

VSI OpenVMS

DEC Text Processing Utility Reference Manual

Document Number: DO–DVDTRM–01A

Publication Date: April 2024

Operating System and Version: VSI OpenVMS Alpha Version 8.4-2L1 or higher

DEC Text Processing Utility Reference Manual



Copyright © 2024 VMS Software, Inc. (VSI), Burlington, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

HPE, HPE Integrity, HPE Alpha, and HPE Proliant are trademarks or registered trademarks of Hewlett Packard Enterprise.

Preface	v
1. About VSI	v
2. Intended Audience	v
3. Document Structure	v
4. VSI Encourages Your Comments	v
5. OpenVMS Documentation	v
6. Conventions	v
Chapter 1. DECTPU Built-In Procedures Grouped According to Function	1
Chapter 2. Descriptions of the DECTPU Built-In Procedures	5
2.1. DECTPU Built-In Procedures	5
Appendix A. Sample DECwindows DECTPU Procedures	431
A.1. Creating a Mouse Pad	431
A.2. Implementing an EDT-Style APPEND Command	436
A.3. Testing and Returning a Select Range	437
A.4. Handling Callbacks from a Scroll Bar Widget	439
A.5. Reactivating a Select Range	441
A.6. Copying Selected Material from EVE to Another DECwindows Application	442
Appendix B. DECTPU Messages	445
Appendix C. DECTPU Cursor Behavior	457
C.1. Cursor Position Compared to Editing Point	457
C.2. Built-In Padding	458

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is a reference for experienced programmers who want to program in the DEC Text Processing Utility (DECTPU). Some features of DECTPU, for example, the callable interface and the built-in procedure FILE_PARSE, are intended for system programmers who have a good understanding of VSI Open VMS operating system concepts.

3. Document Structure

This manual is organized as follows:

- Chapter 1 lists DECTPU built-in features according to their function.
- Chapter 2 contains complete descriptions of the DECTPU built-in functions.
- Appendix A contains sample procedures written in DECwindows DECTPU.
- Appendix B contains DECTPU messages.
- Appendix C discusses cursor behavior in DECTPU applications.

4. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at <https://docs.vmssoftware.com>.

6. Conventions

The following conventions are used in this manual:

Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PFn	PFn indicates that you press the key labeled PFn on the numeric keypad, where n is 1, 2, 3, or 4.

<i>x</i>	A lowercase italic <i>x</i> indicates the generic use of a letter. For example, <i>xxx</i> indicates any combination of three alphabetic characters.
<i>n</i>	A lowercase italic <i>n</i> indicates the generic use of a number. For example, 19 <i>nn</i> indicates a 4-digit number in which the last 2 digits are unknown.
PF1 <i>x</i>	A sequence such as PF1 <i>x</i> indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.
{ }	In command format descriptions, braces indicate required elements; you must choose one of the options listed.
[]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
[]	In command format descriptions, vertical bars separating items inside brackets indicate that you choose one, none, or more than one of the options.
()	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
" "	Quotation marks enclose system messages that are specified in text.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: <ul style="list-style-type: none"> • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
<i>italic text</i>	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error <i>number</i>), in command lines (<i>/ PRODUCER=name</i>), and in command parameters in text (where <i>dd</i> represents the predefined code for the device type).
bold text	This text style represents the introduction of a new term or the name of an argument, an attribute, or a reason.

	In the HTML version of this document, this convention appears as <i>italic text</i> .
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
lowercase	In format descriptions, words in lowercase indicate parameters or arguments to be specified by the user.
Monospace text	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radices—binary, octal, or hexadecimal—are explicitly indicated.
mouse	The term <i>mouse</i> refers 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. MB3 indicates the right mouse button. (Users can redefine the mouse buttons.)

Chapter 1. DECTPU Built-In Procedures Grouped According to Function

This chapter groups each of the DECTPU built-in procedures in a table according to the functions that they perform so you can see at a glance which built-in is related to what task.

When you want to perform editing tasks, use the following table to help you identify which built-in procedures are related to a particular task.

Chapter 2 lists the built-in procedures alphabetically and describes them in detail.

Some entries in this manual describe language elements or keywords that are not built-in procedures. These elements and keywords are included because they are used in the same way built-ins are used. Table 1.1 lists all the DECTPU built-in procedures, grouped by the functions they perform (screen layout, moving the cursor, matching patterns, and so on).

Table 1.1. List of DECTPU Built-In Procedures by Function

Screen Layout		
ADJUST_WINDOW	SET (PAD)	SET (VIDEO)
CREATE_WINDOW	SET (PROMPT_AREA)	SET (WIDTH)
MAP	SET (SCREEN_UPDATE)	SHIFT
REFRESH	SET (SCROLLING)	UNMAP
SET (DISPLAY_VALUE)	SET (STATUS_LINE)	UPDATE
SET (HEIGHT)	SET (TEXT)	
Moving the Cursor		
CURSOR_HORIZONTAL	SET (COLUMN_MOVE_VERTICAL)	SET (DETACHED_ACTION)
CURSOR_VERTICAL	SET (CROSS_WINDOW_BOUNDS)	SET (MOVE_VERTICAL_CONTEXT)
SCROLL		
Moving the Editing Position		
MOVE_HORIZONTAL	MOVE_VERTICAL	POSITION
Manipulating Text		
APPEND_LINE	ERASE_LINE	SEARCH_QUIETLY
BEGINNING_OF	FILE_PARSE	SELECT
CHANGE_CASE	FILE_SEARCH	SELECT_RANGE
COPY_TEXT	FILL	SET (ERASE_UNMODIFIABLE)
CREATE_BUFFER	MARK	SET (MODIFIABLE)
CREATE_RANGE	MESSAGE_TEXT	SET (MODIFIED)
EDIT	MODIFY_RANGE	SPLIT_LINE
END_OF	MOVE_TEXT	TRANSLATE

ERASE	READ_FILE	WRITE_FILE
ERASE_CHARACTER	SEARCH	
Matching Patterns		
ANCHOR	MATCH	SCANL
ANY	NOTANY	SPAN
ARB	PAGE_BREAK	SPANL
LINE_BEGIN	REMAIN	UNANCHOR
LINE_END	SCAN	
Status of the Editing Context		
CURRENT_BUFFER	SET (DEBUG)	SET (NO_WRITE)
CURRENT_CHARACTER	SET (DEFAULT_DIRECTORY)	SET (OUTPUT_FILE)
CURRENT_COLUMN	SET (FACILITY_NAME)	SET (OVERSTRIKE)
CURRENT_DIRECTION	SET (FORWARD)	SET (PAD_OVERSTRUCK_TABS)
CURRENT_LINE	SET (INFORMATIONAL)	SET (PERMANENT)
CURRENT_OFFSET	SET (INSERT)	SET (RECORD_ATTRIBUTE)
CURRENT_ROW	SET (JOURNALING)	SET (RECORD_MODE)
CURRENT_WINDOW	SET (KEYSTROKE_RECOVERY)	SET (REVERSE)
DEBUG_LINE	SET (LEFT_MARGIN)	SET (RIGHT_MARGIN)
ERROR	SET (LEFT_MARGIN_ACTION)	SET (RIGHT_MARGIN_ACTION)
ERROR_LINE	SET (LINE_NUMBER)	SET (SPECIAL_ERROR_SYMBOL)
ERROR_TEXT	SET (MARGINS)	SET (SUCCESS)
GET_INFO	SET (MAX_LINES)	SET (SYSTEM)
LOCATE_MOUSE	SET (MESSAGE_ACTION_LEVEL)	SET (TAB_STOPS)
RECOVER_BUFFER	SET (MESSAGE_ACTION_TYPE)	SET (TIMER)
SET (AUTO_REPEAT)	SET (MESSAGE_FLAGS)	SET (TRACEBACK)
SET (BELL)	SET (MOUSE)	SHOW
Defining Keys		
ADD_KEY_MAP	LAST_KEY	SET (PRE_KEY_PROCEDURE)
CREATE_KEY_MAP	LOOKUP_KEY	SET (SELF_INSERT)
CREATE_KEY_MAP_LIST	REMOVE_KEY_MAP	SET (SHIFT_KEY)
DEFINE_KEY	SET (KEY_MAP_LIST)	SET (UNDEFINED_KEY)
KEY_NAME	SET (POST_KEY_PROCEDURE)	UNDEFINE_KEY
Multiple Processing		
ATTACH	SEND	SPAWN

CREATE_PROCESS	SEND_EOF	
Executing Programs		
ABORT	COMPILE	RETURN
BREAK	EXECUTE	SAVE
Specific to DECwindows		
CREATE_WIDGET	SET (DRM_HIERARCHY)	SET (MENU_POSITION)
DEFINE_WIDGET_CLASS	SET (ENABLE_RESIZE)	SET (RESIZE_ACTION)
GET_CLIPBOARD	SET (FIRST_INPUT_ACTION)	SET (SCREEN_LIMITS)
GET_DEFAULT	SET (GLOBAL_SELECT)	SET (SCROLL_BAR)
GET_GLOBAL_SELECT	SET (GLOBAL_SELECT_GRAB)	SET (SCROLL_BAR_AUTO_THUMB)
LOWER_WIDGET	SET (GLOBAL_SELECT_READ)	SET (UID)
MANAGE_WIDGET	SET (GLOBAL_SELECT_TIME)	SET (WIDGET)
RAISE_WIDGET	SET (GLOBAL_SELECT_UNGRAB)	SET (WIDGET_CALLBACK)
READ_CLIPBOARD	SET (ICON_NAME)	SET (WIDGET_CALL_DATA)
READ_GLOBAL_SELECT	SET (ICON_PIXMAP)	SET (WIDGET_CONTEXT_HELP)
REALIZE_WIDGET	SET (INPUT_FOCUS)	SET (WIDGET_RESOURCE_TYPES)
SEND_CLIENT_MESSAGE	SET (INPUT_FOCUS_GRAB)	UNMANAGE_WIDGET
SET (ACTIVE_AREA)	SET (INPUT_FOCUS_UNGRAB)	WRITE_CLIPBOARD
SET (CLIENT_MESSAGE)	SET (MAPPED_WHEN_MANAGED)	WRITE_GLOBAL_SELECT
SET (DEFAULT_FILE)		
Miscellaneous		
ASCII	INDEX	QUIT
CALL_USER	INT	READ_CHAR
CONVERT	JOURNAL_CLOSE	READ_KEY
CREATE_ARRAY	JOURNAL_OPEN	READ_LINE
DELETE	LEARN_ABORT	SET (EOB_TEXT)
EXIT	LEARN_BEGIN	SLEEP
EXPAND_NAME	LEARN_END	STR
FAO	LENGTH	SUBSTR
HELP_TEXT	MESSAGE	

Chapter 2. Descriptions of the DECTPU Built-In Procedures

This chapter describes the DECTPU built-in procedures. The discussion of each procedure is divided, as applicable, into the following parts:

- Syntax
- Parameter(s)
- Return Value(s)
- Description
- Signaled Errors (listing the warnings and errors signaled, if applicable)
- Examples

2.1. DECTPU Built-In Procedures

This section lists the DECTPU built-in procedures in alphabetical order and describes each in detail.

The descriptions of built-in procedures that return useful values show a return value in the format section. The built-in procedure descriptions that do not show a return value in the format section either return 0 or the following message:

```
TPU$_NORETURNVALUE, "Built-in does not return a value."
```

ABORT

Syntax

ABORT

Parameters

None.

Description

The ABORT procedure stops any executing procedures and causes DECTPU to wait for the next key press. ABORT returns control to DECTPU's main control loop. It causes an immediate exit from all invoked procedures.

Although ABORT behaves much like a built-in, it is actually a DECTPU language element.

ABORT is evaluated for correct syntax at com pile time. In contrast, DECTPU procedures are usually evaluated for a correct parameter count and parameter types at execution time.

Signaled Errors

ABORT is a language element and has no completion codes.

Example

The following example stops execution of the current procedure and returns to DECTPU's main loop. The error handler does not try to recover from an error.

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

ADD_KEY_MAP

Syntax

```
ADD_KEY_MAP (key-map-list-name, { "first",  
                                  "last", } key-map-name [ , ... ])
```

Parameters

key-map-list-name

A string that specifies the name of the key map list.

"first"

A string that directs DECTPU to add the key map to the beginning of the key map list. In cases where a key is defined in multiple key maps, the first definition found for that key in any of the key maps in a key map list is used.

"last"

A string that directs DECTPU to add the key map to the end of the key map list. In cases where a key is defined in multiple key maps, the first definition found for that key in any of the key maps in a key map list is used.

key-map-name

A string that specifies the name of the key map to be added to the key map list. You can specify more than one key map. Key maps are added to the key map list in the order specified. The order of a key map in a key map list determines precedence among any conflicting key definitions.

Description

The `ADD_KEY_MAP` procedure adds one or more key maps to a key map list. Key maps are added, in the order specified, to either the top or the bottom of the key map list. Key map precedence in a key map list is used to resolve any conflicts between key definitions. The key definition in a preceding key map overrides any conflicting key definitions in key maps that follow in the key map list.

See the descriptions of the `DEFINE_KEY`, `CREATE_KEY_MAP`, and `CREATE_KEY_MAP_LIST` built-in procedures for more information on key definitions, key maps, and key map lists, respectively. Also, see the description of the `REMOVE_KEY_MAP` built-in procedure for information on removing key maps from a key map list.

Signaled Errors

TPU\$_NOKEYMAP	WARNING	Third argument is not a defined key map.
TPU\$_KEYMAPNTFND	WARNING	The key map listed in the third argument is not found.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the ADD_KEY_MAP built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the ADD_KEY_MAP built-in.
TPU\$_NOKEYMAPLIST	WARNING	Attempt to access an undefined key map list.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the ADD_KEY_MAP built-in.
TPU\$_ILLREQUEST	WARNING	The position string must be either <i>"first"</i> or <i>"last"</i> .
TPU\$_BADREQUEST	WARNING	The position string must be either <i>"first"</i> or <i>"last"</i> .

Examples

1. The following example adds the default key map TPU\$KEY_MAP to the default key map list, TPU\$KEY_MAP_LIST. Usually (except in the EVE editor), TPU\$KEY_MAP is a member of the default key map list.

```
ADD_KEY_MAP ("TPU$KEY_MAP_LIST", "last", "TPU$KEY_MAP");
```

2. The following example creates a key map called HELP_KEYS and adds it to the beginning of the default key map list, TPU\$KEY_MAP_LIST. Key definitions in the new key map are invoked over definitions in the key maps already in the list.

```
help_keys := CREATE_KEY_MAP ("help_keys");
ADD_KEY_MAP ("TPU$KEY_MAP_LIST", "first", help_keys);
```

ADJUST_WINDOW

Syntax

ADJUST_WINDOW (window, integer1, integer2)

Parameters

window

The window whose size or location you want to change. The window that you specify becomes the current window, and the buffer mapped to that window becomes the current buffer.

integer1

The signed integer value that you add to the screen line number at the top of the window.

integer2

The signed integer value that you add to the screen line number at the bottom of the window.

Description

The `ADJUST_WINDOW` procedure changes the size or screen location, or both, of a window and makes the window that you specify the current window. If you want to check the visible size or location, or both, of a window before making an adjustment to it, use any of the following statements:

```
SHOW (WINDOW);
```

```
SHOW (WINDOWS);
```

```
top := GET_INFO (window, "top", VISIBLE_WINDOW);  
MESSAGE (STR (top));
```

```
bottom := GET_INFO (window, "bottom", VISIBLE_WINDOW);  
MESSAGE (STR (bottom));
```

There are screen line numbers at both the top and bottom of the visible window. Adjust the size of a visible window by changing either or both of these screen line numbers. Make these changes by adding to or subtracting from the current screen line number, not by specifying the screen line number itself.

You can enlarge a window by decreasing the screen line number at the top of the window. (Specify a negative value for *integer1*.) You can also enlarge a window by increasing the screen line number at the bottom of the window. (Specify a positive value for *integer2*.) The following example adds four lines to the current window, provided that the values fall within the screen boundaries:

```
ADJUST_WINDOW (CURRENT_WINDOW, -2, +2)
```

If you specify integers that attempt to set the screen line number beyond the screen boundaries, DECTPU issues a warning message. DECTPU then sets the window boundary at the edge (top or bottom, as appropriate) of the screen.

You can reduce a window by increasing the screen line number at the top of the window. (Specify a positive value for *integer1*.) You can also reduce a window by decreasing the screen line number at the bottom of the window. (Specify a negative value for *integer2*.) If you attempt to make the size of the window smaller than one line (two lines if the window has a status line, three lines if the window has a status line and a horizontal scroll bar), DECTPU issues an error message and no adjustment occurs. The following example reduces the current window by four lines:

```
ADJUST_WINDOW (CURRENT_WINDOW, +2, -2)
```

You can also use `ADJUST_WINDOW` to change the position of the window on the screen without changing the size of the window. The following command moves the current window two lines higher on the screen, provided that the values fall within the screen boundaries:

```
ADJUST_WINDOW (CURRENT_WINDOW, -2, -2)
```

Figure 2.1 below shows a screen layout that appears when you invoke DECTPU with EVE and a user-written command file. In this case, the user-written command file divides the screen into two windows. The top window has 15 text lines (including the "End-of-file" message) and a status line. The bottom window has five text lines and a status line. The two bottom lines of the screen are the command window and message window, each consisting of one line.

Figure 2.1. Screen Layout Before Using ADJUST_WINDOW

```

First line
Second line
Third line
Fourth line
Fifth line
Sixth line
Seventh line
Eighth line
Ninth line
Tenth line
Eleventh line
Twelfth line
Thirteenth line
Fourteenth line
[End of file]
Buffer: MAIN. | Write | Overstrike | Forward
First line
Second line
Third line
Fourth line
Fifth line
Buffer: SECOND_BUFFER. | Write | Insert | Forward

```

The user-written command file uses the variable *second_window* to identify the bottom window. Figure 2.2 shows the screen layout after you enter `ADJUST_WINDOW (second_window, -5, 0)` after the appropriate prompt from EVE. Both the top and bottom windows now contain 10 lines of text and a status line; the cursor is located in the bottom window. The command and message windows still contain one line each.

`ADJUST_WINDOW` adds (+/-) *integer1* to the "visible_top" and (+/-) *integer2* to the "visible_bottom" of a window. The mapping of the window to its buffer is not changed. The new values for the screen line numbers become the values for the original top and original bottom. See the *Guide to the DEC Text Processing Utility* for more information on window dimensions and window values.

Figure 2.2. Screen Layout After Using ADJUST_WINDOW

```

First line
Second line
Third line
Fourth line
Fifth line
Sixth line
Seventh line
Eighth line
Ninth line
Tenth line
Buffer: MAIN. | Write | Overstrike | Forward
First line
Second line
Third line
Fourth line
Fifth line
Sixth line
Seventh line
Eighth line
Ninth line
Tenth line
Buffer: SECOND_BUFFER. | Write | Insert | Forward

```

Using `ADJUST_WINDOW` on a window makes it the current window; that is, DECTPU puts the cursor in that window if the cursor was not already there, and DECTPU marks that window as current in

DECTPU's internal tracking system. DECTPU may scroll or adjust the text in the window to keep the current position visible after the adjustment occurs.

Both ADJUST_WINDOW and MAP may split or occlude other windows.

If you execute ADJUST_WINDOW within a procedure, the screen is not immediately updated to reflect the adjustment. The adjustment is made after the entire procedure is finished executing and control returns to the screen manager. If you want the screen to reflect the adjustment to the window before the entire procedure is executed, you can force the immediate update of a window by adding an UPDATE statement to the procedure. See the UPDATE built-in procedure for more information.

If you have defined a top or bottom scroll margin, and the window is adjusted so that the scroll margins no longer fit, DECTPU signals TPU\$_ADJSCROLLREG and the scroll margins shrink proportionally. For example, if you have a 10-line window, with an 8-line top scroll margin, shrinking the window to a 5-line window also reduces the top scroll margin to four lines.

Signaled Errors

TPU\$_ADJSCROLLREG	INFO	The window's scrolling region has been adjusted to fit the new window.
TPU\$_BOTLINETRUNC	INFO	Bottom line cannot exceed bottom of screen.
TPU\$_TOPLINETRUNC	INFO	Top line cannot exceed top of screen.
TPU\$_WINDNOTMAPPED	WARNING	Cannot adjust a window that is not mapped.
TPU\$_BADWINDADJUST	WARNING	Cannot adjust window to less than the minimum number of lines.
TPU\$_WINDNOTVIS	WARNING	No adjustment if window is not visible.
TPU\$_TOOFEW	ERROR	You specified less than three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Examples

1. The following example reduces the current window by removing five lines from the top of the window. If the top line of the window is screen line number 11, this statement changes the top line of the window to screen line number 16. If the bottom line of the window is less than screen line number 16, DECTPU signals an error.

```
ADJUST_WINDOW (CURRENT_WINDOW, +5, 0)
```

2. The following example removes five lines from the top of a window and puts a help window in their place:

```
PROCEDURE user_display_help
  top_of_window := GET_INFO (CURRENT_WINDOW, "VISIBLE_TOP");
  !
  ! Remove the top five lines from the current window
  ! and replace them with a help window
  !
```

```
ADJUST_WINDOW (CURRENT_WINDOW, +5, 0);
example_window := CREATE_WINDOW (top_of_window, 5, ON);
example_buffer := CREATE_BUFFER ("EXAMPLE",
                                "sys$login:template.txt");
MAP (example_window, example_buffer);
ENDPROCEDURE;
```

ANCHOR

Syntax

ANCHOR

Parameters

None.

Description

The ANCHOR procedure forces the next pattern element either to match immediately or else to fail. When SEARCH fails to find a match for a pattern, it usually tries the search again. To try again, the SEARCH built-in procedure moves the starting position one character forward or backward, depending upon the direction of the search. SEARCH continues this operation until it either finds a match for the pattern or reaches the end or beginning of the buffer or range being searched.

If ANCHOR appears as the first element of a complex pattern, the search does not move the starting position. Instead, the search examines the next (or previous) character to determine if it matches the next character or element in the complex pattern. If the pattern does not match starting in the original position, the search fails. SEARCH does not move the starting position nor try the search again.

When you use the plus sign (+) operator rather than the ampersand (&) operator to build complex patterns, ANCHOR is useful only as the first element of a complex pattern. It is legal elsewhere in a pattern but has no effect.

Although ANCHOR behaves much like a built-in, it is actually a keyword.

For more information on patterns or modes of pattern searching, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

ANCHOR is a keyword and has no completion codes.

Examples

1. The following example creates a pattern that matches the string a123 . Because ANCHOR appears as the first element of the pattern, SEARCH will find a123 only if the string appears at the starting position for the search.

```
pat1 := ANCHOR + "a123";
```

2. The following example starts at the beginning of a buffer and searches forward, removing all comments that begin in column 1. The ANCHOR keyword in this example ties the search to the

first character of a line (the current character). This prevents the search function from finding and removing exclamation points in the middle of a line (for example, in the FAO directive !AS).

```
PROCEDURE user_remove_comments
  LOCAL pat1,
         number_removed,
         end_mark;
  pat1 := ANCHOR + "!";
  number_removed := 0;
  end_mark := END_OF (CURRENT_BUFFER);

  POSITION (BEGINNING_OF (CURRENT_BUFFER));
  LOOP
    EXITIF MARK (NONE) = end_mark;
    r1 := SEARCH_QUIETLY (pat1, FORWARD);
    IF r1 <> 0
      THEN ! comment found so erase it
        ERASE_LINE;
        number_removed := number_removed + 1;
      ENDIF;
    MOVE_VERTICAL (1); ! move to the next line
  ENDLOOP;
  MESSAGE (FAO ("!ZL comment!%S removed.", number_removed));
ENDPROCEDURE;
```

ANY

Syntax

$$\text{pattern} := \text{ANY} \left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \llbracket , \text{integer1} \rrbracket$$

Parameters

buffer

An expression that evaluates to a buffer. ANY matches any of the characters in the resulting buffer.

range

An expression that evaluates to a range. ANY matches any of the characters in the resulting range.

string

An expression that evaluates to a string. ANY matches any of the characters in the resulting string.

integer1

A value that indicates how many contiguous characters ANY matches. The default value for this integer is 1.

Return Value

A pattern matching one or more characters that appear in the string, buffer, or range passed as the first parameter to ANY.

Description

The ANY procedure returns a pattern that matches one or more characters from the set specified. You use ANY to construct patterns.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	ANY must appear in the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	ANY requires at least one argument.
TPU\$_TOOMANY	ERROR	ANY accepts no more than two arguments.
TPU\$_ARGMISMATCH	ERROR	The argument you passed to the ANY built-in was of the wrong type.
TPU\$_INVPARAM	ERROR	The argument you passed to the ANY built-in was of the wrong type.
TPU\$_MINVALUE	WARNING	The argument you passed to the ANY built-in was less than the minimum accepted value.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of the ANY built-in.

Examples

1. The following example creates a pattern that matches any one of the characters *h*, *i*, *j*, *k*, and *l*:

```
pat1 := ANY ("hijkl")
```

2. The following example creates a pattern that matches any one of the characters *a*, *b*, *c*, *x*, and *y*:

```
a_buf := CREATE_BUFFER ("new buffer");
POSITION (a_buf);
COPY_TEXT ("xy");
SPLIT_LINE;
COPY_TEXT ("abc");
pat1 := ANY (a_buf);
```

3. The following example finds an ENDPROCEDURE statement that starts in column 1 and moves the editing point to the end of the statement:

```
PROCEDURE user_find_endprocedure
  LOCAL endprocedure_pattern,
        search_range;
  endprocedure_pattern := (LINE_BEGIN + "ENDPROCEDURE") +
                          (LINE_END | ANY (";! " + ASCII (9)));
  search_range := SEARCH_QUIETLY (endprocedure_pattern, FORWARD);
  IF search_range = 0
  THEN
    MESSAGE ("Endprocedure statement not found");
  ELSE
    POSITION (END_OF (search_range));
  ENDIF;
ENDPROCEDURE;
```

APPEND_LINE

Syntax

APPEND_LINE

Parameters

None.

Description

The APPEND_LINE procedure places the current line at the end of the previous line. You can use APPEND_LINE to delete line terminators.

The editing point in the line that was the current line before APPEND_LINE was executed becomes the editing point.

Using APPEND_LINE may cause DECTPU to insert padding spaces or blank lines in the buffer. APPEND_LINE causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window.

For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.
TPU\$_NOCACHE	ERROR	There is not enough memory to allocate a new cache.
TPU\$_TOOMANY	ERROR	APPEND_LINE does not accept arguments.
TPU\$_NOTMODIFIABLE	WARNING	You cannot modify an unmodifiable buffer.
TPU\$_LINETOOLONG	WARNING	DECTPU cannot append the line because the length of the resulting line would exceed DECTPU's maximum line length.

Examples

1. The following example adds the current line to the end of the previous line:

```
APPEND_LINE
```

2. The following example deletes the character to the left of the cursor. If you are at the beginning of a line, the procedure appends the current line to the end of the previous line. The procedure works correctly even if the window is shifted.

```

! The following procedure deletes the character
! to the left of the cursor. If the cursor is at the
! beginning of a line, it appends the current line
! to the end of the previous line.
!
PROCEDURE user_delete_char
  IF CURRENT_OFFSET = 0
    THEN
      APPEND_LINE;
    ELSE
      ERASE_CHARACTER (-1);
  ENDIF;
ENDPROCEDURE;

```

You can bind this procedure to the delete key with the following statement:

```
DEFINE_KEY ("user_delete_char", DEL_KEY);
```

ARB

Syntax

pattern := ARB (integer)

Parameter

integer

The number of characters in the pattern. This integer must be positive.

Return Value

A pattern that matches an arbitrary sequence of characters starting at the editing point and extending for the length you specify.

Description

The ARB procedure returns a pattern that matches an arbitrary sequence of characters starting at the editing point and extending for the length you specify. You can use ARB for wildcard matches of fixed length.

For more information on patterns, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	ARB must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	ARB requires at least one argument.
TPU\$_TOOMANY	ERROR	ARB accepts no more than one argument.
TPU\$_INVPARAM	ERROR	The argument to ARB must be an integer.
TPU\$_MINVALUE	WARNING	The argument to ARB must be positive.

Examples

1. The following example creates a pattern that matches the next five characters starting at the editing point. The characters themselves are arbitrary; it is the number of characters that is important for a pattern created with ARB.

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

2. The following example replaces a prefix of any three characters followed by an underscore (xxx_) in the current buffer with the string "user_". It does not change the current position.

```
PROCEDURE user_replace_prefix
  LOCAL cur_mode,
         here,
         pat1,
         found_range;

  pat1 := (LINE_BEGIN | NOTANY ("ABCDEFGHJKLMNOPQRSTUVWXYZ_$"))
          + ((ARB (3) + "_") @ found_range);
  here := MARK (NONE);
  cur_mode := GET_INFO (current_buffer, "mode");

  POSITION (BEGINNING_OF (CURRENT_BUFFER));
  LOOP
    found_range := 0;
    SEARCH_QUIETLY (pat1, FORWARD);
    EXITIF found_range = 0;
    ERASE (found_range);
    POSITION (END_OF (found_range));
    COPY_TEXT ("user_");
  ENDLLOOP;
  POSITION (here);
  SET (cur_mode, current_buffer);
ENDPROCEDURE;
```

ASCII

Syntax

$$\left\{ \begin{array}{l} \text{integer2} \\ \text{string2} \end{array} \right\} := \text{ASCII} \left(\left\{ \begin{array}{l} \text{integer1} \\ \text{keyword} \\ \text{string1} \end{array} \right\} \right)$$

Parameters

integer1

The decimal value of a character in the DEC Multinational Character Set.

keyword

Must be a key name. If the key name is the name of a key that produces a printing character, ASCII returns that character; otherwise it returns the character whose ASCII value is 0.

string1

The character whose ASCII value you want. If the string has a length greater than 1, the ASCII built-in returns the ASCII value of the first character in the string.

Return Value

The character with the specified ASCII value (if you specify an integer or keyword parameter).

The ASCII value of the string you specify (if you specify a string parameter).

Description

The ASCII procedure returns the ASCII value of a character or the character that has the specified ASCII value. The result of ASCII depends upon its argument. If the argument is an integer, ASCII returns a string of length 1 that represents the character of the DEC Multinational Character Set corresponding to the integer you specify. If the argument is a string, ASCII takes the first character of the string and returns the integer corresponding to the ASCII value of that character.

If the argument to ASCII is a keyword, that keyword must be a key name. The KEY_NAME built-in produces key names. In addition, there are several predefined keywords that are key names. See the *Guide to the DEC Text Processing Utility* for a list of these keywords.

If the keyword is a key name and the key produces a printing character, ASCII returns that character; otherwise, it returns the character whose ASCII value is 0.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	ASCII must be on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	ASCII requires one argument.
TPU\$_TOOMANY	ERROR	ASCII accepts only one argument.
TPU\$_ARGMISMATCH	ERROR	The parameter you passed to ASCII is of the wrong type.
TPU\$_NULLSTRING	WARNING	You passed a string of length 0 to ASCII.

Examples

1. The following example assigns a string of length 1 to the variable *my_character*. This string contains the form-feed character because that character has the ASCII value 12.

```
my_character := ASCII(12)
```

2. The following example assigns the integer value 97 to the variable *ascii_value*. The *a* is specified in quotation marks because it is a parameter of type string.

```
ascii_value := ASCII ("a");
```

3. The following example prompts you to press a key. When you do so, the procedure reads the key. If the key is associated with a printing character, ASCII tells you what character is produced. If the key is not associated with a printable character, ASCII informs you of this.

```
PROCEDURE user_test_key
```

```
LOCAL key_struck,  
      key_value;  
MESSAGE ("Press a key");  
key_struck := READ_KEY;  
key_value := ASCII (key_struck);  
  
IF key_value = ASCII (0)  
THEN  
  MESSAGE ("That is not a typing key");  
ELSE  
  MESSAGE (FAO ("That key produces the letter "!AS".", key_value));  
ENDIF;  
ENDPROCEDURE;
```

ATTACH

Syntax

```
ATTACH  [( { integer } ) ]
```

Parameters

integer

An integer that DECTPU interprets as the process identification (PID) of the process to which terminal control is to be switched. You must use decimal numbers to specify the PID to DECTPU.

string

A string that DECTPU interprets as the name of the process to which terminal control is to be switched.

Description

The ATTACH procedure enables you to switch control from your current process to another OpenVMS process that you previously created.

To use ATTACH you must have previously created a subprocess. If the process you specify is not part of the current job or does not exist, an error message is displayed. For information on creating subprocesses, see the description of SPAWN.

ATTACH suspends the current DECTPU process and switches context to the process you use as a parameter. If you do not specify a parameter for ATTACH, DECTPU switches control to the parent or owner process. A subsequent use of the DCL ATTACH command (or a logout from any process except the parent process) resumes the execution of the suspended DECTPU process.

In all cases, DECTPU first deassigns the terminal. If a DECTPU process is resumed following a SPAWN or ATTACH command, DECTPU reassigns the terminal and refreshes the screen.

If the current buffer is mapped to a visible window, the ATTACH built-in causes the screen manager to synchronize the editing point (which is a buffer location) with the cursor position (which is a window location). This may result in the insertion of padding spaces or lines into the buffer if the cursor position

is before the beginning of a line, in the middle of a tab, beyond the end of a line, or after the last line in the file.

ATTACH is not a valid built-in in DECwindows DECTPU. However, if you are running non DECwindows DECTPU in a DECwindows terminal emulator, ATTACH works as described.

Signaled Errors

TPU\$_NOPARENT	WARNING	There is no parent process to which you can attach. Your current process is the top-level process.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the ATTACH built-in.
TPU\$_SYSERROR	ERROR	Error requesting information about the process being attached to.
TPU\$_ARGMISMATCH	ERROR	Wrong type of data sent to the ATTACH built-in. Only process name strings and process IDs are allowed.
TPU\$_CREATEFAIL	WARNING	Unable to attach to the process.
TPU\$_REQUIRESTERM	ERROR	Feature requires a terminal.

Examples

1. The following example causes DECTPU to attach to the OpenVMS subprocess with the PID 97899:

```
ATTACH (97899)
```

2. The following example switches the terminal's control to the OpenVMS process JONES_2:

```
ATTACH ("JONES_2")
```

BEGINNING_OF

Syntax

```
marker := BEGINNING_OF ( { buffer } )
```

Parameters

buffer

The buffer whose beginning you want to mark.

range

The range whose beginning you want to mark.

Return Value

A marker that points to the first character position of the specified buffer or range.

Description

The `BEGINNING_OF` procedure returns a marker that points to the first position of a buffer or a range. If you use the marker returned by `BEGINNING_OF` as a parameter for the `POSITION` built-in procedure, the editing point moves to the marker.

Signaled Errors

<code>TPU\$_NEEDTOASSIGN</code>	ERROR	<code>BEGINNING_OF</code> must appear on the right-hand side of an assignment statement.
<code>TPU\$_TOOFEW</code>	ERROR	<code>BEGINNING_OF</code> requires one argument.
<code>TPU\$_TOOMANY</code>	ERROR	<code>BEGINNING_OF</code> accepts only one argument.
<code>TPU\$_ARGMISMATCH</code>	ERROR	You passed something other than a range or a buffer to <code>BEGINNING_OF</code> .

Examples

1. The following example uses two built-in procedures to move your current character position to the beginning of *my_range*. If *my_range* is in a visible buffer in which the cursor is located, the cursor position is also moved to the beginning of *my_range*.

```
POSITION (BEGINNING_OF (my_range))
```

2. The following example creates a new buffer, associates the buffer with the main window, and maps the main window to the screen. It positions to the top of the buffer, prompts you for the name of a file to include, and reads the file into the buffer.

```
PROCEDURE user_include_file
! Create scratch buffer
  b1 := CREATE_BUFFER ("Scratch Buffer");

! Map scratch buffer to main window
  MAP (main_window, b1);

! Read in file name given
  READ_FILE (READ_LINE ("File to Include:" ));

! Go to top of file
  POSITION (BEGINNING_OF (b1));
ENDPROCEDURE;
```

BREAK

Syntax

`BREAK`

Parameters

None.

Description

The `BREAK` procedure activates the debugger if DECTPU was invoked with the `/DEBUG` qualifier. If there is no debugger, `BREAK` causes the following message to be displayed in the message window:

```
Breakpoint at line xxx
```

It has no other effect. Although `BREAK` behaves much like a built-in, it is actually a DECTPU language element.

`BREAK` is evaluated for correct syntax at compile time. In contrast, DECTPU procedures are usually evaluated for a correct parameter count and parameter types at execution time.

Signaled Errors

`BREAK` is a language element and has no completion codes.

Example

The following example contains a break statement. If the statement is executed, DECTPU's debugger is activated, enabling you to debug that section of the code.

```
PROCEDURE user_not_quite_working
  .
  .
  .
  BREAK;
  .
  .
  .
ENDPROCEDURE;
```

CALL_USER

Syntax

```
string2 := CALL_USER (integer, string1)
```

Parameters

integer

The integer that is passed to the user-written program.

string1

The string that is passed to the user-written program.

Return Value

The value returned by the called program.

Description

The `CALL_USER` procedure calls a program written in another language from within DECTPU. The `CALL_USER` parameters are passed to the external routine exactly as you enter them; DECTPU does not process the parameters in any way.

The integer is passed by reference, and *string1* is passed by descriptor. *String2* is the value returned by the external program.

In addition to returning the value *string2* to `CALL_USER`, the external program returns a status code that tells whether the program executed successfully. You can trap this status code in an `ON_ERROR` statement. An even-numbered status code (low bit in `R0` clear) causes the `ON_ERROR` statement to be executed. The `ERROR` lexical element returns the status value from the program in the form of a keyword.

The `CALL_USER` parameters are input parameters for the external program you are calling. DECTPU does not process the parameters in any way but passes them to the external procedure exactly as you enter them. You must supply both parameters even if the routine you are calling does not require that information be passed to it. Enter the following null parameters to indicate that you are not passing any actual values:

```
CALL_USER (0, "")
```

For information on the DECTPU callable interface, see the *VSI OpenVMS Utility Routines Manual*.

Signaled Errors

<code>TPU\$_REQUIRESVMS</code>	<code>ERROR</code>	Feature not available on this operating system.
<code>TPU\$_BADUSERDESC</code>	<code>ERROR</code>	User-written routine incorrectly filled in the return descriptor.
<code>TPU\$_NOCALLUSER</code>	<code>ERROR</code>	Could not find a routine to invoke.
<code>TPU\$_TOOFEW</code>	<code>ERROR</code>	Too few arguments passed to <code>CALL_USER</code> .
<code>TPU\$_TOOMANY</code>	<code>ERROR</code>	Too many arguments passed to <code>CALL_USER</code> .
<code>TPU\$_NEEDTOASSIGN</code>	<code>ERROR</code>	The call to <code>CALL_USER</code> must be on the right-hand side of the assignment statement.
<code>TPU\$_INVPARAM</code>	<code>ERROR</code>	Wrong type of data sent to <code>CALL_USER</code> .
<code>TPU\$_ARGMISMATCH</code>	<code>ERROR</code>	Parameter is of the wrong data type.
<code>TPU\$_CALLUSERFAIL</code>	<code>WARNING</code>	<code>CALL_USER</code> routine failed with status <code>%X'status'</code> . The value returned by <code>ERROR</code> after this type of error will be the status value reported by this message.

