takes, you can find the code for this routine in SYS$EXAMPLES:EVE$CORE.TPU.

4.3.2.1 Dynamic Selection

When you press the Select key or invoke the EVE command SELECT, EVE creates a dynamic selection. A dynamic selection expands and contracts as you move the text cursor. Moving the text cursor away from the text already selected does not cancel the selection. If you use the mouse to start a selection while a dynamic selection is active, the dynamic selection is canceled.

If EVE's current selection is a dynamic selection, the routine EVE$SELECTION returns the selected range and terminates the selection. If, for some reason, you want to use a statement that returns the current dynamic selection but does not terminate it, you can use a statement whose format is similar to the following:

```
r1 := EVE$SELECTION (TRUE, TRUE, TRUE, TRUE, FALSE)
```

The last parameter directs EVE$SELECTION not to terminate the selection. For more information on how to use these parameters, see the EVE$SELECTION routine in SYS$EXAMPLES:EVE$CORE.TPU.

4.3.2.2 Static Selection

EVE creates a static selection if you do any of the following:

- Click the MB1 mouse button two or more times to select a word, line, paragraph, or buffer
- Use the EVE command SELECT ALL
- Press the MB1 mouse button, drag the mouse across text, and then release the mouse button
- Use the MB1 mouse button with the SHIFT key to extend a selection

EVE implements a static selection by creating a range upon which you can perform EVE commands such as STORE TEXT or REMOVE. However, EVE does not start this range using the VAXTPU built-in SELECT. Thus, if you use the SELECT_RANGE built-in while a static selection is active, VAXTPU returns the message "No select active."

If you move the text cursor off the text in the static selection, the selection is canceled.

4.3.2.3 Found Range Selection

When EVE positions to the beginning of a range as the result of the FIND command, WILDCARD FIND command, or pressing the FIND key, EVE creates a found range containing the text EVE found as a match for your search string. If no other selection is active, EVE treats the found range as the current selection.

EVE implements a found range selection by creating a range upon which you can perform EVE commands such as STORE TEXT or REMOVE. However, EVE does not start this range using the VAXTPU built-in SELECT. Thus, if you use the SELECT_RANGE built-in while a found range selection is active, VAXTPU returns the message "No select active."

If you move the text cursor off the text in the found range selection, the selection is canceled.

4.3.2.4 Relation of EVE Selection to DECwindows Global Selection

If EVE has a dynamic selection or a static selection active, that selection is automatically designated as the primary global selection. A found range selection is not designated as the primary global selection.

You can use the routine EVE$SELECTION to obtain the text of the primary global selection when an application other than VAXTPU owns the selection. To do so, the call to EVE$SELECTION must be in code bound to a mouse button other than MB1. The value returned is a string containing the text of the primary global selection.

4.4 Compiling VAXTPU Programs

Before compiling programs in VAXTPU, you should enable the display of informational messages to help you locate errors. EVE automatically enables the display of informational messages for you when you use the EVE command EXTEND EVE. For more information on displaying messages, see the description of the SET (INFORMATIONAL) built-in in the VAXTPU Reference Section.

The VAXTPU compiler numbers the lines of code it compiles. The line numbers begin with 1. For a string, all VAXTPU statements are considered to be on line 1. For a range, line 1 is the first line of the range, regardless of where in the buffer the range begins. Buffers are numbered starting at the first line. When a compilation error occurs, VAXTPU tells you the approximate line number where the error occurred. To move to the line at which the error occurred, use the POSITION (integer) built-in procedure.

In EVE, you can use the LINE command. For example, the command LINE 42 moves the editing point and the cursor to line 42.

To see VAXTPU messages while in EVE, use the EVE command BUFFER MESSAGES. To return to the original buffer or another buffer of your choice, use the EVE command BUFFER *name_of_buffer*.

There are two ways to compile a program in VAXTPU: on the command line of EVE or in a VAXTPU buffer.

4.4.1 Compiling on the EVE Command Line

You can compile a simple VAXTPU program merely by entering it on the EVE command line. For example, if you use the EVE command TPU and then enter the statement SHOW (SUMMARY), VAXTPU compiles and executes the program associated with the SHOW (SUMMARY) statement.

4.4.2 Compiling in a VAXTPU Buffer

VAXTPU programs are usually compiled by entering VAXTPU procedures and statements in a buffer and then compiling the buffer. If you are using EVE, you can enter the statement SHOW (VARIABLES) in a buffer and compile the buffer by using EVE's command TPU and entering the following statement after the prompt:

```
VAXTPU Statement:  COMPILE (CURRENT_BUFFER) ;
```

The program associated with SHOW (VARIABLES) is not executed until you enter the following statement:

```
VAXTPU Statement:  EXECUTE (CURRENT_BUFFER) ;
```

Note that if you use a buffer, a range, or a string as the parameter for the built-in procedure EXECUTE, VAXTPU first compiles and then executes the buffer, range, or string. See the description of EXECUTE in the VAXTPU Reference Section.

The built-in procedure COMPILE optionally returns a program data type. If you want to use the program that you are compiling later in your session, you can assign the program that is returned to a variable. The following example shows how to make this assignment:

```
new_program := COMPILE (CURRENT_BUFFER);
```

If no error messages are issued while you compile the current buffer, you can then execute the program *new_program* with the following statement:

```
EXECUTE (new_program);
```

You can use the built-in procedure COMPILE to compile certain parts of a buffer rather than a whole buffer. To do so, create a range that includes the statements within the buffer that you want compiled, and then specify the range as the parameter for COMPILE.

4.5 Executing VAXTPU Programs

You can use programs that are already compiled as parameters for the built-in procedure EXECUTE. In addition, you can use buffers, ranges, or strings that contain executable VAXTPU statements as parameters for the built-in procedure EXECUTE. VAXTPU compiles the contents of the buffer, range, or string if necessary; then VAXTPU executes the compiled buffer, range, or string.

Suppose you created a program called *new_program* by using the following statement after using the EVE command TPU:

```
VAXTPU Statement:  new_program := COMPILE (CURRENT_BUFFER);
```

You could then execute *new_program* by using the following statement after using the EVE command TPU:

```
VAXTPU Statement:  EXECUTE (new_program);
```

Note, however, that you could also compile and execute the statements in the current buffer by using the following VAXTPU statement after using the EVE command TPU:

```
VAXTPU Statement:  EXECUTE (CURRENT_BUFFER);
```

Small VAXTPU programs can be entered, compiled, and executed on the command line of EVE. The following example shows a small program that you can enter after the prompt *VAXTPU Statement:*

```
VAXTPU Statement:  SET (TIMER, ON, "Executing");
```

The preceding command executes the program associated with the VAXTPU built-in procedure SET (TIMER) and causes the string "Executing" to be displayed at 1-second intervals when a long procedure is executing. The string is displayed in the last 15 spaces of the prompt area at 1-second intervals.

4.5.1 Interrupting Execution with CTRL/C

Pressing CTRL/C causes VAXTPU to stop the execution of a user-written program. You can also stop the execution of the following VAXTPU built-in procedures with CTRL/C:

- LEARN_BEGIN . . . LEARN_END (Execution of a learn sequence)
- READ_FILE
- SEARCH
- WRITE_FILE

Caution: Because VAXTPU does not journal CTRL/C, using CTRL/C may affect the accuracy of your keystroke journal file. In addition, CTRL/C prevents completion of some built-in procedures, such as ERASE_RANGE, MOVE_TEXT, and FILL. VAXTPU behavior after such an interruption is unpredictable. Digital recommends that you exit from the editor after pressing CTRL/C to ensure that you do not lose any work because of an inaccurate keystroke journal file.

Note, however, that buffer change journaling works properly with CTRL/C. Therefore, if you are *not* using keystroke journaling, exiting from the editor is not necessary.

For more information on the effects of pressing CTRL/C, see Section 3.8.4.7 and Section 3.8.4.7.2.

4.5.2 Procedure Execution

If you include procedure declarations as part of a program, the procedure is compiled and the procedure name is added to the VAXTPU list of procedures when you execute the program. Invoke the procedure in one of the following ways:

- Enter the name of the compiled procedure after the *VAXTPU Statement:* prompt from EVE.
- Call the procedure from within a program or another procedure.

4.6 VAXTPU Startup Files

This section discusses VAXTPU startup files. Startup files are files that VAXTPU reads, compiles, and executes during its initialization sequence.

There are three types of VAXTPU startup files:

- Section files
- Command files
- Initialization files

Section Files

A section file is the compiled, binary form of a file containing VAXTPU source code. To direct VAXTPU to execute a section file, either use the /SECTION qualifier to the EDIT/TPU command or allow VAXTPU to execute the default section file. For more information on the /SECTION qualifier, see Chapter 5.

The default section file is TPU$SECTION. When VAXTPU tries to locate the section file, VAXTPU supplies a default directory of SYS$SHARE and a default file type of TPU$SECTION. VMS defines the systemwide logical name TPU$SECTION as EVE$SECTION, so the default section file is the file implementing the EVE editor. To override the VMS default, redefine TPU$SECTION.

Command Files

A command file contains a series of VAXTPU procedures, followed by a sequence of VAXTPU statements. To direct VAXTPU to compile and execute a command file, either use the /COMMAND qualifier to the EDIT/TPU command or allow VAXTPU to compile and execute the default command file. For more information on the /COMMAND qualifier, see Chapter 5.

The default command file is TPU$COMMAND. When VAXTPU tries to locate the command file, it supplies a default file type of TPU. To direct VAXTPU to compile and execute a particular command file, define the logical name TPU$COMMAND to be the file you want VAXTPU to use.

Initialization Files

An initialization file contains commands to be executed by an application layered on VAXTPU. To specify an initialization file to be executed, use the /INITIALIZATION qualifier to the EDIT/TPU command. For more information on the /INITIALIZATION qualifier, see Chapter 5.

VAXTPU does not determine the default handling of an initialization file. Likewise, VAXTPU does not directly load or execute the commands in an initialization file. The application layered on VAXTPU must determine the defaults and must handle the loading and execution of an initialization file. For example, EVE reads an initialization file (if one is present) and interprets the initialization commands when it processes the procedure TPU$INIT_POSTPROCEDURE. Any key definitions in an initialization file override corresponding key definitions saved in a section file and key definitions in a command file.

Typically, you use EVE initialization files to set values that are not usually saved in a section file, such as margins, tab stops, and bound or free cursor. For a list of the EVE default values that you might want to modify by using an EVE initialization file, see the *VMS EVE Reference Manual*.

4.6.1 Sequence in Which VAXTPU Processes Startup Files

When you invoke VAXTPU, by default VAXTPU reads, compiles, and executes several files. The sequence in which VAXTPU performs these tasks is as follows:

1. VAXTPU loads into memory the specified or default section file unless the user specified /NOSECTION on the DCL command line.
2. VAXTPU reads the specified or default command file into a buffer named $LOCAL$INI$ unless the user specified /NOCOMMAND on the DCL command line.
3. If the user specified /DEBUG on the DCL command line, VAXTPU reads the specified or default debugger file into a buffer named $DEBUG$INI$. A debugger file contains VAXTPU procedures and statements to help debug VAXTPU code. For more information on the default VAXTPU debugger, see Section 4.7.
4. If the buffer named $DEBUG$INI$ containing debugger code is present, VAXTPU compiles the buffer and executes the resulting program.
5. VAXTPU calls and executes the procedure named TPU$INIT_PROCEDURE if the procedure is present in the section file or was defined in the debug file.
6. If the command file was read into the buffer named $LOCAL$INI$, VAXTPU compiles that buffer and executes the resulting program.
7. VAXTPU calls and executes the procedure named TPU$INIT_POSTPROCEDURE if the layered application has defined this procedure in the section file, debug file, or command file.

If a layered application makes use of an initialization file, it is the responsibility of the application to define when the initialization file is processed. EVE processes initialization files during the TPU$INIT_POSTPROCEDURE phase.

4.6.2 Section Files

A section file is the binary form of a program implementing a VAXTPU-based editor or application. It is a collection of compiled VAXTPU procedure definitions, variable definitions, and key bindings. The advantage of using a binary file is that the source code does not have to be compiled each time you invoke the editor or application, so startup performance is improved.

4.6.2.1 Creating and Processing a New Section File

To create a section file, begin by writing a program in the VAXTPU language. The program must adhere to all the programming conventions discussed throughout this manual. For examples of programs used to create a section file, see the files in the directory SYS$EXAMPLES. This directory contains the sources used to create the EVE section file. To see a list of the EVE source files, type the following at the DCL prompt:

```
$ DIR SYS$EXAMPLES:EVE$*.TPU
```

If you cannot find these files on your system, see your system manager.

When writing the VAXTPU program implementing your application, place your initializing statements in a procedure named TPU$INIT_PROCEDURE. Such statements might create buffers, create windows, associate windows with buffers, set up screen attributes, initialize variables, define how the journal facility works, and so on. You can put the procedure TPU$INIT_PROCEDURE anywhere in the procedure declaration portion of your program. VAXTPU executes TPU$INIT_PROCEDURE before executing the command file (if there is one). For more information on VAXTPU's initialization sequence, see Section 4.6.1.

Place any statements implementing or handling initialization files in a procedure named TPU$INIT_POSTPROCEDURE. VAXTPU executes this procedure after both the TPU$INIT_PROCEDURE and the command file have been executed. This allows commands or definitions in the initialization file to modify commands or definitions in the command file. EVE defines both TPU$INIT_PROCEDURE and TPU$INIT_POSTPROCEDURE procedures. For more information on how EVE implements initialization files, see Section 4.6.4.

After you put the desired VAXTPU procedures and statements into the program implementing your application, end your program with the following statements:

- A statement containing the built-in procedure SAVE. SAVE is the mechanism by which you store all currently defined procedures, variables, and bound keys in binary form. For more information on SAVE, see the description of this built-in in the VAXTPU Reference Section.

- The built-in procedure QUIT. QUIT ends the VAXTPU session. For more information on QUIT, see the description of this built-in in the VAXTPU Reference Section.

For examples of files using these statements, see Example 4–4 and Example 4–5.

To compile your program into a section file, invoke VAXTPU but do not supply as a parameter the name of a file to be edited. Use the /NOSECTION qualifier to indicate that no existing section file should be loaded. Use the /COMMAND qualifier to specify the file containing your program. For example, to create a section file from a program in a file called MY_APPLICATION.TPU, you would enter the following at the DCL prompt:

```
$ EDIT/TPU/NOSECTION/COMMAND=my_application.TPU
```

This command causes VAXTPU to write the binary form of the file MY_APPLICATION.TPU to the file you specified as the parameter to the SAVE statement in your program. To use the section file, invoke VAXTPU specifying your section file.

For more information on invoking VAXTPU and using the qualifiers to the EDIT/TPU command, see Chapter 5.

4.6.2.2 Extending an Existing Section File

To extend an existing section file, begin by writing a program in the VAXTPU language.

If you are extending the EVE section file, put your initializing statements in an initialization procedure called TPU$LOCAL_INIT. TPU$LOCAL_INIT is an empty procedure in the EVE section file. When you add your VAXTPU statements and procedures to the EVE section file, your procedure named TPU$LOCAL_INIT supersedes EVE's original empty value of TPU$LOCAL_INIT. TPU$LOCAL_INIT is called at the end of the procedure TPU$INIT_PROCEDURE during the initialization sequence. For more information on the initialization sequence, see Section 4.6.1.

If you are extending a non-EVE section file, you must determine whether that section file has implemented the convention of including a TPU$LOCAL_INIT procedure.

After adding VAXTPU procedures and statements implementing your application, end your program with the following statements:

- A statement containing the built-in procedure SAVE. SAVE is the mechanism by which you store all currently defined procedures, variables, and bound keys in binary form. For more information on SAVE, see the description of this built-in in the VAXTPU Reference Section.
- The built-in procedure QUIT. QUIT ends the VAXTPU session. For more information on QUIT, see the description of this built-in in the VAXTPU Reference Section.

For examples of files using these statements, see Example 4–4 and Example 4–5.

Example 4–4 shows the syntax of a program that could be used to create a section file:

Example 4–4 Sample Program for a Section File

```
PROCEDURE tpu$local_init
     .
     .
     .
ENDPROCEDURE;
PROCEDURE vt100_keys
    .
    .
    .
ENDPROCEDURE;
vt100_keys;  !Call the procedure that defines the keys
SAVE ("sys$login:vt100ini");
QUIT;
```

To add your program to an existing section file, invoke VAXTPU but do not supply as a parameter the name of a file to be edited. Use the /SECTION qualifier to specify the section file to which you want to add your program. Use the /COMMAND qualifier to specify the file containing your program. For example, to add a program called MY_CUSTOMIZATIONS.TPU to the EVE section file, you would enter the following at the DCL prompt:

```
$ EDIT/TPU/SECTION=EVE$SECTION/COMMAND=my_customizations.TPU
```

This command causes VAXTPU to load the EVE section file and then read, compile, and execute the command file you specify. A new section file is created. The new file includes both the EVE section file and the binary form of your program. The section file is written to the file you specified as the parameter to the SAVE statement in your program. To use the section file, invoke VAXTPU specifying your section file.

For more information on invoking VAXTPU and using the qualifiers to the EDIT/TPU command, see Chapter 5.

For more information on extending the EVE section file, see the *Guide to VMS Text Processing*.

4.6.2.3 A Sample Section File

If you choose to design an application layered on VAXTPU and not layered on EVE, you must provide certain basic structures and key definitions to be able to use the VAXTPU compiler and interpreter. Example 4–5 is a sample of the source code that creates a minimal interface. It provides the following basic structures:

- A buffer and a window for VAXTPU messages
- A buffer and a window for information from the built-in procedure SHOW
- A buffer and a window in which to enter VAXTPU programs or text

- A prompt area in which to enter VAXTPU commands

Because VAXTPU does not have any keys defined when invoked without a section file, the sample program also contains the following key definitions:

- The RETURN key
- The DELETE key
- Key for exiting from VAXTPU
- Key for entering VAXTPU statements. Example 4–5 uses the Tab key.

By default, VAXTPU looks for TPU$INIT_PROCEDURE, so the statements that create the structures for a minimal interface are contained in TPU$INIT_PROCEDURE. Individual statements that define keys come after any procedures in the file.

If you entered the text from Example 4–5 into a file named MINI.TPU and you wanted to compile that file into a section file, you would enter the following command at the DCL level:

```
$ EDIT/TPU/NOSECTION/COMMAND=mini.TPU
```

When you enter this command, the qualifier /NOSECTION specifies that no section file is to be read. (This ensures that none of the procedures or variables from an existing section file are loaded into the internal VAXTPU tables.) The qualifier /COMMAND specifies that the command file MINI.TPU is to be compiled by VAXTPU. The built-in procedure SAVE at the end of the command file specifies that all of the procedures, variables, and key definitions in the file are to be saved in binary form in the file SYS$LOGIN:MINI.TPU$SECTION. The built-in procedure QUIT then causes you to leave VAXTPU.

Example 4–5 Source Code for Minimal Interface

```
! mini.TPU - minimal VAXTPU interface

PROCEDURE tpu$init_procedure

! Create a buffer and window for messages

    message_buffer := CREATE_BUFFER ("Message Buffer");
    SET (NO_WRITE, message_buffer);
    SET (SYSTEM, message_buffer);
    SET (EOB_TEXT, message_buffer, "");
    message_window := CREATE_WINDOW (21, 4, OFF);
    MAP (message_window, message_buffer);

! Create a buffer and window for SHOW

    show_buffer := CREATE_BUFFER("Show Buffer");
    SET (NO_WRITE, show_buffer);
    SET (SYSTEM, show_buffer);
    info_window := CREATE_WINDOW (1, 20, ON);

! Create a buffer and window for editing

    main_buffer := CREATE_BUFFER ("Main Buffer");
    main_window := CREATE_WINDOW (1, 20, ON);
    MAP (main_window, main_buffer);
```

(continued on next page)

Example 4–5 (Cont.) Source Code for Minimal Interface

```
! Create an area on the screen for prompts

    SET (PROMPT_AREA, 21, 1, NONE);

!Put the editing point in the main buffer

    POSITION (main_buffer);
    tpu$local_init;

ENDPROCEDURE;

PROCEDURE tpu$local_init   !Procedure to allow end users
                           !to add private extensions
ENDPROCEDURE;

! Define the minimal editing keys:

  DEFINE_KEY ("SPLIT_LINE", RET_KEY);
  DEFINE_KEY ("ERASE_CHARACTER(-1)", DEL_KEY);
  DEFINE_KEY ("EXECUTE(READ_LINE('VAXTPU Statement: '))", TAB_KEY);
  DEFINE_KEY ("EXIT", CTRL_Z_KEY);

! Create a section file and then quit

SAVE ("sys$login:mini");
QUIT;

! End of mini.TPU
```

If you created the section file SYS$LOGIN:MINI.TPU$SECTION, you could use the procedures and definitions in that file as an interface to VAXTPU. To invoke VAXTPU with the MINI section file, you would type the following command at the DCL prompt. This command specifies the file YOUR_TEXT.FIL as the file to be edited:

```
$ EDIT/TPU/SECTION=sys$login:mini your_text.fil
```

Rather than enter this long command each time you invoke VAXTPU, define the logical name TPU$SECTION to point to your section file. By default, VAXTPU looks for a file that TPU$SECTION points to, and reads that file as the default section file.

Whenever you want to add new procedures, variables, learn sequences, or key definitions to a section file, edit the command file to include the new items, and then recompile the command file to produce a section file with the new items. For example, if you want to add key definitions for the arrow keys, you could edit the file MINI.TPU and add the following statements after any procedures in the file:

```
DEFINE_KEY  ("MOVE_VERTICAL (-1)",  UP);
DEFINE_KEY  ("MOVE_VERTICAL (1)",   DOWN);
DEFINE_KEY  ("MOVE_HORIZONTAL (1)",  RIGHT);
DEFINE_KEY  ("MOVE_HORIZONTAL (-1)", LEFT);
```

Then you would recompile the command file with the following command:

```
$ EDIT/TPU/NOSECTION/COMMAND=mini.TPU
```

After completing these steps, when you invoke VAXTPU with the section file MINI.TPU$SECTION the new key definitions would be included.

An alternate way of adding these key definitions to your section file is to enter the definitions as text in the current buffer. You could then press the Tab key (the command prompt key for the minimal interface) and enter the following command after the prompt:

```
VAXTPU Statement:  EXECUTE (CURRENT_BUFFER);
```

This causes the new key definitions to be added to your current editing context. To add the definitions to the section file so you can use them in future sessions, enter the following statement after the command prompt:

```
Command:  SAVE ("sys$login:mini");
```

If you want to save the VAXTPU source code for the key definitions, write out the current buffer or use the built-in procedure EXIT to leave the VAXTPU session so that the contents of the buffer are written to a file.

4.6.2.4 Recommended Conventions for Section Files

A section file implementing a layered application should include the following procedures:

- TPU$INIT_PROCEDURE
- TPU$LOCAL_INIT

If your application is to support initialization files, the section file implementing the application should also include a procedure called TPU$INIT_POSTPROCEDURE. This procedure should contain the VAXTPU statements implementing or handling the initialization files.

For information on EVE's implementation of initialization files, see Section 4.6.4.

A section file implementing a layered application should assign values to the following special variables in the procedure TPU$INIT_PROCEDURE:

- TPU$X_MESSAGE_BUFFER or MESSAGE_BUFFER
- TPU$X_SHOW_BUFFER or SHOW_BUFFER
- TPU$X_SHOW_WINDOW or INFO_WINDOW

If you write a section file extending the EVE section file, EVE provides the procedures and variables above. If you choose to write your own application, your application must contain these structures and procedures.

These procedures and variables are discussed in more detail in the following subsections.

4.6.2.4.1 TPU$INIT_PROCEDURE

This procedure should perform the following operations:

- Initialize all global variables to their startup values.
- Create all required work spaces for the editor (see the list of special purpose buffers and windows in Table 4–2).

You can add other functions to TPU$INIT_PROCEDURE, but it should perform at least these two operations.

4.6.2.4.2 TPU$LOCAL_INIT

If your application allows the end user to customize the application using a command file, you may want to make available to the user a procedure called TPU$LOCAL_INIT. (Although this name is not required, it is commonly used by VAXTPU programmers.)

In EVE, the code implementing the initialization sequence calls TPU$LOCAL_INIT as the last step of the sequence. EVE defines this procedure but leaves it empty. The user can use this procedure in a command file to contain VAXTPU statements implementing private initializations.

The code implementing TPU$LOCAL_INIT in EVE can be found in SYS$EXAMPLES:EVE$CORE.TPU.

4.6.2.4.3 Special Variables

VAXTPU creates six variables (three pairs of synonyms) to be used by layered applications. Although VAXTPU automatically declares the variables, the application must assign a value to one of the synonyms in each pair.

Table 4–2 shows the names and uses of these variables.

Table 4–2 Special VAXTPU Variables Requiring a Value from a Layered Application

Recommended Name	Synonym Provided for Backward Compatibility	Data Type Structure	How VAXTPU Uses the Variable
TPU$X_MESSAGE_BUFFER	MESSAGE_BUFFER	Buffer	VAXTPU writes messages in this buffer. If the MESSAGE_BUFFER is associated with a window that is mapped to the screen, VAXTPU updates the window. If the application does not assign a buffer to this variable, VAXTPU writes messages to the screen.
TPU$X_SHOW_BUFFER	SHOW_BUFFER	Buffer	VAXTPU writes information stored by the SHOW built-in in this buffer.
TPU$X_SHOW_WINDOW	INFO_WINDOW	Window	VAXTPU displays information stored by the SHOW built-in and information from the HELP_TEXT built-in in this window.

If you want to use the built-in procedure SHOW in your application, you must create these special variables that VAXTPU uses for SHOW.

4.6.3 Command Files

This section provides an overview of how to use command files. For more detailed information on the relationship between EVE command files and section files, see the *Guide to VMS Text Processing*.