Examples

1. The following example calls a program that you wrote. Before invoking DECTPU, you created a logical name, `TPU$CALLUSER`, that points to the file containing the program you want called by `CALL_USER`. DECTPU passes the first parameter (6) by reference, and the second parameter ("ABC") by descriptor. If, for example, you use an integer and a string as input values, the program processes the integer 6 and the string "ABC". If the program is designed to return a result, the result is returned in the variable `ret_value`.

```
ret_value := CALL_USER (6, "ABC")
```

2. The following example shows the steps required to use the CALL_USER built-in procedure. The routine that is called to do floating-point arithmetic is written in BASIC.

Step-by-Step Example of Using CALL_USER

1. Write a program in BASIC that does floating-point arithmetic on the values passed to it:

```
! Filename:FLOATARITH.BAS

1      sub TPU$CALLUSER ( some_integer% , input_string$ , return_string$ )

10     ! don't check some_integer% because this function only does
       ! floating-point arithmetic

20     ! parse the input string
       ! find and extract the operation
       comma_location = pos ( input_string$, ",", 1% )
       if comma_location = 0 then go to all_done
       end if

       operation$ = seg$( input_string$, 1%, comma_location - 1% )

       ! find and extract the 1st operand
       operand1_location = pos ( input_string$, ",", comma_location + 1 )
       if operand1_location = 0 then go to all_done
       end if

       operand1$ = seg$( input_string$, comma_location + 1% , &
                        operand1_location - 1 )
       ! find and extract the 2nd operand
       operand2_location = pos ( input_string$, ",", operand1_location + 1 )
       if operand2_location = 0 then
           operand2_location = len( input_string$ ) + 1
       end if

       operand2$ = seg$( input_string$, operand1_location + 1% , &
                        operand2_location - 1 )

       select operation$ ! do the operation
       case "+"
           result$ = sum$( operand1$ , operand2$ ) !
       case "-"
           result$ = dif$( operand1$, operand2$ ) !
       case "*"
           result$ = num1$( Val( operand1$ ) * Val( operand2$ ) )
       case "/"
           result$ = num1$( Val( operand1$ ) / Val( operand2$ ) )
       case else
           result$ = "unknown operation."
       end select

       return_string$ = result$

999 all_done: end sub
```

2. Compile the program with the following statement:

```
$ BASIC/LIST floatarith
```

3. Create an options file to be used by the linker when you link the BASIC program.

```
!+
! File: FLOATARITH.OPT
!
! Options file to link floatarith BASIC program with DECTPU
!
!-
floatarith.obj
!
! For OpenVMS VAX, use the following link option:
!
UNIVERSAL=TPU$CALLUSER
!
! For OpenVMS Alpha, use the following link option:
!
SYMBOL_VECTOR=(TPU$CALLUSER=PROCEDURE)
!
```

4. Link the program (using the options file) to create a shareable image.

```
$ LINK floatarith/SHARE/OPT/MAP/FULL
```

5. Define the logical name TPU\$CALLUSER to point to the executable image of the BASIC program.

```
$ DEFINE TPU$CALLUSER device:[directory]floatarith.EXE
```

6. Invoke DECTPU.

7. Write and compile the following DECTPU procedure:

```
PROCEDURE my_call_user

! test the built-in procedure call_user

LOCAL output,
input;

input := READ_LINE ("Call user >"); ! Provide a parameter
for routine
output := CALL_USER ( 0, input); ! Value this routine
returns
MESSAGE (output);
ENDPROCEDURE;
```

8. When you call the procedure my_call_user, you are prompted for parameters to pass to the BASIC routine. The order of the parameters is operator, number, number. For example, if you enter +, 3.33, 4.44 after the prompt, the result 7.77 is displayed in the message area.

CHANGE_CASE

Syntax

$$\left\{ \begin{array}{l} \text{returned_buffer} \\ \text{returned_range} \\ \text{returned_string} \end{array} \right\} := \text{CHANGE_CASE} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{keyword} \\ \text{string} \end{array} \right\}, \left\{ \begin{array}{l} \text{INVERT} \\ \text{LOWER} \\ \text{UPPER} \end{array} \right\} \right. \\ \left. \left[\begin{array}{l} \text{IN_PLACE} \\ \text{NOT_IN_PLACE} \end{array} \right] \right)$$

Parameters

buffer

The buffer in which you want DECTPU to change the case. You cannot use the NOT_IN_PLACE keyword if you specify a buffer for the first parameter.

range

The range in which you want DECTPU to change the case. You cannot use the NOT_IN_PLACE keyword if you specify a range for the first parameter.

string

The string in which you want DECTPU to change the case. If you specify IN_PLACE for the third parameter, CHANGE_CASE makes the specified change to the string specified in the first parameter. If *string* is a constant, IN_PLACE has no effect.

INVERT

A keyword that directs DECTPU to change uppercase letters to lowercase and lowercase letters to uppercase.

LOWER

A keyword that directs DECTPU to change letters to all lowercase.

UPPER

A keyword that directs DECTPU to change letters to all uppercase.

IN_PLACE

A keyword that directs DECTPU to make the indicated change in the buffer, range, or string specified. This is the default.

NOT_IN_PLACE

A keyword that directs DECTPU to leave the specified string unchanged and return a string that is the result of the specified change in case. You cannot use NOT_IN_PLACE if the first parameter is specified as a range or buffer. To use NOT_IN_PLACE, you must specify a return value for CHANGE_CASE.

Return Values

returned_buffer

A variable of type buffer that points to the buffer containing the modified text, if you specify a buffer for the first parameter. The variable *returned_buffer* points to the same buffer pointed to by the buffer variable specified as the first parameter.

returned_range

A range that contains the modified text, if you specify a range for the first parameter. The returned range spans the same text as the range specified as a parameter, but they are two separate ranges. If you subsequently change or delete one of the ranges, this has no effect on the other range.

returned_string

A string that contains the modified text, if you specify a string for the first parameter. CHANGE_CASE can return a string even if you specify IN_PLACE.

The CHANGE_CASE procedure changes the case of all alphabetic characters in a buffer, range, or string, according to the keyword that you specify. Optionally, CHANGE_CASE returns a string, range, or buffer containing the changed text.

Description

The CHANGE_CASE procedure changes the case of all alphabetic characters in a buffer, range, or string, according to the keyword that you specify. Optionally, CHANGE_CASE returns a string, range, or buffer containing the changed text.

Signaled Errors

TPU\$_TOOFEW	ERROR	CHANGE_CASE requires two parameters.
TPU\$_TOOMANY	ERROR	CHANGE_CASE accept s only two parameters.
TPU\$_ARGMISMATCH	ERROR	One of the parameters to CHANGE_CASE is of the wrong data type.
TPU\$_INVPARAM	ERROR	One of the parameters to CHANGE_CASE is of the wrong data type.
TPU\$_BADKEY	WARNING	You gave the wrong keyword to CHANGE_CASE.
TPU\$_NOTMODIFIABLE	WARNING	You cannot change the case of text in an unmodifiable buffer.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of CHANGE_CASE.

Examples

1. The following example makes all the characters in the current buffer uppercase. If you enter this statement on the command line of your interface, you see the effects immediately. If you use this statement within a procedure, you see the effect of the statement at the next screen update.

```
CHANGE_CASE (CURRENT_BUFFER, UPPER)
```

2. The following example puts the current text object in uppercase:

```
PROCEDURE user_upcase_item
  ON_ERROR
  ! In case no string is found during search
  MESSAGE ("No current item.");
  RETURN;
ENDON_ERROR;

delimiters := " " + ASCII(9);
current_item := ANCHOR & SCAN (delimiters);
item_range := SEARCH (current_item, FORWARD, NO_EXACT);
CHANGE_CASE (item_range, UPPER);
ENDPROCEDURE;
```

3. The following example inverts the case of all characters in the string pointed to by *the_string* and returns the modified string in the variable *returned_value*. It does not change *the_string* in any way.

```
returned_value := CHANGE_CASE (the_string, INVERT, NOT_IN_PLACE);
```

COMPILE

Syntax

$$[\text{program} :=] \text{COMPILE} \left(\begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right)$$

Parameters

buffer

A buffer that contains only valid DECTPU declarations and statements.

range

A range that contains only valid DECTPU declarations and statements.

string

A string that contains only valid DECTPU declarations and statements.

Return Value

The program created by compiling the declarations and statements in the string, range, or buffer. If the program fails to compile, an integer zero is returned.

Description

The COMPILE procedure converts DECTPU procedures and statements into an internal, compiled format. Valid items for compilation can be represented by a string, a range, or a buffer. COMPILE optionally returns a program.

The program that `COMPILE` optionally returns is the compiled form of valid DECTPU procedures, statements, or both. You can assign the compiled version of DECTPU code to a variable name. DECTPU statements, as well as procedure definitions, can be stored by DECTPU in the program returned by `COMPILE`. Later in your editing session, you can execute the DECTPU code that you compiled by using the program as a parameter for the `EXECUTE` built-in procedure. You can also use the program as a parameter for the `DEFINE_KEY` built-in procedure to define a key to execute the program. Then you can execute the program by pressing that key.

`COMPILE` returns a program variable only if the compilation generates executable statements. `COMPILE` does not return a program variable if you compile any of the following:

- Null strings or buffers
- Procedure definitions that do not have any executable statements following them
- Programs with syntax errors

DECTPU cannot compile a string or line of text in a buffer or range longer than 256 characters. If DECTPU encounters a longer string or line, DECTPU truncates characters after the 256th character and attempts to compile the truncated string.

If necessary, use the `SET (INFORMATIONAL, ON)` built-in procedure before compiling a procedure interactively to see the compiler messages.

To check the results of a compilation to determine whether execution is possible, use the following statement in a program:

```
x := COMPILE (my_range);
!if the program is nonzero, continue
IF x <> 0

THEN

.
.
.

ENDIF;
```

If `x = 0`, no program is generated because of compilation errors or because there are no executable statements. The statement `"IF x <> 0 THEN"` allows your program to continue as long as a program is generated.

You can also use an `ON_ERROR` statement to check the result of a compilation. This statement tells you whether the compilation completed successfully; it does not tell you whether execution is possible.

Signaled Errors

<code>TPU\$_COMPILEFAIL</code>	ERROR	Compilation aborted because of syntax errors.
<code>TPU\$_ARGMISMATCH</code>	ERROR	The data type of a parameter passed to the <code>COMPILE</code> built-in is unsupported.
<code>TPU\$_TOOFEW</code>	ERROR	Too few arguments.
<code>TPU\$_TOOMANY</code>	ERROR	Too many arguments.

Examples

1. The following example associates the `MOVE_VERTICAL (1)` function with the variable `dwn`. You can use the variable `dwn` with the `EXECUTE` built-in procedure to move the editing point down one line.

```
dwn := COMPILE ("MOVE_VERTICAL (1) ")
```

2. The following example compiles the contents of the main buffer:

```
user_program := COMPILE (main_buffer)
```

If the buffer contains executable statements, DECTPU returns a program that stores these executable commands. If the buffer contains procedure definitions, DECTPU compiles the procedures and lists them in the procedure definition table so that you could call them by either entering the name of the procedure after the appropriate prompt from the interface you are using; or calling the procedure from within other procedures.

CONVERT

Syntax

```
CONVERT ( { DECW_ROOT_WINDOW  
          SCREEN  
          window } , { CHARACTERS  
                    COORDINATES }  
        from_x_integer, from_y_integer,  
          { DECW_ROOT_WINDOW  
          SCREEN  
          window } , { CHARACTERS  
                    COORDINATES }  
        to_x_integer, to_y_integer )
```

Parameters

DECW_ROOT_WINDOW

Specifies the coordinate system to be that used by the root window of the screen on which DECTPU is running.

SCREEN

Specifies the coordinate system to be that used by the DECwindows window associated with DECTPU's top-level widget.

window

Specifies the coordinate system to be that used by the DECTPU window.

CHARACTERS

Specifies a system that measures screen distances in rows and columns, as a character-cell terminal does. In a character-cell-based system, the cell in the top row and the leftmost column has the coordinates (1,1).

COORDINATES

Specifies a DECwindows coordinate system in which coordinate units correspond to pixels. The pixel in the upper left corner has the coordinates (0, 0).

from_x_integer

from_y_integer

Integer values that represent a point in the original coordinate system and units.

to_x_integer

to_y_integer

Variables of type integer that represent a point in the specified coordinate system and units. The previous contents of the parameters are deleted when DECTPU places the resulting values in them. You must specify DECTPU variables for the parameters *to_x_integer* and *to_y_integer*. Passing a constant integer, string, or keyword value causes an error. This requirement does not apply to the parameters *from_x_integer* and *from_y_integer*.

Description

The CONVERT procedure, given the coordinates of a point in one coordinate system, returns the corresponding coordinates for the point in the coordinate system you specify. The converted coordinates are returned using the *to_x_integer* and *to_y_integer* parameters. Coordinate systems are distinguished both by units employed and where each places its origin.

Signaled Errors

TPU\$_ARGMISMATCH	ERROR	The data type of the indicated parameter is not supported by CONVERT.
TPU\$_BADDELETE	ERROR	You are attempting to modify an integer, keyword, or string constant.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_TOOFEW	ERROR	Too few arguments passed to CONVERT.
TPU\$_TOOMANY	ERROR	Too many arguments passed to CONVERT.
TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_WINDNOTVIS	WARNING	CONVERT cannot operate on an invisible window.

Example

The following example converts a point's location from the current window's coordinate system (with the origin in the upper left-hand corner of the window) to the DECTPU screen's coordinate system (with the origin in the upper left-hand corner of the DECTPU screen).

```
PROCEDURE user_convert
```

```
LOCAL source_x,
```

```
    source_y,  
    dest_x,  
    dest_y;  
  
source_x := 1;  
source_y := 1;  
dest_x := 0;  
dest_y := 0;  
  
CONVERT (CURRENT_WINDOW, COORDINATES, source_x, source_y,  
        SCREEN, COORDINATES, dest_x, dest_y);  
  
ENDPROCEDURE;
```

If the current window is not the top window, CONVERT changes the value of the y coordinate to reflect the difference in the DECTPU screen's coordinate system. For more information about the difference between a DECTPU window and the DECTPU screen, see the program development chapter in the *Guide to the DEC Text Processing Utility*.

COPY_TEXT

Syntax

$$\llbracket \text{range2} := \rrbracket \text{COPY_TEXT} \left(\begin{array}{l} \text{buffer} \\ \text{range1} \\ \text{string} \end{array} \right)$$

Parameters

buffer

The buffer containing the text that you want to copy.

range1

The range containing the text that you want to copy.

string

A string representing the text that you want to copy.

Return Value

The range where the copied text has been placed.

Description

The COPY_TEXT procedure makes a copy of the text you specify and places it in the current buffer. If the current buffer is in insert mode, the text you specify is inserted before the current position in the current buffer. If the current buffer is in overstrike mode, the text you specify replaces text starting at the current position and continuing for the length of the string, range, or buffer.

Note

You cannot add a buffer or a range to itself. If you try to add a buffer to itself, DECTPU issues an error message. If you try to insert a range into itself, part of the range is copied before DECTPU signals an error. If you try to overstrike a range into itself, DECTPU may or may not signal an error.

Using `COPY_TEXT` may cause DECTPU to insert padding spaces or blank lines in the buffer. `COPY_TEXT` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

<code>TPU\$_NOCURRENTBUF</code>	WARNING	You are not positioned in a buffer.
<code>TPU\$_NOCOPYBUF</code>	WARNING	Trying to copy a buffer to itself is not allowed.
<code>TPU\$_NOCACHE</code>	ERROR	There is not enough memory to allocate a new cache.
<code>TPU\$_OVERLAPRANGE</code>	ERROR	You tried to put the contents of a range into that same range instead of into another structure.
<code>TPU\$_TOOFEW</code>	ERROR	<code>COPY_TEXT</code> requires one argument.
<code>TPU\$_TOOMANY</code>	ERROR	<code>COPY_TEXT</code> accepts only one argument.
<code>TPU\$_ARGMISMATCH</code>	ERROR	The argument to <code>COPY_TEXT</code> must be a string, range, or buffer.
<code>TPU\$_NOTMODIFIABLE</code>	ERROR	You cannot copy text into an unmodifiable buffer.
<code>TPU\$_LINETOOLONG</code>	WARNING	The line exceeds DECTPU's maximum line length.
<code>TPU\$_TRUNCATE</code>	WARNING	Characters have been truncated because you tried to add text that would exceed the maximum line length.

Examples

1. The following example causes the string "Perseus is near Andromeda" to be placed just before the current position in the current buffer when the buffer is set to insert mode:

```
COPY_TEXT ("Perseus is near Andromeda")
```

2. The following example implements a simple `INSERT HERE` function. It assumes that there is a paste buffer and that this buffer contains the most recently deleted text. The procedure copies the text from that buffer into the current buffer.

```
PROCEDURE user_simple_insert
```



```
IF BEGINNING_OF (paste_buffer) = END_OF (paste_buffer)
THEN
  MESSAGE ("Nothing to INSERT");
ELSE
  COPY_TEXT (paste_buffer);
ENDIF;
ENDPROCEDURE;
```

CREATE_ARRAY

Syntax

```
array := CREATE_ARRAY [(integer1 [, integer2])] ]
```

Parameters

integer1

The number of integer-indexed elements to be created when the array is created. DECTPU processes elements specified by this parameter more quickly than elements created dynamically. You can add integer-indexed elements dynamically, but they are not processed as quickly as predeclared integer-indexed elements.

integer2

The first predeclared integer index of the array. The predeclared integer indexes of the array extend from this integer through to $integer2 + integer1 - 1$. This parameter defaults to 1.

Return Value

The variable that is to contain the newly created array

Description

The CREATE_ARRAY procedure creates an array. In DECTPU, an array is a one-dimensional collection of data values that you can consider or manipulate as a unit.

To create an array variable called *bat*, use the CREATE_ARRAY built-in as follows:

```
bat := CREATE_aARRAY;
```

DECTPU arrays can have a static portion, a dynamic portion, or both. A static array or portion of an array contains predeclared integer-indexed elements.

These elements are allocated contiguous memory locations to support quick processing. To create an array with a static portion, specify the number of contiguous integer-indexed elements when you create the array. You also have the option of specifying a beginning index number other than 1. For example, the following statement creates an array with 100 predeclared integer-indexed elements starting at 15:

```
bat := CREATE_ARRAY (100, 15);
```

All static elements of a newly created array are initialized to the data type unspecified.

A dynamic portion of an array contains elements indexed with expressions evaluating to any DECTPU data type except unspecified, learn, pattern, or program. Dynamic array elements are dynamically created

and deleted as needed. To create a dynamic array element, assign a value to an element of an existing array. For example, the following statement creates a dynamic element in the array *bat* indexed by the string "bar" and assigns the integer value 10 to the element:

```
bat{"bar"} := 10;
```

To create an array with both static and dynamic elements, first create the static portion of the array. Then use assignment statements to create as many dynamic elements as you wish. For example, the following code fragment creates an array stored in the variable *small_array*. The array has 15 static elements and one dynamic element. The first static element is given the value 10. The dynamic element is indexed by the string "fred" and contains the value 100.

```
small_array := CREATE_ARRAY (15);
small_array{1} := 10;
small_array{"fred"} := 100;
```

To delete a dynamic array element, assign to it the constant `TPU$K_UNSPECIFIED`, which is of type unspecified.

One array can contain elements indexed with several data types. For example, you can create an array containing elements indexed with integers, buffers, windows, markers, and strings. An array element can be of any data type. All array elements of a newly created array are of type unspecified.

If the same array has been assigned to more than one variable, DECTPU does not create multiple copies of the array. Instead, each variable points to the array that has been assigned to it. DECTPU arrays are reference counted, meaning that each array has a counter keeping track of how many variables point to it.

DECTPU arrays are autodelete data types, meaning that when no variables point to an array, the array is deleted automatically. You can also delete an array explicitly by using the `DELETE` built-in. For example, the following statement deletes the array *bat*:

```
DELETE (bat);
```

If you delete an array that still has variables pointing to it, the variables receive the data type unspecified after the deletion.

If you modify an array pointed to by more than one variable, modifications made using one variable show up when another variable references the modified element. To duplicate an array, you must write a procedure to create a new array and copy the old array's elements to the new array.

To refer to an array element, use the array variable name followed by an index expression enclosed in braces or parentheses. For example, if *bar* were a variable of type marker, the following statement would assign the integer value 10 to the element indexed by *bar*:

```
bat{bar} := 10;
```

You can perform the same operations on array elements that you can on other DECTPU variables, with one exception: you cannot make partial pattern assignments to array elements.

See the *Guide to the DEC Text Processing Utility* for additional information about arrays.

Signaled Errors

TPU\$_TOOMANY	ERROR	CREATE_ARRAY accepts no more than two arguments.
---------------	-------	--

TPU\$_NEEDTOASSIGN	ERROR	CREATE_ARRAY must appear on the right-hand side of an assignment statement.
TPU\$_INVPARAM	ERROR	The arguments to CREATE_ARRAY must be integers.
TPU\$_MINVALUE	WARNING	The first argument to CREATE_ARRAY must be 1 or greater.
TPU\$_MAXVALUE	WARNING	The first argument to CREATE_ARRAY must be no greater than 65,535.
TPU\$_GETMEM	ERROR	DECTPU could not create the array because DECTPU did not have enough memory.

Examples

1. The following example creates an array that has ten predeclared integer-indexed elements that can be processed quickly by DECTPU. It can also be indexed by any other DECTPU data type except pattern, program, learn, and unspecified.

```
array2 := CREATE_ARRAY( 10 );
```

2. The following example creates an array that can be indexed by the integers –5 through 5. It can also be indexed by any other DECTPU data type other than patterns and learn sequences.

```
array3 := CREATE_ARRAY( 11, -5 );
```

CREATE_BUFFER

Syntax

```
[[buffer2 := ] CREATE_BUFFER (string1 [,string2 [,buffer1] [,string3] )]
```

Parameters

string1

A string representing the name of the buffer that you want to create.

string2

A string representing the file specification of an input file that is read into the buffer.

buffer1

The buffer that you want to use as a template for the buffer to be created. The information copied from the template buffer includes the following:

- End-of-buffer text
- Direction (FORWARD/REVERSE)

- Text entry mode (INSERT/OVERSTRIKE)
- Margins (right and left)
- Margin action routines
- Maximum number of lines
- Write-on-exit status (NO_WRITE)
- Modifiable status
- Tab stops
- Key map list

DECTPU does not copy the following attributes of the template buffer to the new buffer:

- Buffer contents
- Marks or ranges
- Input file name
- Mapping to windows
- Cursor position
- Editing point
- Associated subprocesses
- Buffer name
- Permanent status, if that is an attribute of the template buffer
- System status, if that is an attribute of the template buffer

string3

The name of the journal file to be used with the buffer. DECTPU does not copy the journal file name from the template buffer. Instead, CREATE_BUFFER uses *string3* as the new journal file name. If you do not specify *string3*, DECTPU names the journal file by using its journal file naming algorithm. For more information on the naming algorithm, see the *Guide to the DEC Text Processing Utility*.

EVE turns on buffer-change journaling by default for each new buffer. However, the CREATE_BUFFER built-in procedure does not automatically turn on journaling. If you are layering directly on DECTPU, your application must use SET (JOURNALING) to turn journaling on.

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.

Return Value

The buffer created by `CREATE_BUFFER`.

Description

The `CREATE_BUFFER` procedure defines a new work space for editing text. You can create an empty buffer or you can associate an input file name with the buffer. `CREATE_BUFFER` optionally returns a buffer.

Although you do not have to assign the buffer that you create to a variable, you need to make a variable assignment if you want to refer to the buffer for future use. The buffer variable on the left-hand side of an assignment statement is the item that you must use when you specify a buffer as a parameter for other DECTPU built-in procedures. For example, to move to a buffer for editing, enter the buffer variable after the `POSITION` built-in procedure:

```
my_buffer_variable := CREATE_BUFFER ("my_buffer_name", "my_file_name");  
POSITION (my_buffer_variable);
```

The buffer name that you specify as the first parameter for the `CREATE_BUFFER` built-in procedure (for example, "my_buffer_name") is used by DECTPU to identify the buffer on the status line. To change the status line, use the `SET (STATUS_LINE)` built-in procedure.

If you want to skip an optional parameter and specify a subsequent optional parameter, you must use a comma as a placeholder for the skipped parameter.

You can create multiple buffers. Buffers can be empty or they can contain text. The current buffer is the buffer in which any DECTPU commands that you execute take effect (unless you specify another buffer). Only one buffer can be the current buffer. See the `CURRENT_BUFFER` built-in procedure for more information.

A buffer is visible when it is associated with a window that is mapped to the screen. A buffer can be associated with multiple windows, in which case any edits that you make to the buffer are reflected in all of the windows in which the buffer is visible. To get a list of all the buffers in your editing context, use the `SHOW (BUFFERS)` built-in procedure.

When you use the following keywords with the `SET` built-in procedure, you can establish attributes for buffers. The text describes the default for the attributes:

- `SET (EOB_TEXT, buffer, string)`—The default end-of-buffer text is [EOB].
- `SET (ERASE_UNMODIFIABLE, buffer, { ON
OFF })`—By default, unmodifiable records can be deleted from buffers by built-ins such as `ERASE_LINE`.
- `SET (FORWARD, buffer)`—The default direction is forward.
- `SET (INSERT, buffer)`—The default mode of text entry is insert.
- `SET (JOURNALING, buffer, { ON
OFF })`—By default, buffer-change journaling is turned off.
- `SET (LEFT_MARGIN, buffer, integer)`—The default left margin is 1 (that is, the left margin is set in column 1).
- `SET (LEFT_MARGIN_ACTION, buffer, program_source)`—By default, buffers do not have left margin action routines.

- SET (MARGINS, buffer, integer1, integer2)—The default left margin is 1 and the default right margin is 80.
- SET (MAX_LINES, buffer, integer)—The default maximum number of lines is 0 (in other words, this feature is turned off).
- SET (MODIFIABLE, buffer, { ON
OFF })—By default, a buffer can be modified. Using the OFF keyword makes a buffer unmodifiable.
- SET (MODIFIED, buffer, { ON
OFF })—Turn s on or turns off the bit indicating that the specified buffer has been modified.
- SET (NO_WRITE, buffer [,keyword])—By default, when you exit from DECTPU, the buffer is written if it has been modified.
- SET (OUTPUT_FILE, buffer, string)—The default output file is the input file specification with the highest existing version number for that file plus 1.
- SET (OVERSTRIKE, buffer)—The default mode of text entry is insert.
- SET (PERMANENT, buffer)—By default, the buffer can be deleted.
- SET (RECORD_ATTRIBUTE, marker, range, buffer)
- SET (REVERSE, buffer)—The default direction is forward.
- SET (RIGHT_MARGIN, buffer, integer)—The default right margin is 80.
- SET (RIGHT_MARGIN_ACTION, buffer, program_source)—By default, buffers do not have right margin action routines.
- SET (SYSTEM, buffer)—By default, the buffer is a user buffer.
- SET (TAB_STOPS, buffer, { string
integer })—The default tab stops are set every eight character positions.

See the SET built-in procedure for more information on these keywords.

Signaled Errors

TPU\$_DUPBUFNAME	WARNING	First argument to the CREATE_BUFFER built-in must be a unique string.
TPU\$_TRUNCATE	WARNING	A record was truncated to the maximum record length.
TPU\$_TOOMANY	ERROR	The CREATE_BUFFER built-in takes a maximum of two arguments.
TPU\$_TOOFEW	ERROR	The CREATE_BUFFER built-in requires at least one argument.
TPU\$_INVPARAM	ERROR	The CREATE_BUFFER built-in accepts parameters of type string or buffer only.

TPU\$_GETMEM	ERROR	DECTPU ran out of virtual memory trying to create the buffer.
TPU\$_OPENIN	ERROR	CREATE_BUFFER could not open the specified input file.
TPU\$_OPENOUT	ERROR	CREATE_BUFFER could not open the journal file.

Examples

1. The following example creates a buffer called NEW_BUFFER and stores a pointer to the buffer in the variable *nb*. Use the variable *nb* when you want to specify this buffer as a parameter for DECTPU built-in procedures. The file specification "login.com" reads the input file for NEW_BUFFER from LOGIN.COM.

```
nb := CREATE_BUFFER ("new_buffer", "login.com")
```

2. The first statement in the following example creates a buffer called DEFAULTS and stores a pointer to the buffer in the variable *default_buffer*. The second statement sets the direction of *default_buffer* to reverse. The third statement creates a buffer called BUFFER_B and stores a pointer to the buffer in the variable *b*. This statement takes default information from *default_buffer*. Buffer *b* does not receive any text, marks, or ranges from the buffer *default_buffer*.

```
default_buffer := CREATE_BUFFER ("defaults");
SET (REVERSE, default_buffer);
b := CREATE_BUFFER ("buffer_b", "", default_buffer);
```

3. The following example creates the help buffer:

```
PROCEDURE user_help_buffer
  help_buf := CREATE_BUFFER("help_buf");
  SET (EOB_TEXT, help_buf, "[End of HELP]");
  SET (NO_WRITE, help_buf);
  SET (SYSTEM, help_buf);
ENDPROCEDURE;
```

4. The following example creates a buffer named scratch. It directs DECTPU to name the associated buffer-change journal file SCRATCH_JL.JL. You must use commas as placeholders for the two unspecified optional parameters. Also, by default DECTPU puts journal files in the directory defined by the logical name TPU\$JOURNAL. TPU\$JOURNAL points to the same directory that SYS\$SCRATCH points to. You can reassign TPU\$JOURNAL to point to a different directory.

```
buf1 := CREATE_BUFFER ("Scratch",,, "Scratch_jl.jl");
```

5. The following example creates a template buffer called DEFAULTS, changes the end-of-buffer text for the template buffer, and then creates a user buffer. The user buffer is created with the same end-of-buffer text that the defaults buffer has.

```
defaults_buffer := CREATE_BUFFER ("Defaults");

SET (EOB_TEXT, defaults_buffer, "[That's all, folks!]);

user_buffer := CREATE_BUFFER ("User1.txt", "", defaults_buffer);
```

CREATE_KEY_MAP

Syntax

```
[[string2 := ] CREATE_KEY_MAP (string1)
```

Parameter

string1

A string that specifies the name of the key map you create.

Return Value

A string that is the name of the key map created.

Description

The `CREATE_KEY_MAP` procedure creates and names a key map. `CREATE_KEY_MAP` optionally returns a string that is the name of the key map created. A key map is a set of key definitions. Key maps let you manipulate key definitions as a group. Key maps and their key definitions are saved in section files. The default key map for DECTPU is `TPU$KEY_MAP`, contained in the default key map list `TPU$KEY_MAP_LIST`. See the description on key map lists in `CREATE_KEY_MAP_LIST`.

The EVE editor does not use the default key map `TPU$KEY_MAP`. In EVE, the name of a key map is not the same as the variable that contains the key map. For example, the EVE variable `EVE$X_USER_KEYS` contains the key map named `EVE$USER_KEYS`, which stores your key definitions. EVE stores all its key maps in the default key map list `TPU$KEY_MAP_LIST`. However, the default key map, `TPU$KEY_MAP`, is removed from the default key map list by the standard EVE section file.

When you create a key map, its keys are undefined. Each key map can hold definitions for all characters in the DEC Multinational Character Set and all the keypad keys and the function keys, in both their shifted and unshifted forms. Each key map has its own name (a string). This name cannot be the same as that of either another key map or a key map list.

Signaled Errors

<code>TPU\$_DUPKEYMAP</code>	WARNING	A key map with this name already exists.
<code>TPU\$_TOOFEW</code>	ERROR	Too few arguments passed to the <code>CREATE_KEY_MAP</code> built-in.
<code>TPU\$_TOOMANY</code>	ERROR	Too many arguments passed to the <code>CREATE_KEY_MAP</code> built-in.
<code>TPU\$_INVPARAM</code>	ERROR	Wrong type of data sent to the <code>CREATE_KEY_MAP</code> built-in.

Examples

The following example creates a key map and defines two keys in the key map. The name of the key map is stored in the variable `sample_key_map`.

```
PROCEDURE init_sample_key_map

sample_key_map := CREATE_KEY_MAP ("sample_key_map");
```



```
DEFINE_KEY ("EXIT", Ctrl_Z_KEY, "Exit application", sample_key_map);
DEFINE_KEY ("COPY_TEXT ('XYZZY')", Ctrl_B_KEY, "Magic Word",
  sample_key_map);

ENDPROCEDURE;
```

CREATE_KEY_MAP_LIST

Syntax

```
[[string3 := ]
CREATE_KEY_MAP_LIST (string1, string2 [,...])
```

Parameters

string1

A string that specifies the name of the key map list that you create.

string2

A string that specifies the names of the initial key maps within the key map list you create.

Return Value

A string that is the name of the key map list created.

Description

The `CREATE_KEY_MAP_LIST` procedure creates and names a key map list, and also specifies the initial key maps in the key map list it creates. `CREATE_KEY_MAP_LIST` optionally returns a string that is the name of the key map list created. A key map list is an ordered set of key maps. Key map lists let you change the procedures bound to your keys. To find the definition of a given key, DECTPU searches through the key maps in the specified or default key map list until DECTPU either finds a definition for the key or reaches the end of the last key map in the list.

DECTPU provides the default key map list `TPU$KEY_MAP_LIST`, which contains the default key map `TPU$KEY_MAP`. See the description of the `CREATE_KEY_MAP` built-in procedure for more information on key maps.

The `CREATE_KEY_MAP_LIST` built-in procedure creates a new key map list, names the key map list, and specifies the initial key maps contained in the list.

Key map lists store directions on what DECTPU is to do when you press an undefined key associated with a printable character. By default, a key map list directs DECTPU to insert undefined printable characters into the current buffer. To change the default, use the `SET (SELF_INSERT)` built-in procedure.

A newly created key map list is not bound to any buffer. To bind a key map list to a buffer, use the `SET (KEY_MAP_LIST)` built-in procedure. When you use the `POSITION` built-in to select a current buffer, the key map list that is bound to the buffer is automatically activated.

A newly created key map list has no procedure defined to be called when an undefined key is referenced. You can define such a procedure with the SET (UNDEFINED_KEY) built-in procedure. The default is to display the message "key has no definition".

Key map lists are saved in section files, along with any undefined key procedures and the SELF_INSERT settings.

Signaled Errors

TPU\$_DUPKEYMAP	WARNING	The string argument is already defined as a key map.
TPU\$_DUPKEYMAPLIST	WARNING	The string argument is already defined as a key map list.
TPU\$_NOKEYMAP	WARNING	The string argument is not a defined key map.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the CREATE_KEY_MAP_LIST built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the CREATE_KEY_MAP_LIST built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the CREATE_KEY_MAP_LIST built-in.

Examples

The following example creates two key maps and groups them into a key map list:

```
PROCEDURE init_help_key_map_list

help_user_keys := CREATE_KEY_MAP ("help_user_keys");
help_keys := CREATE_KEY_MAP ("help_keys");
help_key_list := CREATE_KEY_MAP_LIST ("help_key_list", help_user_keys,
                                     help_keys);
ENDPROCEDURE;
```

CREATE_PROCESS

Syntax

```
process := CREATE_PROCESS (buffer [,string])
```

Parameters

buffer

The buffer in which DECTPU stores output from the subprocess.

string

A string that represents the first command that you want to send to the subprocess. If you do not want to include the first command when you use the CREATE_PROCESS built-in procedure, see

the SEND built-in procedure for a description of how to send the first or subsequent commands to a subprocess.

Return Value

The process created.

Description

The CREATE_PROCESS procedure starts a subprocess and associates a buffer with it. You can optionally specify an initial command to send to the subprocess.

You can create multiple subprocesses. When you exit from DECTPU, any subprocesses you have created with CREATE_PROCESS are deleted. If you want to remove a subprocess before exiting, use the DELETE built-in procedure with the process as a parameter (DELETE (proc1)), or set the variable to integer zero, as follows:

```
proc1 := 0
```

CREATE_PROCESS creates a subprocess of a DECTPU session and all of the output from the subprocess goes into a DECTPU buffer. You cannot run a program or utility that takes over control of the screen from a process created with this built-in procedure. You can, however, use the SPAWN built-in procedure to create a subprocess that suspends your DECTPU process and places you directly at the system command prompt. You can then run programs that control the whole screen.

See the *Guide to the DEC Text Processing Utility* for a list of subprocess restrictions.

Signaled Errors

TPU\$_DUPBUFNAME	WARNING	First argument must be a unique string.
TPU\$_CREATEFAIL	WARNING	Unable to activate the subprocess.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the CREATE_PROCESS built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the CREATE_PROCESS built-in.
TPU\$_NEEDTOASSIGN	ERROR	The CREATE_PROCESS built-in call must be on the right-hand side of an assignment statement.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the CREATE_PROCESS built-in.
TPU\$_CAPTIVE	WARNING	Unable to create a subprocess in a captive account.
TPU\$_NOTMODIFIABLE	WARNING	Attempt to change unmodifiable buffer. You can write only the output of the subprocess to a modifiable buffer.
TPU\$_NOPROCESS	WARNING	No subprocess to interact with. The process was deleted between the time that it was created and when DECTPU attempted to send information to it.
TPU\$_SENDFAIL	WARNING	Unable to send data to the subprocess.
TPU\$_DELETEFAIL	WARNING	Unable to terminate the subprocess.

Examples

The following example creates a buffer to hold the output from the DCL commands executed by the OpenVMS subprocess:

```
! Create a buffer to hold the output from the DCL commands
! "SET NOON" and "DIRECTORY".

PROCEDURE user_dcl_process
  dcl_buffer := CREATE_BUFFER ("dcl_buffer");
  MAP (main_window, dcl_buffer);
  my_dcl_process := CREATE_PROCESS (dcl_buffer, "SET NOON");
  MESSAGE ("Creating DCL subprocess...");
  SEND ("DIRECTORY", my_dcl_process);
ENDPROCEDURE;
```

CREATE_RANGE

Syntax

```
range := CREATE_RANGE ( { marker1 } , { marker1 }
                       [, keyword2 ])
```

Parameters

marker1

The marker that indicates the point in the buffer where the range begins.

marker2

The marker that indicates the point in the buffer where the range ends.

keyword1

A keyword that indicates the point in the buffer where you want the range to begin or end. Table 2.1 shows the valid keywords and their meanings.