A command file is a VAXTPU source file that can contain procedures, key definitions, and other VAXTPU executable statements. You can have any number of command files in your directory. You might want to write one command file that customizes your editor for programming in PASCAL, another command file that customizes your editor for text editing, and so on. If you have several command files, give them names that remind you of their contents. If you have one command file that you use most of the time, name it TPU$COMMAND.TPU.

The syntax to invoke VAXTPU with a command file at the DCL command level is as follows:

$ EDIT/TPU/COMMAND [[= filespec]]

If you name your command file TPU$COMMAND.TPU and it is in your default directory, VAXTPU reads the file by default, without your having to use /COMMAND. If you name your file something other than TPU$COMMAND.TPU, or if you put it in a directory other than your default directory, you must use the qualifier /COMMAND explicitly and provide a full file specification after the qualifier.

VAXTPU reads a command file, compiles it, and executes any commands that do not contain syntax errors. If there are errors, VAXTPU writes an error message to the message area. The command file can customize or extend the application implemented by the section file with which you invoked VAXTPU.

Example 4–6 is a sample VAXTPU command file defining a procedure that moves the editing point to the beginning of a segment of text delimited by the characters *%(/** at the beginning and ***/)%* at the end.

Example 4–6 Command File for Go to Text Marker

```
PROCEDURE goto_text_marker

   LOCAL text_marker_pattern,
         text_marker_range;

   text_marker_pattern := '%(/*' + MATCH ('*/)%');
   text_marker_range :=  SEARCH_QUIETLY (text_marker_pattern,
                              GET_INFO (CURRENT_BUFFER, "direction"));
   IF text_marker_range <> 0
   THEN
       POSITION (text_marker_range);
   ELSE
       MESSAGE ("Text_marker not found");
   ENDIF;

   RETURN text_marker_range;

ENDPROCEDURE;
```

If you name the file that contains this procedure TEXT_MARKERS.TPU, you can invoke VAXTPU with EVE and your command file in the following way:

```
$  EDIT/TPU/COMMAND=device:[user]text_markers.tpu
```

If you add procedures or statements to the command file TEXT_MARKERS.TPU, place all procedures before any individual statements that are not listed within a procedure (for example, key definitions to move to the next text marker).

Remember to name your variables and procedures so they do not conflict with VAXTPU reserved words and predefined identifiers. Digital recommends that you prefix your variable and procedure names with three letters (your initials, for example) followed by an underscore (_).

4.6.4 EVE Initialization Files

Any application layered on VAXTPU can support initialization files. This section describes EVE's implementation of initialization files. For more information on EVE initialization files, see the *Guide to VMS Text Processing*.

EVE initialization files enable you to do the following:

- Use EVE commands in a startup file to customize editing sessions
- Set formats for individual buffers

EVE initialization files contain EVE commands that are executed either when you invoke the editor or when you issue the EVE @ (at sign) command.

To create an EVE initialization file, put in the file the EVE commands you want to use to customize the editor. Use one command on each line and one line for each command. Do not separate the commands with semicolons. If a command in an EVE initialization file is incomplete, EVE prompts you for more information, the same as if you were typing the command during an editing session. Comments in EVE initialization files must be on lines separate from commands and must begin with an exclamation point (!). You cannot nest EVE initialization files. Do not use the DO command in an EVE initialization file.

The following sample initialization file sets left and right margins, establishes overstrike mode, binds the QUIT command to the GOLD/Q key sequence, and enables an EDT-like keypad:

```
SET LEFT MARGIN 5
SET RIGHT MARGIN 60
OVERSTRIKE MODE
DEFINE KEY=gold/q QUIT
SET KEYPAD EDT
```

4.6.4.1 Using an EVE Initialization File at Startup

You can cause an initialization file to be executed in any of the following ways when you invoke EVE:

- Name the file EVE$INIT.EVE. This is the default file name for EVE initialization files.
- Specify the name of the initialization file as a qualifier to EDIT/TPU.
- Define a logical name, EVE$INIT, to point to your initialization file.

The first method and third method are appropriate if you intend to use one initialization file most of the time to customize your editing sessions. If you name the file EVE$INIT.EVE and do not specify another EVE initialization file on the command line, EVE automatically executes EVE$INIT.EVE when you issue the EDIT/TPU command.

Use the second method to control which initialization file EVE executes to customize the editing session. For example, if you have an EVE$INIT file but want to use another initialization file, specify the other file using the /INITIALIZATION qualifier to EDIT/TPU. To specify an initialization file called MY_INIT.EVE, enter the following command string at the DCL prompt:

```
$ EDIT/TPU/INITIALIZATION=my_init.eve
```

EVE always executes the initialization file specified on the command line, if such a file is present. If no file is specified on the command line, EVE searches for EVE$INIT.EVE first in the current directory and then in SYS$LOGIN. If it finds EVE$INIT.EVE, the editor executes that file. If the file is not found, the editor checks whether the logical name EVE$INIT has been defined.

If you plan to create several initialization files and to use them equally, you may not want to name one of the files EVE$INIT. For example, if you want one initialization file to set narrow margins and another to set wide margins, create both files and specify the file you want when you invoke EVE.

4.6.4.2 Using an EVE Initialization File During an Editing Session

To execute an EVE initialization file during an editing session, use the @ (at sign) command and specify the file. For example, the following command executes an initialization file called MYEVE.EVE in your current (default) directory.

```
Command:  @myeve
```

Commands for buffer settings apply to the current buffer. This is effectively the same as typing the commands that the file contains. You may want to create initialization files to execute two or more related commands, such as resetting both margins.

4.6.4.3 How an EVE Initialization File Affects Buffer Settings

Commands in an EVE initialization file that set buffer characteristics (such as margins and tab stops) affect a system buffer named $DEFAULTS$. Buffers created during the editing session have the same settings as $DEFAULTS$. For example, if your initialization file contains the command SET RIGHT MARGIN 65, the value 65 is used as the right margin setting for the main buffer and for any buffers you create during the session with GET FILE or BUFFER commands.

To see the settings for the $DEFAULTS$ buffer, use the EVE command SHOW DEFAULTS BUFFER. For example, if you wanted to know what the tab settings were for the $DEFAULTS$ buffer, you would type the following command:

```
Command:  SHOW DEFAULTS BUFFER
```

This command causes EVE to show buffer information in a format similar to the format in Example 4–7 (using values that apply to your editing session):

Example 4–7 SHOW DEFAULTS BUFFER Display

```
Information about buffer $DEFAULTS$

Not modified                        Left margin set to: 1
Mode: Insert                        Right margin set to:  79
Direction: Forward
Max lines: No limit

Tab Stops set every 8 columns

Non-default right margin action
```

To change the characteristics of the $DEFAULTS$ buffer during an editing session, use the command BUFFER $DEFAULTS$ to put the defaults buffer in a window. This buffer is empty and you cannot add text to it. However, when you change the settings of the $DEFAULTS$ buffer, the changes are saved and used to set the characteristics of any user buffers you create. Use commands such as SET RIGHT MARGIN, SET LEFT MARGIN, SET TABS, FORWARD, REVERSE, INSERT, or OVERSTRIKE to change the characteristics of the $DEFAULTS$ buffer. The new characteristics are applied to new buffers but not to existing ones. To leave the $DEFAULTS$ buffer and put a different buffer in the window, use the BUFFER command.

4.7 Debugging VAXTPU Programs

To debug VAXTPU programs, you can either write your own debugger in the VAXTPU language or you can use the VAXTPU debugger provided in TPU$DEBUG.TPU. Regardless of what debugger you use, you may also find it helpful to enable the display of error line numbers using SET (LINE_NUMBER, ON) and to enable the display of procedures called when an error occurs using SET (TRACEBACK, ON).

If you write your own debugger, you can invoke it by using the /DEBUG qualifier to the EDIT/TPU command. For example, if you wanted to use your own debugger, called MY_DEBUGGER.TPU, on a file called MIGHT_BE_BUGGY.TPU, you would type the following at the DCL prompt:

```
$ EDIT/TPU/DEBUG=my_debugger.tpu might_be_buggy.tpu
```

4.7.1 Invoking the VAXTPU Debugger

You invoke the VAXTPU debugger to debug one of the following kinds of files:

- Section files
- Command files
- Files containing VAXTPU programs that are not startup programs

The following subsections contain more information on debugging each kind of file.

4.7.1.1 Section Files

To invoke the debugger for a section file, type the following at the DCL prompt:

```
$ EDIT/TPU/DEBUG
```

The /DEBUG qualifier causes the VAXTPU initialization routine to execute the debugger file before the procedure TPU$INIT_PROCEDURE is run.

The debugger initially creates a window filling most of the screen. The window consists of the following three areas:

- **Source area** — Displays your code when it has been placed in the debugger source buffer.
- **Output area** — Displays one-line messages or one-line results of an EXAMINE command.
- **Debug command line** — Displays the *Debug:* prompt.

When VAXTPU displays the debug window, you can set breakpoints in the section file using the SET BREAKPOINT command. For example, if you wanted to debug a procedure called USER_FUM, you would type the following on the debugger command line:

```
Debug:  SET BREAKPOINT user_fum
```

After setting breakpoints, use the GO command to switch control of execution from the debugger to VAXTPU. After you have used this command, the screen displays the code you specified.

4.7.1.2 Command Files

To invoke the debugger for use on a command file, invoke VAXTPU using the /DEBUG, /COMMAND, and /NOSECTION qualifiers. For example, if you wanted to debug a command file called MY_COMMANDS.TPU, you would type the following at the DCL prompt:

```
$ EDIT/TPU/NOSECTION/COMMAND=my_commands.tpu/DEBUG
```

VAXTPU compiles and executes the debugger and places the debug window on the screen before compiling the command file. As a result, you must set breakpoints in the command file before it has been compiled. When you set breakpoints, VAXTPU notifies you that you have specified breakpoints at nonexistent procedures.

To continue with the debugging session, use the GO command. GO causes VAXTPU to compile the contents of the command file. Recompiling a procedure does not remove any breakpoints set in that procedure.

You cannot use the VAXTPU debugger on a file that does not contain VAXTPU procedures. If your command file does not contain any procedures, you must find a different method of debugging it.

4.7.1.3 Other VAXTPU Source Code

To debug a VAXTPU program that is not a section file or a command file, use the /DEBUG qualifier when you invoke VAXTPU. For example, if you want to debug procedures in a file called USER_APPLICATION.TPU, you invoke the debugger as follows:

```
$ EDIT/TPU/DEBUG user_application.tpu
```

The debugger creates a window filling the screen as described in Section 4.7.1.1.

4.7.2 Getting Started with the VAXTPU Debugger

This section describes using the default VAXTPU debugger with EVE.

If you know which parts of the code you want to debug, use the SET BREAKPOINT command to set breakpoints. If you need to look at the code before setting breakpoints, use the GO command as soon as the debugger window appears. This places on the screen the code in the file you specified on the DCL command line. At this point, EVE commands are available so you can manipulate the text. To return to the debugger so you can set breakpoints, enter the command DEBUG at the EVE command line. You can also gain access to the debugger with the VAXTPU procedure called DEBUGON. To invoke this procedure from within EVE, type the following at the EVE command prompt:

```
Command:  TPU DEBUGON
```

When you use either DEBUG or DEBUGON, the screen displays the debugger window and command line. After setting breakpoints, use the GO command to return control of execution to VAXTPU.

To compile all code in the buffer, use the EVE command EXTEND ALL or use the VAXTPU statement COMPILE (CURRENT_BUFFER). To execute a procedure after compilation, use the EVE command TPU. For example, if you wanted to execute the compiled procedure USER_FUM, you would type the following at the EVE command prompt:

```
Command:  TPU user_fum
```

When VAXTPU encounters a breakpoint (or when you use the STEP command described below), VAXTPU invokes the debugger program. As the debugger assumes control, it receives from VAXTPU the name of the procedure whose execution has been suspended. The debugger searches its source buffer for that procedure.

When VAXTPU encounters the first breakpoint in the session, the code you are debugging has not yet been placed in the debugger's source buffer. The debugger prompts for the name of the file containing your code. Using your response, the debugger places your code in its source buffer. The debugger uses your previous response to supply missing fields, if any, in subsequent file names that you specify. Note that all files read into the source buffer remain there, so that the time VAXTPU takes to find a procedure may increase as more files are read into the source buffer.

You cannot use the EVE command TPU followed by the VAXTPU built-in MESSAGE to examine the contents of a local variable while debugging. To examine a local variable using the MESSAGE built-in, you must write the MESSAGE built-in into the procedure you are debugging. After the statement containing MESSAGE is executed, you can examine the message buffer to see the results. Alternatively, you can use the debugger command EXAMINE to examine local variables and the formal parameters of the suspended procedure.

4.7.3 VAXTPU Debugger Commands

Once you have set breakpoints, compiled code, and started execution, you can use the following commands for debugging:

ATTACH process

Suspends the current editing session and transfers control to another active process or subprocess. DCL process names are case sensitive.

CANCEL BREAKPOINT procedure-name

Cancels a breakpoint set with the SET BREAKPOINT command.

DEPOSIT variable := expression

Enables you to set the values of global variables, local variables, and formal parameters.

DISPLAY SOURCE

Clears text from the screen after use of the HELP or SHOW BREAKPOINTS command. Causes the source display area to display your code. You can enter this command by pressing the key sequence CTRL/Z when you are in the HELP or SHOW display.

EXAMINE variable

Displays the current contents of global and local variables, global constants, formal parameters of the procedure that has been interrupted, and variables local to that procedure. Local constants cannot be examined.

GO

Causes the debugger to relinquish control of execution until it is invoked again by a breakpoint, by the DEBUG command, or by the DEBUGON procedure.

HELP

Lists available debugger commands and keypad bindings.

QUIT

Stops execution of the current procedure. Uses the ABORT statement to return to the main loop of VAXTPU. This command is useful when you have located a problem in a procedure and are ready to get out of the procedure.

SCROLL [-] number-of-lines

Scrolls text in the source display area by the specified number of lines. To scroll backward through the code in the display area, specify a negative number of lines.

To scroll forward by one line less than the number of lines in the display window, press the Next Screen key or the sequence GOLD/↓. To scroll backward in the same way, press the Prev Screen key or the sequence GOLD/↑.

SET BREAKPOINT procedure-name

Invokes the debugger when the specified procedure is entered.

SET WINDOW top, length

Places the top of the debugger window at the line number specified by the **top** parameter. Extends the window down by the number of lines specified by the **length** parameter. The default length is 7 lines. The minimum valid length is 3 lines. The SET WINDOW command only changes the size of the source display area. The output area and command line always occupy exactly one line.

SHIFT [-] number-of-columns

Moves the source display window left or right across the source code to display text wider than the screen.

To move left, you can press the key sequence GOLD/←, then enter the number of columns to move. To move right, you can press the key sequence GOLD/→, then enter the number of columns to move.

SHOW BREAKPOINTS

List the current breakpoints in the debugger source window. To redisplay code in the source window, use the DISPLAY SOURCE command.

SPAWN subprocess

Suspends the current editing session and creates a subprocess.

STEP

Executes one line of VAXTPU code, then returns control to the debugger. If you have several VAXTPU statements on one line, all statements are executed before control returns to the debugger.

TPU statement

Executes the VAXTPU statement you specify. You can enter more than one statement using the TPU command just once.

4.8 Error Handling

Each VAXTPU built-in procedure returns one or more status codes telling you what happened when the built-in was executed. A VAXTPU status code can have one of the following severity levels:

- SUCCESS
- INFORMATIONAL
- WARNING
- ERROR
- FATAL

You can enable or disable the display of informational or success messages with the built-ins SET (INFORMATIONAL) and SET (SUCCESS).

See Chapter 3 for a description of how to use the ON_ERROR language statement to trap error and warning messages.

In addition to messages that are generated by VAXTPU, a built-in procedure may return system messages. Appendix C contains an alphabetized list of all the possible return codes for VAXTPU and their severity levels. The *VMS System Messages and Recovery Procedures Reference Manual* includes all the possible return codes for VAXTPU as well as the appropriate explanations and suggested user actions. In addition, each built-in procedure that can return a warning or error message has the possible messages it can return listed in a section called SIGNALED ERRORS in the individual built-in procedure description.

5 Invoking VAXTPU

The basic DCL command for invoking VAXTPU with EVE (the default editor) is as follows:

```
$  EDIT/TPU
```

This chapter covers the more advanced uses of the EDIT/TPU command, including the following:

- Understanding how to avoid fatal VAXTPU internal errors before using EDIT/TPU. See Section 5.1.
- Invoking VAXTPU from a DCL command procedure. See Section 5.2.
- Invoking VAXTPU from a batch job. See Section 5.3.
- Specifying qualifiers to the EDIT/TPU command. See Section 5.4.
- Understanding how EVE uses the qualifiers that are not processed by VAXTPU. See Section 5.5.
- Specifying a parameter to the EDIT/TPU command. See Section 5.6.

5.1 Avoiding Errors Related to Virtual Address Space

VAXTPU manipulates data in a process's virtual memory space. If the space required by the VAXTPU images, data structures and files in memory exceeds the virtual address space, VAXTPU will try to write part of the data to the work file, thus freeing up space for other parts of the data that it needs immediately.

If the work file is full, VAXTPU attempts to return either a TPU$_GETMEM or TPU$_NOCACHE error message. Although you may be able to free up some space by deleting unused buffers, it is recommended that you terminate the VAXTPU session if you encounter either of these errors. You can then start a new session with fewer or smaller buffers. Alternatively, you may want to put the work file on a disk containing more free space—use one of the following methods to do this:

- Redefine TPU$WORK to point to the disk with more free space, or
- Invoke VAXTPU with the /WORK=*filename* qualifier

VAXTPU may be unable to signal an error when it frees up memory by writing to the work file—in this case, VAXTPU aborts with a fatal internal error.

You may be able to avoid writing to the work file by increasing the virtual address space available to a process. The virtual address space is controlled by the following two factors:

- The SYSGEN parameter VIRTUALPAGECNT

- The page file quota of the account you are using

The VIRTUALPAGECNT parameter controls the number of virtual pages that can be mapped for a process. For more information on VIRTUALPAGECNT, see the description of this parameter in the *VMS System Generation Utility Manual.*

The page file quota controls the number of pages in the system paging file that can be allocated to your process. For more information on the page file quota, see the description of the /PGFLQUOTA qualifier in the *VMS Authorize Utility Manual.*

You may need to modify both the VIRTUALPAGECNT parameter and the page file quota to enlarge the virtual address space.

VAXTPU keeps strings in a different virtual pool than it does other memory. Once VAXTPU starts writing to the work file, the size of the string memory pool is fixed. VAXTPU cannot write strings to the work file, so if it needs to allocate more space in the string memory pool, it will fail with a fatal internal error. If you encounter this problem, you can expand the string memory pool during startup by preallocating several large strings. The following example shows how to do this:

```
PROCEDURE preallocate_strings
LOCAL
    str_len,
    string1,
    string2;

str_len := 65535;
string1 := 'a' * str_len;
string2 := string1;
ENDPROCEDURE;
```

5.2 Invoking VAXTPU from a DCL Command Procedure

There are two reasons that you might want to invoke VAXTPU from a command procedure:

- To set up a special environment for interactive editing
- To execute a noninteractive, VAXTPU-based application

5.2.1 Setting Up a Special Editing Environment

You can run VAXTPU with a special editing environment by writing a DCL command procedure that first establishes the environment that you want, and then invokes VAXTPU. In such a command procedure, you must define SYS$INPUT to have the same value as SYS$COMMAND, because VAXTPU signals an error if SYS$INPUT is not defined as the terminal. To prevent such an error, place the following statement in the command procedure setting up the environment:

```
$ DEFINE/USER SYS$INPUT SYS$COMMAND
```

Example 5–1 shows a DCL command procedure that "remembers" the last file that you were editing and uses it as the input file for VAXTPU. When you edit a file, the file name you specify is saved in the DCL symbol *last_file_edited*. If you do not specify a file name when you invoke the editor the next time, the file name from the previous session is used.

Example 5–1 DCL Command Procedure FILENAME.COM

```
$ IF P1 .NES. "" THEN last_file_edited == P1
$ WRITE SYS$OUTPUT "*** ''last_file_edited' ***"
$ DEFINE/USER SYS$INPUT SYS$COMMAND
$ EDIT/TPU/COMMAND=DISK$:[USER]TPU$COMMAND.TPU 'last_file_edited'
```

Example 5–2 establishes an environment that specifies tab stop settings for FORTRAN programs.

Example 5–2 DCL Command Procedure FORTRAN_TS.COM

```
$ IF P1 .EQS. "" THEN GOTO REGULAR_INVOKE
$ last_file_edited == P1
$ FTN_TEST = F$FILE_ATTRIBUTES (last_file_edited,"RAT")
$ IF FTN_TEST .NES. "FTN" THEN GOTO REGULAR_INVOKE
$ FTN_INVOKE:
$     DEFINE/USER SYS$INPUT SYS$COMMAND
$     EDIT/TPU/COMMAND=FTNTABS 'last_file_edited'
$ GOTO TPU_DONE
$ REGULAR_INVOKE:
$     DEFINE/USER SYS$INPUT SYS$COMMAND
$     EDIT/TPU/ 'last_file_edited'
$ TPU_DONE:
```

5.2.2 Creating a Noninteractive Application

In some situations, you may want to put all of your editing commands in a file and have them read from the file rather than entering the commands interactively. You may also want VAXTPU to perform the edits without displaying them on the screen. You can do this type of editing from a batch job; or, if you want to see the results of the editing session displayed on your screen, you can do this type of editing from a DCL command procedure. Even though the edits are not displayed on your screen as they are being made, your terminal is not free while the command procedure is executing.

Example 5–3 shows a DCL command procedure named INVISIBLE_TPU.COM containing a single command line that invokes VAXTPU using the following qualifiers:

- /NOSECTION — This qualifier prevents VAXTPU from using a section file. All procedures and key definitions must be specified in a command file.
- /COMMAND=gsr.tpu — This qualifier specifies a command file containing the code to be executed (GSR.TPU).

- /NODISPLAY — This qualifier suppresses screen display.

Example 5–3 DCL Command Procedure INVISIBLE_TPU.COM

```
! This command procedure invokes VAXTPU without an editor.
! The file GSR.TPU contains the edits to be made.
! Specify the file to which you want the edits made as p1.
!
$ EDIT/TPU/NOSECTION/COMMAND=gsr.tpu/NODISPLAY 'p1'
!
```

The VAXTPU command file GSR.TPU, which is used as the file specification for the qualifier /COMMAND, performs a search through the current buffer and replaces a string or a pattern with a string. Example 5–4 shows the file GSR.TPU. Note that GSR.TPU does not create or manipulate any windows.

Example 5–4 VAXTPU Command File GSR.TPU

```
PROCEDURE global_search_replace (str_or_pat, str2)

! This procedure performs a search through the current
! buffer and replaces a string or a pattern with a new string

LOCAL src_range, replacement_count;

! Return to caller if string not found
ON_ERROR
   msg_text := FAO ('Completed !UL  replacement!%S', replacement_count);
   MESSAGE (msg_text);
   RETURN;
ENDON_ERROR;

replacement_count := 0;

LOOP
   src_range := SEARCH (str_or_pat, FORWARD);    ! Search returns a range if found
   ERASE (src_range);                            ! Remove first string
   POSITION (END_OF (src_range));                ! Move to right place
   COPY_TEXT (str2);                             ! Replace with second string
   replacement_count := replacement_count + 1;
ENDLOOP;
ENDPROCEDURE;        ! global_search_replace

! Executable statements
input_file := GET_INFO (COMMAND_LINE, "file_name");
main_buffer:= CREATE_BUFFER ("main", input_file);
POSITION (BEGINNING_OF (main_buffer));
global_search_replace ("xyz$_", "user$_");
pat1:= "" & LINE_BEGIN & "t";
POSITION (BEGINNING_OF (main_buffer));
global_search_replace (pat1, "T");
WRITE_FILE (main_buffer, "newfile.dat");
QUIT;
```

To use the DCL command procedure INVISIBLE_TPU.COM interactively, invoke it with the DCL command @ (at sign). For example, to use INVISIBLE_TPU.COM interactively on a file called MY_FILE.TXT, you would type the following at the DCL prompt:

```
$ @invisible_tpu my_file.txt
```

You must explicitly write out any modified buffers before leaving the editor with QUIT or EXIT. If you use QUIT before writing out such buffers, VAXTPU quits without saving the modifications. If you use EXIT, VAXTPU asks if it should write the file before exiting.

5.3 Invoking VAXTPU from a Batch Job

If you want your edits to be made in batch rather than at the terminal, you can use the DCL command SUBMIT to send your job to a batch queue.

For example, if you wanted to use the file GSR.TPU (shown in Example 5–4) to make edits in batch mode to a file called MY_FILE.TXT, you would enter the following command:

```
$ SUBMIT invisible_tpu.COM/LOG=invisible_tpu.LOG/parameter=my_file.txt
```

This job is then entered in the default batch queue for your system. The results are sent to the LOG file that the batch job creates.

Note that in batch VAXTPU, EXIT is the same as QUIT.

5.4 Qualifiers to the DCL Command EDIT/TPU

The DCL command EDIT/TPU has qualifiers for setting attributes of VAXTPU or an application layered on VAXTPU. The qualifiers fall into the following two categories:

- Qualifiers handled by VAXTPU. Qualifiers in this category have their defaults set by VAXTPU.
- Qualifiers handled by the application layered on VAXTPU. Some qualifiers in this category have their defaults set entirely by VAXTPU; some have their defaults set entirely by the layered application, and some have their defaults set partly by each.

Table 5–1 shows, for each qualifier, which program sets the default and which program is responsible for handling the qualifier.