Table 2.1. CREATE_RANGE Keyword Parameters

Keyword	Meaning
LINE_BEGIN	The beginning of the current buffer's current line.
LINE_END	The end of the current buffer's current line.
BUFFER_BEGIN	Line 1, offset 0 in the current buffer. This is the first position where a character could be inserted, regardless of whether there is a character there. This is the same as the point referred to by BEGINNING_OF (CURRENT_BUFFER).
BUFFER_END	The last position in the buffer where a character could be inserted. This is the same as the point referred to by END_OF (CURRENT_BUFFER).

keyword2

The video attribute for the range: BLINK, BOLD, NONE, REVERSE, or UNDERLINE. If you omit the parameter, the default is NONE.

Return Value

The range created by CREATE_RANGE.

Description

The CREATE_RANGE procedure returns a range that includes two delimiters and all the characters between them, and sets the video attributes for displaying the characters when they are visible on the screen. A range delimiter can be a marker, the beginning or end of a line, or the beginning or end of a buffer. The beginning and ending delimiters do not have to be of the same type but must be in the same buffer.

CREATE_RANGE establishes a range that is delimited by the markers you specify. You can create multiple ranges in a buffer. When you apply video attributes to a range, you can see the range if it is in a visible buffer. A range may overlap another range.

If you clear the contents of a range with the ERASE built-in procedure, the range structure still exists. The range and its video attributes, if any, move to the next character or position beyond where the range ended before the range was erased.

To remove the range structure, use the DELETE built-in procedure or set the variable to which the range is assigned to zero (r1 := 0).

In portions of a range that either are associated with nonprintable characters or are not associated with characters at all, DECTPU does not display any of the video attributes of the range. However, if you insert new characters into portions of a range where the video attributes have not been displayed, the new characters do display the video attributes that apply to the range.

CREATE_RANGE checks whether the markers you specify as parameters are free markers. A free marker is a marker not bound to a character. For more information on free markers, see the description of the MARK built-in procedure.

If a marker defining a range is a free marker, DECTPU ties the range to the character or end-of-line nearest to the free marker to use as the range delimiter. An end-of-line is not a character but is a point to which a marker can be bound.

Signaled Errors

TPU\$_NOTSAMEBUF	WARNING	First and second marker are in different buffers.
TPU\$_TOOFEW	ERROR	CREATE_RANGE requires three parameters.
TPU\$_TOOMANY	ERROR	CREATE_RANGE accepts no more than three parameters.
TPU\$_NEEDTOASSIGN	ERROR	CREATE_RANGE must appear on the right-hand side of an assignment statement.

TPU\$_INVPARAM	ERROR	One of your arguments to CREATE_RANGE is of the wrong type.
TPU\$_BADKEY	WARNING	You specified an illegal keyword.

Examples

1. The following example creates a range starting at *start_mark* and ending at *end_mark*. When this range is visible on the screen, the characters in the range are bolded.

```
my_range := CREATE_RANGE (start_mark, end_mark, BOLD)
```

2. The following example erases the text in the current buffer, starting at the editing point and erasing text until the end of the buffer is reached:

```
PROCEDURE user_erase_to_eob

    LOCAL start_of_range,
           here_to_eob;

    start_of_range := MARK (NONE);
    here_to_eob := CREATE_RANGE (start_of_range,
                                END_OF (CURRENT_BUFFER),
                                NONE);

    ERASE (here_to_eob);
ENDPROCEDURE;
```

3. The following example creates a range starting at the first point in the buffer where a character can be inserted and ending at the point marked by *mark2*. If the range is visible on the screen, the characters in it are highlighted with the reverse video attribute.

```
the_range := CREATE_RANGE (BUFFER_BEGIN, mark2, REVERSE);
```

CREATE_WIDGET

CREATE_WIDGET — The CREATE_WIDGET built-in procedure has two variants with separate syntaxes.

Syntax

```
widget := CREATE_WIDGET (widget_class, widget_name, { parent_widget }
                        { SCREEN
                        { buffer
                        { learn_sequence
                        { program
                        { range
                        { string
                        [, closure
                        [, widget_args.. ] ] )
```

Low-Level Variant

This variant uses the Intrinsics or Motif Toolkit low-level creation routine to create and return a widget. Although it has been created, the returned widget is not managed and therefore not visible. The application must call the MANAGE_WIDGET built-in procedure to make the widget visible.

Syntax

```
widget := CREATE_WIDGET (resource_manager_name, hierarchy_id,  
                        { parent_widget }  
                        { SCREEN }  
                        [, { buffer  
                          learn_sequence  
                          program  
                          range  
                          string }  
                        [, closure ] ])
```

Hierarchy Variant

This variant creates and returns an entire hierarchy of widgets (as defined in a Motif Resource Manager database) and returns the topmost widget. All children of the returned widget are also created and managed. The topmost widget is not managed, so none of the widgets created is visible.

Parameters

widget_class

The integer returned by `DEFINE_WIDGET_CLASS` that specifies the class of widget to be created.

widget_name

A string that is the name to be given to the widget.

parent_widget

The widget that is to be the parent of the newly created widget.

SCREEN

A keyword indicating that the newly created widget is to be the child of DECTPU's main window widget.

buffer

The buffer that contains the interface callback routine. This code is executed when the widget performs a callback to DECTPU; all widgets created with a single `CREATE_WIDGET` call use the same callback code. If you do not specify this parameter, DECTPU does not execute any callback code when the widget performs a callback to DECTPU.

learn_sequence

The learn sequence that is the interface callback routine. This is executed when the widget performs a callback to DECTPU; all widgets created with a single `CREATE_WIDGET` call use the same callback code. If you do not specify this parameter, DECTPU does not execute any callback code when the widget performs a callback to DECTPU.

program

The program that is the interface callback routine. This is executed when the widget performs a callback to DECTPU; all widgets created with a single `CREATE_WIDGET` call use the same

callback code. If you do not specify this parameter, DECTPU does not execute any callback code when the widget performs a callback to DECTPU.

range

The range that contains the interface callback routine. This is executed when the widget performs a callback to DECTPU; all widgets created with a single `CREATE_WIDGET` call use the same callback code. If you do not specify this parameter, DECTPU does not execute any callback code when the widget performs a callback to DECTPU.

string

The string that contains the interface callback routine. This is executed when the widget performs a callback to DECTPU; all widgets created with a single `CREATE_WIDGET` call use the same callback code. If you do not specify this parameter, DECTPU does not execute any callback code when the widget performs a callback to DECTPU.

closure

A string or integer. DECTPU passes the value to the application when the widget performs a callback to DECTPU. For more information about using closures, see the *Guide to the DEC Text Processing Utility*.

If you do not specify this parameter, DECTPU passes the closure value (if any) given to the widget in the User Interface Language (UIL) file defining the widget. If you specify the closure value with `CREATE_WIDGET` instead of in the UIL file, all widgets created with the same `CREATE_WIDGET` call have the same closure value.

widget_args

One or more pairs of resource names and resource values. You can specify a pair in an array or as a pair of separate parameters. If you use an array, you index the array with a string that is the name of the resource you want to set.

Resource names are case sensitive. The corresponding array element contains the value you want to assign to that resource. The array can contain any number of elements. If you use a pair of separate parameters, use the following format:

```
resource_name_string, resource_value
```

Arrays and string/value pairs may be interspersed. Each array index and its corresponding element value, or each string and its corresponding value, must be valid widget arguments for the class of widget you are creating.

resource_manager_name

A case-sensitive string that is the name assigned to the widget in the UIL file defining the widget.

hierarchy_id

The hierarchy identifier returned by the `SET (UID)` built-in procedure. This identifier is passed to the Motif Resource Manager, which uses the identifier to find the resource name in the database.

Return Value

The newly created widget.

Description

The `CREATE_WIDGET` procedure creates a widget. The widget name that you specify in the User Interface Definition (UID) file must match the case of the widget name that you specify as a parameter to `CREATE_WIDGET`. If you specify case-sensitive widget names in your User Interface Language (UIL) file, you must use the same widget name case with `CREATE_WIDGET` as you used in the UIL file. If you specify case-insensitive widget names in your UIL file, the UIL compiler translates all widget names to uppercase, so in this instance you must use uppercase widget names with `CREATE_WIDGET`. Example 1 in the Examples section specifies case-insensitive widget names in the UIL file and specifies an uppercase name for the widget with the `CREATE_WIDGET` built-in procedure.

If you specify one or more callback arguments in your UIL file, specify either the routine `TPU$WIDGET_INTEGER_CALLBACK` or the routine `TPU$WIDGET_STRING_CALLBACK`. For more information about specifying callbacks, see the *Guide to the DEC Text Processing Utility*. For more information about UIL files, see the *VMS DECwindows Guide to Application Programming*. DECTPU uses the Motif Version 1.1-3 compatibility libraries and requires that UID files are produced using the 1.1-3 UIL compiler and not the version 1.2 UIL compiler provided with new releases of DECwindows. Procedures for using the 1.1-3 UIL compiler are described in the DECwindows release notes.

Signaled Errors

<code>TPU\$_BADKEY</code>	WARNING	You specified an invalid keyword as a parameter.
<code>TPU\$_UNDWIDCLA</code>	WARNING	You specified a widget class integer that is not known to DECTPU.
<code>TPU\$_INVPARAM</code>	ERROR	You specified one of the parameters with data of the wrong type.
<code>TPU\$_NEEDTOASSIGN</code>	ERROR	<code>CREATE_WIDGET</code> must return a value.
<code>TPU\$_REQUIRESDECW</code>	ERROR	You can use <code>CREATE_WIDGET</code> only if you are using DECwindows DECTPU.
<code>TPU\$_TOOFEW</code>	ERROR	Too few arguments passed to <code>CREATE_WIDGET</code> .
<code>TPU\$_TOOMANY</code>	ERROR	Too many arguments passed to <code>CREATE_WIDGET</code> .
<code>TPU\$_WIDMISMATCH</code>	ERROR	You specified a widget whose class is not supported.
<code>TPU\$_ARGMISMATCH</code>	ERROR	A widget argument was not an array or a string/value pair.
<code>TPU\$_COMPILEFAIL</code>	WARNING	Compilation of the widget interface callback routine failed due to syntax errors.
<code>TPU\$_NONAMES</code>	WARNING	A widget argument is not supported by the specified widget.

TPU\$_EXTRANEOUSARGS	ERROR	You specified one or more extraneous widget arguments.
TPU\$_BADHIERARCHY	ERROR	You specified an invalid hierarchy identifier.

Examples

1. The following example, *eve_display_example*, creates a modal dialog box widget and maps the widget to the DECTPU screen.

The example shows how to use the variant of `CREATE_WIDGET` that returns an entire widget hierarchy. To use this variant to create a widget or widget hierarchy, you must have available the compiled form of a User Interface Language (UIL) file specifying the characteristics of the widgets you want to create. VSI recommends that you use one or more UIL files and the corresponding variant of `CREATE_WIDGET` whenever possible. UIL is efficient and UIL files make it easy to translate your application into other languages.

```
PROCEDURE eve_display_example

LOCAL   example_widget,      ! Variable assigned to the created widget.
        example_widget_name, ! The name of the widget assigned
                                ! to this variable must be uppercase
                                ! if you specified case insensitive
                                ! widget names in the UIL file.

        example_hierarchy;   ! Resource Manager
                                ! hierarchy for this example.

ON_ERROR
  [OTHERWISE]: ! Traps errors.
ENDON_ERROR;

! Set the widget hierarchy. The default file spec is "SYS$LIBRARY: .UID"
example_hierarchy := SET (UID, "mynode$dua0:[smith]example");

! The DECTPU CREATE_WIDGET built-in needs the name of the widget
! defined in the UIL file.

example_widget_name := "EXAMPLE_BOX"; ! The widget EXAMPLE_BOX is
                                        ! defined in the file
                                        EXAMPLE.UIL.

! Create the widget if it has not already been created.

IF GET_INFO (example_widget, "type") <> WIDGET
THEN
  example_widget := CREATE_WIDGET (example_widget_name,
                                  example_hierarchy,
                                  SCREEN, eve$kt_callback_routine);
  ! EVE defines eve$callback_dispatch to be EVE's callback routine.
  ! You do not need to define it again if you are extending EVE.

ENDIF;

! Map "example_widget" to the screen using MANAGE_WIDGET.
```

```
MANAGE_WIDGET (example_widget);

RETURN (TRUE);

ENDPROCEDURE;
```

2. The following example shows a sample UIL file describing the modal dialog box called *example_box*. The UIL file specifies where the widget appears on the screen, what label appears on the box's OK button, and what message the widget displays.

```
module example
version = 'V00-000'

! This is a sample UIL file that creates a message box containing
! the message "Hello World".

names = case_insensitive

value
    example_ok : compound_string ("OK");
    example_message : compound_string ("Hello World");

object
    example_box : XmMessageBox {
        arguments {
            XmNdefaultPosition = true; ! puts box in center work
        area
            XmNokLabelString = example_ok;
            XmNmessageString = example_message;
        };
    };

end module;
```

For an example showing how to use the variant of `CREATE_WIDGET` that calls the Toolkit low-level creation routine, see Example A.1.

CREATE_WINDOW

Syntax

```
[[window := ]
CREATE_WINDOW (integer1, integer2,
              { ON
                OFF
                1
                0
              } )
```

Parameters

integer1

The screen line number at which the window starts.

integer2

The number of rows in the window.

ON, 1

A keyword that directs DECTPU to display a status line in the new window.

The status line occupies the last row of a window. By default, the status line is displayed in reverse video and contains the following information about the buffer that is currently mapped to the window:

- The name of the buffer that is associated with the window
- The name of the file that is associated with the buffer, if one exists

See SET (STATUS_LINE) for information on changing the video attributes of the status line, the information displayed on the status line, or both.

OFF, 0

Suppresses the display of the status line.

Return Value

The window created by CREATE_WINDOW.

Description

The CREATE_WINDOW procedure defines a screen area called a window. You must specify the screen line number at which the window starts, the length of the window, and whether the status line is to be displayed. CREATE_WINDOW optionally returns the newly created window. If you want to use the window that you create as a parameter for any other built-in procedure, then you should specify a variable into which the window is returned.

You can create multiple windows on the screen, but only one window can be the current window. The cursor is positioned in the current window. The current window and the current buffer are not necessarily the same.

To make a window visible, you must associate a buffer with the window and then map the window to the screen. The following command maps *main_window* to the screen:

```
MAP (main_window, main_buffer)
```

See the MAP built-in procedure for more information.

The following keywords used with the SET built-in procedure let you establish attributes for windows. This list shows the defaults for the attributes:

- SET (PAD, window, keyword)—By default, there is no blank padding on the right.
- SET (SCROLL_BAR)—By default, DECTPU does not create vertical and horizontal scroll bars for a window in the DECwindows environment.
- SET (SCROLL_BAR_AUTO_THUMB)—By default, DECTPU controls the slider in any scroll bars in a window.

- SET (SCROLLING, window, keyword, integer1, integer2, integer3)—The default cursor limit for scrolling at the top of the screen is the first line of the window; the default cursor limit for scrolling at the bottom of the screen is the bottom line of the window. If the terminal type you are using does not let you set scrolling regions, the window is repainted.
- SET (STATUS_LINE, window, keyword, string)—The status line may be ON or OFF according to the keyword specified for the CREATE_WINDOW built-in procedure. See the preceding description of the ON keyword for information about the default attributes of a status line.
- SET (TEXT, window, keyword)—By default, the text is set to BLANK_TABS (tabs are displayed as blank spaces).
- SET (VID EO, window, keyword)—There are no video attributes by default.
- SET (WIDTH, window, integer)—By default, the width is the same as the physical width of the terminal screen when the window is created.

See the SET built-in procedure for more information on these keywords.

Use the SHIFT built-in procedure to display text that lies to the right of the window's right edge in an unshifted window. For more information, see the description of the SHIFT built-in in this chapter.

Signaled Errors

TPU\$_TOOFEW	ERROR	The CREATE_WINDOW built-in requires exactly three parameters.
TPU\$_TOOMANY	ERROR	The CREATE_WINDOW built-in accepts exactly three parameters.
TPU\$_BADKEY	ERROR	The keyword must be either ON or OFF.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADWINDLEN	WARNING	Invalid window length.
TPU\$_BADFIRSTLINE	WARNING	Invalid first line for window.

Examples

1. The following example creates a window that starts at screen line 11 and is 10 rows long, and assigns the window to the variable *new_window* :

```
new_window := CREATE_WINDOW (11, 10, ON)
```

2. A status line is displayed as the last line of the window. To make this window visible, you must associate an existing buffer with it and map the window to the screen with the following command:

```
MAP (new_window, buffer_variable)
```

3. The following example creates a window called *new_window* that starts at screen line 1 and is 21 lines long. No status line is displayed. Tabs are displayed as special graphic characters. The buffer *new_buffer*, which is set to NO_WRITE, is associated with the window and the window is mapped to the screen.

```
PROCEDURE user_make_window
```

```

new_window := CREATE_WINDOW(1, 21, OFF);
SET (TEXT, new_window, GRAPHIC_TABS);
new_buffer := CREATE_BUFFER ("user_buffer_name");
SET (NO_WRITE, new_buffer);
MAP (new_window, new_buffer);
ENDPROCEDURE;

```

CURRENT_BUFFER

Syntax

buffer := CURRENT_BUFFER

Parameters

None.

Return Value

The buffer in which you are currently positioned.

Description

The CURRENT_BUFFER procedure returns the buffer in which you are currently positioned. The current buffer is the work space in which any DECTPU statements you execute take effect. The editing point is in the current buffer. The editing point is not necessarily the same as the cursor position.

Signaled Errors

TPU\$_TOOMANY	ERROR	CURRENT_BUFFER takes no parameters.
TPU\$_NEEDTOASSIGN	ERROR	The CURRENT_BUFFER built-in must be on the right-hand side of an assignment statement.
TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.

Examples

1. The following example stores a pointer to the current buffer in the variable *my_cur_buf*:

```
my_cur_buf := CURRENT_BUFFER
```

2. The following example reverses the direction of the current buffer:

```

PROCEDURE user_toggle_direction
  IF CURRENT_DIRECTION = FORWARD
  THEN
    SET (REVERSE, CURRENT_BUFFER);
  ELSE
    SET (FORWARD, CURRENT_BUFFER);
  ENDIF;

```

```
ENDPROCEDURE;
```

CURRENT_CHARACTER

Syntax

```
string := CURRENT_CHARACTER
```

Parameters

None.

Return Value

A string that consists of the character at the editing point in the current buffer.

Description

The `CURRENT_CHARACTER` procedure returns the character at the editing point in the current buffer at which most editing operations are carried out. Each buffer maintains its own editing point, but only the editing point in the current buffer is the active editing point. An editing point, which always refers to a character position in a buffer, is not necessarily the same as the cursor position, which always refers to a location in a window. For more information on the distinction between the editing point and the cursor position, see Appendix C.

If the editing point is at the end of a line, `CURRENT_CHARACTER` returns a null string. If the editing point is at the end of a buffer, `CURRENT_CHARACTER` returns a null string and also signals a warning.

Using `CURRENT_CHARACTER` may cause DECTPU to insert padding spaces or blank lines in the buffer. `CURRENT_CHARACTER` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

TPU\$_TOOMANY	ERROR	<code>CURRENT_CHARACTER</code> takes no parameters.
TPU\$_NEEDTOASSIGN	ERROR	The <code>CURRENT_CHARACTER</code> built-in must be on the right-hand side of
TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.
TPU\$_NOEOBSTR	WARNING	You are positioned at the EOB (end-of-buffer) mark.

Examples

1. The following example stores the string that represents the editing point in the variable `my_cur_char`:

```
my_cur_char := CURRENT_CHARACTER
```

2. The following example writes the character that is at the current character position into the message area:

```
PROCEDURE user_display_current_character
```

```
! This procedure returns the ASCII character in the editing point.
```

```
    ascii_char := CURRENT_CHARACTER;
    IF ascii_char <> ""
        THEN
            MESSAGE ("The current character is '" + ascii_char + "'");
        ELSE
            MESSAGE ("There is no current character.");
    ENDIF;
ENDPROCEDURE;
```

CURRENT_COLUMN

Syntax

```
integer := CURRENT_COLUMN
```

Parameters

None.

Return Value

An integer that is the column number of the current cursor position on the screen.

Description

The `CURRENT_COLUMN` procedure returns an integer that is the current column number of the cursor position on the screen. The column numbers range from 1 on the extreme left of the screen to the maximum value allowed for the terminal type you are using on the extreme right of the screen.

The value returned by `CURRENT_COLUMN` and the value returned by `GET_INFO (SCREEN, "current_column")` are equivalent.

When used in a procedure, `CURRENT_COLUMN` does not necessarily return the position where the cursor has been placed by other statements in the procedure. DECTPU generally does not update the screen until all statements in a procedure are executed. If you want the cursor position to reflect the actual editing location, put an `UPDATE` statement in your procedure immediately before any statements containing `CURRENT_COLUMN`, as follows:

```
UPDATE (CURRENT_WINDOW);
```

If you do not want to update a window to get the current value for `CURRENT_COLUMN`, you can use the `GET_INFO` built-in procedure (`buffer_variable, "offset_column"`). This built-in returns the column number that the current offset in the buffer would have if it were mapped to a window, and if you were to force a screen update. This built-in returns an accurate value only if both of the following conditions are true:

- You are using bound cursor movement (`MOVE_VERTICAL`, `MOVE_HORIZONTAL`) or other built-in procedures that cause cursor movement because of character movement within a buffer.
- The window is not shifted.

`GET_INFO` (`window_variable`, "current_column") does not necessarily return the column number that the cursor would occupy if you caused an explicit screen update.

If a window is shifted, `CURRENT_COLUMN` still returns the current column number of the cursor on the screen. However, the value returned by `x := GET_INFO (buffer, "offset_column")` includes the number of columns by which the window is shifted. For example, if a window is shifted to the left by eight columns, `CURRENT_COLUMN` returns the value 1, while `x := GET_INFO (buffer, "offset_column")` returns the value 9.

Signaled Errors

<code>TPU\$_TOOMANY</code>	ERROR	<code>CURRENT_COLUMN</code> takes no parameters.
<code>TPU\$_NEEDTOASSIGN</code>	ERROR	The <code>CURRENT_COLUMN</code> built-in must be on the right-hand side of an assignment statement.
<code>TPU\$_NOCURRENTBUF</code>	WARNING	You are not positioned in a buffer.

Examples

1. The following example combines three DECTPU built-in procedures. `CURRENT_COLUMN` returns the integer that is the current column position, `STR` converts the integer to a string, and `MESSAGE` writes this string to the message buffer.

```
MESSAGE (STR (CURRENT_COLUMN))
```

2. The following example splits a line at the editing point. If the editing point is row 1, column 1, the procedure causes the screen to scroll.

```
PROCEDURE user_split_line
  LOCAL old_position, new_position;

  SPLIT_LINE;
  IF (CURRENT_ROW = 1) AND (CURRENT_COLUMN = 1)
  THEN
    old_position := MARK (NONE);
    SCROLL (CURRENT_WINDOW, -1);
    new_position := MARK (NONE);
    !Make sure we scrolled before doing CURSOR_VERTICAL
    IF new_position <> old_position
    THEN
      CURSOR_VERTICAL (1);
    ENDIF;
  ENDIF;
ENDPROCEDURE;
```

CURRENT_DIRECTION

Syntax

keyword := CURRENT_DIRECTION

Parameters

None.

Return Value

A keyword (FORWARD or REVERSE) that indicates the current direction of the current buffer.

Description

The CURRENT_DIRECTION procedure returns a keyword (FORWARD or REVERSE) that indicates the current direction of the current buffer. See also the descriptions of the SET (FORWARD) and SET (REVERSE) built-in procedures. If the FORWARD keyword is returned, the current direction is toward the end of the buffer. If the REVERSE keyword is returned, the current direction is toward the beginning of the buffer.

Signaled Errors

TPU\$_TOOMANY	ERROR	CURRENT_DIRECTION takes no parameters.
TPU\$_NEEDTOASSIGN	ERROR	The CURRENT_DIRECTION built-in must be on the right-hand side of an assignment statement.
TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.

Examples

1. The following example stores in the variable *my_cur_dir* the keyword that indicates whether the current direction setting for the buffer is FORWARD or REVERSE:

```
my_cur_dir := CURRENT_DIRECTION
```

2. The following example writes to the message buffer a message indicating the current direction of character movement in the buffer:

```
PROCEDURE user_show_direction

    IF CURRENT_DIRECTION = FORWARD
    THEN
        my_message1 := MESSAGE ("Forward");
    ELSE
        my_message2 := MESSAGE ("Reverse");
    ENDIF;
ENDPROCEDURE;
```

CURRENT_LINE

Syntax

```
string := CURRENT_LINE
```

Parameters

None.

Return Value

A string that represents the current line.

Description

The `CURRENT_LINE` procedure returns a string that represents the current line. The current line is the line that contains the editing point. If you are positioned on a line that has a length of 0, `CURRENT_LINE` returns a null string. If you are positioned at the end of the buffer, `CURRENT_LINE` returns a null string and also signals a warning.

Using `CURRENT_LINE` may cause DECTPU to insert padding spaces or blank lines in the buffer. `CURRENT_LINE` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

TPU\$_TOOMANY	ERROR	<code>CURRENT_LINE</code> takes no parameters.
TPU\$_NEEDTOASSIGN	ERROR	The <code>CURRENT_LINE</code> built-in must be on the right-hand side of an assignment statement.
TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.
TPU\$_NOEOBSTR	WARNING	You are positioned at or beyond the EOB (end-of- buffer) mark.

Examples

1. The following example stores in the variable `my_cur_lin` the string that represents the current line. The current line is the line in the current buffer that contains the editing point.

```
my_cur_lin := CURRENT_LINE
```

2. The following example returns true if the current line has the format of a DSR command (starts with a period followed by an alphabetic character, a semicolon, or an exclamation point). If not, the procedure returns false. The procedure assumes that the cursor was at the beginning of the line and moves it back to the beginning of the line when done.

```
PROCEDURE user_runoff_line
  IF LENGTH (CURRENT_LINE) < 2
```

```
THEN
  user_runoff_line := 0;
ELSE
  IF CURRENT_CHARACTER <> "."
  THEN
    user_runoff_line := 0;
  ELSE
    MOVE_HORIZONTAL (1);
    IF INDEX
      ("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNopQRSTUVWXYZ!;",
      CURRENT_CHARACTER) = 0
    THEN
      user_runoff_line := 0;
    ELSE
      user_runoff_line := 1;
    ENDIF;
    MOVE_HORIZONTAL (-1);
  ENDIF;
ENDIF;
ENDPROCEDURE;
```

CURRENT_OFFSET

Syntax

integer := CURRENT_OFFSET

Parameters

None.

Return Value

An integer that is the offset of the editing point within the current line.

Description

The CURRENT_OFFSET procedure returns an integer for the offset of the editing point within the current line. The current offset is the number of positions a character is located from the first character position in the current line (offset 0). In DECTPU, the leftmost character position is offset 0, and this offset is increased by 1 for each character position (including the tab character) to the right. DECTPU numbers columns starting with the leftmost position on the screen where a character could be placed, regardless of where the margin is. This leftmost position is numbered 1.

Note

The current offset value is not the same as the position of the cursor on the screen. See the CURRENT_COLUMN built-in procedure if you want to determine where the cursor is. For example, if you have a line with a left margin of 10 and if the cursor is on the first character in that line, then CURRENT_OFFSET returns 0 while CURRENT_COLUMN returns 10.

Using CURRENT_OFFSET may cause DECTPU to insert padding spaces or blank lines in the buffer. CURRENT_OFFSET causes the screen manager to place the editing point at the cursor position if the

current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

If you are using an interface with free cursor motion, when you move beyond the end of a line `CURRENT_OFFSET` makes the current cursor position the new end-of-line.

If the current offset equals the length of the current line, you are positioned at the end of the line.

Signaled Errors

<code>TPU\$_TOOMANY</code>	ERROR	<code>CURRENT_OFFSET</code> takes no parameters.
<code>TPU\$_NEEDTOASSIGN</code>	ERROR	The <code>CURRENT_OFFSET</code> built-in must be on the right-hand side of an assignment statement.
<code>TPU\$_NOCURRENTBUF</code>	WARNING	You are not positioned in a buffer.

Examples

1. The following example stores the integer that is the offset position of the current character in the variable `my_cur_off`:

```
my_cur_off := CURRENT_OFFSET
```

2. The following example uses the `CURRENT_OFFSET` built-in procedure to determine whether the editing position is at the beginning of a line. If the position is at the beginning, the procedure appends the current line to the previous line; otherwise, it deletes the previous character. Compare this procedure with the procedure used as an example for the `APPEND_LINE` built-in procedure.

```
PROCEDURE user_delete
    IF CURRENT_OFFSET = 0
    THEN
        APPEND_LINE;
    ELSE
        ERASE_CHARACTER (-1);
    ENDIF;
ENDPROCEDURE;
```

CURRENT_ROW

Syntax

```
integer := CURRENT_ROW
```

Parameters

None.

Return Value

An integer that represents the screen line on which the cursor is located.

Description

The `CURRENT_ROW` procedure returns an integer that is the screen line on which the cursor is located. The screen lines are numbered from 1 at the top of the screen to the maximum number of lines available on the terminal. You can get the value of the current row by using the `GET_INFO (SCREEN, "current_row")` built-in procedure.

When used in a procedure, `CURRENT_ROW` does not necessarily return the position where the cursor has been placed by other statements in the procedure. The value returned by `CURRENT_ROW` may not be the current value because DECTPU generally does not update the screen until all statements in a procedure are executed. If you want the cursor position to reflect the actual editing location, put an `UPDATE` statement in your procedure immediately before any statements containing `CURRENT_ROW`, as follows:

```
UPDATE (CURRENT_WINDOW);
```

Signaled Errors

<code>TPU\$_NEEDTOASSIGN</code>	ERROR	The <code>CURRENT_ROW</code> built-in must be on the right-hand side of an assignment statement.
<code>TPU\$_TOOMANY</code>	ERROR	<code>CURRENT_ROW</code> takes no parameters.

Examples

1. The following example causes the cursor to move up the screen:

```
PROCEDURE user_go_up
  IF CURRENT_ROW = GET_INFO (CURRENT_WINDOW, "visible_top")
  THEN
    SCROLL (CURRENT_WINDOW, -1);
  ELSE
    CURSOR_VERTICAL (-1);
  ENDIF;
ENDPROCEDURE;
```

2. The following example causes the cursor to move down the screen. Because `CURSOR_VERTICAL` crosses window boundaries, you must use the `SCROLL` built-in procedure to keep the cursor motion within a single window if you are using free cursor motion. See `CURSOR_HORIZONTAL` and `CURSOR_VERTICAL` for more information.

If the movement of the cursor would take it outside the window, the preceding procedures scroll text into the window to keep the cursor visible. You can bind these procedures to a key so that the cursor motion can be accomplished with a single keystroke.

```
PROCEDURE user_go_down
  IF CURRENT_ROW = GET_INFO (CURRENT_WINDOW, "visible_bottom")
  THEN
    SCROLL (CURRENT_WINDOW, 1);
  ELSE
```

```

        CURSOR_VERTICAL (1);
    ENDIF;
ENDPROCEDURE;

```

CURRENT_WINDOW

Syntax

```
window := CURRENT_WINDOW
```

Parameters

None.

Return Value

The window in which the cursor is visible.

Description

The `CURRENT_WINDOW` procedure returns the window in which the cursor is visible. The current window is the window on which you have most recently performed one of the following operations:

- Selection by using the `POSITION` built-in
- Mapping to the screen by using the `MAP` built-in
- Adjustment by using the `ADJUST_WINDOW` built-in

The current window contains the cursor at the screen coordinates *current_row* and *current_column*. The current buffer is not necessarily associated with the current window.

Signaled Errors

TPU\$_TOOMANY	ERROR	<code>CURRENT_WINDOW</code> takes no parameters.
TPU\$_NEEDTOASSIGN	ERROR	The <code>CURRENT_WINDOW</code> built-in must be on the right-hand side of an assignment statement.
TPU\$_WINDNOTMAPPED	WARNING	No windows are mapped to the screen.

Examples

1. The following example stores the window that holds the cursor in the variable *my_cur_win*:

```
my_cur_win := CURRENT_WINDOW
```

2. The following example determines the length of the current window and then uses that value as a parameter for the `SCROLL` built-in procedure.

```
PROCEDURE user_next_screen

    LOCAL how_much_scroll;

    how_much_scroll := GET_INFO (CURRENT_WINDOW, "visible_length");

    SCROLL (CURRENT_WINDOW, how_much_scroll);
ENDPROCEDURE;
```

CURSOR_HORIZONTAL

Syntax

```
[ [ integer2 := ] CURSOR_VERTICAL (integer1)
```

Parameter

integer1

The signed plus or minus integer value that specifies the number of screen columns to move the cursor position. A positive value directs DECTPU to move the cursor to the right; a negative value directs DECTPU to move the cursor to the left. The value 0 causes DECTPU to synchronize the active editing point with the cursor position.

Return Value

An integer that represents the number of columns the cursor moved. If DECTPU cannot move the cursor as many columns as specified by *integer1*, DECTPU moves the cursor as many columns as possible. The return value may be negative. This notation is reserved for future versions of DECTPU. A negative return value does *not* denote that the cursor moved to the left. Rather, the integer shows the number of spaces that the cursor moved right or left. If the cursor did not move, *integer2* has the value 0. If the CURSOR_HORIZONTAL built-in produces an error, the value of *integer2* is indeterminate.

Description

The CURSOR_HORIZONTAL procedure moves the cursor position across the screen and optionally returns the cursor movement status. You can use CURSOR_HORIZONTAL to provide free cursor movement in a horizontal direction. Free cursor movement means that the cursor is not tied to text, but can move across all available columns in a screen line.

If you move before the beginning of a line, after the end of a line, in the middle of a tab, or beyond the end-of-file mark, other built-ins may cause padding lines or spaces to be added to the buffer.

If you use the CURSOR_HORIZONTAL built-in within a procedure, screen updating occurs in the following manner:

- When you execute a built-in that modifies the buffer or the editing point before you issue the call to CURSOR_HORIZONTAL, the screen is updated before CURSOR_HORIZONTAL is executed. This action ensures that the horizontal movement of the cursor starts at the correct character position. Otherwise, the screen manager does not update the screen until the procedure has finished executing and control is returned to the screen manager.

- `CURSOR_HORIZONTAL` does not move the cursor beyond the left or right edge of the window in which it is located. You cannot move the cursor outside the bounds of a window.
- `CURSOR_HORIZONTAL` has no effect if you use any input device other than a video terminal supported by DECTPU.

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>CURSOR_HORIZONTAL</code> requires one parameter.
<code>TPU\$_TOOMANY</code>	ERROR	<code>CURSOR_HORIZONTAL</code> accepts only one parameter.
<code>TPU\$_INVPARAM</code>	ERROR	One or more of the specified parameters have the wrong type.

Examples

1. The following example moves the cursor position one screen column to the right:

```
int_x := CURSOR_HORIZONTAL (1)
```

2. The following example provides for free cursor motion to the left. You can bind these procedures to keys (for example, the arrow keys) so that the movement can be accomplished with a single keystroke.

```
PROCEDURE user_free_cursor_left
  move_left := CURSOR_HORIZONTAL (-1);
ENDPROCEDURE;
```

CURSOR_VERTICAL

Syntax

```
[[integer2]] := CURSOR_VERTICAL (integer1)
```

Parameters

integer1

The signed integer value that specifies how many screen lines to move the cursor position. A positive value for *integer1* moves the cursor position down. A negative integer moves the cursor position up.

Return Value

An integer that represents the number of rows that the cursor moved up or down. If DECTPU could not move the cursor as many rows as specified by *integer1*, DECTPU moves the cursor as many rows as possible.

If `CROSS_WINDOW_BOUNDS` is set to `ON`, `CURSOR_VERTICAL` may position the cursor to another window. In this case, `CURSOR_VERTICAL` returns the negative of the number of rows the cursor moved. A negative return value does *not* denote that the cursor moved upward.

If the cursor did not move, *integer2* has the value 0. If the `CURSOR_VERTICAL` built-in procedure produced an error, the value of *integer2* is indeterminate.

Description

The `CURSOR_VERTICAL` procedure moves the cursor position up or down the screen and optionally returns the cursor movement status. You can use `CURSOR_VERTICAL` to provide free cursor movement in a vertical direction. Free cursor movement means that the cursor is not tied to text, but that it can move up and down to all lines on the screen that can be edited, whether or not there is text at that column in the new line.

The cursor does not move beyond the top or the bottom edges of the screen. However, `CURSOR_VERTICAL` can cross window boundaries, depending upon the current setting of the `CROSS_WINDOW_BOUNDS` flag. See `SET (CROSS_WINDOW_BOUNDS)` for information on how to set this flag. Use the `POSITION` built-in to move the cursor to a different window on the screen.

When `CROSS_WINDOW_BOUNDS` is set to `ON`, `CURSOR_VERTICAL` can move the cursor position to a new window. The new window in which the cursor is positioned becomes the current window. The column position of the cursor remains unchanged unless vertical movement would position the cursor outside the bounds of a window narrower than the previous window. In this instance, the cursor moves to the left until it is positioned within the right boundary of the narrower window.

When `CROSS_WINDOW_BOUNDS` is set to `OFF`, `CURSOR_VERTICAL` does not move the cursor outside the current window. If the `SET (SCROLLING)` built-in has been used to set scrolling margins, `CURSOR_VERTICAL` also attempts to keep the cursor within the scroll margins.

`CURSOR_VERTICAL` positions the cursor only in screen areas in which editing can occur. For example, `CURSOR_VERTICAL` does not position the cursor on the status line of a window, in the prompt area, or in an area of the screen that is not part of a window. The blank portion of a segmented window is not considered part of a window for this purpose.

If you use `CURSOR_VERTICAL` within a procedure, screen updating occurs in the following manner:

- When you execute a built-in that modifies the buffer or the current character position before you issue the call to `CURSOR_VERTICAL`, the screen is updated before `CURSOR_VERTICAL` is executed. This action ensures that the vertical movement of the cursor starts at the correct character position. Otherwise, the screen manager does not update the screen until the procedure has finished executing and control is returned to the screen manager.
- `CURSOR_VERTICAL` has no effect if you use an input device other than a video terminal supported by DECTPU.

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>CURSOR_VERTICAL</code> requires at least one parameter.
<code>TPU\$_TOOMANY</code>	ERROR	<code>CURSOR_VERTICAL</code> accepts at most one parameter.
<code>TPU\$_INVPARAM</code>	ERROR	You did not specify an integer as the parameter.

Examples

1. The following example moves the cursor position five lines toward the bottom of the screen:

```
int_y := CURSOR_VERTICAL (5)
```

- The following example provides for free cursor motion up and down the screen. These procedures can be bound to keys (for example, the arrow keys) so that you can make the movement with a single keystroke.

```
PROCEDURE user_free_cursor_down

    IF GET_INFO (CURRENT_WINDOW, "CURRENT_ROW") =
        GET_INFO (CURRENT_WINDOW, "VISIBLE_BOTTOM")
    THEN
        SCROLL (CURRENT_WINDOW, 1);
    ELSE
        right_x := CURSOR_VERTICAL (1);
    ENDIF;
ENDPROCEDURE;
```

These examples work regardless of the setting of `CROSS_WINDOW_BOUNDS` because the `SCROLL` built-in procedure keeps the cursor motion within a single window.