Table 5–1 Summary of How VAXTPU and the Application Layered on VAXTPU Relate to the Qualifiers to EDIT/TPU

Qualifier	Program That Sets the Qualifier's Default	Program Responsible for Handling the Qualifier
/[NO]COMMAND[=filespec]	VAXTPU	VAXTPU
/[NO]CREATE	Both VAXTPU and the application layered on VAXTPU	The application layered on VAXTPU
/[NO]DEBUG[=filespec]	VAXTPU	VAXTPU
/[NO]DISPLAY[=keyword]	VAXTPU	VAXTPU
/[NO]INITIALIZATION [=filespec]	Both VAXTPU and the application layered on VAXTPU	The application layered on VAXTPU

(continued on next page)

Table 5–1 (Cont.) Summary of How VAXTPU and the Application Layered on VAXTPU Relate to the Qualifiers to EDIT/TPU

Qualifier	Program That Sets the Qualifier's Default	Program Responsible for Handling the Qualifier
/INTERFACE[= interface]	VAXTPU	VAXTPU
/[NO]JOURNAL[=filespec]	Both VAXTPU and the application layered on VAXTPU	The application layered on VAXTPU
/[NO]MODIFY	The application layered on VAXTPU	The application layered on VAXTPU
/[NO]OUTPUT[=filespec]	Both VAXTPU and the application layered on VAXTPU	The application layered on VAXTPU
/[NO]READ_ONLY	Both VAXTPU and the application layered on VAXTPU	The application layered on VAXTPU
/[NO]RECOVER	VAXTPU	VAXTPU
/[NO]SECTION[=filespec]	VAXTPU	VAXTPU
/START_POSITION[=(line,column)]	VAXTPU	The application layered on VAXTPU
/[NO]WRITE	Both VAXTPU and the application layered on VAXTPU	The application layered on VAXTPU

The following subsections present the qualifiers in alphabetical order, giving a more detailed description of each qualifier. The examples in the following sections show the qualifiers directly after the EDIT/TPU command and before the input file specification. You can place the qualifiers anywhere on the command line after EDIT/TPU. These subsections show the defaults that are set if you use EVE. The subsections explain how EVE handles each qualifier that can be processed by a layered application. Applications not based on EVE may handle such qualifiers differently.

5.4.1 /COMMAND

/COMMAND[=filespec]
/NOCOMMAND
/COMMAND=TPU$COMMAND.TPU (default)

Determines whether VAXTPU compiles and executes a command file (a file of VAXTPU procedures and statements) at startup time. Command files extend or modify a VAXTPU-based application or create a new application. The default file type for VAXTPU command files is TPU. You cannot use wildcards in the file specification.

By default, VAXTPU tries to read a command file called TPU$COMMAND.TPU in your default directory. You can use a full file specification after the qualifier /COMMAND or define the logical name TPU$COMMAND to point to a command file other than the default one.

To determine whether the user specified /COMMAND on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "command");
```

The preceding call returns 1 if /COMMAND was specified, 0 otherwise. To fetch the name of the command file specified on the command line, use the following call:

```
x := GET_INFO (COMMAND_LINE, "command_file");
```

For more information on GET_INFO, see the VAXTPU Reference Section.

The following command causes VAXTPU to read a command file named SYS$LOGIN:MY_TPU$COMMAND.TPU and uses LETTER.RNO as the input file for an editing session:

```
$ EDIT/TPU/COMMAND=sys$login:my_tpu$command.tpu letter.rno
```

To prevent VAXTPU from processing a command file, use the qualifier /NOCOMMAND. If you usually invoke VAXTPU without a command file, define a symbol similar to the following:

```
$ EVE == "EDIT/TPU/NOCOMMAND"
```

Using /NOCOMMAND when you do not want to use a command file decreases startup time by eliminating the search for a command file.

If you specify a command file that does not exist, VAXTPU terminates the editing session and returns you to DCL.

For more information on writing and using command files, see Chapter 4.

5.4.2 /CREATE

/CREATE (default)
/NOCREATE

Controls whether a VAXTPU-based application creates a new file when the specified input file is not found. If the user specifies neither /CREATE nor /NOCREATE on the command line, VAXTPU sets the default to /CREATE but does not specify a default name for the file to be created.

The application layered on VAXTPU is responsible for handling this qualifier.

To determine if the user specified /CREATE on the DCL command line, include the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "create");
```

The preceding call returns 1 if /CREATE was specified, 0 otherwise. For more information on GET_INFO, see the VAXTPU Reference Section.

By default, EVE creates a new file if the specified input file does not exist. If you use /NOCREATE and specify an input file that does not exist, EVE aborts the editing session and returns you to the DCL command level. For example, if your default device and directory are DISK$:[USER] and

you specify a nonexistent file, NEWFILE.DAT, your command and EVE's response would be as follows:

```
$  EDIT/TPU/NOCREATE newfile.dat
Input file does not exist: DISK$:[USER]NEWFILE.DAT;
```

5.4.3 /DEBUG

/DEBUG[[=debug_source_filename]]
/NODEBUG (default)

Determines whether VAXTPU loads, compiles, and executes a file implementing a VAXTPU debugger. If /DEBUG is specified, VAXTPU reads, compiles, and executes the contents of a debugger file before executing the procedure TPU$INIT_PROCEDURE and before executing the command file. For more information on the VAXTPU initialization sequence, see Chapter 4.

By default, VAXTPU does not load a debugger. If you specify that a debugger is to be loaded but do not supply a file specification, VAXTPU loads the file SYS$SHARE:TPU$DEBUG.TPU. For more information on how to use the default VAXTPU debugger, see Chapter 4.

To use a debugger file other than the default, use the /DEBUG qualifier and specify the device, directory, and file name of the debugger to be used. If you specify only the file name, VAXTPU searches SYS$SHARE for the file. You can define the logical name TPU$DEBUG to specify a file containing a debugger program. Once you define this logical name, using /DEBUG without specifying a file calls the file specified by TPU$DEBUG.

5.4.4 /DISPLAY

/DISPLAY [[= CHARACTER_CELL (default) | = DECWINDOWS]]
/NODISPLAY

To choose the DECwindows or the non-DECwindows version of VAXTPU, use the command qualifier /DISPLAY on the DCL command line when you invoke VAXTPU.

The /DISPLAY command qualifier is optional. By default, VAXTPU uses /DISPLAY=CHARACTER_CELL, regardless of whether you are running VAXTPU on a workstation or a terminal.

If you specify /DISPLAY = CHARACTER_CELL, VAXTPU uses its character-cell screen manager, which implements the non-DECwindows version of VAXTPU by running in a DECterm (or VWS) terminal emulator or on a physical terminal.

If you specify /DISPLAY=DECWINDOWS, and if the DECwindows environment is available, VAXTPU uses the DECwindows screen manager, which creates a DECwindows window in which to run VAXTPU.

If you specify /DISPLAY=DECWINDOWS and the DECwindows environment is not available, VAXTPU uses its character-cell screen manager to implement the non-DECwindows version of VAXTPU.

For more information about the difference between a DECwindows window and a VAXTPU window, see Chapter 4.

The qualifier /NODISPLAY causes VAXTPU to run without using the screen display and the keyboard functions of a terminal. Use the qualifier /NODISPLAY in the following cases:

- When running VAXTPU procedures in a batch job
- When using VAXTPU on an unsupported terminal

When you use /NODISPLAY, all operations continue as normal, except that no output occurs. (The only exception is that information normally put into the message buffer will appear on SYS$OUTPUT if no message buffer is available.)

The following command causes VAXTPU to edit the file MY_BATCH_FILE.RNO without using terminal functions such as screen display:

```
$ EDIT/TPU/NODISPLAY my_batch_file.rno
```

5.4.5 /INITIALIZATION

/INITIALIZATION[[=filespec]] (default)
/NOINITIALIZATION

Determines whether the VAXTPU-based application being run executes a file of initialization commands. The application layered on VAXTPU is responsible for processing this qualifier.

To determine whether the user specified /INITIALIZATION on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "initialization");
```

The preceding call returns 1 if /INITIALIZATION was specified, 0 otherwise. To fetch the name of the initialization file specified on the command line, use the following call:

```
x := GET_INFO (COMMAND_LINE, "initialization_file");
```

For more information on GET_INFO, see the VAXTPU Reference Section.

If the user does not specify any form of /INITIALIZATION on the DCL command line, VAXTPU specifies /INITIALIZATION but does not supply a default file specification. The default file specification for /INITIALIZATION is set by the application. Digital recommends that a user-written application define the default file specification of an initialization file using the following format:

```
facility$init.facility
```

For example, the default initialization file for the EVE editor is EVE$INIT.EVE.

In EVE, if the user does not specify a device or directory, EVE first checks the current directory. If the specified (or default) initialization file is not there, EVE checks SYS$LOGIN. If EVE finds the specified (or default) initialization file, EVE executes the commands in the file.

For more information on using initialization files with EVE, see Chapter 4 and the *Guide to VMS Text Processing*.

5.4.6 /INTERFACE

/INTERFACE [[= CHARACTER_CELL | = DECWINDOWS]]
/INTERFACE= CHARACTER_CELL (default)

Determines the interface or screen display you want (same as /DISPLAY). The default is CHARACTER_CELL.

For example, to invoke EVE with the DECwindows interface, you can use the following command:

```
$ EDIT/TPU /INTERFACE=DECWINDOWS
```

Then, if DECwindows is available, VAXTPU displays the editing session in a separate window on your workstation screen, and enables DECwindows features—for example, the EVE screen layout includes a menu bar and scroll bars. If DECwindows is not available, VAXTPU works as if on a character-cell terminal.

5.4.7 /JOURNAL

/JOURNAL[[=input_file.TJL]] (default for EVE)
/NOJOURNAL (default for VAXTPU)

Determines whether VAXTPU keeps a journal file of an editing session so the session can be recovered if it is unexpectedly interrupted. VAXTPU offers two forms of journaling:

- Keystroke—keeps track of each keystroke you make in a single journal file, regardless of which buffer is in use when you press the key.
- Buffer change—keeps track of changes made to buffers in a separate journal file for each buffer created during the session.

The application layered on VAXTPU is responsible for processing this qualifier.

To determine whether the user specified /JOURNAL on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "journal");
```

The preceding call returns 1 if /JOURNAL was specified, 0 otherwise.

To determine whether buffer change journaling is turned on for a buffer, use a statement similar to the following:

```
status := GET_INFO (buffer_name, "journaling");
```

To determine the name of the keystroke journal file specified on the command line, use the following call:

```
x := GET_INFO (COMMAND_LINE, "journal_file");
```

For more information on GET_INFO, see the VAXTPU Reference Section.

In EVE, if the user does not specify any form of /JOURNAL or specifies /JOURNAL but not a journal file, buffer change journaling is turned on. The buffer change journal file's default file type is TPU$JOURNAL.

If the user specifies /JOURNAL=*filename*, then EVE also turns on keystroke journaling. The keystroke journal file's default file type is TJL.

To prevent EVE from creating either a keystroke or buffer change journal file for an editing session, use the qualifier /NOJOURNAL. For example, the following command causes EVE to turn off buffer change journaling when you edit the input file MEMO.TXT:

```
$ EDIT/TPU/NOJOURNAL memo.txt
```

If you are developing an application layered on VAXTPU, you can direct VAXTPU to create a keystroke journal file for an editing session by using the built-in JOURNAL_OPEN. Using JOURNAL_OPEN causes VAXTPU to provide a 500-byte buffer in which to journal keystrokes. By default, VAXTPU writes the contents of the buffer to the journal file when the buffer is full.

You can use the built-in procedure SET (JOURNALING) to turn on buffer change journaling, even if you have used /NOJOURNAL to turn it off initially. You can also use SET (JOURNALING) to adjust the journaling frequency.

For more information on JOURNAL_OPEN and SET (JOURNALING), see the descriptions of these built-ins in the VAXTPU Reference Section.

For more information on buffer change journaling, see Section 1.7.

Once a keystroke journal file is created, use the qualifier /RECOVER to direct VAXTPU to process the commands in the keystroke journal file. For example, the following command causes VAXTPU to recover a previous editing session on an input file named MEMO.TXT. Because the journal file has a name different from the input file name, both /JOURNAL and /RECOVER are used. The name of the keystroke journal file is MEMO.TJL:

```
$ EDIT/TPU/RECOVER/JOURNAL=memo.tjl memo.txt
```

In buffer change journaling, to recover the changes made to a specifed buffer, use the RECOVER_BUFFER built-in procedure. For more information on RECOVER_BUFFER, see its description in the VAXTPU Reference Section.

For more information on how to recover from an interrupted EVE editing session, see the *Guide to VMS Text Processing* and the *VMS EVE Reference Manual.*

5.4.8 /MODIFY

/MODIFY (default)
/NOMODIFY

Determines whether the first user buffer in an editing session is modifiable. The application layered on VAXTPU is responsible for processing /MODIFY.

To determine what form of the /MODIFY qualifier was used on the DCL command line, use the following calls:

```
x := GET_INFO (COMMAND_LINE, "modify");
x := GET_INFO (COMMAND_LINE, "nomodify");
```

The first statement returns 1 if /MODIFY was explicitly specified on the command line, 0 otherwise. The second statement returns 1 if /NOMODIFY was explicitly specified on the command line, 0 otherwise. If both statements return 0, then the application is expected to determine the default behavior. For more information on GET_INFO, see the VAXTPU Reference Section.