DEBUG_LINE

Syntax

```
integer := DEBUG_LINE
```

Parameters

None.

Return Value

An integer that represents the line number of the current breakpoint.

Description

The `DEBUG_LINE` procedure returns the line number of the current breakpoint. Use `DEBUG_LINE` when writing your own DECTPU debugger.

VSI recommends that you use the debugger provided in `SYSS$SHARE:TPU$DEBUG.TPU`.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	The <code>DEBUG_LINE</code> built-in must appear on the right-hand side of an assignment statement.
--------------------	-------	---

Examples

- In the following example, the code fragment first uses `GET_INFO` to request the line number of the breakpoint in the current procedure. If the line number is 0, meaning that the breakpoint is not in a

procedure, the code uses `DEBUG_LINE` to determine the breakpoint's line number relative to the buffer.

```
the_line := GET_INFO (DEBUG, "line_number");
IF the_line = 0
    THEN the_line := DEBUG_LINE;
ENDIF;
```

DEFINE_KEY

Syntax

$$\text{DEFINE_KEY} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{learn} \\ \text{program} \\ \text{range} \\ \text{string1} \end{array} \right\} \text{key-name} \right. \\ \left. \left[\text{,string2} \left[\text{,string3} \right] \right] \right)$$

Parameters

buffer

A buffer that contains the DECTPU statements to be associated with a key.

learn

A learn sequence that specifies the executable code associated with a key.

program

A program that contains the executable code to be associated with a key.

range

A range that contains the DECTPU statements to be associated with a key.

string1

A string that specifies the DECTPU statements to be associated with a key.

A DECTPU key name for a key or a combination of keys. See the *Guide to the DEC Text Processing Utility* for a list of the DECTPU key names for the LK0201 and LK0401 series of keyboards. You can also use the `SHOW (KEYWORDS)` built-in procedure to display all the DECTPU keywords.

See the Description section of this built-in procedure for information on keys that you cannot define.

To define a key for which there is no DECTPU key name, use the `KEY_NAME` built-in procedure to create your own key name for the key. For example, `KEY_NAME ("A", SHIFT_KEY)` creates a key name for the combination of PF1, the default shift key for DECTPU, and the keyboard character A. For more information, see the description of the `KEY_NAME` built-in procedure.

string2

An optional string associated with a key that you define. The string is treated as a comment that you can retrieve with the LOOKUP_KEY built-in procedure. You might want to use the comment if you are creating a help procedure for keys that you have defined.

string3

A key map or a key map list in which the key is to be defined. If a key map list is specified, the key is defined in the first key map in the key map list. If neither a key map nor a key map list is specified, the key is defined in the first key map in the key map list bound to the current buffer. See the descriptions of the CREATE_KEY_MAP, CREATE_KEY_MAP_LIST, and SET (KEY_MAP_LIST) built-in procedures for more information on key maps and key map lists.

Description

The DEFINE_KEY procedure associates executable DECTPU code with a key or a combination of keys. DEFINE_KEY com piles the first parameter if it is a string, buffer, or range.

If you use DEFINE_KEY to change the definition of a key that was previously defined, DECTPU does not save the previous definition.

You can define all the keys on the LK201 and LK401 keyboards and keypads with the following exceptions:

- The Compose Character key and Alt function key
- The Shift keys
- The Escape key
- The keys F1 through F5

There are some keys that you can define but that VSI strongly recommends you avoid defining. DECTPU does not signal an error when you use them as keyword parameters. However, on character-cell terminals the definitions you assign to these key combinations are not executed unless you set your terminal in special ways at the DCL level. VSI recommends that you do not use the following special terminal settings. The settings may cause unpredictable results if you do not understand all the implications of changing the default settings:

- Ctrl/C, Ctrl/O, Ctrl/X, and F6—To execute programs that you bind to these keys, you must first enter the DCL SET TERMINAL/PASTHRU command.
- Ctrl/T, Ctrl/Y—To execute programs that you bind to these keys, you must first enter the DCL SET TERMINAL/PASTHRU command, the DCL SET NOCONTROL command, or both.
- Ctrl/S, Ctrl/Q—To execute programs that you bind to these keys, you must first enter the DCL SET TERMINAL/NOTTSYNC command.

The PF1 key is the default shift key for the editor. You cannot define PF1 unless you use either the SET (SHIFT_KEY, keyword) built-in procedure or the EVE SET GOLD KEY command to define a different key as the shift key for the editor.

Whenever you extend EVE by writing a procedure that can be bound to a key, the procedure must return true or false, as needed, to indicate whether execution of the procedure completed successfully. EVE's

REPEAT command relies on this return value to determine whether to halt repetition of a command, a procedure bound to a key, or a learn sequence.

Signaled Errors

TPU\$_NOTDEFINABLE	WARNING	Second argument is not a valid reference to a key.
TPU\$_RECURLEARN	WARNING	This key definition was used as a part of a learn sequence. You cannot use it in this context.
TPU\$_NOKEYMAP	WARNING	Fourth argument is not a defined key map.
TPU\$_NOKEYMAPLIST	WARNING	Fourth argument is not a defined key map list.
TPU\$_KEYMAPNTFND	WARNING	The key map listed in the fourth argument is not found.
TPU\$_EMPTYKMLIST	WARNING	The key map list specified in the fourth argument contains no key maps.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the DEFINE_KEY built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the DEFINE_KEY built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the DEFINE_KEY built-in.
TPU\$_COMPILEFAIL	WARNING	Compilation aborted.
TPU\$_UNKKEYWORD	ERROR	An unknown keyword has been used as an argument.
TPU\$_BADKEY	ERROR	An unknown keyword has been used as an argument.
TPU\$_KEYSUPERSEDED	INFORMATIONAL	Key definition superseded.

Examples

1. The following example associates the DECTPU statement POSITION (main_window) with the key combination Ctrl/B. You must use quotation marks around the DECTPU statement.

```
DEFINE_KEY ("POSITION (main_window)", Ctrl_B_KEY)
```

2. The following example prompts you for the DECTPU statements to be bound to the key that you specify:

```
PROCEDURE user_define_key

  def := READ_LINE ("Definition: ");
  key := READ_LINE ("Press key to define.", 1);

  IF LENGTH (key) > 0
  THEN
    key := KEY_NAME (key)
```

```
ELSE
    key := LAST_KEY;
ENDIF;

DEFINE_KEY (def, key);
ENDPROCEDURE;
```

3. The following example changes the mode of text entry from insert to overstrike, or from overstrike to insert:

```
PROCEDURE user_change_mode

! Toggle mode between insert and overstrike

    IF GET_INFO (CURRENT_BUFFER, "mode") = OVERSTRIKE
    THEN
        SET (INSERT, CURRENT_BUFFER);
    ELSE
        SET (OVERSTRIKE, CURRENT_BUFFER);
    ENDIF;
ENDPROCEDURE;

!The following statement binds this procedure to the
! key combination Ctrl/A. This emulates the OpenVMS key binding
! that toggles between insert and overstrike for text entry
! in command line editing.

DEFINE_KEY ("user_change_mode", Ctrl_A_KEY);
```

DEFINE_WIDGET_CLASS

Syntax

```
integer := DEFINE_WIDGET_CLASS (class_name
    [ , creation_routine_name
    [ , creation_routine_image_name ] ])
```

Parameters

class_name

A string that is the name of the desired widget class record. This string is a universal symbol exported by the Toolkit or the widget writer.

creation_routine_name

A string that is the name of the low-level widget creation routine for this widget class.

If you do not specify this parameter, DECTPU uses the X Toolkit XtCreateWidget routine to create the widget. The routine you specify must have the same calling sequence as the Motif Toolkit widget creation routines.

You can specify a C-binding or OpenVMS binding name for this parameter, as follows:

- C-binding name

If you specify a C-binding name for this parameter, be sure not to use a dollar sign (\$) in your binding name. C-binding creation routine names are case sensitive. For example, *XmCreateScrollBar* is not identical to *xmcreatescrollbar*. To determine the correct case of the string, consult the documentation for the widget whose class you are defining.

- OpenVMS binding name

If you specify an OpenVMS binding name for this parameter, you can use the dollar sign (\$) character in the name. OpenVMS binding names are not case sensitive.

creation_routine_image_name

A string that is the name of the shareable image that contains the widget class record. If you specify a low-level creation routine in the second parameter,

DEFINE_WIDGET_CLASS also looks for the routine in the program image. If you do not specify an image, DECTPU assumes the widget is defined in the Motif image SYS\$LIBRARY:DECW\$XMLIBSHR.EXE. This parameter can specify only the name of the shareable image. If the parameter contains anything else, such as a device name, directory name, file type, or version number, DECTPU signals an error.

Return Value

An integer used by the CREATE_WIDGET built-in to identify the class of widget to be created.

Description

The DEFINE_WIDGET_CLASS procedure defines a widget class and optional creation routine for later use in creating widgets of that class.

Defining a class that is already defined returns the existing class integer. Defining a new class also defines the widget creation routine as the second parameter, if specified, or the X toolkit routine XtCreateWidget.

Signaled Errors

TPU\$_ARGMISMATCH	ERROR	The data type of the indicated parameter is not supported by DEFINE_WIDGET_CLASS.
TPU\$_NEEDTOASSIGN	ERROR	DEFINE_WIDGET_CLASS must return a value.
TPU\$_TOOFEW	ERROR	Too few arguments passed to DEFINE_WIDGET_CLASS.
TPU\$_TOOMANY	ERROR	Too many arguments passed to DEFINE_WIDGET_CLASS.
TPU\$_REQUIRESDECW	ERROR	You can use DEFINE_WIDGET_CLASS only if you are using DECwindows DECTPU.
TPU\$_SYSERROR	ERROR	Could not find class record or creation routine in shareable image.

TPU\$_INVWIDGETCLASS	ERROR	The widget class cannot be defined.
----------------------	-------	-------------------------------------

Examples

For a sample procedure using the DEFINE_WIDGET_CLASS built-in, see Example A.1.

DELETE

Syntax

```
DELETE ( {
    array
    buffer
    integer
    keyword
    learn
    marker
    pattern
    process
    program
    range
    string
    unspecified
    widget
    window
} )
```

Parameters

array

The array that you want to delete. The memory used by the array is freed for later use. If some other data structure, such as a pattern, is referenced only in the array, then that data structure is deleted when the array is deleted.

buffer

The buffer that you want to delete. Any ranges or markers that point to this buffer, any subprocess that is associated with this buffer, the memory for the buffer control structure, the pages for storing text, and the memory for ranges and markers associated with the buffer are deleted also. If the buffer is associated with a window that is mapped to the screen, the window is unmapped. Any associated buffer-change journal file is also closed and deleted.

integer

The integer that you want to delete. Integers use no internal structures or resources, so deleting a variable of type integer simply changes that variable to type unspecified.

keyword

The keyword that you want to delete. Keywords use no internal structures or resources, so deleting a variable of type keyword simply assigns to that variable the type unspecified.

learn

The learn sequence that you want to delete. The memory used by the learn sequence is freed for later use.

marker

The marker that you want to delete. The memory for the marker control structure is deleted also.

pattern

The pattern that you want to delete. The memory used by the pattern is freed for later use. If you delete a pattern that has multiple references to it, the pattern does not go away. If you delete a pattern that has no other references to it, the pattern goes away.

process

The process that you want to delete. The memory for the process control structure and the subprocess are deleted also.

program

The program that you want to delete. The memory for the program control structure and the memory for the program code are deleted also.

range

The range that you want to delete. The memory for the range control structure is deleted also. The text in a range does not belong to the range; rather, it belongs to the buffer in which it is located. A range is merely a way of manipulating sections of text within a buffer. When you delete a range, the text delimited by the range is not deleted. See the ERASE built-in procedure for a description of how to remove the text in a range.

string

The string that you want to delete. The memory used by the string is freed for later use.

unspecified

Deleting a variable of type unspecified is allowed but does nothing.

widget

The widget that you want to delete. When you use the DELETE (widget) built-in, all variables and array elements that refer to the widget are set to unspecified. If an array element is indexed by the deleted widget, the array element is deleted as well.

window

The window that you want to delete. Along with the window, the memory for the window control structure and the record history associated with the window are deleted. If you delete a window that is mapped to the screen, DECTPU unmaps the window before deleting it. The screen appears just as it does when you use the UNMAP built-in procedure.

Description

The DELETE procedure removes DECTPU structures from your editing context. When you delete a structure (for example, a range), all variables that refer to that structure are reset to unspecified. If the

deleted structure had any associated resources, these resources are returned to the editor. When a buffer is deleted, the associated journal file (if any) is closed and deleted.

Depending upon how many variables are referencing an entity, or how many other entities are associated with the entity you are deleting, processing the DELETE built-in procedure can be time consuming. DELETE cannot be terminated by a Ctrl/C.

Any variables that reference the deleted entity are set to unspecified and all other entities that are associated with the deleted entity are also deleted. Use the DELETE built-in procedure with caution.

Signaled Errors

TPU\$_TOOFEW	ERROR	DELETE requires one argument.
TPU\$_TOOMANY	ERROR	DELETE accepts only one argument.
TPU\$_BADDELETE	ERROR	You attempted to delete a constant.
TPU\$_DELETEFAIL	WARNING	DELETE could not delete the process.
TPU\$_INVBUFFERDELETE	WARNING	You cannot delete a permanent buffer.

Examples

1. The following example deletes the main buffer and any associated resources that DECTPU allocated for the main buffer. As a result of this command, the SHOW (BUFFERS) command does not list MAIN_BUFFER as one of the buffers in your editing context.

```
DELETE (main_buffer)
```

2. The following example creates a modal dialog box widget and later deletes it. For purposes of this example, the procedure *user_callback_dispatch_routine* is assumed to be a user-written procedure that handles widget callbacks. For a sample DECwindows User Interface Language (UIL) file to be used with DECTPU code creating a modal dialog box widget, see the example in the description of the CREATE_WIDGET built-in procedure.

```
PROCEDURE sample_create_and_delete

LOCAL example_widget,
      example_widget_name,
      example_hierarchy;

example_hierarchy := SET (UID, "mynode$dua0:[smith]example.uid");
example_widget_name := "EXAMPLE_BOX";
example_widget := CREATE_WIDGET (example_widget_name,
                                example_hierarchy, SCREEN,
                                "user_callback_dispatch_routine");

! .
! .
! .

DELETE (example_widget);

ENDPROCEDURE;
```

EDIT

Syntax

$$\left\{ \begin{array}{l} \text{buffer1} \\ \text{range1} \\ \text{string1} \end{array} \right\} := \text{EDIT} \left(\left\{ \begin{array}{l} \text{buffer2} \\ \text{range2} \\ \text{string2} \end{array} \right\}, \text{keyword1} [, \dots] [, \text{keyword2}] [, \text{keyword3}] \right)$$

Parameters

buffer2

The buffer in which you want DECTPU to edit text. You cannot use the NOT_IN_PLACE keyword if you specify a buffer for the first parameter.

range2

The range in which you want DECTPU to edit text. You cannot use the NOT_IN_PLACE keyword if you specify a range for the first parameter.

string2

The string that you want to modify. If you specify a return value, the returned string consists of the string you specify for the first parameter, modified in the way you specify in the second and subsequent parameters. If you specify IN_PLACE for the third parameter, EDIT makes the specified change to the string specified in the first parameter. If *string2* is a constant, IN_PLACE has no effect.

keyword1

A keyword specifying the editing operation that you want to perform on the string. The valid keywords and their meaning are as follows.

Keyword	Meaning
COLLAPSE	Removes all spaces and tabs.
COMPRESS	Replaces multiple spaces and tabs with a single space.
TRIM	Removes leading and trailing spaces and tabs.
TRIM_LEADING	Removes leading spaces and tabs.
TRIM_TRAILING	Removes trailing spaces and tabs.
LOWER	Converts all uppercase characters to lowercase.
UPPER	Converts all lowercase characters to uppercase.
INVERT	Changes the current case of the specified characters; uppercase characters become lowercase and lowercase characters become uppercase.

keyword2

A keyword specifying whether DECTPU quote characters are used as quote characters or as regular text. The valid keywords are ON, OFF, 1, or 0. The integer 1 is equivalent to ON. The integer 0 is equivalent to OFF. The default is ON or 1.

keyword3

A keyword indicating where DECTPU is to make the indicated change. The valid keywords and their meanings are as follows:

Keyword	Meaning
IN_PLACE	Makes the indicated change in place. This is the default.
NOT_IN_PLACE	Leaves the specified string unchanged and returns a string that is the result of the specified editing. You cannot use NOT_IN_PLACE if the first parameter is specified as a range or buffer. To use NOT_IN_PLACE, you must specify a return value for EDIT.

This keyword is ignored if *string2* is a string constant. EDIT never edits string constants in place. It does return the edited string.

Return Values**buffer1**

A variable of type buffer pointing to the buffer containing the modified text, if you specify a buffer for the first parameter. The variable *returned_buffer* points to the same buffer pointed to by the buffer variable specified as the first parameter.

range1

A range containing the modified text, if you specify a range for the first parameter. The returned range spans the same text as the range specified as a parameter, but they are two separate ranges. If you subsequently change or delete one of the ranges, this has no effect on the other range.

string1

A string containing the modified text, when you specify a string for the first parameter. EDIT can return a string even if you specify IN_PLACE.

Description

The EDIT procedure modifies a string according to the keywords you specify. EDIT is similar (although not identical) to the DCL lexical function F\$EDIT. DECTPU modifies the first parameter of the EDIT built-in in place. EDIT does not modify a literal string.

By default, EDIT does not modify quoted text that occurs within a string. For example, the following code does not change the case of WELL:

```
string_to_change := 'HE SANG "WELL"'; edit (string_to_change, LOWER);
```

The variable *string_to_change* has the value *he sang "WELL"*.

If you specify more than one of the TRIM keywords (TRIM, TRIM_LEADING, TRIM_TRAILING), all of the TRIM operations you specify are performed.

If you specify more than one of the case conversion keywords (UPPER, LOWER, INVERT), the last keyword that you specify determines how the characters in the string are modified.

If you specify both of the quote recognition keywords (ON, OFF), the last keyword you specify determines whether EDIT modifies quoted text.

If you specify no keywords, EDIT does nothing to the passed string.

You can disable the recognition of quotation marks and apostrophes as DECTPU quote characters by using the OFF keyword as a parameter for EDIT. When you use the OFF keyword, DECTPU preserves any quotation marks and apostrophes in the edited text and performs the editing tasks you specify on the text within the quotation marks and apostrophes. OFF may appear anywhere in the keyword list. It need not be the final parameter.

If the string you specify has opening quotation marks but not closing quotation marks, the status TPU \$ _MISSINGQUOTE is returned. All text starting at the unclosed opening quotation mark and continuing to the end of the string is considered to be part of the quoted string and is not modified.

EDIT is similar to the DCL lexical function F\$EDIT, with the following differences:

- EDIT modifies the characters in place, while F\$EDIT returns a result.
- EDIT takes keywords as parameters, while F\$EDIT requires that the edit commands be specified by a string.

Signaled Errors

TPU\$_MISSINGQUOTE	ERROR	Character string is missing terminating quotation marks.
TPU\$_TOOFEW	ERROR	EDIT requires at least one parameter.
TPU\$_TOOMANY	ERROR	You supplied keywords that are the same or contradictory.
TPU\$_ARGMISMATCH	ERROR	One of the parameters to EDIT is of the wrong data type.
TPU\$_INVPARAM	ERROR	One of the parameters to EDIT is of the wrong data type.
TPU\$_BADKEY	WARNING	You gave the wrong keyword to EDIT.

Examples

1. The following example edits the string "PRODUCT NAME" by changing it to lowercase, and displays the edited string in the message window:

```
pn := "PRODUCT NAME";
EDIT (pn, LOWER);
MESSAGE (pn);
```

2. The following example shows a generalized way of changing any input string to lowercase:

```
PROCEDURE user_edit_string (input_string)

    is := input_string;

    EDIT (is, LOWER);
    MESSAGE (is);
ENDPROCEDURE;
```

After compiling the preceding procedure, you can direct DECTPU to print the lowercase word *zephyr* in the message area by entering the following command:

```
user_edit_string ("ZEPHYR")
```

END_OF

Syntax

```
marker := END_OF ( { buffer } )
```

Parameters

buffer

The buffer whose last character position you want to mark.

range

The range whose last character position you want to mark.

Return Value

A marker pointing to the last character position in a buffer or range.

Description

The END_OF procedure returns a marker that points to the last character position in a buffer or a range. If you use the marker returned by the END_OF built-in as a parameter for the POSITION built-in procedure, the editing point moves to this marker.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	END_OF must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	END_OF requires one argument.
TPU\$_TOOMANY	ERROR	END_OF accept s only one argument.
TPU\$_ARGMISMATCH	ERROR	You passed something other than a range or a buffer to END_OF.

Examples

1. The following example stores the last position in the current buffer in the variable

```
the_end := END_OF (CURRENT_BUFFER)
```

2. The following example implements a simple INSERT HERE function. The variable *paste_buffer* points to a buffer that holds previously cut text.

```

PROCEDURE user_paste

    LOCAL paste_text;

    IF (BEGINNING_OF (paste_buffer) <> END_OF (paste_buffer))
    THEN
        COPY_TEXT (paste_buffer);
    ENDIF;

ENDPROCEDURE;

```

ERASE

Syntax

```
ERASE ( { buffer } )
```

Parameters

buffer

The buffer whose contents you want to remove.

range

The range whose contents you want to remove.

Description

The ERASE procedure removes the contents of the buffer or range that you specify. However, the buffer structure still remains a part of your editing context and the editing point remains in the buffer even if you remove the contents of the buffer. The space that was occupied by the contents of the buffer is returned to the system and is available for reuse. Only the end-of-buffer line remains.

When you erase a range, the contents of the range are removed from the buffer. The range structure is still a part of your editing context. You can use the range structure later in your editing session to delimit an area of text within a buffer.

Note that text does not belong to a range; it belongs to a buffer. Ranges are merely a way of manipulating portions of text within a buffer. For more information on ranges, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

TPU\$_TOOFEW	ERROR	ERASE requires one argument.
TPU\$_TOOMANY	ERROR	ERASE accepts only one argument.
TPU\$_INVPARAM	ERROR	The argument to ERASE is of the wrong type.
TPU\$_NOTMODIFIABLE	WARNING	You cannot erase text in an unmodifiable buffer.

Examples

1. The following example erases all the text in the buffer referenced by *main_buffer*. Because the buffer still exists, you can select the buffer by using the POSITION built-in or by mapping the buffer to a window. The procedure simply removes all text from the buffer. All markers in the buffer now mark the end of the buffer.

```
ERASE (main_buffer)
```

2. The following example deletes embedded carriage-return/line-feed pairs:

```
PROCEDURE user_remove_crlfs

    LOCAL crlf,
           here,
           cr_range;

    crlf := ASCII (13) + ASCII (10);
    here := MARK (NONE);
    POSITION (BEGINNING_OF (CURRENT_BUFFER));

    LOOP
        cr_range := SEARCH_QUIETLY (crlf, FORWARD, EXACT);
        EXITIF cr_range = 0;
        ERASE (cr_range);
        POSITION (cr_range);
    ENDLLOOP;

    POSITION (here);
ENDPROCEDURE;
```

ERASE_CHARACTER

Syntax

```
[ string := ] ERASE_CHARACTER (integer)
```

Parameters

integer

An expression that evaluates to an integer, which may be signed. The value indicates which characters, and how many of them, are to be erased.

Return Value

A string that represents the characters deleted by ERASE_CHARACTER.

Description

The ERASE_CHARACTER procedure deletes up to the number of characters that you specify and optionally returns a string that represents the characters you deleted.

If the argument to ERASE_CHARACTER is a positive integer, ERASE_CHARACTER deletes that many characters, starting at the current position and continuing toward the end of the line. If the

argument is negative, `ERASE_CHARACTER` deletes characters to the left of the current character. It uses the absolute value of the parameter to determine the number of characters to delete.

`ERASE_CHARACTER` stops deleting characters if it reaches the beginning or the end of the line before deleting the specified number of characters.

Using `ERASE_CHARACTER` may cause DECTPU to insert padding spaces or blank lines in the buffer. `ERASE_CHARACTER` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

`ERASE_CHARACTER` optionally returns a string that contains the characters that it deleted.

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>ERASE_CHARACTER</code> requires one argument.
<code>TPU\$_TOOMANY</code>	ERROR	<code>ERASE_CHARACTER</code> accepts only one argument.
<code>TPU\$_INVPARAM</code>	ERROR	The argument to <code>ERASE_CHARACTER</code> must be an integer.
<code>TPU\$_NOCURRENTBUF</code>	WARNING	There is no current buffer to erase characters from.
<code>TPU\$_NOTMODIFIABLE</code>	WARNING	You cannot modify an unmodifiable buffer.

Examples

1. The following example removes the current character and the nine characters following it and copies them in the string variable `take_out_chars`. If there are only five characters following the current character, then this statement deletes only the current character and the five following it. It does not also delete characters on the next line.

```
take_out_chars := ERASE_CHARACTER (10)
```

2. The following example deletes the character to the left of the editing point. If the editing point is at the beginning of a line, the procedure appends the current line to the previous line.

```
! This procedure deletes the character to the
! left of the current character. If at the
! beginning of a line, it appends the current
! line to the previous line.
```

```
PROCEDURE user_delete_key
```

```
    LOCAL deleted_char;
```

```
    deleted_char := ERASE_CHARACTER (-1);
```

```
IF deleted_char = "" ! nothing deleted
THEN
    APPEND_LINE;
ENDIF;
ENDPROCEDURE;
```

ERASE_LINE

Syntax

[[string :=] ERASE_LINE

Parameters

None.

Return Value

A string that contains the text of the deleted line.

Description

The ERASE_LINE procedure removes the current line from the current buffer. The current position moves to the first character of the line following the deleted line. ERASE_LINE optionally returns a string containing the text of the deleted line.

Using ERASE_LINE may cause DECTPU to insert padding spaces or blank lines in the buffer. ERASE_LINE causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

If the screen manager inserts padding spaces, ERASE_LINE deletes these spaces when it deletes the line. The spaces appear in the returned string. If the screen manager inserts padding lines into the buffer, ERASE_LINE deletes only the last of these lines.

Signaled Errors

TPU\$_TOOMANY	ERROR	ERASE_LINE accepts no arguments.
TPU\$_NOTMODIFIABLE	WARNING	You cannot erase a line in an unmodifiable buffer.
TPU\$_NOCURRENTBUF	ERROR	You must select a buffer before erasing a line.

Examples

1. The following example removes the current line from the current buffer:

```
ERASE_LINE
```

2. The following example removes the current line from the current buffer and stores the string of characters representing that line in the variable *take_out_line*:

```
take_out_line := ERASE_LINE
```

ERROR

Syntax

keyword := ERROR

Parameters

None.

Return Value

A keyword that represents the most recent error.

Description

The ERROR procedure returns a keyword for the latest error. The possible error and warning codes for each built-in procedure are included in the description of each built-in procedure. The *OpenVMS System Messages and Recovery Procedures Reference Manual* includes all the possible completion codes for DECTPU as well as the appropriate explanations and suggested user actions.

The value returned by ERROR is meaningful only inside an error handler after an error has occurred. The value outside an error handler is indeterminate.

Although ERROR behaves much like a built-in, it is actually a DECTPU language element.

ERROR is evaluated for correct syntax at compile time. In contrast, DECTPU procedures are usually evaluated for a correct parameter count and parameter types at execution.

Signaled Errors

ERROR is a language element and has no completion codes.

Examples

The following example uses the ERROR language element to determine the error that invoked the error handler. If the error was that SEARCH could not find the specified string, then the procedure returns normally. If the error was something else, then the text of the error message is written to the MESSAGES buffer and any executing procedures are terminated.

```
PROCEDURE strip_blanks
```

```
! Remove trailing blanks from all the lines in a buffer
```

```
    LOCAL blank_chars,
```

```
    blank_pattern,  
    blank_range;  
  
ON_ERROR  
  IF ERROR = TPU$_STRNOTFOUND  
  THEN  
    RETURN;  
  ELSE  
    MESSAGE (ERROR_TEXT);  
    ABORT;  
  ENDIF;  
ENDON_ERROR;  
  blank_chars := ASCII (32) + ASCII (9);  
  blank_pattern := (SPAN (blank_chars) @ blank_range) + LINE_END;  
  LOOP  
    SEARCH (blank_pattern, FORWARD);  
    POSITION (BEGINNING_OF (blank_range));  
    ERASE (blank_range);  
  ENDLOOP;  
ENDPROCEDURE;
```

ERROR_LINE

Syntax

integer := ERROR_LINE

Parameters

None.

Return Value

An integer that represents the line number of the most recent error.

Description

The ERROR_LINE procedure returns the line number at which the latest 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.

The value returned by ERROR_LINE is meaningful only inside an error handler after an error has occurred. The value outside an error handler is indeterminate.

Although ERROR_LINE behaves much like a built-in, it is actually a DECTPU language element.

ERROR_LINE is evaluated for correct syntax at compile time. In contrast, DECTPU procedures are usually evaluated for a correct parameter count and parameter types at execution.

Signaled Errors

ERROR is a language element and has no completion codes.

Examples

The following example uses the `ERROR_LINE` built-in procedure to report the line in which the error occurred:

```
PROCEDURE strip_blanks

! Remove trailing blanks from all the lines in a buffer

    LOCAL blank_chars,
           blank_pattern,
           blank_range;

    ON_ERROR
        MESSAGE (ERROR_TEXT);
        MESSAGE ("Error on line " + STR (ERROR_LINE));
        RETURN;
    ENDON_ERROR;

    blank_chars := ASCII (32) + ASCII (9);
    blank_pattern := (SPAN (blank_chars) @ blank_range) + LINE_END;

    LOOP
        SEARCH (blank_pattern, FORWARD);
        POSITION (blank_range);
        ERASE (blank_range);
    ENDLLOOP;
ENDPROCEDURE;
```

ERROR_TEXT

Syntax

```
string := ERROR_TEXT
```

Parameters

None.

Return Value

A string that contains the text of the most recent error message.

Description

The `ERROR_TEXT` procedure returns the text of the most recent error or warning message.

The possible error and warning codes for each built-in procedure are included in the description of each built-in procedure. The *OpenVMS System Messages and Recovery Procedures Reference Manual* includes all the possible completion codes for DECTPU as well as the appropriate explanations and suggested user actions.

The value returned by `ERROR_TEXT` is meaningful only inside an error handler after an error has occurred. The value outside an error handler is indeterminate.

Although `ERROR_TEXT` behaves much like a built-in, it is actually a DECTPU language element.

`ERROR_TEXT` is evaluated for correct syntax at compile time. In contrast, DECTPU procedures are usually evaluated for a correct parameter count and parameter types at execution.

Signaled Errors

`ERROR_TEXT` is a language element and has no completion codes.

Examples

The following example uses the `ERROR_TEXT` built-in procedure to report what happened and where:

```
PROCEDURE strip_blanks

! Remove Trailing blanks from all the lines in a buffer

    LOCAL blank_chars,
           blank_pattern,
           blank_range;

    ON_ERROR
        MESSAGE (ERROR_TEXT);
        MESSAGE ("Error on line " + STR (ERROR_LINE));
        RETURN;
    ENDON_ERROR;

blank_chars := ASCII (32) + ASCII (9);
blank_pattern := (SPAN (blank_chars) @ blank_range) + LINE_END;

    LOOP
        SEARCH (blank_pattern, FORWARD);
        POSITION (BEGINNING_OF (blank_range));
        ERASE (blank_range);
    ENDLLOOP;
ENDPROCEDURE;
```

EXECUTE

Syntax

$$\text{EXECUTE} \left(\begin{array}{l} \text{buffer} \\ \text{key-name} \left[\begin{array}{l} \text{, key-map-list-name} \\ \text{, key-map-name} \end{array} \right] \\ \text{learn} \\ \text{program} \\ \text{range} \\ \text{string} \end{array} \right)$$

Parameters

buffer

The buffer that you want to execute.

key-name

The DECTPU key name for a key or a combination of keys. DECTPU locates and executes the definition bound to the key.

key-map-list-name

The name of the key map list in which the key is defined. This optional parameter is valid only when the first parameter is a key name. If you specify a key map list as the second parameter, DECTPU uses the first definition of the key specified by *key_name* found in any of the key maps specified by the key map list. If you do not specify any value for the second parameter, DECTPU uses the first definition of the key specified by *key_name* found in the key map list bound to the current buffer.

key-map-name

The name of the key map in which the key is defined. This optional parameter is valid only when the first parameter is a key name. Use this parameter only if the key specified by the first parameter is defined in the key map specified as the second parameter. If you do not specify any value for the second parameter, DECTPU uses the first definition of the key specified by *key_name* found in the key map list bound to the current buffer.

learn

The learn sequence that you want to replay.

program

The program that you want to execute.

range

The range that you want to execute.

string

The string that you want to execute.

Description

The EXECUTE procedure does one of the following:

- Executes programs that you have previously compiled
- Compiles and then executes any executable statements in a buffer, a range, or a string
- Replays a learn sequence
- Executes a program bound to a key

EXECUTE performs different actions depending upon the data type of the parameter.

If the parameter is a string or the contents of a buffer or range, it must contain only valid DECTPU statements; otherwise, you get an error message and no action is taken. See the description of the COMPILE built-in procedure for restrictions and other information on compiling strings or the contents

of a buffer or range. When you pass a string to EXECUTE, the string cannot be longer than 256 characters.

Procedures are usually executed by entering the name of a compiled procedure at the appropriate prompt from your editing interface, or by calling the procedure from within another procedure. However, you can execute procedures with the EXECUTE built-in procedure if the procedure returns a data type that is a valid parameter.

Signaled Errors

TPU\$_NODEFINITION	WARNING	There is no definition for this key.
TPU\$_REPLAYWARNING	WARNING	Inconsistency during the execution of a learn sequence . . . sequence is proceeding.
TPU\$_REPLAYFAIL	WARNING	Inconsistency during the execution of a learn sequence . . . execution stopped.
TPU\$_RECURLEARN	ERROR	You cannot execute learn sequences recursively.
TPU\$_CONTROLC	ERROR	The execution of the command terminated because you pressed Ctrl/C.
TPU\$_EXECUTEFAIL	WARNING	Execution of the indicated item halted because it contains an error.
TPU\$_COMPILEFAIL	WARNING	Compilation aborted because of syntax errors.
TPU\$_ARGMISMATCH	ERROR	A parameter's data type is unsupported.
TPU\$_TOOFEW	ERROR	Too few arguments.
TPU\$_TOOMANY	ERROR	Too many arguments.
TPU\$_NOTDEFINABLE	WARNING	Key cannot be defined.
TPU\$_NOCURRENTBUF	WARNING	Key map or key map list not specified, and there is no current buffer.
TPU\$_NOKEYMAP	WARNING	Key map or key map list not defined.
TPU\$_NOTMODIFIABLE	WARNING	You cannot copy text into an unmodifiable buffer.
TPU\$_NODEFINITION	WARNING	Key not defined.

Examples

1. In the following example, the procedure *test* returns a program data type. If you execute a buffer or range that contains the following code, DECTPU compiles and executes the procedure *test*. A program data type is then returned, the program is used as the parameter for the EXECUTE built-in procedure, and the string "abc" is written to the message area.

```
PROCEDURE test
```

```
! After compiling the string 'MESSAGE ("abc")',
! DECTPU returns a program that is the compiled
! form of the string.
```

```
RETURN COMPILE ('MESSAGE ("abc")');  
ENDPROCEDURE;
```

! The built-in procedure EXECUTE executes the
! program returned by the procedure "test."

```
EXECUTE (test);
```

2. The following example compiles the contents of *main_buffer* and then executes any executable statements. If you have any text in the main buffer other than DECTPU statements, you get an error message. If there are procedure definitions in *main_buffer*, they are compiled; they are not executed until you run the procedure (either by entering the procedure name after the appropriate prompt from your interface or by calling the procedure from within another procedure).

```
EXECUTE (main_buffer)
```

3. The following example prompts you for a DECTPU command to execute and then executes the command:

```
PROCEDURE user_do  
  
    command_string := READ_LINE ("Enter DECTPU command to execute: ");  
    EXECUTE (command_string);  
ENDPROCEDURE;
```

4. The following example executes a command with informational messages turned on, and then turns the informational messages off after the command is executed. You must replace the parameter *TPU_COMMAND* with the DECTPU statement that you want.

```
PROCEDURE user_tpu (TPU_COMMAND)  
  
    SET (INFORMATIONAL, ON);  
    EXECUTE (TPU_COMMAND);  
    SET (INFORMATIONAL, OFF);  
ENDPROCEDURE;
```

EXIT

Syntax

EXIT

Parameters

None.

Description

The EXIT command terminates the editing session and writes out any modified buffers that have associated files. DECTPU queries you for a file name for any buffer that you have modified that does not already have an associated file.

Buffers that have the NO_WRITE attribute are not written out. See SET (NO_WRITE, buffer).

If you do not modify a buffer, DECTPU does not write out the buffer to a file when you use EXIT. If you modify a buffer that has an associated file name (because you specified a file name for the second parameter of CREATE_BUFFER),

DECTPU writes out a new version of the file. DECTPU requires the application to make backup copies of existing files before using EXIT.

If you modify a buffer that does not have an associated file name, DECTPU asks you to specify a file name if you want to write the buffer. If you press the Return key rather than entering a file name, the modified buffer is discarded. DECTPU queries you about all modified buffers that do not have associated file names. The order of the query is the order in which the buffers were created.

DECTPU deletes journal files (if any) upon exiting.

If an error occurs while you are exiting, the exit halts and control returns to the application.

Signaled Errors

TPU\$_EXITFAIL	WARNING	The EXIT did not complete successfully because of problems writing modified buffers.
TPU\$_TOOMANY	ERROR	EXIT takes no arguments.

EXPAND_NAME

Syntax

$$\text{string2} := \text{EXPAND_NAME} (\text{string1} \left\{ \begin{array}{l} , \text{ ALL} \\ , \text{ KEYWORDS} \\ , \text{ PROCEDURES} \\ , \text{ VARIABLES} \end{array} \right\})$$

Parameters

string1

An expression that evaluates to a string. If the string contains one or more asterisks (*) or percent signs (%), then the string is a wildcard specification of the DECTPU names to match. An asterisk matches zero or more characters and a percent sign matches exactly one character. If the string does not contain any asterisks or percent signs, then the string is the initial substring of a DECTPU name.

ALL

A keyword specifying that you want DECTPU to match all names.

KEYWORDS

A keyword specifying that you want DECTPU to match only keyword names.

PROCEDURES

A keyword specifying that you want DECTPU to match only procedure names.

VARIABLES

A keyword specifying that you want DECTPU to match only global variable names. EXPAND_NAME does not expand the names of local variables.

Return Value

Returns a string that contains the names that begin with the string you specify.

Description

The EXPAND_NAME procedure returns a string that contains the names of any DECTPU global variables, keywords, or procedures (built-in or user-written) that begin with the string that you specify. DECTPU searches its internal symbol tables to find a match, using your input string as the directive for the match.

If there are no matches for the substring you specify, a null string is returned and a warning (TPU\$_NONAMES) is signaled. If only one DECTPU name matches the substring you specify, the name is returned with no trailing space. If more than one DECTPU name matches your substring, all of the matching names are returned. The matching names are returned as a concatenated string with words separated by a single space. Multiple names signal a warning (TPU\$_MULTIPLenames).

Use EXPAND_NAME in procedures that perform command completion or that interpret abbreviated names.

EXPAND_NAME does not expand the names of local variables.

Signaled Errors

TPU\$_NONAMES	WARNING	No names were found matching the one requested.
TPU\$_MULTIPLenames	WARNING	More than one name matching the one requested was found.
TPU\$_NEEDTOASSIGN	ERROR	EXPAND_NAME must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	EXPAND_NAME requires two arguments.
TPU\$_TOOMANY	ERROR	EXPAND_NAME accepts no more than two arguments.
TPU\$_INVPARAM	ERROR	One of the arguments you passed to EXPAND_NAME has the wrong data type.
TPU\$_BADKEY	WARNING	You specified an invalid keyword as the second argument.

Examples

1. In the following example, the assignment statement requests all the keywords whose names are two characters long:

```
full_name := EXPAND_NAME ("%%", KEYWORDS)
```

This assignment statement returns the following DECTPU keyword names in the string *full_name*:

```
ON UP DO E5 F6 E4 F7 F4 F5 E6 E1 F2 F3 E3 F1 E2 F8 F9
```

- The following example uses the string that you enter as the parameter, and puts the expanded form of a valid DECTPU procedure name that matches the string in the message area. If the initial string matches multiple procedure names, or if it is not a valid DECTPU procedure name, an explanatory message is written to the message area.

```
PROCEDURE user_quick_parse (abbreviated_name)

  ON_ERROR
    IF ERROR = TPU$_NONAMES
    THEN
      MESSAGE ("No such procedure.");
    ELSE
      IF ERROR = TPU$_MULTIPLenames
      THEN
        MESSAGE ("Ambiguous abbreviation.");
      ENDIF;
    ENDIF;
  RETURN;
ENDON_ERROR;

  expanded_name := EXPAND_NAME (abbreviated_name, PROCEDURES);
  MESSAGE ("The procedure is " + expanded_name + ".");
ENDPROCEDURE;
```

FAO

Syntax

```
string2 := FAO (string1 [ { integer1 } [ , ... { integer_n } ] ])
```

Parameters

string1

A string that consists of the fixed text of the output string and Formatted ASCII Output (FAO) directives.

Some FAO directives that you can use as part of the string are the following:

!AS	Inserts a string as is
!OL	Converts a longword to octal notation
!XL	Converts a longword to hexadecimal notation
!ZL	Converts a longword to decimal notation
!UL	Converts a longword to decimal notation without adjusting for negative numbers
!SL	Converts a longword to decimal notation with negative numbers converted

!/	Inserts a new line (carriage return/line feed)
!_	Inserts a tab
!}	Inserts a form feed
!!	Inserts an exclamation mark
!%S	Inserts an <i>s</i> if the most recently converted number is not 1
!%T	Inserts the current time if you enter 0 as the parameter (you cannot pass a specific time because DECTPU does not use quadwords)
!%D	Inserts the current date and time if you enter 0 as the parameter (you cannot pass a specific date because DECTPU does not use quadwords)

integer1 ... integer_{*n*}

An expression that evaluates to an integer. \$FAO uses these integers as arguments to the FAO directives in *string2* to form *string1*.

string3 ... string_{*n*}

An expression that evaluates to a string. \$FAO uses these strings as arguments to the FAO directives in *string2* to form *string1*.

Return Value

A string that contains the output you specify in ASCII format.

Description

The FAO procedure invokes the Formatted ASCII Output (\$FAO) system service to convert a control string to a formatted ASCII output string. By specifying arguments for FAO directives in the control string, you can control the processing performed by the \$FAO system service. The FAO procedure returns a string that contains the formatted ASCII output, constructed according to the rules of the \$FAO system service. The control string directs the formatting process, and the optional arguments are values to be substituted into the control string.

FAO accepts up to 127 parameters. It can return strings of 65535 characters maximum.

For complete information on the \$FAO system service, see the [VSI OpenVMS System Services Reference Manual](https://docs.vmssoftware.com/vsi-openvms-system-services-reference-manual-a-getuai/) [https://docs.vmssoftware.com/vsi-openvms-system-services-reference-manual-a-getuai/].

To ensure that you get meaningful results, you should use the !AS directive for strings and the !OL, !XL, !ZL, !UL, or !SL directive for integers.

Signaled Errors

TPU\$_INVFAOPARAM	WARNING	Argument was not a string or an integer.
TPU\$_NEEDTOASSIGN	ERROR	FAO must appear on the right-hand side of an assignment statement.
TPU\$_INVPARAM	ERROR	The first argument to FAO must be a string.
TPU\$_TOOFEW	ERROR	FAO requires at least one parameter.

Examples

1. The following example stores the current date and time in the variable *date_and_time*:

```
date_and_time := FAO ("!%D", 0)
```

2. The following example uses the FAO directive !SL in a control string to convert the number equated to the variable *count* to a string. The converted string is stored in the variable *report* and then written to the message area.

```
PROCEDURE user_fao_conversion (count)

    report := FAO ("number of forms = !SL", count);
    MESSAGE (report);
ENDPROCEDURE;
```

FILE_PARSE

Syntax

```
string3 := FILE_PARSE (filespec [ , string1
                        [ , string2
                        [ , NODE ]
                        [ , DEVICE ]
                        [ , DIRECTORY ]
                        [ , NAME ] [ , TYPE ]
                        [ , VERSION ]
                        [ , HEAD ]
                        [ , TAIL ] ]])
```

Parameters

filespec

The file specification to be parsed.

string1

A default file specification. If you fail to specify a field in *filespec* and that field is present in the default file specification, DECTPU substitutes the field from *string1* in the output string.

string2

A related file specification. If you fail to specify a field in *filespec* and *string1* and that field is present in the related file specification and is not the version field, DECTPU substitutes the field from *string2* in the output string.

NODE

Keyword specifying that FILE_PARSE should return a file specification, including the node if one of the files specified in *filespec*, *string1*, or *string2* contains a node field. For more information on using the optional keyword parameters to FILE_PARSE, see the Description section. DECTPU can

parse file specifications that contain a node field, but it cannot search, read, or write them. DECTPU parses the node field only for compatibility with OpenVMS file specifications.

DEVICE

VMS keyword specifying that FILE_PARSE should return a file specification, including the device. For more information on using the optional keyword parameters to FILE_PARSE, see the the section called “Description”.

DIRECTORY

Keyword specifying that FILE_PARSE should return a file specification, including the directory. For more information on using the optional keyword parameters to FILE_PARSE, see the the section called “Description”.

NAME

Keyword specifying that FILE_PARSE should return a file specification, including the name. For more information on using the optional keyword parameters to FILE_PARSE, see the the section called “Description”.

TYPE

Keyword specifying that FILE_PARSE should return a file specification, including the type. For more information on using the optional keyword parameters to FILE_PARSE, see the the section called “Description”.

VERSION

Keyword specifying that FILE_PARSE should return a file specification, including the version. For more information on using the optional keyword parameters to FILE_PARSE, see the the section called “Description”.

HEAD

Keyword specifying that FILE_PARSE should return a file specification, including the node, device, and directory fields. For more information on using the optional keyword parameters to FILE_PARSE, see the Description section.

TAIL

Keyword specifying that FILE_PARSE should return a file specification, including the file name, type, and version fields. For more information on using the optional keyword parameters to FILE_PARSE, see the Description section.

Return Value

A string that contains an expanded file specification or the file specification field that you specify.

Description

The FILE_PARSE procedure parses a file specification and returns a string that contains the expanded file specification or the field that you specify. If you do not provide a complete file specification, FILE_PARSE supplies defaults in the return string.

If an error occurs during the parse, `FILE_PARSE` returns a null string. With `FILE_PARSE`, you can parse file specifications into their individual fields and merge fields from three file specifications into one file specification.

Specify the first three parameters as strings. The remaining parameters are keywords. File specifications that include OpenVMS logical names and device names must terminate with a colon. If you omit optional parameters to the left of a given parameter, you must include null strings as placeholders for the missing parameters.

If you omit any fields from the file specified in *filespec*, `FILE_PARSE` supplies defaults, first from *string1* and then from *string2*. The exception to this is that the version field is not supplied from *string2*.

If you omit the device, directory, type, or version fields from the files specified in *filespec*, *string1*, or *string2*, `FILE_PARSE` supplies default values. The default values are the current device and directory, the file type delimiter (.), and the file version delimiter (;). The exception to this is that the current device and directory are not supplied if either *string1* or *string2* contains a node field.

You can specify as many of the keywords for field names as you wish as long as you do not specify fields that are duplicates of fields returned by the head or tail keywords. For example, you cannot request the head field along with the node, device, or directory fields; and you cannot request the tail field along with the name, type, or version fields. If valid keyword combinations are present, `FILE_PARSE` returns a string containing only those fields requested. The fields are returned in normal file specification order. The normal delimiters are included, but there are no other characters separating the fields. For example, suppose you direct DECTPU to execute the following statements:

```
result := FILE_PARSE ("junk.txt", "work::", "disk1:", NODE, DEPROVICE, TYPE);
MESSAGE (result);
```

When the statements execute, DECTPU displays the following string:

```
WORK::DISK1:.TXT
```

The `FILE_PARSE` built-in procedure parses the file specification provided and returns the portions of the resultant file specification requested. It does not check that the file exists.

You can use wildcard directives in supplying file specifications.

The table gives an example of the parsing of the following OpenVMS file specification:

```
node1::usera$:[flamingo.work]eve$section.tpu$section;12
```

Table 2.2. OpenVMS File Parse Example

Requested Element	Returned Information	Example
NODE	Node name, including trailing colons	NODE1::
DEVICE	Device name, including colon	USERA\$:
DIRECTORY	Entire directory string	[FLAMINGO.WORK]
NAME	File name	EVE\$SECTION
TYPE	File type, including period	.TPU\$SECTION
VERSION	File version, including semicolon	;12
HEAD	Node, device, and directory	NODE1::USERA\$: [FLAMINGO.WORK]

Requested Element	Returned Information	Example
TAIL	Name, type, and version	EVE\$SECTION.TPU\$SECTION;12

Signaled Errors

TPU\$_PARSEFAIL	WARNING	FILE_PARSE detected an error while parsing the file specification.
TPU\$_NEEDTOASSIGN	ERROR	FILE_PARSE must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	FILE_PARSE requires at least one argument.
TPU\$_INVPARAM	ERROR	One of the parameters to FILE_PARSE has the wrong data type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword to FILE_PARSE.
TPU\$_INCKWDCOM	WARNING	You specified HEAD along with NODE, DEVICE, or DIRECTORY; or TAIL along with NAME, TYPE, or VERSION.

Examples

The following example returns a full file specification for the file PROGRAM.PAS. The second parameter provides the name of the directory in which the file can be found. Because the device and version fields are missing from the two parameters, FILE_PARSE includes the current device and the version delimiter (;) in the returned file specification.

```
spec := FILE_PARSE ("program.pass", "[abbott]")
```

FILE_SEARCH

Syntax

```
string3 := FILE_SEARCH (filespec
                        [ , string1
                        [ , string2
                        [ , NODE ]
                        [ , DEVICE ]
                        [ , DIRECTORY ]
                        [ , NAME ] [ , TYPE ]
                        [ , VERSION ]
                        [ , HEAD ]
                        [ , TAIL ] ]])
```

Parameters

filespec

The file specification that you want to find.

string1

A default file specification. If you fail to specify a field in *filespec* and that field is present in the default file specification, DECTPU uses the field from *string1* when searching for the file.

string2

A related file specification. If you fail to specify a field in *filespec* and *string1* and that field is present in the related file specification and is not the version field, DECTPU uses the field from *string2* when searching for the file.

NODE

Keyword specifying that FILE_SEARCH should return a file specification, including the node. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

DEVICE

Keyword specifying that FILE_SEARCH should return a file specification, including the device. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

DIRECTORY

Keyword specifying that FILE_SEARCH should return a file specification, including the directory. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

NAME

Keyword specifying that FILE_SEARCH should return a file specification, including the name. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

TYPE

Keyword specifying that FILE_SEARCH should return a file specification, including the type. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

VERSION

Keyword specifying that FILE_SEARCH should return a file specification, including the version. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

HEAD

Keyword specifying that FILE_SEARCH should return a file specification, including the node, device, and directory fields. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

TAIL

Keyword specifying that FILE_SEARCH should return a file specification, including the file name, type, and version fields. For more information on using the optional keyword parameters to FILE_SEARCH, see the Description section.

Return Value

A string that contains the partial or full file specification that you request from \$SEARCH.

Description

The FILE_SEARCH procedure searches one or more directories and returns the partial or full file specification that matches your request. You must use this built-in procedure multiple times with the same parameter to get all of the occurrences of a file name in the directories.

Specify the first three parameters as strings. The remaining parameters are keywords. File specifications that include OpenVMS logical names and device names must terminate with a colon. If you omit optional parameters to the left of a given parameter, you must include null strings as placeholders for the missing parameters.

Unlike the FILE_PARSE built-in, FILE_SEARCH verifies that the file you specify exists.

If FILE_SEARCH does not find a matching file, or if the built-in finds one or more matches but is invoked again and does not find another match, FILE_SEARCH returns a null string but not an error status. Thus, the null string can act as an "end of matching files" indicator. When FILE_SEARCH returns the status TPU\$_SEARCHFAIL, look in the message buffer to see why the search was unsuccessful.

Refer to the description of the FILE_PARSE built-in for more information on using the optional parameters to FILE_SEARCH.

Signaled Errors

TPU\$_SEARCHFAIL	WARNING	FILE_SEARCH detected an error while searching for the file.
TPU\$_TOOFEW	ERROR	FILE_SEARCH requires at least one parameter.
TPU\$_NEEDTOASSIGN	ERROR	FILE_SEARCH must be on the right-hand side of an assignment statement.
TPU\$_INVPARAM	ERROR	One of the arguments you passed to FILE_SEARCH has the wrong type.
TPU\$_BADKEY	WARNING	One of the keyword arguments you specified is not one of those FILE_SEARCH accepts.
TPU\$_INCKWDCOM	WARNING	You specified HEAD along with NODE, DEVICE, or DIRECTORY; or TAIL along with NAME, TYPE, or VERSION.

Examples

1. In the following example, each time this assignment statement is executed on OpenVMS systems, it returns a string that contains the resulting file specification of a file of type .EXE in SYS\$SYSTEM.

Because no version number is specified, only the latest version is returned. When executing the statement returns a null string, there are no more .EXE files in the directory.

```
fil := FILE_SEARCH ("SYS$SYSTEM:*.EXE")
```

2. The following example is similar to the previous example. It makes use of the fact that you are looking for files in the current OpenVMS directory and that FILE_SEARCH can return parts of the file specification to eliminate the call to FILE_PARSE.

```
PROCEDURE user_collect_rnos

    LOCAL filename;

    ! Reset the file_search context

    filename := FILE_SEARCH ("");

    LOOP
        filename := FILE_SEARCH ("*.RNO", "", "", NAME, TYPE);
        EXITIF filespec = "";
        CREATE_BUFFER (filename, filename);
    ENDLOOP;
ENDPROCEDURE;
```

FILL

Syntax

```
FILL ( { buffer } [, string [, integer1 [, integer2
      [, integer3 ] ] ] ] )
```

Parameters

buffer

The buffer whose text you want to fill.

range

The range whose text you want to fill.

string

The list of additional word separators. The space character is always a word separator.

integer1

The value for the left margin. The left margin value must be at least 1 and must be less than the right margin value. This value defaults to the same value as the buffer's left margin.

integer2

The value for the right margin. This value defaults to the same value as the buffer's right margin. *Integer2* must be greater than the left margin and cannot exceed the maximum record size for the buffer.

integer3

The value for the first line indent. This value modifies the left margin of the first filled line. It can be positive or negative. The result of adding the first line indent to the left margin must be greater than 1 and less than the right margin. This value defaults to 0.

Description

The FILL procedure reformat s the text in the specified buffer or range so that the lines of text are approximately the same length. FILL recognizes two classes of characters: text characters and word separators. Any character can be a word separator, and any character other than the space character can be a text character. The space character is always a word separator, even if it is not present in the second parameter passed to FILL.

A word is a contiguous sequence of text characters, all of which are included on the same line, immediately preceded by a word separator or a line break, and immediately followed by a word separator or line break. If the first or last character in the specified range is a text character, that character marks the beginning or end of a word, regardless of any characters outside the range. Filling a range that starts or ends in the middle of a word may result in the insertion of a line break between that part of the word inside the filled range and that part of the word outside the range.

When filling a range or buffer, FILL does the following to each line:

- Removes any spaces at the beginning of the line
- Sets the left margin of the line
- Moves text up to the previous line if it fits
- Deletes the line if it contains no text
- Splits the line if it is too long

FILL sets the line's left margin to the default left margin unless that line is the first line of the buffer or range being filled. In this case, FILL sets the line's left margin to the fill left margin plus the first line indent. However, if you are filling a range and the range does not start at the beginning of a line, FILL does not change the left margin of that line.

FILL moves a word up to the previous line if the previous line is in the range to be filled and if the word fits on the previous line without extending beyond the fill right margin. Before moving the word up, FILL appends a space to the end of the previous line if that line ends in a space or a text character. It does not append a space if the previous line ends in a word separator other than the space character.

When moving a word up, FILL also moves up any word separators that follow the word, even if these word separators extend beyond the default right margin. Fill does not move up any word separator that would cause the length of the previous line to exceed the buffer's maximum record size. If the previous line now ends in a space, FILL deletes that space. FILL does not delete more than one such space.

FILL moves any word separators at the beginning of a line up to the previous line. It does this even if the word separators extend beyond the fill right margin.

FILL splits a line into two lines whenever the line contains two or more words and one of the words extends beyond the fill right margin. FILL splits the line at the first character of the first word that

contains characters to the right of the fill right margin, unless that word starts at the beginning of the line. In this case, FILL does not split the line.

When operating on a range that does not begin at the first character of a line but does begin left of the fill left margin, FILL splits the line at the first character of the range.

FILL places the cursor at the end of the filled text after completing the previously described tasks.

Signaled Errors

TPU\$_INVRANGE	WARNING	You specified an invalid range enclosure.
TPU\$_TOOFEW	ERROR	FILL requires at least one argument.
TPU\$_TOOMANY	ERROR	FILL accepts no more than five arguments.
TPU\$_ARGMISMATCH	ERROR	One of the parameters to FILL is of the wrong type.
TPU\$_BADMARGINS	WARNING	You specified one of the fill margins incorrectly.
TPU\$_INVPARAM	ERROR	One of the parameters to FILL is of the wrong type.
TPU\$_NOTMODIFIABLE	WARNING	You cannot fill text in an unmodifiable buffer.
TPU\$_NOCACHE	ERROR	FILL could not create a new line because there was no memory allocated for it.
TPU\$_CONTROLC	ERROR	FILL terminated because you pressed Ctrl/C.

Examples

1. The following example fills the current buffer. It uses the buffer's left and right margins for the fill left and right margins. The space character is the only word separator. Upon completion, the current buffer contains no blank lines. All lines begin with a word. Unless the buffer contains a word too long to fit between the left and right margins, all text is between the buffer's left and right margins. Spaces may appear beyond the buffer's right margin.

```
FILL (current_buffer)
```

2. In the following example, if *paragraph_range* references a range that contains a paragraph, this statement fills a paragraph. FILL uses a left margin of 5 and a right margin of 65. It indents the first line of the paragraph an additional five characters. The space character and the hyphen are the two word separators. If the paragraph contains a hyphenated word, FILL breaks the word after the hyphen if necessary.

```
FILL (paragraph_range, "-", 5, 65, 5)
```

3. The next example is like the previous one except that FILL unindents the first line of the paragraph by three characters. This is useful for filling numbered paragraphs.

```
FILL (paragraph_range, "-", 10, 65, -3)
```

GET_CLIPBOARD

Syntax

```
string := GET_CLIPBOARD
```

Parameters

None.

Return Value

A string that consists of the data read from the clipboard. Line breaks are indicated by a line-feed character (ASCII (10)).

Description

The GET_CLIPBOARD procedure reads STRING format data from the clipboard and returns a string that contains this data. DECwindows provides a clipboard that lets you move data between applications. Applications can write to the clipboard to replace previous data, and can read from the clipboard to get a copy of existing data. The data in the clipboard can be in multiple formats, but all the information in the clipboard must be written at the same time.

DECTPU provides no clipboard support for applications not written for DECwindows.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	GET_CLIPBOARD must return a value.
TPU\$_TOOMANY	ERROR	Too many arguments passed to GET_CLIPBOARD.
TPU\$_CLIPBOARDFAIL	WARNING	The clipboard did not return any data.
TPU\$_CLIPBOARDLOCKED	WARNING	DECTPU cannot read from the clipboard because some other application has locked it.
TPU\$_CLIPBOARDNODATA	WARNING	There is no string format data in the clipboard.
TPU\$_TRUNCATE	WARNING	Characters were truncated because you tried to add text that would exceed the maximum line length.
TPU\$_STRTOOLARGE	ERROR	The amount of data in the clipboard exceeds 65535 characters.

TPU\$_REQUIRESDECW	ERROR	You can use GET_CLIPBOARD only if you are using DECwindows DECTPU.
--------------------	-------	--

Examples

The following statement reads what is currently in the clipboard and assigns it to *new_string* :

```
new_string := GET_CLIPBOARD;
```

GET_DEFAULT

Syntax

```
{ string3 } := GET_DEFAULT (string1, string2 )
{ integer }
```

Parameters

string1

The name of the resource whose value you want GET_DEFAULT to fetch. Resource names are case sensitive.

string2

The class of the resource. Resource class names are case sensitive.

Return Value

The string equivalent of the resource value or 0 if the specified resource is not defined. If necessary, the application must convert the string to the data type appropriate to the resource.

Description

The GET_DEFAULT procedure returns the value of an X resource from the X resources database. GET_DEFAULT is useful for initializing a layered application that uses an X defaults file. You can use GET_DEFAULT only in the DECwindows environment.

If you use the SET (DEFAULT_FILE) built-in to merge a new X resource database into the display's database, this affects the values returned by GET_DEFAULT.

Signaled Errors

TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_TOOFEW	ERROR	Too few arguments passed to GET_DEFAULT.
TPU\$_TOOMANY	ERROR	Too many arguments passed to GET_DEFAULT.

TPU\$_NEEDTOASSIGN	ERROR	GET_DEFAULT must return a value.
TPU\$_REQUIRESDECW	ERROR	You can use GET_DEFAULT only if you are using DECwindows DECTPU.

Examples

The following example shows the portion of a module_init procedure directing DECTPU to fetch the value of a resource from the X resources database. For more information on module_init procedures, see the *Extensible Versatile Editor Reference Manual*.

If you want to create an extension of EVE that enables use of an X defaults file to choose a keypad setting, you can use a GET_DEFAULT statement in a module_init procedure.

```
PROCEDURE application_module_init

LOCAL
    keypad_name;

    :
    :
    :

keypad_name := GET_DEFAULT ("user.keypad", "User.Keypad");

EDIT (keypad_name, UPPER); ! Convert the returned string to uppercase.

IF keypad_name <> '0'
THEN

    CASE keypad_name

        "EDT" : eve_set_keypad_edt ();
        "NOEDT" : eve_set_keypad_noedt ();
        "WPS" : eve_set_keypad_wps ();
        "NOWPS" : eve_set_keypad_nowps ();
        "NUMERIC" : eve_set_keypad_numeric ();
        "VT100" : eve_set_keypad_vt100 ();
        [INRANGE, OUTRANGE] : eve_set_keypad_numeric; ! If you have
                                                    ! used invalid value,
                                                    ! set the keypad to
                                                    ! NUMERIC setting.

    ENDCASE;

ENDIF;

    :
    :
    :

ENDPROCEDURE;
```

To provide a value for the GET_DEFAULT statement to fetch, an X defaults file would contain an entry similar to the following:

```
User.Keypad : EDT
```

GET_GLOBAL_SELECT

Syntax

$$\left\{ \begin{array}{l} \text{array} \\ \text{integer} \\ \text{string} \\ \text{unspecified} \end{array} \right\} := \text{GET_GLOBAL_SELECT} \left(\left\{ \begin{array}{l} \text{PRIMARY} \\ \text{SECONDARY} \\ \text{selection_name} \end{array} \right\}, \text{selection_property_name} \right)$$

Parameters

PRIMARY

A keyword indicating that the layered application is requesting information about a property of the primary global selection.

SECONDARY

A keyword indicating that the layered application is requesting information about a property of the secondary global selection.

selection_name

A string identifying the global selection whose property is the subject of the layered application's information request. Specify the selection name as a string if the layered application needs information about a selection other than the primary or secondary global selection.

selection_property_name

A string specifying the property whose value the layered application is requesting.

Return Values

array

An array that passes information about a global selection whose contents describe information that is not of a data type supported by DECTPU.

DECTPU does not use or alter the information in the array; the application layered on DECTPU is responsible for determining how the information is used, if at all. Because the array is used to receive information from other DECwindows applications, all applications that exchange information whose data type is not supported by DECTPU must adopt a convention on how the information is to be used.

The element *array {0}* contains a string that names the data type of the information being passed. For example, if the information being passed is a span, the element contains the string "SPAN". The element *array {1}* contains either the integer 8, indicating that the information is passed as a series of bytes, or the integer 32, indicating that the information is passed as a series of longwords.

If *array {1}* contains the value 8, the element *array {2}* contains a string and there are no array elements after *array {2}*. The string does not name anything, but rather is a series of bytes of information. As mentioned, the meaning and use of the information is agreed upon by convention among the DECwindows applications. To interpret this string, the application can use the SUBSTR

built-in procedure to obtain substrings one at a time, and the ASCII built-in procedure to convert the data to integer format if necessary. For more information about using these DECTPU elements, see the description of the SUBSTR and ASCII built-in procedures.

integer

The value of the specified global selection property. The return value is of type integer if the value of the specified global selection property is of type integer.

string

The value of the specified global selection property. The return value is of type string if the value of the specified global selection property is of type string.

unspecified

A data type that indicates that the information requested by the layered application was not available.

Description

The GET_GLOBAL_SELECT procedure supplies information about a global selection. If an owner for the global selection exists, and if the owner provides the information requested in a format that DECTPU can recognize, GET_GLOBAL_SELECT returns the information.

Signaled Errors

TPU\$_ARGMISMATCH	ERROR	Wrong type of data sent to GLOBAL_SELECT.
TPU\$_NEEDTOASSIGN	ERROR	GLOBAL_SELECT must return a value.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_REQUIRESDECW	ERROR	You can use GLOBAL_SELECT only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to GLOBAL_SELECT.
TPU\$_TOOMANY	ERROR	Too many arguments passed to GLOBAL_SELECT.
TPU\$_GBLSELOWNER	WARNING	DECTPU owns the global selection.
TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVGBLSELDATA	WARNING	The global selection owner provided data that DECTPU cannot process.
TPU\$_NOGBLSELDATA	WARNING	The global selection owner indicated that it cannot provide the information requested.

TPU\$_NOGBLSELOWNER	WARNING	You requested information about an unowned global selection.
TPU\$_TIMEOUT	WARNING	The global selection owner did not respond before the timeout period expired.

Examples

The following example fetches the text in the primary global selection and assigns it to the variable *string_to_paste*:

```
string_to_paste := GET_GLOBAL_SELECT (PRIMARY, "STRING");
```

For another example of how to use the GET_GLOBAL_SELECT built-in procedure, see Example A.3.

GET_INFO

Description

The GET_INFO procedure returns information about the current status of the editor.

For information on how to get a screen display of the status of your editor, see the description of the SHOW built-in procedure.

This description provides general information on the GET_INFO built-in procedures. Included are descriptions of individual GET_INFO built-ins. The individual GET_INFO built-ins are grouped according to the value of their first parameter. For a list of the groups of GET_INFO built-ins, see Table 2.3.

All GET_INFO built-in procedures have the following two characteristics in common:

- They return a value that is the piece of information you have requested.
- They consist of the GET_INFO statement followed by at least two parameters, as follows:
 - The first parameter specifies the general topic about which you want information. If you want the GET_INFO built-in to return information on a given variable, use that variable as the first parameter. For example, if you want to know what row contains the cursor in a window stored in the variable *command_window*, you would specify the variable *command_window* as the first parameter. For example:

```
the_row := GET_INFO (command_window, "current_row");
```

Otherwise, the first parameter is a keyword specifying the general subject about which GET_INFO is to return information. The valid keywords for the first parameter are as follows:

```
ARRAY
BUFFER
COMMAND_LINE
DEBUG
DEFINED_KEY
KEY_MAP
KEY_MAP_LIST
```

mouse_event_keyword
PROCEDURES
PROCESS
SCREEN
SYSTEM
WIDGET
WINDOW

For a list of valid mouse event keywords, see Table 2.4.

Do not confuse a GET_INFO built-in whose first parameter is a keyword (such as ARRAY) with a GET_INFO built-in whose first parameter is a variable of a given data type, such as *array_variable*. For example, the GET_INFO (*array_variable*) built-in procedure shows what string constants can be used when the first parameter is an array variable; the GET_INFO (ARRAY) built-in shows what can be used when the first parameter is the ARRAY keyword.

- The second parameter (a DECTPU string) specifies the exact piece of information you want.
- The third and subsequent parameters, if necessary, provide additional information that DECTPU uses to identify and return the requested value or structure.

Each GET_INFO built-in procedure in this section shows the possible return values for a given combination of the first and second parameters. For example, the GET_INFO (*any_variable*) built-in shows that when you use any variable as the first parameter and the string "type" as the second parameter, GET_INFO returns a keyword for the data type of the variable.

Depending upon the kind of information requested, GET_INFO returns any one of the following:

- An array
- A buffer
- An integer
- A keyword
- A marker
- A process
- A range
- A string
- A window

DECTPU maintains internal lists of the following items:

- Arrays
- Array elements
- Breakpoints
- Buffers
- Defined keys

- Key maps
- Key map lists
- Processes
- Windows

You can step through an internally maintained list by using *"first"*, *"next"*, *"previous"*, or *"last"* as the second parameter to GET_INFO. The order in which DECTPU maintains these lists is private and may change in a future version.

Do not write code that depends on a list being maintained in a particular order. When you write code to search a list, remember that DECTPU keeps only one pointer for each list. If you create nested loops that attempt to search the same list, the results are unpredictable.

For example, suppose that a program intended to search two key map lists for common key maps sets up a loop within a loop. The outer loop might contain the following statement:

```
GET_INFO (KEY_MAP, "previous", name_of_second_key_map)
```

The inner loop might contain the following statement:

```
GET_INFO (KEY_MAP, "next", name_of_first_key_map)
```

In DECTPU, the behavior of such a nested loop is unpredictable.

Unless documented otherwise, the order of the internal list is not defined.

The syntax of GET_INFO depends on the kind of information you are trying to get. For more information on specific GET_INFO built-in procedures, see the descriptions in this section. GET_INFO built-ins whose first parameter is a keyword are grouped separately from GET_INFO built-ins whose first parameter is a variable.

Table 2.3. GET_INFO Built-In Procedures by First Parameter

Variable	Keyword	Any Keyword or Key Name
GET_INFO (any_variable)	GET_INFO (ARRAY)	GET_INFO (any_keyname)
GET_INFO (array_variable)	GET_INFO (BUFFER)	GET_INFO (any_keyword)
GET_INFO (buffer_variable)	GET_INFO (COMMAND_LINE)	
GET_INFO (integer_variable)	GET_INFO (DEBUG)	
GET_INFO (marker_variable)	GET_INFO (DEFINED_KEY)	
GET_INFO (process_variable)	GET_INFO (KEY_MAP)	
GET_INFO (range_variable)	GET_INFO (KEY_MAP_LIST)	
GET_INFO (string_variable)	GET_INFO (mouse_event_keyword)	
GET_INFO (widget_variable)	GET_INFO (PROCEDURES)	
GET_INFO (window_variable)	GET_INFO (PROCESS)	
	GET_INFO (SCREEN)	
	GET_INFO (SYSTEM)	

Variable	Keyword	Any Keyword or Key Name
	GET_INFO (WIDGET)	
	GET_INFO (WINDOW)	

Signaled Errors

TPU\$_BADREQUEST	WARNING	Request represented by second argument is not understood for data type of first argument.
TPU\$_BADKEY	WARNING	Bad keyword value or unrecognized data type is passed as the first argument.
TPU\$_NOCURRENTBUF	WARNING	Current buffer is not defined.
TPU\$_NOKEYMAP	WARNING	Key map is not defined.
TPU\$_NOKEYMAPLIST	WARNING	Key map list is not defined.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong data type.
TPU\$_NEEDTOASSIGN	ERROR	The GET_INFO built-in can be used only on the right-hand side of an assignment statement.
TPU\$_NOBREAKPOINT	WARNING	This string constant is valid only after a breakpoint.
TPU\$_NONAMES	WARNING	There are no names matching the one requested.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the GET_INFO built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the GET_INFO built-in.
TPU\$_UNKKEYWORD	ERROR	An unknown keyword was used as an argument.

Examples

1. The following example stores the pointer to the current buffer in the variable *my_buffer*:

```
my_buffer := GET_INFO (BUFFERS, "current");
```

2. The following example stores the integer 1 or 0 in the variable *is_buf_mod*. A value of 1 means the current buffer has been modified. A value of 0 means the current buffer has not been modified.

```
is_buf_mod := GET_INFO (CURRENT_BUFFER, "modified");
```

3. The following example uses GET_INFO to find the top of the current window. It then removes the top five lines and replaces them with an example window.

```
PROCEDURE user_getinfo
```

```
    top_of_window := GET_INFO (CURRENT_WINDOW, "top", visible_window);
```

```
    ! Remove the top five lines from the main window
```



```

ADJUST_WINDOW (CURRENT_WINDOW, +5, 0);

! Replace removed lines with an example window
example_window := CREATE_WINDOW (top_of_window, 5, ON);
example_buffer := CREATE_BUFFER ("EXAMPLE",
                                "sys$login:template.txt");
MAP (example_window, example_buffer);
ENDPROCEDURE;

```

4. The following example shows whether the key map list associated with the current buffer inserts undefined printable characters:

```

PROCEDURE show_self_insert

LOCAL key_map_list_name;

key_map_list_name := GET_INFO (CURRENT_BUFFER, "key_map_list");

IF GET_INFO (key_map_list_name, "self_insert")
THEN
MESSAGE ("Undefined printable characters will be inserted");
ELSE
MESSAGE ("Undefined printable characters will cause an error");
ENDIF;
ENDPROCEDURE;

```

GET_INFO (any_keyname)

Syntax

$$\left\{ \begin{array}{l} \text{integer} \\ \text{keyword} \end{array} \right\} := \text{GET_INFO} (\text{any_keyname}, \left\{ \begin{array}{l} \text{"key_modifiers"} \\ \text{"key_type"} \\ \text{"unmodified"} \end{array} \right\})$$

Parameters

"key_modifiers"

Returns a bit-encoded integer that indicates what key modifier or modifiers were used to create the DECTPU key name specified by the parameter *any_keyname*. For more information about the meaning and possible values of key modifiers, see the description of the KEY_NAME built-in procedure.

DECTPU defines four constants to be used when referring to or testing the numerical value of key modifiers. The correspondence between key modifiers, defined constants, and bit-encoded integers is as follows:

Key Modifier	Constant	Bit-Encoded Integer
SHIFT_MODIFIED	TPU\$K_SHIFT_MODIFIED	1
CTRL_MODIFIED	TPU\$K_CTRL_MODIFIED	2
HELP_MODIFIED	TPU\$K_HELP_MODIFIED	4
ALT_MODIFIED	TPU\$K_ALT_MODIFIED	8

You may have used the `SHIFT_KEY` keyword to create a DECTPU key name. `SHIFT_KEY` is not a modifier; it is a prefix. The Shift key is also called the `GOLD` key by the EVE editor. To use the Shift key, press and release it before you press any other key.

In DECwindows, to use modifying keys (such as `Ctrl`), press and hold the modifying key while you press the modified key.

If you use more than one key modifier with the `KEY_NAME` built-in procedure, the value of the returned integer is the sum of the integer representations of the key modifiers. For example, if you create a key name by using the modifiers `HELP_MODIFIED` and `ALT_MODIFIED`, the `GET_INFO` (`any_keyname`, "key_ modifiers") built-in returns the integer 12.

"key_type"

Returns a keyword that describes the type of key named by *any_keyname*. The keywords that can be returned are `PRINTING`, `KEYPAD`, `FUNCTION`, `CONTROL`, `SHIFT_PRINTING`, `SHIFT_KEYPAD`, `SHIFT_FUNCTION`, and `SHIFT_CONTROL`. Returns 0 if *parameter1* is not a valid key name. There are cases in which `GET_INFO` (`any_keyname`, "name") returns the `PRINTING` keyword but the key described by the key name is not associated with a printable character. For example, if you use the `KEY_NAME` built-in to define a key name as the combination of the character `A` and the `ALT` modifier, and if you then use `GET_INFO` (`any_keyname`, "name") to find out how DECTPU classifies the key, the `GET_INFO` built-in returns the `PRINTING` keyword. However, if you use the `ASCII` built-in to obtain the string representation of the key, the `ASCII` built-in returns a null string because `ALT/A` is not printable.

"unmodified"

Returns a keyword that describes the key name specified by *any_keyname* without any modifiers. For example, if you create a key name for the `F20` key with the `ALT_MODIFIED` modifier, the `GET_INFO` (`any_keyname`, "unmodified") built-in returns the `F20` keyword.

Return Values

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

Description

The `GET_INFO` (`any_keyname`) procedure returns a keyword that describes the type of key named by *any_keyname*.

For general information about using all forms of `GET_INFO` built-in procedures, see the description of `GET_INFO`.