If you invoke EVE and do not specify /MODIFY, /NOMODIFY, /READ_ONLY, or /NOWRITE, EVE makes the first user buffer of the editing session modifiable. If you specify /NOMODIFY, EVE makes the first user buffer unmodifiable. Regardless of what qualifiers you use on the DCL command line, EVE makes all user buffers after the first buffer modifiable.

If you do not specify either form of the /MODIFY qualifier, EVE checks whether you have used any form of the /READ_ONLY or /WRITE qualifiers. By default, a read-only buffer is unmodifiable and a write buffer is modifiable. However, if you specify /READ_ONLY and /MODIFY or /NOWRITE and /MODIFY, the buffer is modifiable. Similarly, if you specify /WRITE and /NOMODIFY or /NOREAD_ONLY and /NOMODIFY, the buffer is unmodifiable.

5.4.9 /OUTPUT

/OUTPUT=input_file.type (default)
/NOOUTPUT

Determines whether the output of your VAXTPU session is written to a file. The application layered on VAXTPU is responsible for processing this qualifier.

To determine whether the user specified /OUTPUT on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "output");
```

The preceding call returns 1 if /OUTPUT was specified, 0 otherwise. To fetch the name of the output file specified on the command line, use the following call:

```
x := GET_INFO (COMMAND_LINE, "output_file");
```

For more information on GET_INFO, see the VAXTPU Reference Section.

If you do not specify any form of /OUTPUT on the DCL command line, VAXTPU specifies /OUTPUT but does not supply a default file specification.

In EVE, using /OUTPUT allows you to name the file created from the main buffer when you exit from VAXTPU. For example, the following command causes VAXTPU to read in a file called LETTER.RNO and to write the contents of the main buffer to the file NEWLET.RNO upon exiting from VAXTPU:

```
$ EDIT/TPU/OUTPUT=newlet.rno letter.rno
```

By default, the output file has the same name as the input file, and the version number is one higher than the highest existing version of the input file. You can specify a different name for the output file by using the file specification argument for the qualifier /OUTPUT.

In EVE, specifying /NOOUTPUT causes EVE to suppress creation of an output file for the first buffer of the editing session. Using /NOOUTPUT does not suppress creation of a journal file.

Using /NOOUTPUT, you can develop an application letting the user control the output of a file. For example, an application could be coded so that if the user specifies /NOOUTPUT on the DCL command line, VAXTPU would set the NO_WRITE attribute for the main buffer and suppress creation of an output file for that buffer.

5.4.10 /READ_ONLY

/READ_ONLY
/NOREAD_ONLY (default)

Determines whether the application layered on VAXTPU creates an output file from the contents of the main buffer if the contents are modified.

The processing of the /READ_ONLY qualifier is interrelated with the processing of the /WRITE qualifier. /READ_ONLY is equivalent to /NOWRITE; /NOREAD_ONLY is equivalent to /WRITE.

VAXTPU signals an error and returns control to DCL if VAXTPU encounters either of the following combinations of qualifiers on the DCL command line:

- /READ_ONLY and /WRITE
- /NOREAD_ONLY and /NO_WRITE

The application layered on VAXTPU is responsible for processing this qualifier.

To determine whether either the /READ_ONLY or /NOWRITE qualifier was used on the DCL command line, use the following call in an application:

```
x := GET_INFO (COMMAND_LINE, "read_only");
```

This statement returns 1 if /READ_ONLY or /NOWRITE was explicitly specified on the command line.

To determine whether either /NOREAD_ONLY or /WRITE was used on the DCL command line, use the following call in an application:

```
x := GET_INFO (COMMAND_LINE, "write");
```

This statement returns 1 if /NOREAD_ONLY or /WRITE was explicitly specified on the command line.

If both GET_INFO calls return false, the application is expected to determine the default behavior. For more information on GET_INFO, see the VAXTPU Reference Section.

In EVE, using the qualifier /READ_ONLY is equivalent to using the qualifiers /NOJOURNAL, /NOMODIFY, and /NOOUTPUT. If you specify /READ_ONLY, VAXTPU does not maintain a journal file for your editing session, and the NO_WRITE and NO_MODIFY attributes are set for the main buffer. When a buffer is set to NO_WRITE, the contents of the buffer are not written out upon exit, regardless of whether the session is terminated with the EXIT built-in or the QUIT built-in. For example, if you want to edit a file called MEETING.MEM but not write out the contents when exiting or quitting, you would use the following command:

```
$ EDIT/TPU/READ_ONLY meeting.mem
```

In response to /NOREAD_ONLY, EVE writes out the main buffer (if the buffer has been modified) when an EXIT command is issued. This is the default behavior.

5.4.11 /RECOVER

/RECOVER
/NORECOVER (default)

Determines whether VAXTPU reads a keystroke journal file at the start of an editing session to recover edits made during a prior interrupted editing session. For example, the following command causes VAXTPU to recover the edits made in a previous EVE editing session on the file NOTES.TXT:

```
$ EDIT/TPU/RECOVER notes.txt
```

To determine whether the user specified /RECOVER on the DCL command line, use the following call:

```
x := GET_INFO (COMMAND_LINE, "recover");
```

The preceding call returns 1 if /RECOVER was specified, 0 otherwise. For more information on GET_INFO, see the VAXTPU Reference Section.

Note that VAXTPU uses /RECOVER to recover a keystroke journal file *only*. In buffer change journaling, to recover the changes made to a specifed buffer, use the RECOVER_BUFFER built-in procedure. For more information on RECOVER_BUFFER, see its description in the VAXTPU Reference Section.

If VAXTPU encounters and executes the built-in procedure JOURNAL_OPEN while running a layered application, by default VAXTPU opens the journal file for output only. If the user specifies /RECOVER when invoking VAXTPU with a layered application, then when the built-in procedure JOURNAL_OPEN is executed, the keystroke journal file is opened for input and output. VAXTPU opens the input file to restore whatever commands it contains. Then VAXTPU continues to journal keystrokes for the rest of the editing session or until a statement containing the built-in JOURNAL_CLOSE is executed.

When you recover an editing session, every file used during the session must be in the same state as it was at the start of the session being recovered. Each terminal characteristic must also be in the same state as it was at the start of the editing session being recovered. If you have changed the width or page length of the terminal, you must change the attribute back to the value it had at the start of the editing session you want to recover. Check especially the following values:

- Device type
- Edit mode
- Eight bit
- Page length
- Width

If the journal file has a different name from the input file, you must include both /JOURNAL and /RECOVER with the EDIT/TPU command. For example, if you wanted to recover the edits you had made to a file called LETTER.DAT using the keystroke journal file SAVE.TJL, you would enter the following command on the DCL command line:

```
$ EDIT/TPU/RECOVER/JOURNAL=save.TJL letter.dat
```

In EVE, /RECOVER may be used to recover either an editing session from a keystroke journal file or a single buffer from a buffer change journal file. If you specify /JOURNAL=*filename*, EVE recovers from the specified keystroke journal file. Otherwise, EVE recovers from a buffer change journal file that corresponds to the input parameter (or the buffer Main if no input parameter is specified).

For more information on journaling and recovery in EVE, see the *VMS EVE Reference Manual* or the *Guide to VMS Text Processing*.

5.4.12 /SECTION

/SECTION[[=filespec]]
/NOSECTION
/SECTION=TPU$SECTION (default)

Determines whether VAXTPU loads a section file. A section file is a startup file containing key definitions and compiled procedures in binary form.

The default section file is TPU$SECTION. When VAXTPU tries to locate the section file, VAXTPU supplies a default directory of SYS$SHARE and a default file type of TPU$SECTION. VMS defines the systemwide logical name TPU$SECTION as EVE$SECTION, so the default section file is the file implementing the EVE editor. To override the VMS default, redefine TPU$SECTION.

You can specify a different section file. The preferred method is to define the logical name TPU$SECTION to point to a section file other than the default file. You can also supply a full file specification for the qualifier /SECTION. For example, if your device is called DISK$USER and your directory is called [SMITH], the following command causes VAXTPU to read a section file called VT100INI.TPU$SECTION:

```
$ EDIT/TPU/SECTION=disk$user:[smith]vt100ini
```

If you omit the device and directory in the file specification, VAXTPU assumes the file is in SYS$SHARE. The section file must be located on the same node on which you are running VAXTPU.

To determine whether /SECTION was specified on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "section");
```

The preceding call returns 1 if /SECTION was specified, 0 otherwise. To fetch the name of the section file specified on the command line, use the following call:

```
x := GET_INFO (COMMAND_LINE, "section_file");
```

For more information on GET_INFO, see the VAXTPU Reference Section.

The file used as the value for the /SECTION qualifier must be compiled by running the source code version of the file through VAXTPU and then using the built-in procedure SAVE. This process converts the file to the proper binary form. For more information on creating and using section files, see Chapter 4 and the *Guide to VMS Text Processing*.

If you specify /NOSECTION, VAXTPU does not load a section file. Unless you use the qualifier /COMMAND with /NOSECTION, VAXTPU has no user interface and no keys are defined. In this state, the only way to exit from VAXTPU is to press CTRL/Y. Typically, you use /NOSECTION when creating your own layered VAXTPU application without using EVE as a base.

5.4.13 /START_POSITION

/START_POSITION=(line,column)
/START_POSITION=(1,1) (default)

Determines where the application layered on VAXTPU positions the cursor when the user invokes the application.

The application layered on VAXTPU is responsible for processing this qualifier.

To determine the row and column that the user has specified on the DCL command line using /START_POSITION, use the following calls in the application:

```
start_line := GET_INFO (COMMAND_LINE, "start_record");
start_char := GET_INFO (COMMAND_LINE, "start_character");
```

For more information on GET_INFO, see the VAXTPU Reference Section.

VAXTPU sets the starting row and starting column to 1 if the user does not use /START_POSITION on the DCL command line.

EVE uses this qualifier to determine the row and column in the main buffer where the cursor first appears. By default, the start position is row 1, column 1 (the upper left corner) of the buffer. Typically, you use /START_POSITION when you want to begin editing at a particular line or column, such as when you want to skip over a standard heading in a file.

5.4.14 /WRITE

/WRITE (default)
/NOWRITE

Determines whether the application layered on VAXTPU creates an output file from the contents of the main buffer if the contents are modified.

The processing of the /WRITE qualifier is interrelated with the processing of the /READ_ONLY qualifier. /WRITE is equivalent to /NOREAD_ONLY; /NOWRITE is equivalent to /READ_ONLY.

VAXTPU signals an error and returns control to DCL if VAXTPU encounters either of the following combinations of qualifiers on the DCL command line:

- /READ_ONLY and /WRITE
- /NOREAD_ONLY and /NO_WRITE

The application layered on VAXTPU is responsible for processing this qualifier.

To determine whether the /WRITE or the /NOREAD_ONLY qualifier was used on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "write");
```

This statement returns 1 if /NOREAD_ONLY or /WRITE was explicitly specified on the command line.

To determine whether the /NOWRITE or /READ_ONLY qualifier was used on the DCL command line, use the following call in the application:

```
x := GET_INFO (COMMAND_LINE, "read_only");
```

This statement returns 1 if /READ_ONLY or /NOWRITE was explicitly specified on the command line.

If both GET_INFO calls return false, the application is expected to determine the default behavior. For more information on GET_INFO, see the VAXTPU Reference Section.

In EVE, using the qualifier /NOWRITE is equivalent to using the qualifiers /NOJOURNAL, /NOMODIFY, and /NOOUTPUT. If you specify /NOWRITE, VAXTPU does not maintain a journal file for your editing session, and the NO_WRITE and NO_MODIFY attributes are set for the main buffer. When a buffer is set to NO_WRITE, the contents of the buffer are not written out upon exit, regardless of whether the session is terminated with the EXIT built-in or the QUIT built-in. For example, if you want to edit a file called MEETING.MEM but not write out the contents when exiting or quitting, you use the following command:

```
$ EDIT/TPU/READ_ONLY meeting.mem
```

5.5 How EVE Uses /MODIFY, /OUTPUT, /READ_ONLY, and /WRITE

EVE uses the qualifiers /MODIFY, /OUTPUT, /READ_ONLY, and /WRITE to determine whether to make the first user buffer of an EVE editing session modifiable and whether to write the contents of the buffer, if modified, to a file when the user exits. (By default, all EVE user buffers created after the first buffer in an editing session start out modifiable and, if modified, are written to a file when the user exits.)

Because these qualifiers are interrelated, this section covers the order in which EVE processes the qualifiers. Note that if you layer an application on top of EVE, then EVE handles these qualifiers for your application unless you explicitly override EVE's actions.

To process these four interrelated qualifiers, EVE performs the following steps in the order shown:

1. EVE makes the first user buffer modifiable and makes it a write buffer.
2. EVE checks whether /NOOUTPUT was specified on the DCL command line. If so, the call GET_INFO (COMMAND_LINE, "output") returns the value false and the callable interface bit TPU$V_OUTPUT is set to 0. EVE prevents the buffer from being written out by specifying the ON parameter with the built-in SET (NO_WRITE).
3. EVE checks whether /READ_ONLY was specified on the DCL command line. If so, the call GET_INFO (COMMAND_LINE, "read_only") returns the value true and the callable interface bit TPU$V_READ is set to 1. EVE prevents the buffer from being written out by specifying the ON parameter with the built-in SET (NO_WRITE). EVE

also prevents the buffer from being modified by specifying the OFF parameter with the built-in SET (MODIFIABLE).

4 EVE checks whether /WRITE was specified on the DCL command line. If so, the call GET_INFO (COMMAND_LINE, "write") returns the value true and the callable interface bit TPU$V_WRITE is set to 1. EVE makes the buffer writable by specifying the OFF parameter with the built-in SET (NO_WRITE). EVE also makes the buffer modifiable by specifying the ON parameter with the built-in SET (MODIFIABLE).

5 EVE checks whether /MODIFY was specified on the DCL command line. If so, the call GET_INFO (COMMAND_LINE, "modify") returns the value true and the callable interface bit TPU$V_MODIFY is set to 1. EVE makes the buffer modifiable by specifying the ON parameter with the built-in SET (MODIFIABLE).

6 EVE checks whether /NOMODIFY was specified on the DCL command line. If so, the call GET_INFO (COMMAND_LINE, "nomodify") returns the value true and the callable interface bit TPU$V_NOMODIFY is set to 1. EVE prevents the buffer from being modified by specifying the OFF parameter with the built-in SET (MODIFIABLE).

7 EVE checks whether the user has both specified /NOWRITE and specified /OUTPUT with a file specification. If so, EVE signals an error and terminates the editing session.

5.6 Specifying a Parameter to EDIT/TPU

You can use a VMS file specification as a parameter to the command EDIT/TPU. The syntax for invoking VAXTPU with a parameter is as follows:

```
$ EDIT/TPU [[/qualifier,...]] [[filespec]]
```

The parameter is the name of the file you want to create or edit using VAXTPU. For example, the following command invokes VAXTPU with the section file EVE$SECTION and specifies as a parameter a file named HISTORY.TXT:

```
$  EDIT/TPU/SECTION=sys$library:eve$section history.txt
```

VAXTPU never requires the parameter. However, most applications use the parameter to the EDIT/TPU command to specify the file that is to be processed. For example, EVE accepts a file specification as an optional parameter. You can start an EVE editing session without specifying an input file, but if you enter any data into a buffer, EVE prompts you for a file name when you exit.

A file specification can be a full file specification or just the file name. For example, if your device is called DISK$USER and your directory is called [SMITH], the following command invokes VAXTPU with the file LETTER.DAT:

```
$  EDIT/TPU disk$user:[smith]letter.dat
```

To determine what file has been specified as a parameter, use the following call in an application:

```
x := GET_INFO (COMMAND_LINE, "file_name");
```

The application layered on VAXTPU determines whether VAXTPU recognizes wildcard characters in the input file specification. For example, EVE handles wildcard characters if there is one unique file that matches the wildcard specification. Otherwise, EVE does not read a file. Other applications can handle wildcard characters differently.

You do not have to include the version number as part of the file specification. If you do not specify a version number, EVE opens the file that has the highest version number. To edit an earlier version, include the version number in the file specification.

The handling of the specified file at exit time depends on the application layered on VAXTPU. For example, EVE uses the input file name as the name of the output file unless the user specifies the name of an output file using the qualifier /OUTPUT. EVE leaves the original version of the input file, unaltered, in its directory unless the system manager has set a version limit. When you exit from EVE, a new file is created in the input file's directory (unless the user has specified a different directory). The file has the same name as the input file but has a version number that is one higher than the input file.

6 VAXTPU Screen Management

The VAXTPU screen manager handles the display of windows and the buffers mapped to those windows. This chapter discusses how to invoke the screen manager, what you can expect it to do, and how the screen manager handles various display situations.

To disable the screen manager, use the /NODISPLAY qualifier when you invoke VAXTPU. By default, the screen manager is enabled, causing the screen to display all VAXTPU operations.

6.1 How the Screen Manager Handles Windows and Buffers

A window is an area of the terminal screen used to display the contents of a buffer. There are two ways to modify the way information is displayed on the screen:

- Modify the size, attributes, or location of the display area
- Modify the information that is presented

The screen manager automatically updates the window when VAXTPU finishes processing a keystroke or series of keystrokes. When input is entered, VAXTPU queues the keystrokes for processing. As the input is processed, either by inserting characters into the buffer or by executing the procedures bound to the keys, the input is taken off the queue. When the queue is completely empty, the screen manager is called to reflect the changes. For more information on what happens during an update, see Section 6.2.1, Section 6.2.2, and Section 6.2.3.

6.1.1 Buffer Changes

Buffers can be changed by:

- Inserting records
- Deleting records
- Modifying characters in a record
- Modifying video attributes associated with characters
- Modifying record attributes

To make the screen display modifications to a buffer, use the UPDATE built-in. Note, however, that a screen update does not reflect any modifications to portions of a buffer that are not visible in the window mapped to the buffer. VAXTPU has a restriction on the screen display of modifications to a buffer. If two or more windows are mapped to the same portion of the same buffer and a select range is created in the current window, the other windows do not display the select range unless the user

or subsequent code invokes the REFRESH built-in or the EVE REFRESH command.

6.1.2 Window Changes

Changes to windows occur at the following times:

- When a window is mapped to a buffer
- When a window is deleted
- When a window becomes the current window
- When a window is shifted
- When a window changes size or location

Creating a window causes no visible effects. To become visible, a window must be mapped to a buffer. For more information, see the descriptions of the MAP, DELETE, POSITION, SHIFT, ADJUST_WINDOW, and CREATE_WINDOW built-ins in the VAXTPU Reference Section.

When you create a window, you specify the following:

- The screen line where the top of the window is to be located
- The number of rows in the window
- Whether a status line is associated with the window

6.1.2.1 Making a Window Current

There are three ways to make a window the current window:

- Map the window to a buffer
- Position to the window
- Adjust the size or location of the window

For more information, see the descriptions of the MAP, POSITION, and ADJUST_WINDOW built-ins in the VAXTPU Reference Section.

The screen manager makes the current window fully visible. If the current window overlaps any other windows, the overlapped portions of the other windows are not visible. A window that is partly hidden in such a fashion is said to be *partially occluded*; a window that is completely hidden is said to be *fully occluded*.

A fully occluded window is not visible on the screen. The window data structures are not modified in any way, but screen updates ignore the fully occluded window.

A partially occluded window is displayed as if it were a smaller window. For example, if a window's status line is occluded by another window, the next screen update makes the window smaller by one line. This creates space to redisplay the window's status line. The screen manager always displays the current record in the shrunken window.

Making a window the current window may cut another underlying window into two discontiguous pieces. If this occurs, only the top portion of the occluded window is displayed. The remaining lines of that window are blank either until the occluding window is removed from view or until another window is mapped to those remaining lines.

For example: window A is created from lines 1 through 24 of the screen, and window B is created from lines 5 through 10. Each window has its own status line. The buffer mapped to window A is visible in lines 1 through 3; the status line for window A is in line 4. The buffer mapped to window B is visible in lines 5 through 9; the status line for window B is in line 10. Because window B occludes window A — cutting it into two discontiguous pieces — lines 11 through 24 are blank until one of the following occurs:

- Window B is deleted (so that window A is no longer occluded)
- A new window is created in lines 11 through 24 (to display those lines of the buffer)
- Window A becomes the current window (which, in this example, would fully occlude window B)

6.1.2.2 Mapping a Window

To become visible, a window must be mapped to a buffer. Mapping a window to a buffer makes that window the current window and makes that buffer the current buffer.

You can map more than one window to a buffer. For example, you could display the text at the top of a buffer in one window and the text at the bottom of the same buffer in another window. However, you can map only one buffer to a window.

If a window is already mapped to a buffer, mapping the window to the same buffer makes that window the current window and makes that buffer the current buffer. Doing this has no other screen effects and does not alter the cursor position of the window.

For more information, see the descriptions of the MAP and CREATE_WINDOW built-ins in the VAXTPU Reference Section.

6.1.2.3 Shifting a Window

Windows are normally displayed with the first character on a line of text in the leftmost column of the window. Shifting a window causes the leftmost column of a window to display a different character on the current line.

Once you shift a window, that window displays the shifted view of any buffer to which the shifted window is mapped.

When a window is shifted, all the lines displayed in the window are updated.

For more information, see the description of the SHIFT built-in in the VAXTPU Reference Section.

6.1.2.4 Deleting a Window

When you delete a window, its screen lines are returned to any windows it was occluding. Any lines from the deleted window that did not occlude another window become blank and remain so until you map them to a different window.

If you delete the current window, VAXTPU makes another window the current window. VAXTPU tries to determine which other window, if any, was most recently the current window, and automatically makes that window current. The new current window may occlude other windows on the screen.

An update refreshes the display of any occluded windows that became unoccluded before the update.

For more information, see the description of the DELETE built-in in the VAXTPU Reference Section.

6.1.2.5 How VAXTPU Window Size Affects a Terminal Emulator

If you are using VAXTPU on a VAXstation or other machine running VWS or DECwindows and you increase or decrease the width of a window, the terminal emulator resizes itself to match the width of the widest visible window. This resizing causes a refresh operation, which clears the screen and redisplays all visible windows.

When you use VAXTPU in a VWS or DECwindows environment, you should not use the mouse to resize the terminal emulator window while you are in VAXTPU. VAXTPU does not record the fact that the terminal emulator window has been resized. As a result, VAXTPU may unexpectedly truncate text at the edges of the terminal emulator window.

You should not create a window wider or taller than the widest or tallest possible setting of the terminal. If you do, VAXTPU may unexpectedly truncate text at the edges of the window.

For more information, see the description of the ADJUST_WINDOW built-in in the VAXTPU Reference Section.

6.1.2.6 How VAXTPU Window Size Affects the Display on a Terminal

If you are using VAXTPU on a VT300-, VT200-, or VT100-series terminal, there are only two possible modes for displaying text on the screen: 80-column mode and 132-column mode. You can specify any window width between 1 and 255 using the SET (WIDTH) command. However, the new window width does not necessarily cause any visible change to the terminal display unless you change the width to 132 or to 80 columns. In these two cases, VAXTPU sends the DECCOLM escape sequence to the terminal. This sequence changes the display mode.

You should not create a window wider or taller than the widest or tallest possible setting of the terminal. If you do, VAXTPU may unexpectedly truncate text at the edges of the window.

For more information, see the description of the ADJUST_WINDOW built-in in the VAXTPU Reference Section.

6.1.2.7 How a Window Displays Insertion of Records into a Buffer

If scrolling is disabled and the screen manager finds records that have been inserted since the last update, the inserted records and all records following the inserted records are repainted over whatever was previously on the screen. The repainting stops when the window is completely repainted or the last record in the buffer has been displayed.

If scrolling is enabled, the effect of updating depends upon whether the inserted lines are followed by deleted lines, as follows:

- If the inserted records are followed by deleted records, the screen manager puts the new records in the space vacated by the deleted records.
- If there are too many new records to fit in the space vacated by the deleted records, the screen manager scrolls currently displayed records out of the window to make room for the rest of the new records.
- If there are fewer inserted records than deleted records, the screen manager scrolls records into the screen to fill in the lines vacated by the excess deleted lines.
- If an inserted record or a series of inserted records is not followed by a deleted record or series of deleted records, the screen manager scrolls the screen to make room for the new records. The screen manager tries to scroll lines off the bottom of the screen whenever possible.
- If there are not enough lines in the buffer below the bottom of the window to fill the entire window, the screen manager scrolls lines in from the top. If there are still not enough lines to fill the window, the screen manager scrolls the end-of-buffer text up from the bottom line of the window. If the end-of-buffer text has been scrolled up, all lines below the end-of-buffer text are cleared.

6.1.2.8 How a Window Displays Deletion of Records from a Buffer

The treatment of deleted records is similar to the treatment of inserted records.

Inserted records are used to replace deleted records. If there are more deleted records than inserted records, the extra deleted records are replaced using the following algorithm:

If scrolling is disabled:

- When records are deleted, the screen manager takes records from below the deleted records and paints the records into the vacated area.
- If there are not enough lines in the buffer to fill the entire window, lines below the end-of-buffer text are cleared.

If scrolling is enabled:

- The screen manager tries to minimize scrolling by using inserted lines below the deleted lines to fill in the deleted area.
- If there are no inserted lines following the area, the screen manager scrolls lines in from the bottom of the window to cover the deleted area.

- If there are not enough lines below the bottom line of the window to fill the deleted area, the screen manager scrolls lines in from above the top of the window.
- If there are still not enough lines to fill the deleted area, the screen manager moves the bottom of the buffer up and clears the screen lines below the end-of-buffer text.

If there are more inserted records than deleted records, the screen manager scrolls lines off the bottom of the window and paints the new records in the cleared area.

6.1.2.9 How a Window Displays Changes to a Record in a Buffer

When characters are inserted or deleted or when the video attributes of characters are changed, the screen manager is informed of the first changed character, the last changed character, and the nature of the change to the characters.

If the window is set to NO_TRANSLATE mode, then each time a line is modified the screen manager redisplays the line. The screen manager truncates any of the line's text that lies to the left of the window's left edge. The screen manager then sends the rest of the line to the terminal.

If the window is not set to NO_TRANSLATE, then the screen manager updates a changed line by positioning to the first changed character and repainting the rest of the characters in the line. If the change makes the line too long for the window, a diamond character appears in the rightmost column of the window to indicate that there is more text on the line.

If the line being updated has a left margin greater than 1 (that is, not at the extreme left edge of the screen), the screen manager ensures that the area left of the left margin is cleared or, if SET (PAD) is on, padded with blank spaces.

After the characters on the line are painted, if SET (PAD) is on, the screen manager appends blank spaces. Otherwise, the screen manager erases the remainder of the line if there are leftover characters on the line.

6.2 Invoking the Screen Manager

When you write VAXTPU procedures, you can prevent updates or cause immediate updates by using the UPDATE, REFRESH, or SET (SCREEN_UPDATE) built-in. The SCROLL built-in causes an immediate update.

6.2.1 Enabling Screen Updates

To suppress screen updates, or to reenable updates after they have been disabled, use the SET (SCREEN_UPDATE) built-in. When screen updates are turned off, the screen is frozen in its current state.

While screen updating is off, built-ins that normally update the screen (such as SCROLL, REFRESH, and UPDATE) have no effect or return an error status.

Turning on screen updating causes an immediate update. If a refresh was requested while screen updating was off, the screen is immediately refreshed and repainted.

Updates can be turned on or off only on a global basis. That is, you cannot prevent updating of one window while causing it in other windows.

6.2.2 Automatic Updates

When input is entered, VAXTPU queues the keystrokes for processing. As the input is processed, either by inserting characters into the buffer or by executing procedures bound to keys, the input is taken off the queue. When the queue is empty, the screen manager updates the screen to reflect the changes that have occurred.

Note that a stream of input arriving as fast as VAXTPU can process it prevents the screen manager from running. For example, if you bind a large, relatively slow procedure to an autorepeating key, a user holding down that key may see no screen updates until the key is released. This is because new input has arrived while the screen manager was handling the first keystroke. After the key is released, the screen manager updates the screen, rolling all the previous user input into one update.

Windows are updated from the top to bottom of the screen, except that multiple windows mapped to the same buffer are updated one after another. For example, given the following mapping of windows to buffers, VAXTPU updates windows in the order shown in the four-step list following:

Window A mapped to buffer 1
Window B mapped to buffer 2
Window C mapped to buffer 1
Window D mapped to buffer 3

1. Window A is updated first, because it is the top window on the screen.
2. Window C is updated next because it is also mapped to buffer 1.
3. Window B is then updated, because it is the next window in the screen's top-to-bottom order after window A.
4. Because no other window is mapped to buffer 2, the update proceeds to the next window down — window C. However, since this window has already been updated, the screen manager skips it and updates the last window, window D.

When an automatic update occurs, the screen manager performs the following operations:

1. Returns immediately if VAXTPU is running with /NODISPLAY or if screen updating is off
2. Clears the prompt window if that window contains output and is not occluded by another window
3. Clears any lines that no longer have windows mapped to them

4 Makes sure that the width of the screen (on a VAXstation) is set correctly for the width of the widest window

5 If no windows are mapped, exits without taking further action

6 If mapped windows are present and a refresh request is still pending, refreshes the screen

7 Updates each visible window (including the status line) from the top to the bottom

8 Updates the status line if there is a status line and it has changed

9 Updates the cursor position in the current window after updating all windows

6.2.3 Updating Windows

You can update a specific window by using the UPDATE built-in with the appropriate window variable as the parameter. The update occurs immediately. When updating a specific window, the screen manager performs the following operations:

1 Returns a success status if VAXTPU is running with /NODISPLAY or if screen updating is off

2 Returns the error TPU$_WINDOWNOTMAPPED if the window is not mapped to a buffer

3 Marks the cursor position as unknown if the window is not visible (repaints the window the next time it becomes visible)

4 If a window needs to be completely repainted (for example, because a new buffer is mapped to the window), determines the new first, last, and current records in the window, and repaints all lines from the top to the bottom

5 Updates the status line if there is a status line and it has changed

6 Determines the cursor position for this window

7 Updates any other windows mapped to the same buffer

8 Repositions the cursor to the active cursor position in the current window

9 Enables the timer message (if it was disabled)

If a partial update is being done:

1 The screen manager determines which record contains the window's cursor position. This record is the current record.

2 If the window being updated is the current window, and if there is a select range active, the screen manager determines whether any of the lines needs to have its video attributes updated.

3 The screen manager places the appropriate record at the top of the window. If the cursor is on a record between the window's scroll margins, the screen manager places the same record at the top of the updated window as it placed at the top of the old window. If the cursor is on a record that is not between the window's scroll margins, then the screen manager places the record containing the cursor at the top of the updated window. Usually the screen manager accomplishes this by scrolling text. However, if this would mean scrolling more than one window's worth of text, the screen manager repaints the window instead. After placing the appropriate record at the top of the window, the screen manager determines the video attributes to be applied to the beginning of that record.

4 The screen manager disables the timer message.

5 The screen manager updates each line currently on the screen, from the top to the bottom. If no records have been inserted or deleted in the buffer, the screen manager paints in any video or text modifications that have occurred.

6 If there are deleted records that were visible, the screen manager checks whether there are any newly inserted records following and paints the new records over the deleted records. If there are no newly inserted records following, the screen manager scrolls lines in to fill the vacated area.

7 If scrolling is turned off for the window, the screen manager repaints the window. If the end-of-buffer text is on the screen and there are records above the first line of the window, the screen manager scrolls lines down from above the top of the window. Otherwise, the screen manager scrolls lines up to replace the deleted records.

8 If there are newly inserted records and there are more inserted records than will fit on the screen, the screen manager repaints the window. Otherwise, the screen manager checks whether the inserted records are followed by records that were visible but are now deleted. If so, the new records are painted over the deleted records. Otherwise, the screen manager scrolls lines down to make room for the new records.

9 If scrolling is turned on for the window, the screen manager makes room for the inserted lines and paints them in. If scrolling is turned off for the window, or if the inserted records reach the bottom of the window, the screen manager repaints the rest of the lines in the window without checking for deleted records.

6.2.4 Updating the Whole Screen

To update all the windows visible on the screen, use the UPDATE (ALL) built-in. If there is a refresh request, this causes a refresh to take place. Otherwise, UPDATE (ALL) forces an automatic update, just as if all procedures have finished execution and there is no user input waiting to be processed. The screen is updated immediately in either case.

If screen updating has been turned off, UPDATE (ALL) has no effect.

6.2.5 The REFRESH Built-In

REFRESH clears the screen, reinitializes terminal settings such as autorepeat, and repaints the windows from the top to the bottom of the screen. Use REFRESH when line noise, power failure, or other events external to VAXTPU cause the screen to be disrupted.

If screen updating has been turned off, REFRESH does nothing immediately. However, the next update refreshes the screen.

6.2.6 The SCROLL Built-In

SCROLL requires that the screen be up to date. If there are modifications to the buffers or to the sizes of windows since the last update, SCROLL updates the screen before starting the scrolling operation. The scrolling operation occurs immediately after the update.

You cannot use SCROLL when screen updating is off.

Although SCROLL updates the text on the screen, it does not update changed video attributes. Thus, if you use SCROLL operations while a select range is active, the video attributes of the screen may not be correct until the next automatic update—unless you explicitly use the UPDATE or REFRESH built-in in your procedure.

6.3 Cursor Position Compared to Editing Point

Cursor position is the location of the cursor in a window. Each window has an independent cursor position—the location of the cursor when that window becomes the current window.

The cursor position must be within the bounds of the visible window. To move the cursor position, use the CURSOR_HORIZONTAL or CURSOR_VERTICAL built-in. The cursor position is not necessarily bound to text.

VAXTPU keeps the cursor position as close as possible to the *editing point*, which is the point in the buffer where text operations occur. However, the cursor position is not always exactly the same as the editing point. The editing point may be at a location in a buffer that is not visible in the current window, or the current buffer may not be mapped to a window at all. In either of these situations, text operations take place at a point different from the cursor position. In this situation, the editing point is said to be *detached*. Being *detached* is not the same as being *free*. The editing point is free when it is in a location not occupied by a character. The editing point is detached when its location is not visible on the screen. Whenever possible, keep the cursor position synchronized with the editing point so that text operations are visible.

To move the editing point, use the MOVE_HORIZONTAL, MOVE_VERTICAL, or POSITION built-in.

The editing point is *free* if it is located before the beginning of a line, after the end of a line, in the middle of a tab, or beyond the end of a buffer.

Each buffer has its own editing point, which becomes active when that buffer becomes the current buffer.

Whenever the screen is updated, the cursor position in a window moves to the editing point of the buffer mapped to that window.

To move the editing point of a buffer to the cursor position of a window, use the POSITION built-in with a window variable as the parameter. The MAP and ADJUST_WINDOW built-ins position to the window implicitly and thus also move the editing point to the cursor position.

It is possible to move the editing point without moving the cursor position and the reverse. However, to avoid confusion, the cursor position and the editing point should be synchronized when an operation manipulates the contents of a buffer. That is, both the cursor position and the editing point should point to the same place, or as close as possible. For example, using POSITION (*buffer_variable*) or POSITION (*marker_variable*) may reposition to another buffer without changing the current window. In this state, if the user adds self-inserting characters to a buffer, the cursor may not be visible in a window mapped to the buffer where the characters are inserted. Moreover, if the current buffer is not mapped to a visible window, there is no visual feedback of the input at all.

There are various ways to avoid this discrepancy between the cursor position and the editing point, depending on where a given text operation is to be carried out. If you use POSITION (*buffer_variable*) or POSITION (*marker_variable*) to implement user operations in a given buffer, either map the buffer to a visible window or position to a window to which the buffer is already mapped and then update the window. Remember that simply exiting from your procedure may allow the screen manager to update the window automatically.

If you position to a buffer or marker to perform some housekeeping operation and then want to restore the cursor position to its previous location, you should position to the current window (the window in which the visible cursor is located). This makes the buffer mapped to the current window the current buffer, and moves the editing point to the cursor position. Updating the screen at this point has no effect, because the positions are already synchronized.

6.4 Built-In Padding

The cursor position is not necessarily bound to text. The cursor position can be moved to locations where there is no underlying text, such as left of the left margin, right of the end-of-line, in the middle of a tab, or on or below the end-of-buffer text.

However, some built-ins require an accurate offset into the current line. If you use such a built-in when the cursor position points to an area where there is no text, the screen manager inserts padding records and spaces to bind the current cursor position to a text offset.

The following built-ins cause this padding effect:

APPEND_LINE	MOVE_HORIZONTAL
ATTACH	MOVE_TEXT
COPY_TEXT	MOVE_VERTICAL
CURRENT_CHARACTER	READ_FILE
CURRENT_LINE	SELECT
CURRENT_OFFSET	SELECT_RANGE
ERASE_CHARACTER	SPAWN
ERASE_LINE	SPLIT_LINE
MARK	

The insertion of self-inserting characters also causes padding if the cursor is free.

To determine whether padding will occur if you use one of the built-ins listed above, use the following call:

```
GET_INFO (window_variable, "bound");
```

If the cursor is to the left of the left margin, the margin is moved to the cursor position and spaces are inserted to fill the line from the cursor to where the text begins. If the cursor is to the left of the left margin on a blank line, the margin is moved to the cursor position and no spaces are inserted.

To find out if the cursor position is before the beginning of a line in a particular window, use the following call:

```
GET_INFO (window_variable,"before_bol");
```

If the cursor is to the right of the end-of-line, spaces are inserted from the end of the line to the cursor position. To find out if the cursor is to the right of the end of a line in a particular window, use the following call:

```
GET_INFO (window_variable,"beyond_eol");
```

If the cursor is in the middle of a tab, spaces are inserted from the tab character to the current cursor position. The tab character is not destroyed; it is simply moved to the left. To find out if the cursor is in the middle of a tab in a particular window, use the following call:

```
GET_INFO (window_variable, "middle_of_tab");
```

If the cursor is below the bottom of the buffer, blank lines are added from the end-of-buffer text to the line the cursor is on. These blank lines are inserted using the left margin set for the buffer. If necessary, the line the cursor is on is then padded, depending on whether the cursor is to the left or right of the left margin. To find out if the cursor is below the bottom of the buffer, use the following call:

```
GET_INFO (window_variable, "beyond_eol");
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX Text Processing Utility
Manual: Part I
AA–PBTMA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line

Reader's Comments

VAX Text Processing Utility
Manual: Part I
AA–PBTMA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line