Examples

In the following example, the first statement creates a DECTPU key name for the key sequence produced by pressing the `Ctrl` key, the `Shift` key, and the `4` key on the keypad all at once. The new key name is assigned to the variable *new_key*. The second statement fetches the integer equivalent of this combination of key modifiers. The third statement displays the integer 3 in the message buffer. The

IF clause of the fourth statement shows how to test whether a key name was created using a modifier. This statement does not detect whether a key name was created using the `SHIFT_KEY` keyword. The THEN clause shows how to fetch the key modifier keyword or keywords used to create a key name. The final statement displays the string `KEY_NAME (KP4, SHIFT_MODIFIED, CTRL_MODIFIED)` in the message buffer.

```
new_key := KEY_NAME (KP4, SHIFT_MODIFIED, CTRL_MODIFIED);
modifier_value := GET_INFO (new_key, "key_modifiers");
MESSAGE (STR (modifier_value));
IF GET_INFO (new_key, "key_modifiers")
THEN
    the_name := GET_INFO (new_key, "name")
    MESSAGE (STR (the_name));
ENDIF;
```

GET_INFO (any_keyword)

Syntax

```
string := GET_INFO (any_keyword, "name")
```

Parameters

"name"

Returns the string equivalent of the specified keyword.

You can use `GET_INFO (any_keyword, "name")` to obtain the string equivalent of a key name. This is useful for displaying screen messages about keys. For example, to obtain the string equivalent of the key name `PF1`, you could use the following statement:

```
the_string := GET_INFO (PF1, "name");
```

If a key name is created using several key modifiers, the built-in returns the string representations of all the keywords used to create the key name. For more information on creating key names, see the description of the `KEY_NAME` built-in procedure.

Return Value

Returns the requested information about the string you specify.

Description

The `GET_INFO (any_keyword)` procedure returns the string representation of the keyword specified in the first parameter to `GET_INFO`.

For general information about using all forms of `GET_INFO` built-in procedures, see the description of `GET_INFO`. See also the description of `GET_INFO (integer_variable)`.

Examples

The following example shows one possible use of `GET_INFO (any_keyword, "name")`. The first statement creates a DECTPU key name for the key sequence produced by pressing the Ctrl key, the Shift

key, and the 4 key on the keypad all at once. The new key name is assigned to the variable *key_name*. The IF clause of the statement shows how to test whether a key name was created using one or more key modifier keywords. This statement does not detect whether a key name was created using the SHIFT_KEY keyword. The GET_INFO (any_keyname, "key_modifiers") built-in returns 0 even if the key name was created using SHIFT_KEY. The THEN clause shows how to fetch the key modifier keyword or keywords used to create a key name. The final statement displays the string KEY_NAME (KP4, SHIFT_MODIFIED, ALT_MODIFIED) in the message buffer.

```
new_key := KEY_NAME (KP4, SHIFT_MODIFIED, CTRL_MODIFIED);
!      .
!      .
!      .
IF GET_INFO (new_key, "key_modifiers") <> 0
THEN
    the_name := GET_INFO (new_key, "name")
ENDIF;
MESSAGE (STR (the_name));
```

GET_INFO (any_variable)

Syntax

```
keyword := GET_INFO (any_variable, "type")
```

Parameter

"type"

Returns a keyword that is the data type of the variable specified in *any_variable*.

Return Value

Returns the requested information about the keyword you specify.

Description

The GET_INFO (any_variable) procedure returns a keyword that specifies the data type of the variable.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

Examples

The following example tests whether the variable *select_popup* has been assigned a value of type widget. If not, the code causes a message to be displayed on the screen.

```
IF GET_INFO (select_popup, "type") <> WIDGET
THEN
    MESSAGE ("Select_popup widget not created.")
ENDIF;
```

GET_INFO (ARRAY)

Syntax

```
array := GET_INFO (ARRAY { "current"  
                          "first"  
                          "last"  
                          "next"  
                          "previous" } )
```

Parameters

"current"

Returns the current array in DECTPU's internal list of arrays. You must use either GET_INFO (ARRAY, "first") or GET_INFO (ARRAY, "last") before you can use GET_INFO (ARRAY, "current"). If you use these built-ins in the wrong order or if no arrays have been created, GET_INFO (ARRAY, "current") returns 0.

"first"

Returns the first array in the DECTPU internal list of arrays; returns 0 if no arrays are defined.

"last"

Returns the last array in the DECTPU internal list of arrays; returns 0 if no arrays are defined.

"next"

Returns the next array in DECTPU's internal list of arrays. This parameter is valid whenever GET_INFO (ARRAY, "current") would return an array. That is, the intended usage of this parameter is that you use GET_INFO (ARRAY, "first") before you can use GET_INFO (ARRAY, "next").

Returns 0 if you use this parameter immediately after GET_INFO (ARRAY, "last"), or if you have not created an array.

"previous"

Returns the previous array in DECTPU's internal list of arrays. The intended usage of this parameter is that you use either GET_INFO (ARRAY, "current") or GET_INFO (ARRAY, "last") before you use GET_INFO (ARRAY, "previous").

Returns 0 if you use this parameter immediately after GET_INFO (ARRAY, "first"), or if you have not created an array.

Return Value

Returns the requested information about the array you specify.

Description

The GET_INFO (ARRAY) procedure returns an array in DECTPU's internal list of arrays.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (array_variable)

Syntax

array buffer integer keyword marker process range string widget window	:= GET_INFO (array_variable,	"current" "first" "high_index" "last" "low_index" "next" "previous")
---	------------------------------	---	---

Parameters

"current"

Returns the index value of the current element of the specified array, whether the index is of type integer or some other type. Returns any type except program, pattern, or learn. Returns the UNSPECIFIED keyword if there is no current element.

You must use either GET_INFO (array_variable, "first") or GET_INFO (array_variable, "last") before you can use GET_INFO (array_variable, "current").

"first"

Returns the index value of the first element of the specified array, whether the index is of type integer or some other type. Returns any type except program, pattern, or learn. Returns the UNSPECIFIED keyword if there is no first element.

"high_index"

Returns an integer that is the highest valid integer index for the static predeclared portion of the array. If the GET_INFO call returns a high index lower than the low index, the array has no static portion.

"last"

Returns the index value of the last element of the specified array, whether the index is of type integer or some other type. Returns any type except program, pattern, or learn. Returns the UNSPECIFIED keyword if there is no last element.

"low_index"

Returns an integer that is the lowest valid integer index for the static predeclared portion of the array. If the GET_INFO call returns a high index lower than the low index, the array has no static portion.

"next"

Returns the index value of the next element of the specified array, whether the index is of type integer or some other type. Returns any type except program, pattern, or learn. Returns the UNSPECIFIED keyword if there is no next element.

You must use GET_INFO (array_variable, "first") before you can use GET_INFO (array_variable, "next").

"previous"

Returns the index value of the previous element of the specified array, whether the index is of type integer or some other type. Returns any type except program, pattern, or learn. Returns the UNSPECIFIED keyword if there is no previous element.

You must use either GET_INFO (array_variable, "current") or GET_INFO (array_variable, "last") before you can use GET_INFO (array_variable, "previous").

Return Values

array

Returns requested information about the array you specify.

buffer

Returns requested information about the buffer you specify.

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

marker

Returns requested information about the marker you specify.

process

Returns requested information about the process you specify.

range

Returns requested information about the range you specify.

string

Returns requested information about the string you specify.

widget

Returns requested information about the widget you specify.

window

Returns requested information about the window you specify.

Description

The GET_INFO (array_variable) procedure returns information about a specified array.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (BUFFER)

Syntax

```
buffer := GET_INFO (BUFFER[[S]] { "current"  
                                "find_buffer", "find_name"  
                                "first"  
                                "last"  
                                "next"  
                                "previous" } )
```

Parameters

"current"

Returns the current buffer in DECTPU's internal list of buffers; returns 0 if there is no current buffer.

GET_INFO (BUFFER[[S]], "current") always returns the current buffer, regardless of whether you have first used GET_INFO (BUFFER[[S]], "first") or GET_INFO (BUFFER[[S]], "last"). Thus, GET_INFO (BUFFER[[S]], "current") is equivalent to the CURRENT_BUFFER built-in procedure.

"find_buffer", buffer_name

Returns the buffer whose name you specify (as a string) as the third parameter; returns 0 if no buffer with the name you specify is found.

"first"

Returns the first buffer in DECTPU's internal list of buffers; returns 0 if there is none.

"last"

Returns the last buffer in DECTPU's internal list of buffers; returns 0 if there is none.

"next"

The next buffer in DECTPU's internal list of buffers; returns 0 if there are no more.

"previous"

Returns the preceding buffer in DECTPU's internal list of buffers; returns 0 if there is none.

Return Value

Returns the requested information about the buffer you specify.

Description

The GET_INFO (BUFFER) procedure returns a buffer in DECTPU's internal list of buffers.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (buffer_variable)

Syntax

<pre> { integer keyword learn_sequence marker program range string } </pre>	<pre> := GET_INFO (buffer_variable, </pre>	<pre> "before_bol" "beyond_eob" "beyond_eol" "bound" "character" "direction" "eob_text" "erase_unmodifiable" "file_name" "first_marker" "first_range" "journaling" "journal_file" "journal_name" "key_map_list" "left_margin" "left_margin_action" "line" "map_count" "max_lines" "middle_of_tab" "mode" "modifiable" "modified" "move_vertical_context" "name" "next_marker" "next_range" "no_write" "offset" "offset_column" "output_file" "permanent" "read_routine", GLOBAL_SELECT "record_count" "record_number" "record_size" "right_margin" "right_margin_action" "safe_for_journaling" "system" "tab_stops" "unmodifiable_records") </pre>
---	--	---

Parameters

"before_bol"

Returns an integer (1 or 0) that indicates whether the editing point is located before the beginning of a line.

"beyond_eob"

Returns an integer (1 or 0) that indicates whether the editing point is located beyond the end of a buffer.

"beyond_eol"

Returns an integer (1 or 0) that indicates whether the editing point is located beyond the end of a line.

"bound"

Returns an integer (1 or 0) that indicates whether the marker that is the specified buffer's editing point is bound to text. For more information about bound markers, see the *Guide to the DEC Text Processing Utility*.

"character"

Returns a string that is the character at the editing point for the buffer.

"direction"

Returns the FORWARD or REVERSE keyword. Use the SET (FORWARD) and SET (REVERSE) built-in procedures to establish or change this parameter.

"eob_text"

Returns a string that represents the end-of-buffer text. Use the SET (EOB_TEXT) built-in procedure to establish or change this parameter.

"erase_unmodifiable"

Returns 1 if unmodifiable records can be erased from the specified buffer and returns 0 if the records cannot be erased.

"file_name"

Returns a string that is the name of a file given as the second parameter to CREATE_BUFFER. Returns the null string if none was specified.

"first_marker"

Returns the first marker in DECTPU's internal list of markers for the buffer. Returns 0 if there is none. You must use GET_INFO (buffer_variable, "first_marker") before the first use of GET_INFO (buffer_variable, "next_marker"). If you do not follow this rule, GET_INFO (buffer_variable, "next_marker") returns 0.

There is no corresponding "last_marker" or "prev_marker" parameter.

Do not write code that relies on DECTPU storing markers in one particular order. Creating markers or ranges may alter the internal order.

"first_range"

Returns the first range in DECTPU's internal list of ranges for the buffer. Returns 0 if there are none. You must use GET_INFO (buffer_variable, "first_range") before you use GET_INFO (buffer_variable, "next_range") or the "next_range" parameter returns 0.

There is no corresponding *"last_range"* or *"prev_range"* parameter.

Do not write code that relies on DECTPU storing ranges in one particular order. Creating markers or ranges may alter the internal order.

"journaling"

Returns 1 if the specified buffer is being journaled or returns 0 if it is not.

"journal_file"

Returns a string that is the name of the journal file for the specified buffer. If the buffer is not being journaled, the call returns 0.

"journal_name"

Converts a buffer's name to a journal file name by using the DECTPU default journal file name algorithm. DECTPU converts the buffer name to a journal file name regardless of journaling status. The GET_INFO call does not require journaling to be turned on for the specified buffer. For more information on this algorithm, see the *Guide to the DEC Text Processing Utility*.

"key_map_list"

Returns a string that is the key map list bound to the buffer. Use the SET built-in procedure to establish or change this parameter.

"left_margin"

Returns an integer that is the current left margin setting. Use the SET (LEFT_MARGIN) built-in procedure to establish or change this parameter.

"left_margin_action"

Returns a program or learn sequence that specifies what DECTPU should do if you try to insert text to the left of the left margin. Returns the UNSPECIFIED keyword if no left margin action routine has been set. Use the SET (LEFT_MARGIN_ACTION) built-in procedure to establish or change this parameter.

"line"

Returns a string that is the line of text at the editing point for the buffer.

"map_count"

Returns an integer that is the number of windows associated with the buffer.

"max_lines"

Returns an integer that is the maximum number of records (lines) in the buffer. Use the SET built-in procedure to establish or change this parameter.

"middle_of_tab"

Returns an integer (1 or 0) that indicates whether the editing point is located in the white space within a tab.

"mode"

Returns the INSERT or OVERSTRIKE keyword. Use the SET (INSERT) and SET (OVERSTRIKE) built-in procedures to establish or change this parameter.

"modifiable"

Returns an integer (1 or 0) that indicates whether the buffer is modifiable.

"modified"

Returns an integer (1 or 0) that indicates whether the buffer has been modified.

"move_vertical_context"

Returns the encoded integer that describes the column where DECTPU attempts to position the cursor during MOVE_VERTICAL operations. See the SET (MOVE_VERTICAL_CONTEXT) built-in procedure for more information.

"name"

Returns a string that is the name given to the buffer when it was created.

"next_marker"

Returns the next marker in DECTPU's internal list of markers for the buffer. Returns 0 if there are no more. You must use GET_INFO (buffer_variable, "first_marker") before you use GET_INFO (buffer_variable, "next_marker") or the *"next_marker"* parameter returns 0.

There is no corresponding *"last_marker"* or *"prev_marke"* parameter.

Do not write code that relies on DECTPU storing markers in one particular order. Creating markers or ranges may alter the internal order.

"next_range"

Returns the next range in DECTPU's internal list of ranges for the buffer. Returns 0 if there are no more. You must use GET_INFO (buffer_variable, "first_range") before you use GET_INFO (buffer_variable, "next_range") or the *"next_range"* parameter returns 0.

There is no corresponding *"last_range"* or *"prev_range"* parameter.

Do not write code that relies on DECTPU storing ranges in one particular order. Creating markers or ranges may alter the internal order.

"no_write"

Returns an integer (1 or 0) that indicates whether the buffer should be written to a file at exit time. DECTPU writes the buffer to a file only if the buffer has been modified during the editing session. Use the SET (NO_WRITE) built-in procedure to establish or change this parameter.

"offset"

Returns an integer that is the number of characters between the left margin and the editing point. The left margin is counted as character 0. A tab is counted as one character, regardless of width.

Window shifts have no effect on the value returned when you use *"offset"*. The value returned has no relation to the visible screen column in which a character is displayed.

"offset_column"

Returns an integer that is the screen column in which DECTPU displays the character at the editing point. When calculating this value, DECTPU does not take window shifts into account; DECTPU assumes that any window mapped to the current buffer is not shifted. The value returned when you use *"offset_column"* reflects the location of the left margin and the width of tabs preceding the editing point. In contrast, the value returned when you use *"offset"* is not affected by the location of the left margin or the width of tabs.

"output_file"

Returns a string that is the name of the file used with the SET (OUTPUT_FILE) built-in procedure. Returns 0 if there is no output file associated with the specified buffer. Use the SET (OUTPUT_FILE) built-in procedure to establish or change this parameter.

"permanent"

Returns an integer (1 or 0) that indicates whether the buffer is permanent or can be deleted. Use the SET (PERMANENT) built-in procedure to establish or change the parameter.

"read_routine"

Use with DECwindows only.

Returns the program or learn sequence that DECTPU executes when it owns a global selection and another application has requested information about that selection. If the application has not specified a global selection read routine, 0 is returned.

GLOBAL_SELECT is a keyword indicating that the built-in is to return the global selection read routine. When you use *"read_routine"* as the second parameter to this built-in, you must use the GLOBAL_SELECT keyword as the third parameter, as follows:

```
GET_INFO (buffer_variable, "read_routine", GLOBAL_SELECT)
```

"record_count"

Returns an integer that is the number of records (lines) in the buffer. GET_INFO (buffer, "record_count") does not count the end-of-buffer text as a record, but GET_INFO (marker, "record_number") does if the specified marker is on the end-of-buffer text. Thus, the maximum value returned by GET_INFO (buffer, "record_count") is one less than the maximum value returned by GET_INFO (marker, "record_number") if the specified marker is on the end-of-buffer text.

"record_number"

Returns the record number of the editing point.

"record_size"

Returns an integer that is the maximum length for records (lines) in the buffer.

"right_margin"

Returns an integer that is the current right margin setting. Use the SET (RIGHT_MARGIN) built-in procedure to establish or change the parameter.

"right_margin_action"

Returns a program or learn sequence that specifies what DECTPU should do if you try to insert text to the right of the right margin. Returns the UNSPECIFIED keyword if the buffer does not have a right margin action.

Use the SET (RIGHT_MARGIN_ACTION) built-in procedure to establish or change the parameter.

"safe_for_journaling"

Returns 1 if the specified buffer is safe for journaling or returns 0 if it is not. "Safe for journaling" means that you can use the SET (JOURNALING) built-in procedure to turn on journaling. A buffer is safe for journaling if it is empty, has never been modified, or has not been modified since the last time it was written to a file.

"system"

Returns an integer (1 or 0) that indicates whether the buffer is a system buffer. Use the SET (SYSTEM) built-in procedure to establish or change the parameter.

"tab_stops"

Returns either an integer or a string. Use the SET (TAB_STOPS) built-in procedure to determine the data type of the return value. If you specify a return value of type string, the GET_INFO (buffer_variable, "tab_stops") built-in procedure returns a string representation of all the column numbers where tab stops are set. The column numbers are separated by spaces. If you specify a return value of type integer, the return value is the number of columns between tab stops.

"unmodifiable_records"

Returns 1 if the specified buffer contains one or more unmodifiable records. The call returns 0 if no unmodifiable records are present in the specified buffer.

Return Values

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

learn_sequence

Returns requested information about the learn_sequence you specify.

marker

Returns requested information about the marker you specify.

program

Returns requested information about the program you specify.

range

Returns requested information about the range you specify.

string

Returns requested information about the string you specify.

Description

The GET_INFO (buffer_variable) procedure returns information about a specified buffer.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (COMMAND_LINE)**Syntax**

```

{ integer } := GET_INFO (COMMAND_LINE,
{ string  }
"character"
"character_set"
"command"
"command_file"
"create"
"display"
"file_name"
"first_file_name"
"initialization"
"initialization_file"
"init_file"
"journal"
"journal_file"
"line"
"modify"
"next_file_name"
"nomodify"
"output"
"output_file"
"read_only"
"recover"
"section"
"section_file"
"start_character"
"start_record"
"work"
"work_file"
"write"
)

```

Parameters**"character"**

Returns an integer that is the column number of the character position specified by the / START_POSITION qualifier. This parameter is useful in a procedure to determine where DECTPU

should place the cursor at startup time. The default is 1 if you do not specify the qualifier or option. This parameter is the same as the *"start_character"* parameter.

"character_set"

Returns a keyword that indicates the character set that you specified by the /CHARACTER_SET qualifier. DECTPU can return the following keywords: DEC_MCS (default), ISO_LATIN1, and GENERAL.

"command"

Returns an integer (1 or 0) that indicates whether the /COMMAND qualifier is active (either by default or because the qualifier or option was specified when you invoked DECTPU).

"command_file"

Returns a string that is the command file specification from the /COMMAND qualifier.

"create"

Returns an integer (1 or 0) that indicates whether the /CREATE qualifier is active (either by default or because the qualifier was specified when you invoked DECTPU).

"display"

Returns an integer (1 or 0) that indicates whether the /DISPLAY or /INTERFACE qualifier is active (either by default or because /DISPLAY or /INTERFACE was specified when you invoked DECTPU).

"file_name"

Returns a string that is the first file specification used as a parameter when you invoke DECTPU. Returns a null string if you did not specify an input file name on the command line.

"first_file_name"

Returns a string that is the first file specification used as a parameter when you invoke DECTPU. Returns 0 if you did not specify any file name on the command line. There is a single input parameter that can be a list of comma-separated file specifications. This GET_INFO returns only the first file specification.

"initialization"

Returns an integer (1 or 0) that indicates whether the /INITIALIZATION qualifier is active (either by default or because the qualifier was specified when you invoked DECTPU).

"initialization_file"

Returns a string that is the initialization file specification for the /INITIALIZATION qualifier.

"init_file"

This is a synonym for GET_INFO (COMMAND_LINE, "initialization_file").

"journal"

Returns an integer (1 or 0) that indicates whether the /JOURNAL qualifier is active (either by default or because the qualifier was specified when you invoked DECTPU).

"journal_file"

Returns a string that is the journal file specification for the /JOURNAL qualifier.

"line"

Returns an integer that is the record number of the line specified by the /START_POSITION qualifier. This parameter is useful in a procedure to determine where DECTPU should place the cursor at startup time. The default is 1 if the qualifier or option is not specified. This parameter is the same as the *"start_record"* parameter.

"modify"

Returns an integer (1 or 0) that indicates whether the /MODIFY qualifier was specified when you invoked DECTPU.

"next_file_name"

Returns the next file name entered on the command line that invoked TPU. Returns 0 if no file name was specified on the command line, or if there are no more file names to return.

"nomodify"

Returns an integer (1 or 0) that indicates whether the /NOMODIFY qualifier was specified when you invoked DECTPU.

"output"

Returns an integer (1 or 0) that indicates whether the /OUTPUT qualifier is active (either by default or because the qualifier was specified when you invoked DECTPU).

"output_file"

Returns a string that is the output file specification for the /OUTPUT qualifier.

"read_only"

Returns an integer (1 or 0) that indicates whether the /READ_ONLY qualifier was specified when you invoked DECTPU.

"recover"

Returns an integer (1 or 0) that indicates whether the /RECOVER qualifier was specified when you invoked DECTPU.

"section"

Returns an integer (1 or 0) that indicates whether the /SECTION qualifier is active (either by default or because the qualifier was specified when you invoked DECTPU).

"section_file"

Returns a string that is the section file specification for the /SECTION qualifier.

"start_character"

Returns an integer that is the column number of the character position specified by the /START_POSITION qualifier. This parameter is useful in a procedure to determine where DECTPU should place the cursor at startup time. The default is 1 if you do not specify qualifier. This parameter is a synonym for *"character"*.

"start_record"

Returns an integer that is the record number of the line specified by the /START_POSITION qualifier. This parameter is useful in a procedure to determine where DECTPU should place the cursor at startup time. The default is 1 if you do not specify the qualifier. This parameter is a synonym for *"line"*.

"work"

Returns an integer (1 or 0) that indicates whether the /WORK qualifier is active (either by default or because the qualifier was specified when you invoked DECTPU).

"work_file"

Returns a string that is the work file specification for the /WORK qualifier.

"write"

Returns an integer (1 or 0) that indicates whether the /WRITE qualifier was specified when you invoked DECTPU.

Return Values

integer

Returns requested information about the integer you specify.

string

Returns requested information about the string you specify.

Description

The GET_INFO (COMMAND_LINE) procedure returns information about the command line used to invoke DECTPU.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (DEBUG)

Syntax

$$\left\{ \begin{array}{l} \text{contents} \\ \text{integer} \\ \text{parameter} \\ \text{string} \\ \text{variable} \end{array} \right\} := \text{GET_INFO} (\text{DEBUG}, \left\{ \begin{array}{l} \text{"breakpoint"} \\ \text{"examine", variable_name} \\ \text{"line_number"} \\ \text{"local"} \\ \text{"next"} \\ \text{"parameter"} \\ \text{"previous"} \\ \text{"procedure"} \end{array} \right\})$$

Parameters

"breakpoint"

Returns a string that is the name of the first breakpoint. This establishes a breakpoint context for the *next* and *previous* parameters. TPU\$_NONAMES is returned if there are no breakpoints.

"examine", variable_name

Returns the contents of the specified variable. TPU\$_NONAMES is returned if the specified variable cannot be found.

You must specify a string that contains the name of the variable as the third parameter to GET_INFO (DEBUG, "examine").

"line_number"

Returns an integer that is the line number of the breakpoint within the procedure. If the procedure is unnamed, 0 is returned.

"local"

Returns the first local variable in the procedure. This establishes a context for the *next* and *previous* parameters. TPU\$_NONAMES is returned if there are no local variables.

"next"

Returns the next parameter, local variable, or breakpoint. Before using GET_INFO (DEBUG, "next"), you must use one of the following built-ins:

- GET_INFO (DEBUG, "local")
- GET_INFO (DEBUG, "breakpoint")
- GET_INFO (DEBUG, "parameter")

TPU\$_NONAMES is returned if there are no more.

"parameter"

Returns the first parameter of the procedure. GET_INFO (DEBUG, "parameter") causes the DECTPU Debugger to construct a list of all the formal parameters of the procedure you are

debugging. Once this list is constructed, you can use GET_INFO (DEBUG, "next") and GET_INFO (DEBUG, "previous"). DECTPU signals TPU\$_NONAMES if the procedure you are debugging does not have any parameters.

"previous"

Returns the previous parameter, local variable, or breakpoint. TPU\$_NONAMES is returned if there are no more.

"procedure"

Returns a string that is the name of the procedure containing the breakpoint. The null string is returned if the procedure has no name.

Return Values

contents

Returns requested information about the contents you specify.

integer

Returns requested information about the integer you specify.

parameter

Returns requested information about the parameter you specify.

string

Returns requested information about the string you specify.

variable

Returns requested information about the variable you specify.

Description

The GET_INFO (DEBUG) procedure returns information about the status of a debugging session when you are using the DECTPU Debugger.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (DEFINED_KEY)

Syntax

$$\text{keyword} := \text{GET_INFO} (\text{DEFINED_KEY} \left\{ \begin{array}{l} \text{"first",} \\ \text{"last",} \\ \text{"next",} \\ \text{"previous",} \end{array} \right\}, \text{string})$$

Parameters

"first"

Returns a keyword that is the key name of the first key in the specified key map or key map list.

"last"

Returns a keyword that is the key name of the last key in the specified key map or key map list.

"next"

Returns a keyword that is the key name of the next key in the specified key map or key map list.
Returns 0 if last. Use *"first"* before *"next"*.

"previous"

Returns a keyword that is the key name of the previous key in the specified key map or key map list.
Returns 0 if first. Use *"last"* before *"previous"*.

string

The string that specifies the name of either the key map or key map list to be searched.

Return Value

Returns the requested information about the keyword you specify.

Description

The GET_INFO (DEFINED_KEY) procedure returns a keyword that is the key name of a specified key.

"Current" is not valid when the first parameter is DEFINED_KEY or KEY_MAP, although it is valid when the first parameter is KEY_MAP_LIST.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (integer_variable)

Syntax

```
string := GET_INFO (integer, "name")
```

Parameters

integer

Returns an integer that is the equivalent of a DECTPU keyword. When you use GET_INFO (integer, "name"), the built-in returns the string representation of the keyword that is equivalent to the specified integer.

For example, the following statement assigns the string *process* to the variable *equiv_string*:

```
equiv_string := GET_INFO (10, "name");
```

The value 10 is the integer equivalent of the PROCESS keyword.

You should not use the integer equivalents of keywords in DECTPU code. VSI does not guarantee that the existing equivalences between integers and keywords will always remain the same.

"name"

Returns the string equivalent of the specified integer or keyword.

Return Value

Returns the string representation of any integer that is an equivalent of a keyword.

Description

The GET_INFO (integer_variable) procedure returns the string representation of any integer that is an equivalent of a keyword.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO. See also the description of GET_INFO (any_keyword).

GET_INFO (KEY_MAP)

Syntax

$$\left\{ \begin{array}{l} \text{integer} \\ \text{string} \end{array} \right\} := \text{GET_INFO} (\text{KEY_MAP}, \left\{ \begin{array}{l} \text{"first"} \\ \text{"last"} \\ \text{"next"} \\ \text{"previous"} \end{array} \right\}, \text{name_string})$$

Parameters

"first"

Returns a string that is the name of the first key map in the key map list; returns 0 if there is none.

"last"

Returns a string that is the name of the last key map in the key map list; returns 0 if there is none.

"next"

Returns a string that is the name of the next key map in the key map list; returns 0 if there is none. Use "first" before "next".

"previous"

Returns a string that is the name of the previous key map in the key map list; returns 0 if there is none. Use "last" before "previous".

name_string

The string that specifies the name of either the key map or key map list to be searched.

Return Values**integer**

Returns requested information about the integer you specify.

string

Returns requested information about the string you specify.

Description

The GET_INFO (KEY_MAP) procedure returns information about a key map in a specified key map list. GET_INFO (KEY_MAP) takes a string as a third parameter. The string specifies the name of the key map list to be searched.

The parameter *"current"* is not valid when the first keyword is DEFINED_KEY or KEY_MAP, although it is valid when the first keyword is KEY_MAP_LIST.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (KEY_MAP_LIST)**Syntax**
$$\left\{ \begin{array}{l} \text{integer} \\ \text{string} \end{array} \right\} := \text{GET_INFO} \left(\text{KEY_MAP_LIST}, \left\{ \begin{array}{l} \text{"current"} \\ \text{"first"} \\ \text{"last"} \\ \text{"next"} \\ \text{"previous"} \end{array} \right\} \right)$$
Parameters**"current"**

Returns a string that is the name of the current key map list; returns 0 if there is none.

"first"

Returns a string that is the name of the first key map list; returns 0 if there is none.

"last"

Returns a string that is the name of the last key map list; returns 0 if there is none.

"next"

Returns a string that is the name of the next key map list; returns 0 if there is none. Use *"current"* or *"first"* before *"next"*.

"previous"

Returns a string that is the name of the previous key map list; returns 0 if there is none. Use *"current"* or *"last"* before *"previous"*.

Return Values**integer**

Returns requested information about the integer you specify.

string

Returns requested information about the string you specify.

Description

The GET_INFO (KEY_MAP_LIST) procedure returns information about a key map list.

The parameter *"current"* is not valid when the first keyword is DEFINED_KEY or KEY_MAP, although it is valid when the first keyword is KEY_MAP_LIST.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (marker_variable)**Syntax**

$$\left\{ \begin{array}{l} \text{buffer} \\ \text{integer} \\ \text{keyword} \end{array} \right\} := \text{GET_INFO} (\text{marker_variable}, \left\{ \begin{array}{l} \text{"before_bol"} \\ \text{"beyond_eob"} \\ \text{"beyond_eol"} \\ \text{"bound"} \\ \text{"buffer"} \\ \text{"display_value"} \\ \text{"left_margin"} \\ \text{"middle_of_tab"} \\ \text{"offset"} \\ \text{"offset_column"} \\ \text{"record_number"} \\ \text{"right_margin"} \\ \text{"unmodifiable_records"} \\ \text{"video"} \\ \text{"within_range", range} \end{array} \right\})$$
Parameters**marker_variable**

The marker for which the information is requested.

"before_bol"

Returns 1 if the specified marker is located before the beginning of a line; returns 0 if it is not.

"beyond_eob"

Returns 1 if the specified marker is located beyond the end of a buffer; returns 0 if it is not.

"beyond_eol"

Returns 1 if the specified marker is located beyond the end of a line; returns 0 if it is not.

"bound"

Returns 1 if the specified marker is attached to a character; returns 0 if the marker is free. For more information on bound and free markers, see the *Guide to the DEC Text Processing Utility*.

"buffer"

Returns the buffer in which the marker is located.

"display_value"

Returns the display value of the record in which the specified marker is located. For more information about display values, see the descriptions of the SET (DISPLAY_VALUE) and SET (RECORD_ATTRIBUTE S) built-in procedures.

"left_margin"

Returns an integer that is the current left margin setting of the line containing the marker.

"middle_of_tab"

Returns an integer (1 or 0) that indicates whether the marker is located in the white space created by a tab.

"offset"

Returns an integer that is the number of characters between the left margin and the marker. The left margin is counted as character 0. A tab is counted as one character, regardless of width. Window shifts have no effect on the value returned when you use *"offset"*. The value returned has no relation to the visible screen column in which the character bound to the marker is displayed.

"offset_column"

Returns an integer that is the screen column in which DECTPU displays the character to which the marker is bound. When calculating this value, DECTPU does not take window shifts into account; DECTPU assumes that any window mapped to the current buffer is not shifted. The value returned when you use *"offset_column"* does reflect the location of the left margin and the width of tabs preceding the editing point. In contrast, the value returned when you use *"offset"* is not affected by the location of the left margin or the width of tabs.

"record_number"

Returns an integer that is the number associated with the record (line) containing the specified marker.

A record number indicates the location of a record in a buffer. Record numbers are dynamic. As you add or delete records, DECTPU changes the number associated with a particular record, as appropriate. DECTPU counts each record in a buffer, regardless of whether the line is visible in a window or whether the record contains text. GET_INFO (marker, "record_number") counts the end-of-buffer text as a record if the specified marker is on the end-of-buffer text, but GET_INFO (buffer, "record_count") never counts the end-of-buffer text as a record. Thus, it is possible for the value returned by GET_INFO (buffer, "record_count") to be one less than the maximum value returned by GET_INFO (marker, "record_number").

"right_margin"

Returns an integer that is the current right margin setting of the line containing the marker.

"unmodifiable_records"

Returns 1 if the record that contains the specified marker is unmodifiable; returns 0 if the record is modifiable.

"video"

Returns a keyword that is the video attribute of the marker; returns 0 if the marker is a free marker.

"within_range"

Returns an integer (1 or 0) that indicates whether the marker is in the range specified by the third parameter.

Return Values

buffer

Returns requested information about the buffer you specify.

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

Description

The GET_INFO (marker_variable) procedure returns information about a specified marker.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (mouse_event_keyword)

Syntax

$$\left\{ \begin{array}{l} \text{integer} \\ \text{window} \end{array} \right\} := \text{GET_INFO} \left(\text{mouse_event_keyboard}, \left\{ \begin{array}{l} \text{"mouse_button"} \\ \text{"window"} \end{array} \right\} \right)$$

Parameters

"mouse_button"

Returns an integer that is the number of the mouse button specified with a mouse event keyword.

Table 2.4 lists the valid keywords for the first parameter when you use *"mouse_button"* as the second parameter.

Table 2.4. DECTPU Keywords Representing Mouse Events

M1UP	M2UP	M3UP	M4UP	M5UP
M1DOWN	M2DOWN	M3DOWN	M4DOWN	M5DOWN
M1DRAG	M2DRAG	M3DRAG	M4DRAG	M5DRAG
M1CLICK	M2CLICK	M3CLICK	M4CLICK	M5CLICK
M1CLICK2	M2CLICK2	M3CLICK2	M4CLICK2	M5CLICK2
M1CLICK3	M2CLICK3	M3CLICK3	M4CLICK3	M5CLICK3
M1CLICK4	M2CLICK4	M3CLICK4	M4CLICK4	M5CLICK4
M1CLICK5	M2CLICK5	M3CLICK5	M4CLICK5	M5CLICK5

"window"

Returns the window in which the downstroke occurred that started the current drag operation. Returns 0 if no drag operation is in progress for the specified mouse button when the built-in is executed.

The valid keywords for the first parameter when you use *"window"* as the second parameter are M1DOWN, M2DOWN, M3DOWN, M4DOWN, and M5DOWN.

Return Values

integer

Returns requested information about the integer you specify.

window

Returns requested information about the window you specify.

Description

The GET_INFO (mouse_event_keyword) procedure returns information about a mouse event. A *mouse_event_keyword* is a keyword that represents a single click, multiple clicks, upstroke, downstroke, or drag of a mouse button.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

Examples

1. In the following example, the statement causes DECTPU to assign the value 3 to the variable *x*:

```
x := GET_INFO (M3CLICK2, "mouse_button");
```

2. In the following example, when bound to MIDRAG, that procedure is called by DECTPU to respond to a drag event by checking whether you have dragged the mouse across window boundaries; if you have, the procedure displays a message. If not, the procedure outputs a message that you are dragging the mouse.

```
PROCEDURE sample_m1_drag

LOCAL the_window,
      new_window,
      column,
      row,
      temp;

the_window := GET_INFO (M1DOWN, "window");
IF the_window = 0
THEN
  RETURN (FALSE)
ENDIF;

LOCATE_MOUSE (new_window, column, row);

IF the_window <> new_window
THEN
  MESSAGE ("Invalid drag of pointer across window boundaries.");
ENDIF;
MESSAGE ("Dragging the mouse...");
ENDPROCEDURE;
```

GET_INFO (PROCEDURES)

Syntax

$$\text{integer} := \text{GET_INFO} (\text{PROCEDURES}, \left\{ \begin{array}{l} \text{"defined"} \\ \text{"minimum_parameters"} \\ \text{"maximum_parameters"} \end{array} \right\}, \text{string})$$

Parameters

"defined"

Returns an integer (1 or 0) that indicates whether the specified procedure is user defined.

"minimum_parameters"

Returns an integer that is the minimum number of parameters required for the specified user-defined procedure.

"maximum_parameters"

Returns an integer that is the maximum number of parameters required for the specified user-defined procedure.

string

A string that is the name of the procedure about which you want information.

Return Value

Returns the requested information about the integer you specify.

Description

The GET_INFO (PROCEDURES) procedure returns information about a specified procedure. GET_INFO (PROCEDURES) takes a string as a third parameter. The string specifies the name of the procedure about which you are requesting information.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (PROCESS)

Syntax

$$\text{process} := \text{GET_INFO} (\text{PROCESS}, \left\{ \begin{array}{l} \text{"current"} \\ \text{"first"} \\ \text{"last"} \\ \text{"next"} \\ \text{"previous"} \end{array} \right\})$$

Parameters

"current"

Returns the current process in DECTPU's internal list of processes. You can use GET_INFO (PROCESS, "current") only after you have used GET_INFO (PROCESS, "first") or GET_INFO (PROCESS, "last"). The built-in returns 0 if you do not use these GET_INFO built-ins in the correct order.

"first"

Returns the first process in DECTPU's internal list of processes; returns 0 if there is none.

"last"

Returns the last process in DECTPU's internal list of processes; returns 0 if there is none.

"next"

Returns the next process in DECTPU's internal list of processes; returns 0 if there are no more processes. Use "first" before "next".

"previous"

Returns the preceding process in DECTPU's internal list of processes; returns 0 if there is no previous process. Use "last" before "previous".

Return Value

Returns the requested information about the process you specify.

Description

The GET_INFO (PROCESS) procedure returns a specified process in DECTPU's internal list of processes.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (process_variable)

Syntax

$$\left\{ \begin{array}{l} \text{buffer} \\ \text{integer} \end{array} \right\} := \text{GET_INFO} (\text{process_variable}, \left\{ \begin{array}{l} \text{"buffer"} \\ \text{"pid"} \end{array} \right\})$$

Parameters

"buffer"

Returns the buffer associated with the process.

"pid"

Returns an integer that is the process identification number.

Return Values

buffer

Returns requested information about the buffer you specify.

integer

Returns requested information about the integer you specify.

Description

The GET_INFO (process_variable) procedure returns information about a specified process.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (range_variable)

Syntax

$$\left\{ \begin{array}{l} \text{buffer} \\ \text{keyword} \end{array} \right\} := \text{GET_INFO} (\text{range_variable}, \left\{ \begin{array}{l} \text{"buffer"} \\ \text{"unmodifiable_records"} \\ \text{"video"} \end{array} \right\})$$

Parameters

"buffer"

Returns the buffer in which the range is located.

"unmodifiable_records"

Returns 1 if the specified range contains one or more unmodifiable records; returns 0 if no unmodifiable records are present in the specified range.

"video"

Returns a keyword that is the video attribute of the range.

Return Values

buffer

Returns requested information about the buffer you specify.

keyword

Returns requested information about the keyword you specify.

Description

The GET_INFO (range_variable) procedure returns information about a specified range.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (SCREEN)

Syntax

<pre> array integer keyword learn_sequence PRIMARY program SECONDARY selection_name string </pre>	<pre> := GET_INFO (SCREEN, </pre>	<pre> "active_area" "ansi_crt" "auto_repeat" "avo" "client_message" "client_message_routine" "cross_window_bounds" "current_column" "current_row" "dec_crt" "dec_crt2" "dec_crt3" "dec_crt4" "decwindows" "default_file" "detached_action" "detached_reason" "edit_mode" "eightbit" "event", GLOBAL_SELECT "first_input" "first_input_routine" "global_select", { PRIMARY SECONDARY } selection_name } "grab_routine" { GLOBAL_SELECT INPUT_FOCUS } "icon_name" "input_focus" "jump_scroll" "length" "line_editing" "motif" "mouse" "new_length" "new_width" "old_length" "old_width" "pixel_length" "pixel_width" "pop_up_parent_widget" "prompt_length" "prompt_row" "read_routine", GLOBAL_SELECT "screen_limits" "screen_update" "scroll" "time", GLOBAL_SELECT "ungrab_routine," { GLOBAL_SELECT INPUT_FOCUS } "visible_length" "vk100" "vt100" "vt200" "vt300" "vt400" "widget" "width" "xui" </pre>
---	-----------------------------------	--

Parameters

"active_area"

Returns an array that contains information on the location and dimensions of the application's active area; returns the integer 0 if there is no active area. The active area is the region in a window in which DECTPU ignores movements of the pointer cursor for purposes of distinguishing clicks from drags. When you press down a mouse button, DECTPU interprets the event as a click if the upstroke occurs in the active area with the downstroke. If the upstroke occurs outside the active area, DECTPU interprets the event as a drag operation.

A DECTPU layered application can have only one active area at a time, even if the application has more than one window visible on the screen. An active area is valid only if you are pressing a mouse button. The default active area occupies one character cell. By default, the active area is located on the character cell pointed to by the cursor.

GET_INFO (SCREEN, "active_area") returns five pieces of information about the active area in integer-indexed elements of the returned array. You need not use the CREATE_ARRAY built-in procedure before using GET_INFO (SCREEN, "active_area"); DECTPU assigns a properly structured array to the return variable you specify. The structure of the array is as follows:

Array Element	Contents
array {1}	Window that contains the active area
array {2}	Column that forms the leftmost edge of the active area
array {3}	Row that forms the top edge of the active area
array {4}	Width of the active area, expressed in columns
array {5}	Height of the active area, expressed in rows

"ansi_crt"

Returns an integer (1 or 0) that indicates whether the terminal is an ANSI_CRT.

"auto_repeat"

Returns an integer (1 or 0) that indicates whether the terminal's autorepeat feature is on.

"avo"

Returns an integer (1 or 0) that indicates whether the ADVANCED_VID EO attribute has been set for the terminal.

"client_message"

Returns a keyword that indicates whether DECTPU has received a KILL_SELECTION client message or a STUFF_SELECTION client message. If the call is used when there is no current client message, the integer 0 is returned.

GET_INFO (SCREEN, "client_message") is used in a DECTPU layered or EVE layered application's client message routine. This routine provides the application's response to a client message received from another application.

GET_INFO (SCREEN, "client_message") returns the KILL_SELECTION keyword when you are copying the primary global selection between DECwindows applications.

GET_INFO (SCREEN, "client_message") returns the STUFF_SELECTION keyword when you are copying the secondary global selection between DECwindows applications. For more information, see SEND_CLIENT_MESSAGE.

"client_message_routine"

Returns the program or learn sequence designated as an application's client message action routine; returns 0 if none is designated.

"cross_window_bounds"

Returns an integer (1 or 0) that indicates whether the CURSOR_VERTICAL built-in procedure causes the cursor to cross a window boundary if the cursor is at the top or bottom of the window.

"current_column"

Returns an integer that is the number of the current column.

"current_row"

Returns an integer that is the number of the current row.

"dec_crt"

Returns an integer (1 or 0) that indicates whether the terminal is a DEC_CRT. For more information on this terminal characteristic, see the SET TERMINAL command in your VMS overview documentation.

"dec_crt2"

Returns an integer (1 or 0) that indicates whether the terminal is a DEC_CRT2. For more information on this terminal characteristic, see the SET TERMINAL command in your OpenVMS overview documentation.

"dec_crt3"

Returns an integer (1 or 0) that indicates whether the terminal is a DEC_CRT3. For more information on this terminal characteristic, see the SET TERMINAL command in your VMS overview documentation.

"dec_crt4"

Returns an integer (1 or 0) that indicates whether the terminal is a DEC_CRT4. For more information on this terminal characteristic, see the SET TERMINAL command in your VMS overview documentation.

"decwindows"

Returns 1 if your system is running the DECwindows Motif updater version of DECTPU. Returns 0 if you are using the character-cell updater. For more information about the DECwindows version of DECTPU, see the *Guide to the DEC Text Processing Utility*.

"default_file"

Returns the name of the X resource file merged into the display's database during editor initialization or by SET (DEFAULT_FILE).

"detached_action"

Returns the current detached action routine. If no such routine is designated, returns the UNSPECIFIED keyword.

"detached_reason"

Returns a bit-encoded integer indicating which of the five possible detached states the cursor is in.

VSI recommends that you use the DECTPU predefined constants rather than the actual integers to refer to the reasons for detachment. Table 2.5 shows the correspondence of constants, integers, and reasons.

Table 2.5. Detached Cursor Flag Constants

Constant	Value	Reason
TPU\$K_OFF_LEFT	1	The editing point is off the left side of the current window.
TPU\$K_OFF_RIGHT	2	The editing point is off the right side of the current window.
TPU\$K_INVISIBLE	4	The editing point is on a record that is invisible in the current window.
TPU\$K_DISJOINT	8	The current buffer is not mapped to the current window.
TPU\$K_UNMAPPED	16	No current window exists.
TPU\$K_NO_UPDATE	32	The current window is a no-update window.

You can set TPU\$K_INVISIBLE in combination with either the TPU\$K_OFF_LEFT or TPU\$K_OFF_RIGHT flags.

You can set TPU\$K_NO_UPDATE in conjunction with any other detached reason, with the exception of TPU\$K_UNMAPPED. Use of TPU\$K_UNMAPPED is a detached cursor situation because, with its use, the cursor does not accurately reflect the editing point within the current buffer. Applications that use "no_update" windows should trap the TPU\$K_UNMAPPED detached cursor reason, and position to a normal window. EVE traps this condition in its detached cursor action routine and sets its position to the topmost normal window.

"edit_mode"

Returns an integer (1 or 0) that indicates whether the terminal is set to edit mode.

"eightbit"

Returns an integer (1 or 0) that indicates whether the terminal uses 8-bit characters.

"event"

Use with DECwindows only.

When you use "event" as the second parameter, you must specify the GLOBAL_SELECT keyword as the third parameter. GLOBAL_SELECT indicates that GET_INFO is to supply information about a global selection.

If called from within a global selection grab or ungrab routine, GET_INFO (SCREEN, "event", GLOBAL_SELECT) identifies the global selection that was grabbed or lost. GET_INFO (SCREEN, "event", GLOBAL_SELECT) returns a keyword if the global selection was the primary or secondary selection. The built-in returns a string naming the global selection if the grab or ungrab involves a global selection other than the primary or secondary selection.

If called from within a routine that responds to requests for information about a global selection, GET_INFO (SCREEN, "event", GLOBAL_SELECT) returns an array. The array contains the information an application needs to respond to the request for information about the global selection. The array contains the following information:

array {1}	The PRIMARY keyword, the SECONDARY keyword, or a string. This element identifies the global selection about which information was requested.
array {2}	A string. This element identifies the global selection property about which information has been requested.

The GET_INFO (SCREEN, "event") built-in returns 0 if the built-in is not responding to a grab, an ungrab, or a selection information request.

For more information about grabbing and ungrabbing a global selection, see the *VMS DECwindows Guide to Application Programming*.

"first_input"

Use with DECwindows only.

Returns integer 1 if DECTPU has received its first key or button event; otherwise returns 0.

"first_input_routine"

Use with DECwindows only.

Returns the program or learn sequence that implements the application's first input action routine. Returns 0 if no input action routine is set.

"global_select"

Use with DECwindows only.

Returns the integer 1 if DECTPU currently owns the specified global selection; returns 0 if it does not.

You must specify one of the following parameters as a third parameter to GET_INFO (SCREEN, "global_select"):

PRIMARY	Keyword that directs DECTPU to get information on the primary global selection
SECONDARY	Keyword that directs DECTPU to get information on the secondary global selection

selection_name	String that identifies the global selection about which DECTPU is to get information
----------------	--

For more information about grabbing and ungrabbing a global selection, see the *VMS DECwindows Guide to Application Programming*.

"grab_routine"

Use with DECwindows only.

Returns the program or learn sequence designated as the application's global selection or input focus grab routine. Returns the integer 0 if the requested grab routine is not present.

You must specify one of the following keywords as a third parameter to GET_INFO (SCREEN, "grab_routine"):

GLOBAL_SELECT	Keyword indicating that GET_INFO is to return the global selection grab routine
INPUT_FOCUS	Keyword indicating that GET_INFO is to return the input focus grab routine

"icon_name"

Use with DECwindows only.

Returns the string used as the layered application's name in the DECwindows icon box.

"input_focus"

Use with DECwindows only.

Returns an integer (1 or 0) that indicates whether DECTPU currently owns the input focus. Input focus is the ability to process user input from the keyboard.

"jump_scroll"

Returns an integer (1 or 0) that indicates whether the SET (SCROLLING, JUMP) built-in procedure has been used to direct DECTPU to use the JUMP mode of scrolling (that is, to perform all currently specified scrolling before repainting the screen).

"length"

Returns an integer that is the current length of the screen (in rows).

"line_editing"

Returns information that indicates whether you are using the insert or overstrike method of line editing; returns 0 if you are using neither method. In DECwindows DECTPU, this parameter always returns 0.

"motif"

Returns 1 if DECTPU is using the VMS Motif screen updater TPU\$MOTIFSHR; returns 0 if any other screen updater is in use.

"mouse"

Returns an integer (1 or 0) that indicates whether DECTPU's mouse support capability is turned on.

"new_length"

Use with DECwindows only.

Returns an integer that is the length (in rows) of the screen after the resize action routine is executed.

Resize action routines should use the length returned by GET_INFO (SCREEN, "new_length") to determine the length of their windows. If the call is made outside a resize action routine, this length is the same as the current length of the screen.

"new_width"

Use with DECwindows only.

Returns an integer that is the width (in columns) of the screen after the resize action routine is executed.

Resize action routines should use the length returned by GET_INFO (SCREEN, "new_width") to determine the width of their windows. If the call is made outside a resize action routine, this width is the same as the current width of the screen.

"old_length"

Use with DECwindows only.

Returns an integer that is the length (in rows) of the screen before the most recent resize event.

The *"old_length"* value is initially set to the length of the screen at startup. This value is reset after DECTPU processes a resize event and before DECTPU executes the resize action routine.

"old_width"

Use with DECwindows only.

Returns the width (in columns) of the screen before the most recent resize event. The *"old_width"* value is initially set to the width of the screen at startup. This value is reset after DECTPU processes a resize event and before DECTPU executes the resize action routine.

"original_length"

Returns an integer that is the number of lines the screen had when DECTPU was invoked.

"original_width"

Returns an integer that is the width of the screen when DECTPU was invoked.

"pixel_length"

Use with DECwindows only.

Returns the length (height) of the current display device in pixels. If you use this parameter on a character-cell terminal, you get the error message TPU\$_REQUIRESDECW.

"pixel_width"

Use with DECwindows only.

Returns the width of the current display device in pixels. If you use this parameter on a character-cell terminal, you get the error message TPU\$_REQUIRESDECW.

"pop_up_parent_widget"

Use with DECwindows only.

Returns the Motif parent widget for application pop-up widgets. You must specify the parent widget when you create pop-up widgets that use the CREATE_WIDGET built-in procedure.

"prompt_length"

Returns an integer that is the number of lines in the prompt area.

"prompt_row"

Returns an integer that is the screen line number at which the prompt area begins.

"read_routine"

Use with DECwindows only.

Returns the program or learn sequence that DECTPU executes when it owns a global selection and another application has requested information about that selection. If the application has not specified a global selection read routine, 0 is returned.

You must specify the GLOBAL_SELECT keyword as the third parameter to GET_INFO (SCREEN, "read_routine"). GLOBAL_SELECT indicates that GET_INFO is to return the global selection read routine.

"screen_limits"

Returns an integer-indexed array that specifies the minimum and maximum screen length and width.

An integer-indexed array uses four elements to specify the minimum and maximum screen width and length. The array indices and the contents of their corresponding elements are as follows:

Array Element	Contents
array {1}	Minimum screen width, in columns. This value must be at least 0 and less than or equal to the maximum screen width. The default value is 0.
array {2}	Minimum screen length, in lines. This value must be at least 0 and less than or equal to the maximum screen length. The default value is 0.
array {3}	Maximum screen width, in columns. This value must be greater than or equal to the minimum screen width and less than or equal to 255. The default value is 255.
array {4}	Maximum screen length, in lines. This value must be greater than or equal to the minimum

screen length and less than or equal to 255. The default value is 255.
--

"screen_update"

Returns an integer (1 or 0) that indicates whether screen updating is turned on.

"scroll"

Returns an integer (1 or 0) that indicates whether the terminal has scrolling regions. For more information on scrolling regions, see the description of the SET (SCROLLING) built-in procedure.

"time"

Use with DECwindows only.

Returns a string in OpenVMS delta time format that indicates the amount of time after requesting global selection information that DECTPU waits for a reply. When the time has expired, DECTPU assumes the request will not be answered.

You must specify the GLOBAL_SELECT keyword as the third parameter to GET_INFO (SCREEN, "time").

"ungrab_routine"

Use with DECwindows only.

Returns the program or learn sequence that DECTPU executes when it loses ownership of a global selection or of the input focus. Returns 0 if the requested ungrab routine is not present.

You must specify one of the following keywords as a third parameter to GET_INFO (SCREEN, "ungrab_routine"):

GLOBAL_SELECT	Keyword indicating that GET_INFO is to return the global selection ungrab routine
INPUT_FOCUS	Keyword indicating that GET_INFO is to return the input focus ungrab routine

"visible_length"

Returns an integer that is the page length of the terminal.

"vk100"

Returns an integer (1 or 0) that indicates whether the terminal is a GIGI.

"vt100"

Returns an integer (1 or 0) that indicates whether the terminal is in the VT100 series.

"vt200"

Returns an integer (1 or 0) that indicates whether the terminal is in the VT200 series.

"vt300"

Returns an integer (1 or 0) that indicates whether the terminal is in the VT300 series.

"vt400"

Returns an integer (1 or 0) that indicates whether the terminal is in the VT400 series.

"widget"

Use with DECwindows only.

Returns DECTPU's top-level widget regardless of the active DECwindows user interface. In Motif this widget is created by XtAPPCreateShell. In character-cell environments, this built-in signals the error message TPU\$_REQUIRESDECW.

"width"

Returns an integer that is the current physical width of the screen.

"xui"

Returns 0. DECTPU no longer supports the XUI interface.

Return Values

array

Returns requested information about the array you specify.

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

learn_sequence

Returns requested information about the learn sequence you specify.

PRIMARY

Returns requested information about the primary global selection you specify.

program

Returns requested information about the program you specify.

SECONDARY

Returns requested information about the secondary global selection you specify.

selection_name

Returns requested information about the selection name you specify.

string

Returns requested information about the string you specify.

Description

The GET_INFO (SCREEN) procedure returns information about the screen.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (string_variable)

Syntax

$$\left. \begin{array}{l} \text{array} \\ \text{integer} \\ \text{keyword} \\ \text{program} \end{array} \right\} := \text{GET_INFO} (\text{string_variable}, \left. \begin{array}{l} \text{"journal"} \\ \text{"pre_key_procedure"} \\ \text{"post_key_procedure"} \\ \text{"self_insert"} \\ \text{"sheft_key"} \text{"undefined_key"} \end{array} \right\})$$

Parameters

"journal"

Returns an array that contains information about the buffer-change journal file whose name you specify with the string parameter. If the specified file is not a journal file, the integer 0 is returned.

The array indices and the contents of the corresponding elements of the returned array are as follows (all elements are of type string):

Index	Contents of Element
1	Name of the buffer whose contents were journaled.
2	Date and time the journal file was created.
3	Date and time the edit session started.
4	Name of the source file. A source file is a file to which the buffer has been written. The journal file maintains a pointer to the source file. This enables the journal file to retrieve from the source file the buffer contents as they were after the last write operation. If the buffer has not been written out, or if none of the source files is available during recovery, this element contains a null string.
5	Name of the output file associated with the buffer. This is the file name specified with the SET (OUTPUT_FILE) built-in.
6	Name of the original input file associated with the buffer by the CREATE_BUFFER built-in. If there is no associated input file or if the input file is not available during a recovery, this element contains a null string.

7

Identification string for the version of DECTPU that wrote the journal file.

"pre_key_procedure"

Returns the program (stored in the specified key map or key map list) that is called before execution of code bound to keys. Returns 0 if no procedure was defined by SET (PRE_KEY_PROCEDURE).

"post_key_procedure"

Returns the program (stored in the specified key map or key map list) that is called before execution of code bound to keys. Returns 0 if no procedure was defined by SET (POST_KEY_PROCEDURE).

"self_insert"

Returns an integer (1 or 0) that indicates whether printable characters are to be inserted into the buffer if they are not defined. Use the SET (SELF_INSERT) built-in procedure to establish or change this parameter.

"shift_key"

Returns a keyword that is the key name for the key currently used as the shift key. Use the SET (SHIFT_KEY) built-in procedure to establish or change this parameter.

"undefined_key"

Returns the program that is called when an undefined character is entered. Returns 0 if the program issues the default message. Use the SET (UNDEFINED_KEY) built-in procedure to establish or change this parameter.

Return Values

array

Returns requested information about the array you specify.

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

program

Returns requested information about the program you specify.

Description

The GET_INFO (string_variable) procedure returns information about the specified string. The string must be the name of a key map or key map list.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (SYSTEM)

Syntax

$\left. \begin{array}{l} \text{integer} \\ \text{keyword} \\ \text{learn_sequence} \\ \text{program} \\ \text{string} \end{array} \right\} := \text{GET_INFO (SYSTEM,}$	$\left. \begin{array}{l} \text{"bell"} \\ \text{"column_move_vertical"} \\ \text{"default_directory"} \\ \text{"display"} \\ \text{"enable_resize"} \\ \text{"facility_name"} \\ \text{"journaling_frequency"} \\ \text{"journal_file"} \\ \text{"line_number"} \\ \text{"message_action_level"} \\ \text{"message_action_type"} \\ \text{"message_flags"} \\ \text{"operating_system"} \\ \text{"pad_overstruck_tabs"} \\ \text{"record_mode"} \\ \text{"recover"} \\ \text{"resize_action"} \\ \text{"section_file"} \\ \text{"shift_key"} \\ \text{"success"} \\ \text{"system_default"} \\ \text{"timed_message"} \\ \text{"timer"} \\ \text{"traceback"} \\ \text{"update"} \\ \text{"version"} \\ \text{"work_file"} \end{array} \right\})$
---	---

Parameters

"bell"

Returns the ALL keyword if the bell is on for all messages. Returns the BROADCAST keyword if the bell is on for broadcast messages only. Returns 0 if the SET (BELL) feature is off. Use the SET built-in procedure to establish or change this parameter.

"column_move_vertical"

Returns 1 if the MOVE_VERTICAL built-in procedure is set to keep the cursor in the same column as the cursor moves from line to line. Returns 0 if the MOVE_VERTICAL built-in preserves the offset, rather than the column position, from line to line. Use the SET (COLUMN_MOVE_VERTICAL) built-in procedure to establish or change this parameter.

"default_directory"

Returns the name of the current default directory.

"display"

Returns 1 if you have specified the /DISPLAY qualifier or if it is the default; otherwise, returns 0.

"enable_resize"

Returns 1 if resize operations are enabled; otherwise returns 0. By default, resize operations are not enabled. You can turn resizing on or off with the SET (ENABLE_RESIZE) built-in procedure.

"facility_name"

Returns a string that is the current facility name. Use the SET (FACILITY_NAME) built-in procedure to establish or change this parameter.

"informational"

Returns an integer (1 or 0) that indicates whether informational messages are displayed. Use the SET (INFORMATIONAL) built-in procedure to establish or change this parameter.

"journaling_frequency"

Returns an integer that indicates how frequently records are written to the journal file. Use the SET (JOURNALING) built-in procedure to establish or change this parameter.

"journal_file"

Returns a string that is the name of the journal file.

"line_number"

Returns an integer (1 or 0) that indicates whether DECTPU displays the line number at which an error occurred. Use the SET (LINE_NUMBER) built-in procedure to establish or change this parameter.

"message_action_level"

Returns an integer that is the completion status severity level at which DECTPU performs the message action you specify. The valid values, in ascending order of severity, are as follows: 1 (success), 3 (informational), 0 (warning), and 2 (error). Use the SET (MESSAGE_ACTION_LEVEL) built-in procedure to establish or change this parameter.

"message_action_type"

Returns a keyword describing the action to be taken when DECTPU signals an error, warning, or message whose severity level is greater than or equal to the level set with SET (MESSAGE_ACTION_LEVEL). The possible keywords are NONE, BELL, and REVERSE. Use the SET (MESSAGE_ACTION_TYPE) built-in procedure to establish or change this parameter.

"message_flags"

Returns an integer that is the current value of the message flag setting. Use the SET (MESSAGE_FLAGS) built-in procedure to establish or change this parameter.

"operating_system"

Returns a DECTPU keyword that indicates which operating system is in use. When operating on OpenVMS VAX systems, OPENVMS is returned. When operating on OpenVMS Alpha,

OPENVMS_ALPHA is returned. There is no GET_INFO call to determine the type of CPU being used.

"pad_overstruck_tabs"

Returns an integer (1 or 0) that indicates whether DECTPU preserves the white space created by a tab character. Use the SET (PAD_OVERSTRUCK_TABS) built-in procedure to establish or change this parameter.

"record_mode"

Returns a keyword for the default record format and attributes for all files written from buffers having no input file. To change the default record mode for the operating system VARIABLE_CR, use the SET (RECORD_MODE, SYSTEM) built-in procedure. The possible keyword returns, and what record format and attributes they imply, are as follows:

Keyword	Record Format	Record Attributes
VARIABLE_NONE	fab\$c_var	0
VARIABLE_FTN	fab\$c_var	fab\$m_ftn
VARIABLE_CR	fab\$c_var	fab\$m_cr
STREAM	fab\$c_stm	fab\$m_cr
STREAM_LF	fab\$c_stmlf	fab\$m_cr
STREAM_CR	fab\$c_stmcr	fab\$m_cr

"recover"

Returns an integer (1 or 0) that indicates whether a recovery using a keystroke journal file is currently in progress. Be careful when using this built-in— specifying different DECTPU actions during a recovery rather than during an ordinary editing session may cause DECTPU journaling to fail.

"resize_action"

Returns the program or learn sequence designated as the application's resize action routine. Returns the UNSPECIFIED keyword if the requested resize action routine is not present. You can designate a resize action routine by using the SET (RESIZE_ACTION) built-in procedure.

"section_file"

Returns a string that is the name of the section file used when you invoked DECTPU.

"shift_key"

Returns a keyword that is the value of the current shift key set with SET (SHIFT_KEY) for the current buffer.

"success"

Returns an integer (1 or 0) that indicates whether success messages are displayed. Use the SET (SUCCESS) built-in procedure to establish or change this parameter.

"system_default"

Keyword for the operating system's default record format and attributes for all files written from buffers having no input file: VARIABLE_CR for VMS systems.

"timed_message"

Returns a string of text that DECTPU displays at 1-second intervals in the prompt area if the SET (TIMER) feature is on.

"timer"

Returns the integer 1 if SET (TIMER) has been enabled; otherwise returns 0.

"traceback"

Returns an integer (1 or 0) that indicates whether DECTPU displays the call stack for DECTPU procedures when an error occurs. Use the SET (TRACEBACK) built-in procedure to establish or change this parameter.

"update"

Returns an integer that is the update number of this version of DECTPU.

"version"

Returns an integer that is the version number of DECTPU.

"work_file"

Returns a string that is the name of the work file opened during startup.

Return Values

integer

Returns requested information about the integer you specify.

keyword

Returns requested information about the keyword you specify.

learn_sequence

Returns requested information about the learn sequence you specify.

program

Returns requested information about the program you specify.

string

Returns requested information about the string you specify.

Description

The GET_INFO (SYSTEM) procedure returns information about the system.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (WIDGET)

Syntax

$$\left. \begin{array}{l} \text{array} \\ \text{integer} \\ \text{widget} \\ \text{NONE} \end{array} \right\} := \text{GET_INFO} (\text{WIDGET}, \left. \begin{array}{l} \text{"callback_parameters", array} \\ \text{"children", } \left\{ \begin{array}{l} \text{widget} \\ \text{SCREEN} \end{array} \right\}, \text{array} \\ \text{"menu_position", mouse_down_button} \\ \text{"widget_id", } \left\{ \begin{array}{l} \text{parent_widget} \\ \text{SCREEN} \end{array} \right\}, \text{widget_name} \\ \text{"widget_resource_types"} \end{array} \right\})$$

Parameters

"callback_parameters"

Returns the widget that performs the callback, the closure value associated with the widget, and the reason for the callback. In DECwindows documentation, the closure is called the *tag*.

array

An array used to return values for the callback, the closure, and the reason. The array has the following indices of type string: *"widget"*, *"closure"*, and *"reason_code"*. `GET_INFO (WIDGET, "callback_parameters")` places the corresponding values in the array elements. DECTPU automatically creates the array in which the return values are placed.

To use this parameter, specify a variable that has been declared or initialized before you use it. The initial type and value of the variable are unimportant. When `GET_INFO (WIDGET, "callback_parameters")` places the return values in the array, the initial values are lost.

The integer on the left side of the assignment operator indicates whether `GET_INFO` was used correctly.

`GET_INFO (WIDGET, "callback_parameters")` should be used in a widget callback procedure. If you use this built-in outside a widget callback procedure, the value returned is indeterminate. If you use the built-in inside a widget callback procedure and callback information is available, the built-in returns 1.

For more information about callbacks and closure values in DECwindows DECTPU, see the *Guide to the DEC Text Processing Utility*. For general information about using callbacks and closure values, see the VMS overview documentation.

"children"

Returns the number of widget children controlled by the specified widget. The array parameter returns the children themselves. If the `SCREEN` keyword is specified instead of a widget, the built-in returns the number of children controlled by the DECTPU main window widget.

"menu_position"

Returns information about any pop-up widgets that are set for menu positioning when you press the specified mouse button. If no pop-up widgets are set, returns the `NONE` keyword; otherwise, returns an integer-indexed array of all pop-ups set for menu positioning.

mouse_down_button	This keyword (<code>M1DOWN</code> , <code>M2DOWN</code> , <code>M3DOWN</code> , <code>M4DOWN</code> , or <code>M5DOWN</code>) indicates the mouse button associated with the pop-up menus.
-------------------	--

"widget_id"

Returns the widget whose name matches the specified widget name. The remaining parameters are as follows:

parent_widget	The widget that is an ancestor of the widget returned by the GET_INFO (WIDGET) built-in procedure.
SCREEN	A keyword indicating that DECTPU's main window widget is the ancestor of the widget that you want the GET_INFO (WIDGET) built-in procedure to return.
widget_name	<p>A string that is the fully qualified name of the widget you want the built-in to return. To specify this parameter correctly, start the string with either the name of the widget's parent, or the name of a child of the parent. If you used the SCREEN parameter instead of the <i>parent_widget</i> parameter, start the string with the name of a child of that widget.</p> <p>Next, specify the names of the ancestors, if any, that occur in the widget hierarchy between the widget named above and the widget itself. Finally, specify the name of the widget you want the GET_INFO (WIDGET) built-in procedure to return. Separate all widget names with periods.</p> <p>The fully qualified widget name is case sensitive.</p>

"widget_resource_types"

Returns an array indexed by strings that are the widget resource data types supported by DECTPU, such as boolean or callback. Each array element is another array that is integer-indexed from 0, and contains the names of widget resources or resource types that are of the specified data type.

For more information on DECwindows concepts such as parent widgets, ancestor widgets, and the distinction between widget classes and widgets, see the *VMS DECwindows Guide to Application Programming*.

Return Values**array**

Returns requested information about the array you specify.

integer

Returns requested information about the integer you specify.

widget

Returns requested information about the widget you specify.

Description

The GET_INFO (WIDGET) procedure returns information about DECTPU widgets in general or about a specific widget whose name you do not know at the time you use the built-in.

Use the GET_INFO (WIDGET) built-in procedure with DECwindows only.

For general information about using all forms of GET_INFO built-in procedure, see the description of GET_INFO.

Examples

The following example is a simplified version of the EVE EVE\$CALLBACK_DISPATCH procedure. The original version is in SYS\$EXAMPLE S:EVE\$MENUS.TPU. (For more information about using the files in SYS\$EXAMPLES as examples, see Appendix A.)

1. PROCEDURE eve\$callback_dispatch

```

LOCAL    the_program,
         status,
         temp_array;

ON_ERROR
  [TPU$_CONTROL]:
    eve$$x_state_array {eve$$k_command_line_flag} := eve
    $k_invoked_by_key;
    eve$learn_abort;
    ABORT;
  [OTHERWISE]:
    eve$$x_state_array {eve$$k_command_line_flag} := eve
    $k_invoked_by_key;
ENDON_ERROR

IF NOT eve$x_decwindows_active
THEN
  RETURN (FALSE);
ENDIF;

eve$$x_state_array {eve$$k_command_line_flag} := eve$k_invoked_by_menu;

status :=
  GET_INFO (WIDGET, "callback_parameters", temp_array); ! This
  statement using                                     ! GET_INFO
  (WIDGET)                                             ! returns the
  calling                                              ! widget, the
  closure,                                             ! and the
  reason code.

! The following statements make the contents of "temp_array"
! available to all the eve$$widget_xxx procedures

```

```
eve$x_widget := temp_array {"widget"};
                ! This array element contains the widget
                ! that called back.
eve$x_widget_tag := temp_array {"closure"};
                ! This array element contains the widget tag
                ! that is assigned to the widget in the UIL file.
eve$x_widget_reason := temp_array {"reason_code"};
                ! This array element contains callback reason code.

! The next statements get the callback routine from the widget arrays.
loop
    exitif an_array = tpu$k_unspecified; ! silence if no widget matches
    an_array := eve$$x_widget_arrays {an_array};
    the_program := an_array {eve$x_widget_tag};
    if the_program <> tpu$k_unspecified
    then
        execute (the_program);
        eve$$found_post_filter; ! in case menu function moved cursor
    endif;
endloop;

eve$$x_state_array {eve$$k_command_line_flag} := eve$k_invoked_by_key;
RETURN;

ENDPROCEDURE;
```

This version of `EVE$CALLBACK_DISPATCH` handles callbacks from EVE widgets. The statement `GET_INFO (WIDGET, "callback_parameters", temp_array)` copies the following three items into elements of the array *temp_array*:

- Widget that is calling back
- Widget's integer closure
- Reason why the widget is calling back

The array *eve\$\$x_widget_array* contains pointers to all of EVE's callback routines in elements indexed by the appropriate integer closure values. This procedure locates the correct index in the array and executes the corresponding callback routine.

Warning

This simplified version of `EVE$CALLBACK_DISPATCH` does not completely replace the version in existing EVE code. This example is presented solely to illustrate how EVE uses the `GET_INFO (WIDGET, "callback_parameters", array)` built-in procedure in a callback handling procedure.

The following example assigns to the variable *the_text_widget* the widget named by the string *NEW_DIALOG.NEW_TEXT*. The name of the parent widget, *NEW_DIALOG*, is optional. The returned widget is the child of the widget assigned to the variable *new_dialog*.

2.

```
the_text_widget := GET_INFO (WIDGET, "widget_id", new_dialog,
                            "NEW_DIALOG.NEW_TEXT");
```

The following example shows how to use `GET_INFO (WIDGET, "children")` to display the entire hierarchy of widgets known to a DECTPU session:

```
3. PROCEDURE eve_show_widgets           ! Display the widget hierarchy

local
    num_topmost,
    widget_array;

widget_array := 0;
num_topmost := GET_INFO (WIDGET, "children", SCREEN, widget_array);

IF num_topmost > 0
THEN
    show_widget_tree (widget_array, "");
ENDIF;

ENDPROCEDURE;

PROCEDURE show_widget_tree           ! Recursively display the widget tree
    (the_array, the_string)

LOCAL
    child_array,
    highest,
    loop_index,
    num_children;

child_array := 0;
loop_index := 1;
highest := get_info (the_array, "high_index");
LOOP
    EXITIF loop_index > highest;
    MESSAGE (the_string + GET_INFO (the_array {loop_index}, "name")
            + ASCII (%o11)
            + GET_INFO (the_array {loop_index}, "class"));
    num_children := GET_INFO (WIDGET, "children",
                            the_array {loop_index}, child_array);
    IF num_children > 0
    THEN
        show_widget_tree (child_array, the_string + "    ");
    ENDIF;
    loop_index := loop_index + 1;
ENDLOOP;

ENDPROCEDURE;
```

GET_INFO (widget_variable)

Syntax

$\left. \begin{array}{l} \text{array} \\ \text{integer} \\ \text{learn_sequence} \\ \text{program} \\ \text{string} \\ \text{widget} \end{array} \right\} := \text{GET_INFO} (\text{widget_variable},$	$\left\{ \begin{array}{l} \text{"callback_routine"} \\ \text{"class"} \\ \text{"input_focus"} \\ \text{"insertion_position"} \\ \text{"is_managed"} \\ \text{"is_subclass"}, \text{widget_class} \\ \text{"parent"} \\ \text{"resources"} \\ \text{"text"} \\ \text{"widget_info"}, \left\{ \begin{array}{l} \text{array} \\ \text{arg_pair } [], \text{arg_pair} \end{array} \right. \end{array} \right.$
---	---

Parameters

"callback_routine"

Returns the program or learn sequence designated as the application's callback routine for the specified widget. This is the program or learn sequence that DECTPU should execute when a widget callback occurs for the specified widget. For more information about callbacks, see the *Guide to the DEC Text Processing Utility*.

"class"

Returns the name of the class to which the specified widget belongs.

"input_focus"

Returns 1 if the specified widget has input focus; otherwise, it returns 0. You cannot set or get the input_focus state of a widget unless that widget is a shell widget that has a resource named XtNinput.

"insertion_position"

Returns the location of the insertion position in the specified text widget. The insertion position is between characters in a text widget and starts at position 0 when to the left of the first character. Returns the NONE keyword if the specified widget is not a text widget.

"is_managed"

Returns 1 if the specified widget is managed; otherwise, it returns 0.

"is_subclass"

Returns 1 if the specified widget belongs to the class referred to by the specified integer or belongs to a subclass of that class. A 1 value indicates only that the widget is equal to or is a subclass of the specified class. The value does not indicate how far down the class hierarchy the widget's class or subclass is. If the widget is not in the class, or one of its subclasses, this GET_INFO call returns 0.

widget_class	The integer specifying the widget class to use in the subclass test. This value is returned from the DEFINE_WIDGET_CLASS built-in procedure.
--------------	--

"name"

Returns a string that is the name of the specified widget.

"parent"

Returns the parent of the specified widget. If the widget has no parent, the call returns 0.

"resources"

Returns a string-indexed array in which each index is a valid resource name for the specified widget. The corresponding array element is a string that contains the resource's data type and class, separated by a line feed (ASCII (10)).

"text"

Returns a string that is the value of the specified simple text widget. (The value of a text widget is the text you enter into the text widget in response to a prompt in a dialog box.) If the specified widget is not a text widget, DECTPU returns the NONE keyword.

"widget_info"

Returns the current values for one or more resources of the specified widget. The values are returned in the array or series of argument pairs that is passed as the third parameter. The integer on the left side of the assignment operator indicates whether the built-in executed successfully.

The third parameter is either an array or a series of paired arguments, specified as follows:

array	Each array index must be a string that names a valid resource for the specified widget. Resource names are case sensitive. The corresponding array element contains the value of the resource. The array can contain any number of elements.
arg_pair	<p>A string that names a valid resource for the widget followed by a variable to store the value of the resource. Separate the resource name string from the variable with a comma and a space, as follows:</p> <pre>resource_name_string, resource_value</pre> <p>You can fetch as many resources as you want by using multiple pairs of arguments.</p>

If you specify the name of a resource that the widget does not support, DECTPU signals the error TPU\$_ARGMISMATCH.

If the requested resource is a list of items and the list contains no entries, the GET_INFO call uses either the element of the array parameter or the value parameter to return an array that has no elements.

For more information about specifying resources, see the *Guide to the DEC Text Processing Utility*.

Return Values

array

Returns requested information about the array you specify.

integer

Returns requested information about the integer you specify.

learn_sequence

Returns requested information about the learn sequence you specify.

program

Returns requested information about the program you specify.

string

Returns requested information about the string you specify.

widget

Returns requested information about the widget you specify.

Description

The GET_INFO (widget_variable) procedure returns information about a specified widget variable.

Use the GET_INFO (widget_variable) built-in procedure with DECwindows only.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

Examples

The following example executes the callback routine for the widget *eve\$x_replace_dialog*. The statement is valid only after the Replace dialog box has been used at least once because EVE does not create any dialog box until you have invoked it.

1. EXECUTE (GET_INFO (eve\$x_replace_dialog,
"callback_routine"));

The following example displays the name of the widget specified by the variable *eve \$x_replace_dialog*. To confirm that the widget has been created as expected, the procedure also displays a message that identifies the data type of the variable's contents. The procedure is valid only after the *Replace* dialog box has been used at least once because EVE does not create any dialog box until you have invoked it.

A statement that contains the GET_INFO (widget, "name") built-in procedure is useful in code implementing a debugging command that evaluates DECTPU statements, expressions, and variables.

2. PROCEDURE sample_return_name

```
LOCAL status;
```

```

status := GET_INFO (eve$x_replace_dialog,
                    "name");

MESSAGE ("The data type of status is: ");
MESSAGE (STR (GET_INFO(status, "type")));
MESSAGE ("The value of status is: ");
MESSAGE (STR (status));

ENDPROCEDURE;

```

The following example creates an EVE file name dialog box widget and assigns the widget to the variable *eve\$x_needfilename_dialog*. Next, the fragment assigns to the variable *the_value* a string that prompts you for the name of a file to which the buffer's contents should be written. The fragment uses the GET_INFO (WIDGET, "widget_id") built-in procedure to assign the dialog box's label widget to the variable *child_of_box*. Finally, the fragment assigns to the label widget's "labelString" resource the string contained in *the_value*.

```

3.
eve$x_needfilename_dialog := CREATE_WIDGET ("NEEDFILENAME_DIALOG",
                                           eve$x_widget_hierarchy,
                                           SCREEN,
                                           eve$kt_callback_routine);

the_value := "Type filename for writing buffer " +
             get_info (the_buffer, "name");

child_of_box := get_info (WIDGET, "widget_id",
                         eve$x_needfilename_dialog,
                         "NEEDFILENAME_DIALOG.NEEDFILENAME_LABEL");

status := set (WIDGET, child_of_box, "labelString", the_value);

```

GET_INFO (WINDOW)

Syntax

```

window := GET_INFO (WINDOW[S], {
    "current"
    "first"
    "last"
    "next"
    "previous"
})

```

Parameters

"current"

Returns the current window on the screen; returns 0 if there is none. GET_INFO (WINDOW[[S]], "current") always returns the current window, regardless of whether you have first used GET_INFO (WINDOW[[S]], "first") or GET_INFO (WINDOW[[S]], "last").

"first"

Returns the first window in DECTPU's internal list of windows; returns 0 if there is none.

"last"

Returns the last window in DECTPU's internal list of windows; returns 0 if there is none.

"next"

Returns the next window in DECTPU's internal list of windows; returns 0 if there are no more windows in the list. Use *"current"* or *"first"* before *"next"*.

"previous"

Returns the preceding window in DECTPU's internal list of windows; returns 0 if there are no previous windows in the list. Use *"current"* or *"last"* before *"previous"*.

Return Value

Returns the requested information about the window you specify.

Description

The GET_INFO (WINDOW) procedure returns a window from DECTPU's internal list of windows or the current window on the screen. For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

GET_INFO (window_variable)

Syntax

$\left\{ \begin{array}{l} \text{buffer} \\ \text{integer} \\ \text{keyword} \\ \text{string} \\ \text{widget} \\ \text{window} \end{array} \right\} := \text{GET_INFO} (\text{window_variable},$

$\left[\begin{array}{l} , \text{WINDOW} \\ , \text{TEXT} \\ , \text{VISIBLE_WINDOW} \\ , \text{VISIBLE_TEXT} \end{array} \right]$
 $\left[\begin{array}{l} , \text{WINDOW} \\ , \text{TEXT} \\ , \text{VISIBLE_WINDOW} \\ , \text{VISIBLE_TEXT} \end{array} \right]$
 "length"
 "middle_of_tab"
 "next"
 "no_video"
 "no_video_status"
 "original_bottom"
 "original_length"
 "original_top"
 "pad"
 "previous"
 "reverse_status"
 "reverse_video"
 $\left[\begin{array}{l} , \text{WINDOW} \\ , \text{TEXT} \\ , \text{VISIBLE_WINDOW} \\ , \text{VISIBLE_TEXT} \end{array} \right]$
 "right"
 "screen_update"
 "scroll"
 "scroll_amount"
 "scroll_bar", { HORIZONTAL }
 { VERTICAL }
 "scroll_bar_auto_thumb", { HORIZONTAL }
 { VERTICAL }
 "scroll_bottom"
 "scroll_top"
 "shift_amount"
 "special_graphics_status"
 "status_line"
 "status_video"
 "text"
 $\left[\begin{array}{l} , \text{WINDOW} \\ , \text{TEXT} \\ , \text{VISIBLE_WINDOW} \\ , \text{VISIBLE_TEXT} \end{array} \right]$
 "top"
 "underline_status"
 "underline_video"
 "video"
 "visible"
 "visible_bottom"
 "visible_length"
 $\left[\begin{array}{l} , \text{WINDOW} \\ , \text{TEXT} \\ , \text{VISIBLE_WINDOW} \\ , \text{VISIBLE_TEXT} \end{array} \right]$
 "visible_top" "width"

Parameters

"before_bol"

Returns an integer (1 or 0) that indicates whether the cursor is to the left of the current line's left margin. The return value has no meaning if *"beyond_eob"* is true. Returns 0 if the window you specified is not mapped.

"beyond_eob"

Returns an integer (1 or 0) that indicates whether the cursor is below the bottom of the buffer. Returns 0 if the window you specified is not mapped.

"beyond_eol"

Returns an integer (1 or 0) that indicates whether the cursor is beyond the end of the current line. The return value has no meaning if *"beyond_eob"* is true. Returns 0 if the window you specified is not mapped.

"blink_status"

Returns an integer (1 or 0) that indicates whether BLINK is one of the video attributes of the window's status line. Use the SET (STATUS_LINE) built-in procedure to establish or change this parameter.

"blink_video"

Returns an integer (1 or 0) that indicates whether BLINK is one of the video attributes of the window. Use the SET (VID EO) built-in procedure to establish or change this parameter.

"bold_status"

Returns an integer (1 or 0) that indicates whether BOLD is one of the video attributes of the window's status line. Use the SET (STATUS) built-in procedure to establish or change this parameter.

"bold_video"

Returns an integer (1 or 0) that indicates whether BOLD is one of the video attributes of the window. Use the SET (VID EO) built-in procedure to establish or change this parameter.

"bound"

Returns an integer (1 or 0) that indicates whether the cursor is located on a character.

"bottom"

Returns an integer that is the number of the last row or last visible row of the specified window, or the specified window's text area. The window row whose number is returned depends on the keyword you specify as the third parameter. If you do not specify a keyword, the default is TEXT. Valid keywords for the third parameter when the second parameter is "bottom", "left", "length", "right", "top", or "width" are listed in Table 2.6.

Table 2.6. GET_INFO (Window_Variable): Valid Keywords for the Third Parameter

Keyword	Definition
TEXT	A keyword that directs the built-in to return the specified (left, right, top, or bottom) window

Keyword	Definition
	row or column or the number of window rows or columns on which text can be displayed. By specifying TEXT instead of VISIBLE_TEXT, you obtain information about a window's rows and columns even if they are invisible because the window is occluded. If the window is not occluded, the value returned is the same as the value returned with VISIBLE_TEXT.
VISIBLE_TEXT	A keyword that directs the built-in to return the specified (left, right, top, or bottom) visible window row or column or the number of visible window rows or columns on which text can be displayed. When DECTPU determines a window's last visible text row, DECTPU does not consider the status line or the bottom scroll bar to be a text row.
VISIBLE_WINDOW	A keyword that directs the built-in to return the specified (left, right, top, or bottom) visible window row or column or the number of visible window rows or columns in the window.
WINDOW	<p>A keyword that directs the built-in to return the specified (left, right, top, or bottom) window row or column or the number of window rows or columns in the window. By specifying WINDOW instead of TEXT, you get the window's last row or column, even if it cannot contain text because it contains a scroll bar or status line.</p> <p>By specifying WINDOW instead of VISIBLE_WINDOW, you get information about a window's rows and columns even if they are invisible because the window is occluded. If the window is not occluded, the value returned is the same as the value returned with VISIBLE_WINDOW.</p>

GET_INFO (window_variable, "bottom", TEXT) is a synonym for GET_INFO (window_variable, "original_bottom"). The call GET_INFO (window_variable, "bottom", VISIBLE_TEXT) is a synonym for GET_INFO (window_variable, "visible_bottom").

"buffer"

Returns the buffer that is associated with the window; returns 0 if there is none.

"current_column"

Returns an integer that is the column in which the cursor was most recently located.

"current_row"

Returns an integer that is the row in which the cursor was most recently located.

"display_value"

Returns the display value of the specified window.

"key_map_list"

Returns the string that is the name of the key map list associated with the window you specify.

"left"

Returns an integer that is the number of the leftmost column or leftmost visible column of the specified window, or the specified window's text area. The column whose number is returned depends on the keyword you specify as the third parameter. If you do not specify a keyword, the default is TEXT. Valid keywords are shown in Table 2–6.

"length"

Returns an integer that is the number of rows or visible rows in the specified window or the specified window's text area. The number of rows returned depends on the keyword you specify as the third parameter. If you do not specify a keyword, the default is TEXT. Valid keywords are shown in Table 2–6.

"middle_of_tab"

Returns an integer (1 or 0) that indicates whether the cursor is in the middle of a tab. The return value has no meaning if *"beyond_eob"* is true. This call returns 0 if the window you specified is not mapped.

"next"

Returns the next window in DECTPU's internal list of windows; returns 0 if there are no more windows in the list.

"no_video"

Returns an integer (1 or 0) that indicates whether the video attribute of the window is NONE. Use the SET (VIDEO) built-in procedure to establish or change this parameter.

"no_video_status"

Returns an integer (1 or 0) that indicates whether the video attribute of the window's status line is NONE. Use the SET (STATUS) built-in procedure to establish or change this parameter.

"original_bottom"

Returns an integer that is the screen line number of the bottom of the window when it was created or last adjusted (does not include status line or scroll bar). Compaq recommends that you use GET_INFO (window, "bottom", text) to retrieve this information.

"original_length"

Returns an integer that is the number of lines in the window when it was created. The value returned includes the status line. Compaq recommends that you use GET_INFO (window, "length", window) to retrieve this information.

"original_top"

Returns an integer that is the screen line number of the top of the window when it was created.

"pad"

Returns an integer (1 or 0) that indicates whether padding blanks have been displayed from column 1 to the left margin (if the left margin is greater than 1) and from the ends of lines to the right margin. Use the SET (PAD) built-in procedure to establish or change this parameter.

"previous"

Returns the previous window in DECTPU's internal list of windows; returns 0 if there are no previous windows in the list.

"reverse_status"

Returns an integer (1 or 0) that indicates whether REVERSE is one of the video attributes of the window's status line. Use the SET (STATUS) built-in procedure to establish or change this parameter.

"reverse_video"

Returns an integer (1 or 0) that indicates whether REVERSE is one of the video attributes of the window. Use the SET (VIDEO) built-in procedure to establish or change this parameter.

"right"

Returns an integer that is the number of the last column or last visible column of the specified window or the specified window's text area. The window column whose number is returned depends on the keyword you specify as the third parameter. If you do not specify a keyword, the default is TEXT. Valid keywords are shown in Table 2.6.

"screen_update"

Returns the update status of a window. A 0 indicates that updates are off. A 1 indicates that the window is updated normally. See the SET (SCREEN_UPDATE) built-in procedure in this chapter for more information.

"scroll"

Returns an integer (1 or 0) that indicates whether scrolling is enabled for the window. Use the SET (SCROLLING) built-in procedure to establish or change this parameter.

"scroll_amount"

Returns an integer that is the number of lines to scroll. Use the SET built-in procedure to establish or change this parameter.

"scroll_bar"

Use with DECwindows only.

Returns the specified scroll bar widget that implements the scroll bar associated with a window if it exists; otherwise it returns 0.

You must specify the HORIZONTAL or VERTICAL keyword as the third parameter to GET_INFO (window_variable, "scroll_bar"). HORIZONTAL directs DECTPU to return the window's horizontal scroll bar; VERTICAL directs DECTPU to return the window's vertical scroll bar.

"scroll_bar_auto_thumb"

Use with DECwindows only.

Returns an integer (1 or 0) that indicates whether automatic adjustment of the specified scroll bar slider is enabled. Returns 1 if automatic adjustment is enabled and 0 if it is disabled.

You must specify the HORIZONTAL or VERTICAL keyword as the third parameter to GET_INFO (window_variable, "scroll_bar_auto_thumb"). HORIZONTAL directs DECTPU to return information about the window's horizontal scroll bar; VERTICAL directs DECTPU to return information about the window's vertical scroll bar.

"scroll_bottom"

Returns an integer that is the bottom of the scrolling area, an offset from the bottom screen line. Use the SET (SCROLLING) built-in procedure to establish or change this parameter.

"scroll_top"

Returns an integer that is the top of the scrolling area, an offset from the top screen line. Use the SET (SCROLLING) built-in procedure to establish or change this parameter.

"shift_amount"

Returns an integer that is the number of columns the window is shifted to the left.

"special_graphics_status"

Returns an integer (1 or 0) that indicates whether SPECIAL_GRAPHICS is one of the video attributes of the window's status line. Use the SET (STATUS_LINE) built-in procedure to establish or change this parameter.

"status_line"

Returns a string that is the text of the status line; returns 0 if there is none. Use the SET (STATUS_LINE) built-in procedure to establish or change this parameter.

"status_video"

If there is no video attribute or only one video attribute for the window's status line, the appropriate video keyword (NONE, BLINK, BOLD, REVERSE, UNDERLINE, or SPECIAL_GRAPHICS) is returned. If there are multiple video attributes for the window's status line, the integer 1 is returned. If there is no status line for the window, the integer 0 is returned. Use the SET (STATUS_LINE) built-in procedure to establish or change this parameter.

"text"

Returns a keyword that indicates which keyword was used with SET (TEXT). SET (TEXT) controls text display in a window. Valid keywords are as follows: BLANK_TABS, GRAPHIC_TABS, or NO_TRANSLATE.

"top"

Returns an integer that is the number of the first row or first visible row of the specified window or the specified window's text area. The window row whose number is returned depends on the

keyword you specify as the third parameter. If you do not specify a keyword, the default is TEXT. Valid keywords are shown in Table 2–6.

"underline_status"

Returns an integer (1 or 0) that indicates whether UNDERLINE is one of the video attributes of the window's status line. Use the SET (STATUS_LINE) built-in procedure to establish or change this parameter.

"underline_video"

Returns an integer (1 or 0) that indicates whether UNDERLINE is one of the video attributes of the window. Use the SET (VIDEO) built-in procedure to establish or change this parameter.

"video"

If there is no video attribute or only one video attribute for the window, the appropriate video keyword (NONE, BLINK, BOLD, REVERSE, or UNDERLINE) is returned. If there are multiple video attributes for the window, the integer 1 is returned. If you get the return value 1 and you want to know more about the window's video attributes, use the specific parameters, such as *"underline_video"* and *"reverse_video"*.

Use the SET (VIDEO) built-in procedure to establish or change this parameter.

"visible"

Returns an integer (1 or 0) that indicates whether the window is mapped to the screen and whether it is occluded.

"visible_bottom"

Returns an integer that is the screen line number of the visible bottom of the window (does not include status line). Use the ADJUST_WINDOW built-in procedure, create other windows, or map a window to change this value.

Compaq recommends that you use GET_INFO (window, "bottom", visible_text) to retrieve this information.

"visible_length"

Returns an integer that is the visible length of the window (includes status line). This value differs from the value returned by GET_INFO (window_variable, "original_length") in that the value returned by *"visible_length"* is the original length minus the number of window lines (if any) hidden by occluding windows. Use the ADJUST_WINDOW built-in procedure, create other windows, or map a window to change this value.

Compaq recommends that you use GET_INFO (window, "length", visible_window) to retrieve this information.

"visible_top"

Returns an integer that is the screen line number of the visible top of the window. Use the ADJUST_WINDOW built-in procedure, create other windows, or map a window on top of the current window to change this value.

Compaq recommends that you use GET_INFO (window, "top", visible_window) to retrieve this information.

"width"

Returns an integer that is the number of columns or the number of visible columns in the specified window or the specified window's text area. The number of columns returned depends on the keyword you specify as the third parameter. If you do not specify a keyword, the default is TEXT. Valid keywords are shown in Table 2.6.

Use the SET built-in procedure to establish or change this parameter.

Return Values**buffer**

Returns requested information about the buffer you specify.

integer

Returns requested information about the array you specify.

keyword

Returns requested information about the keyword you specify.

string

Returns requested information about the string you specify.

widget

Returns requested information about the widget you specify.

window

Returns requested information about the window you specify.

Description

The GET_INFO (window_variable) procedure returns information about a specified window.

For general information about using all forms of GET_INFO built-in procedures, see the description of GET_INFO.

Examples

1. The following example returns the last line of the window *bottom_window*. The value returned is the line that contains the status line or scroll bar, whichever comes last, if the window has a status line or scroll bar.

```
last_line := GET_INFO (bottom_window, "bottom", WINDOW);
```

2. The following example returns the number of the rightmost column in the current window. The column whose number is returned can be occupied by a vertical scroll bar if one is present. Also, the returned value changes if you widen the window, but not if you move the window without widening it.

```
last_column := GET_INFO (CURRENT_WINDOW, "right", WINDOW);
```

3. The following example returns the number of the first row in the current window. The row number returned is relative to the top of the DECTPU screen. Thus, if the current window is not the top window on the DECTPU screen, the row number returned is not 1.

```
first_row := GET_INFO (CURRENT_WINDOW, "top", WINDOW);
```

HELP_TEXT

Syntax

$$\text{HELP_TEXT (library-file, topic, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\} , \text{buffer)}$$

Parameters

library-file

A string that is the file specification of the help library. The string can be a logical name.

topic

A string that is the initial library topic. If this string is empty, the top level of help is displayed.

ON, 1

A keyword or integer that specifies that the VMS Help utility should prompt you for input.

OFF, 0

A keyword or integer that specifies that prompting should be turned off.

buffer

The buffer to which the help information is written.

Description

The HELP_TEXT procedure provides help information on the topic you specify. You must specify the help library to be used for help information, the initial library topic, the prompting mode for the Help utility, and the buffer to which DECTPU will write the help information. You can enter a complete file specification for the help library as the first parameter. However, if you enter only a file name, the Help utility provides a default device (SYS\$HELP) and default file type (.HLB).

If you do not specify an initial topic as the second parameter, you must enter a null string as a placeholder. The Help utility then displays the top level of help available in the specified library.

When the prompting mode is ON for the HELP_TEXT built-in procedure, the following prompt appears if the help text contains more than one window of information:

Press RETURN to continue ...

Before DECTPU invokes the Help utility, it erases the buffer specified as the help buffer. (In EVE, the buffer to which the help information is written is represented by the variable *help_buffer*.) If the help buffer is associated with a window that is mapped to the screen, the window is updated each time DECTPU prompts you for input. If you set the prompting mode to OFF, the window is not updated.

If *help_buffer* is not associated with a window that is mapped to the screen, the information from the Help utility is not visible.

Signaled Errors

TPU\$_BADKEY	ERROR	Only ON and OFF are allowed.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_NOTMODIFIABLE	WARNING	The output buffer is currently unmodifiable.
TPU\$_OPENIN	ERROR	Error opening help library.
TPU\$_SYSERROR	ERROR	Error activating the help library.
TPU\$_TOOFEW	ERROR	The HELP_TEXT built-in requires four parameters.
TPU\$_TOOMANY	ERROR	You specified more than four parameters.

Examples

1. The code in the following example causes the top level of help information from the SYS \$HELP:TPUHELP.HLB library to be written to the help buffer. The Help utility prompting mode is not turned on.

```
HELP_TEXT ("tpuhelp", "", OFF ,help_buffer)
```

2. This procedure displays information about getting out of help mode on the status line, prompts you for input, and maps *help_buffer* to the screen:

```
! Interactive HELP
```

```
PROCEDURE user_help
```

```
    SET (STATUS_LINE, info_window, UNDERLINE,  
        "Press Ctrl/Z to leave prompts then Ctrl/F to resume editing");
```

```
    MAP (info_window, help_buffer);
```

```
    HELP_TEXT ("USERHELP", READ_LINE ("Topic: "), ON, help_buffer);
```

```
ENDPROCEDURE;
```

INDEX

Syntax

```
integer:=INDEX (string, substring)
```

Parameters

string

The string within which you want to find a character or a substring.

substring

A character or a substring whose leftmost character location you want to find within *string1* .

Return Value

An integer that shows the character position within a string of the substring you specify.

Description

The INDEX procedure locates a character or a substring within a string and returns its location within the string. INDEX finds the leftmost occurrence of *substring* within *string*. It returns an integer that indicates the character position in *string* at which *substring* was found. If *string* is not found, DECTPU returns a 0. The character positions within *string* start at the left with 1.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	INDEX must be on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	INDEX requires two arguments.
TPU\$_TOOMANY	ERROR	INDEX accepts only two arguments.
TPU\$_INVPARAM	ERROR	The arguments to INDEX must be strings.

Examples

1. The following example stores an integer value of 6 in the variable *loc* because the substring "67" is found starting at character position 6 within the string "1234567":

```
loc := INDEX ("1234567", "67")
```

2. The following example uses the INDEX built-in procedure to return true if a given item is an alphanumeric character; otherwise, it returns false. (The list of characters in this example does not include characters that are not in the ASCII range of the DEC Multinational Character Set. However, you can write a procedure that uses such characters because DECTPU supports the DEC Multinational Character Set.) The parameter that is passed to this procedure is assumed to be a single character.

```
PROCEDURE user_is_character (c)

    LOCAL symbol_characters;

    symbol_characters :=
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";

    RETURN INDEX( symbol_characters, c ) > 0;
```

```
ENDPROCEDURE;
```

INT

Syntax

$$\text{integer3} := \text{INT} \left(\begin{array}{l} \text{integer1} \\ \text{keyword} \\ \text{string } [, \text{integer2 }] \end{array} \right)$$

Parameters

integer1

Any integer value. INT accepts a parameter of type integer so you need not check the type of the parameter you supply.

keyword

A keyword whose internal value you want.

string

A string that consists of numeric characters.

integer2

An integer that specifies the radix (base) of the string being converted. The default radix is 10. The other allowable values are 8 and 16.

Return Value

The integer equivalent of the parameter you specify.

Description

The INT procedure converts a keyword or a string that consists of numeric characters into an integer. You can use INT to store an integer value for a keyword or a string of numeric characters in a variable. You can then use the variable name in operations that require integer data types.

INT signals a warning and returns 0 if the string is not a number.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	INT returns a value that must be used.
TPU\$_TOOFEW	ERROR	INT requires one parameter.
TPU\$_TOOMANY	ERROR	INT accepts only one parameter.
TPU\$_ARGMISMATCH		The parameter to INT was not a keyword or string.
TPU\$_INVNUMSTR	WARNING	The string you passed to INT was not a number.

TPU\$_NULLSTRING	WARNING	You passed a string of length 0 to INT.
TPU\$_BADVALUE	ERROR	You specified a value other than 8, 10, or 16 for the radix parameter.

Examples

1. The following example converts the string "12345" into an integer value and stores it in the variable *user_int*:

```
user_int := INT ("12345")
```

2. The following example is used by commands that prompt for integers. The procedure returns true if prompting worked or was not needed; otherwise it returns false. The number that is returned is returned in the output parameter.

```
! Parameters:
!
!   new_number           New integer value - output
!   prompt_string       Text of prompt - input
!   no_value_message     Message printed if you press the
!                       RETURN key to get out of the command - input
!

PROCEDURE user_prompt_number (new_number, prompt_string,
                             no_value_message)

  LOCAL read_line_string;

  ON_ERROR
    IF ERROR = TPU$_NULLSTRING
    THEN
      MESSAGE (no_value_message);
    ELSE
      IF ERROR = TPU$_INVNUMSTR
      THEN
        MESSAGE (FAO ("Don't understand !AS",
                     read_line_string));
      ELSE
        MESSAGE (ERROR_TEXT);
      ENDIF;
    ENDIF;
    user_prompt_number := 0;
  ENDON_ERROR;

  user_prompt_number := 1;
  read_line_string := READ_LINE (prompt_string);

  EDIT (read_line_string, TRIM);
  TRANSLATE (read_line_string, "1", "1");

  new_number := INT (read_line_string);
ENDPROCEDURE;
```

JOURNAL_CLOSE

Syntax

JOURNAL_CLOSE

Parameters

None

Description

JOURNAL_CLOSE None. The JOURNAL_CLOSE procedure closes an open keystroke journal file (if one exists for your session) and saves the journal file. JOURNAL_CLOSE applies only to *keystroke* journaling. Once you specify JOURNAL_CLOSE, DECTPU does not keep a keystroke journal of your work until you specify JOURNAL_OPEN. Calling the JOURNAL_OPEN built-in procedure causes DECTPU to open a new keystroke journal file for your session.

To turn off buffer-change journaling, see the description of the SET (JOURNALING) built-in procedure.

Caution

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

Signaled Errors

TPU\$_TOOMANY	ERROR	JOURNAL_CLOSE accepts no arguments.
---------------	-------	-------------------------------------

JOURNAL_OPEN

Syntax

[string :=] JOURNAL_OPEN (file-name)

Parameters

file-name

A string that is the name of the keystroke journal file created for your editing session.

Return Value

The file specification of the file journaled.

Description

The JOURNAL_OPEN procedure opens a keystroke journal file and starts making a copy of your editing session by recording every keystroke you make. If you invoked DECTPU with the /RECOVER qualifier, then DECTPU recovers the previous aborted session before recording new keystrokes. JOURNAL_OPEN optionally returns a string that contains the file specification of the file journaled. JOURNAL_OPEN applies only to *keystroke* journaling.

DECTPU saves the keystrokes of your editing session by storing them in a buffer. DECTPU writes the contents of this buffer to the file that you specify as a journal file. If DECTPU terminates unexpectedly, you can recover your editing session by using this journal file. To do this, invoke DECTPU with the / RECOVER qualifier. See the *Guide to the DEC Text Processing Utility* for information on recovering files.

To turn on buffer-change journaling, see the description of the SET (JOURNALING) built-in procedure.

By default, DECTPU writes keystrokes to the journal file whenever the journal buffer contains 500 bytes of data. DECTPU also tries to write keystrokes to the journal file when it aborts.

When you recover a DECTPU session, your terminal characteristics should be the same as they were when the journal file was created. If they are not the same, DECTPU informs you what characteristics are different and asks whether you want to continue recovering. If you answer yes, DECTPU tries to recover; however, the different terminal settings may cause differences between the recovered session and the original session.

There are no keystrokes in batch mode. You can use JOURNAL_OPEN nodisplay mode with the / NODISPLAY qualifier; however, when you do this, nothing is journaled.

Caution

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

Signaled Errors

TPU\$_BADJOUFILE	ERROR	JOURNAL_OPEN could not open the journal file.
TPU\$_TOOFEW	ERROR	JOURNAL_OPEN requires one argument.
TPU\$_TOOMANY	ERROR	JOURNAL_OPEN accepts only one argument.
TPU\$_INVPARAM	ERROR	The parameter to JOURNAL_OPEN must be a string.
TPU\$_ASYNCACTIVE	WARNING	You cannot journal with asynchronous handlers declared.
TPU\$_JNLOPEN	ERROR	A journal file is already open.

Examples

1. The following example causes DECTPU to open a file named TEST.FIL as the journal file for your editing session. DECTPU uses your current default device and directory to complete the file specification.

```
JOURNAL_OPEN ("test.fil")
```

2. The following example starts journaling. It can be called from the TPU\$INIT_ PROCEDURE after a file is read into the current buffer.

```
PROCEDURE user_start_journal
```



```

LOCAL default_journal_name, ! Default journal name
      aux_journal_name;    ! Auxiliary journal name derived
                          ! from file name
IF (GET_INFO (COMMAND_LINE, "journal") = 1)
AND
  (GET_INFO (COMMAND_LINE, "read_only") <> 1)
THEN
  aux_journal_name := GET_INFO (CURRENT_BUFFER, "file_name");

  IF aux_journal_name = ""
  THEN
    aux_journal_name := GET_INFO (CURRENT_BUFFER, "output_file");
  ENDIF;

  IF (aux_journal_name = "") or (aux_journal_name = 0)
  THEN
    default_journal_name := "user.TJL";
  ELSE
    default_journal_name := ".TJL";
  ENDIF;

  journal_file := GET_INFO (COMMAND_LINE, "journal_file");
  journal_file := FILE_PARSE (journal_file, default_journal_name,
                             aux_journal_name);
  JOURNAL_OPEN (journal_file);
ENDIF;
ENDPROCEDURE;

```

KEY_NAME

Syntax

$$\text{keyword2} := \text{KEY_NAME} \left(\begin{array}{l} \text{integer} \\ \text{key_name} \\ \text{string} \end{array} \right. \left. \begin{array}{l} \left[\begin{array}{l} \text{SHIFT_KEY} \\ \text{SHIFT_MODIFIED} \\ \text{ALT_MODIFIED} \\ \text{CTRL_MODIFIED} \\ \text{HELP_MODIFIED} \end{array} \right] \left[\dots \right] \left[\begin{array}{l} \text{FUNCTION} \\ \text{KEYPAD} \end{array} \right] \end{array} \right)$$

Parameters

integer

An integer that is either the integer representation of a keyword for a key, or is a value between 0 and 255 that DECTPU interprets as the value of a character in the DEC Multinational Character Set.

key_name

A keyword that is the DECTPU name for a key.

string

A string that is the value of a key from the main keyboard.

SHIFT_KEY

A keyword that specifies that the key name created includes one or more shift keys. The **SHIFT_KEY** keyword specifies the DECTPU shift key, not the key on the keyboard marked Shift. The shift key is also referred to as the GOLD key in EVE. (See the description of the SET (**SHIFT_KEY**) built-in procedure in the *VAX Text Processing Utility Manual*.)

SHIFT_MODIFIED

A keyword that specifies that the key name created by the built-in includes the key marked Shift on the keyboard that toggles between uppercase and lowercase, not the key known as the GOLD key.

Compaq recommends that you avoid using this keyword in the non-DECwindows version of DECTPU. In non-DECwindows DECTPU, when you use this keyword to create a key name, the keyboard cannot generate a corresponding key.

ALT_MODIFIED

A keyword that specifies that the key name created by the built-in includes the Alt key. On most Compaq keyboards, the Alt key is labeled Compose Character.

ALT_MODIFIED modifies only function keys and keypad keys.

Compaq recommends that you avoid using this keyword in the non-DECwindows version of DECTPU. In non-DECwindows DECTPU, when you use this keyword to create a key name, the keyboard cannot generate a corresponding key.

CTRL_MODIFIED

A keyword that specifies that the key name created by the built-in includes the Ctrl key.

CTRL_MODIFIED modifies only function keys and keypad keys.

Compaq recommends that you avoid using this keyword in the non-DECwindows version of DECTPU. In non-DECwindows DECTPU, when you use this keyword to create a key name, the keyboard cannot generate a corresponding key.

HELP_MODIFIED

A keyword that specifies that the key name created by the built-in includes the Help key. **HELP_MODIFIED** modifies only function keys and keypad keys.

Compaq recommends that you avoid using this keyword in the non-DECwindows version of DECTPU. In non-DECwindows DECTPU, when you use this keyword to create a key name, the keyboard cannot generate a corresponding key.

FUNCTION

A parameter that specifies that the resulting key name is to be that of a function key.

KEYPAD

A parameter that specifies that the resulting key name is to be that of a keypad key.

Return Value

A DECTPU keyword to be used as the name of a key.

Description

The `KEY_NAME` procedure returns a DECTPU keyword for a key or a combination of keys, or creates a keyword used as a key name by DECTPU. With `KEY_NAME`, you can create key names that are modified by more than one key. For example, you can create a name for a key sequence that consists of the `GOLD` key, the `Ctrl` key, and an alphanumeric or keypad key.

The `GET_INFO` (`key_name`, "key_modifiers") built-in procedure returns a bit-encoded integer whose value represents the key modifier or combination of key modifiers used to create a given key name. For more information about interpreting the integer returned, see the description of `GET_INFO` (`key_name`, "key_modifiers").

The `GET_INFO` (`keyword`, "name") built-in procedure has been extended to return a string that includes all the key modifier keywords used to create a key name. For more information about fetching the string equivalent of a key name, see the description of `GET_INFO` (`keyword`, "name").

If you specify only one DECTPU key name as an argument to `KEY_NAME`, `KEY_NAME` is sensitive to the case of the argument. For example, the following expressions do not evaluate to the same value:

```
KEY_NAME ("Z");
KEY_NAME ("z");
```

When you use the optional parameter `SHIFT_KEY` with `KEY_NAME`, however, `KEY_NAME` is case insensitive and the following statements return the same keyword:

```
KEY_NAME ("Z", SHIFT_KEY);
KEY_NAME ("z", SHIFT_KEY);
```

Signaled Errors

<code>TPU&_INCKWDCOM</code>	WARNING	Inconsistent keyword combination.
<code>TPU\$_MUSTBEONE</code>	WARNING	String must be one character long.
<code>TPU\$_NOTDEFINABLE</code>	WARNING	Second argument is not a valid reference to a key.
<code>TPU\$_NEEDTOASSIGN</code>	ERROR	<code>KEY_NAME</code> call must be on the right-hand side of an assignment statement.
<code>TPU\$_ARGMISMATCH</code>	ERROR	Wrong type of data sent to the <code>KEY_NAME</code> built-in.
<code>TPU\$_BADKEY</code>	ERROR	<code>KEY_NAME</code> accepts <code>SHIFT_KEY</code> , <code>FUNCTION</code> , or <code>KEYPAD</code> as a keyword argument.
<code>TPU\$_TOOFEW</code>	ERROR	Too few arguments passed to the <code>KEY_NAME</code> built-in.

TPU\$_TOOMANY	ERROR	Too many arguments passed to the KEY_NAME built-in.
---------------	-------	---

Examples

1. The following example creates a name for the key sequence GOLD/Ctrl/KP4 and binds the EVE FILL command to the resulting key sequence:

```
new_key := KEY_NAME (KP4, Ctrl_MODIFIED, SHIFT_KEY);
DEFINE_KEY ("eve_fill", new_key);
```

2. The following example shows a portion of a command file that defines the keys for an editing interface that emulates EDT:

```
! Procedure to define keys to emulate EDT

PROCEDURE user_define_edtkey

! Bind the EDT Fndnxt function to PF3

    DEFINE_KEY ("edt$search_next", PF3);

! Bind the EDT Find function to SHIFT PF3

    DEFINE_KEY ("edt$search", KEY_NAME (PF3, SHIFT_KEY));
ENDPROCEDURE;
```

LAST_KEY

Syntax

keyword := LAST_KEY

Parameters

None.

Return Value

A marker that returns a keyword for the last key that was entered, read, or executed.

Description

The LAST_KEY procedure returns a DECTPU keyword for the last key that was entered, read, or executed. When DECTPU is replaying a learn sequence or executing the program bound to a key, LAST_KEY returns the last key replayed or processed so far—not the last key that was pressed to invoke the learn sequence or program.

When you invoke DECTPU with the /NODISPLAY qualifier, the value 0 is returned for LAST_KEY, except in the following case: If you precede the LAST_KEY statement with a READ_LINE statement, LAST_KEY can return a key name that represents the last key read by READ_LINE, Ctrl/Z, or the Return key. See the description of READ_LINE for more information on the values that LAST_KEY can return when you use LAST_KEY while running DECTPU in /NO_DISPLAY mode.

Signaled Errors

TPU\$_TOOMANY	ERROR	Too many arguments passed to the LAST_KEY built-in.
---------------	-------	---

Examples

The following example prompts you for input for key definitions:

```
PROCEDURE user_define_key

    def := READ_LINE ("Definition: ");
    key := READ_LINE ("Press key to define.",1);
    IF LENGTH (key) > 0
    THEN
        key := KEY_NAME (key)
    ELSE
        key := LAST_KEY;
    ENDIF;
    DEFINE_KEY (def, key);
ENDPROCEDURE;
```

LEARN_ABORT

Syntax

```
[[integer := ] LEARN_ABORT
```

Parameters

None

Return Value

An integer that indicates whether a learn sequence was actually replaying at the time the LEARN_ABORT statement was executed. The value 1 is returned if a learn sequence was being replayed; otherwise it returns 0. The value 1 is returned each time LEARN_ABORT executes until DECTPU gets back to its main key-reading loop.

Description

The LEARN_ABORT procedure causes a learn sequence being replayed to be terminated whether or not the learn sequence has completed. Only the currently executing learn sequence is aborted.

Whenever you write a procedure that can be bound to a key, the procedure should invoke the LEARN_ABORT built-in procedure in case of error. Using LEARN_ABORT prevents a learn sequence from finishing if the learn sequence calls the user-written procedure and the procedure is not executed successfully.

Signaled Errors

TPU\$_TOOMANY	ERROR	The LEARN_ABORT built-in takes no parameters.
---------------	-------	---

Example

If an error occurs, the following error handler aborts any executing learn sequence:

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

LEARN_BEGIN and LEARN_END

Syntax

```
LEARN_BEGIN ( { EXACT
               NO_EXACT } )
.
.
.
learn := LEARN_END
```

Parameters

EXACT

Causes DECTPU to reuse the input, or learn sequence, that you entered for each READ_LINE, READ_KEY, or READ_CHAR built-in procedure.

NO_EXACT

Causes DECTPU to prompt for new input each time a READ_LINE, READ_KEY, or READ_CHAR built-in procedure is replayed within a learn sequence.

Return Value

A variable of type learn that stores the keystrokes you specify.

Description

The LEARN_BEGIN and LEARN_END procedures saves all keystrokes typed between LEARN_BEGIN and LEARN_END. LEARN_BEGIN starts saving all keystrokes that you type. LEARN_END stops the “learn mode” of DECTPU and returns a learn sequence that consists of all the keystrokes that you entered.

You can use the variable name that you assign to a learn sequence as the parameter for the EXECUTE built-in procedure to replay a learn sequence. You can also use the variable name with the DEFINE_KEY built-in procedure to bind the sequence to a key so that the learn sequence is executed when you press that key.

Learn sequences are different from other DECTPU programs because they are created with keystrokes rather than with DECTPU statements. You create the learn sequence as you are entering text and executing DECTPU commands. Because learn sequences make it easy to collect and execute a sequence

of DECTPU commands, they are convenient for creating temporary “programs”. You can replay these learn sequences during the editing session in which you create them.

Learn sequences are not flexible enough to use for writing general programs. Learn sequences are best suited for saving a series of editing actions that you perform many times during a single editing session.

You can save learn sequences from session to session so that you can replay them in an editing session other than the one in which you created them. To save a learn sequence, bind it to a key. Before ending your editing session, use the SAVE built-in procedure to do an incremental save to the section file you are using. Using the SAVE built-in procedure causes the new definitions from the current session to be added to the section file with which you invoked DECTPU. For more information, see the SAVE built-in procedure. Learn sequences cannot be transferred from one version of DECTPU to another.

Note

You should not use built-in procedures that can return WARNING or ERROR messages as a part of a learn sequence because learn sequences do not stop on error conditions. Because the learn sequence continues executing after an error or warning condition, the editing actions that are executed after an error or a warning may not take effect at the character position you desire.

If, for example, a SEARCH built-in procedure that you use as a part of a learn sequence fails to find the string you specify and issues a warning, the learn sequence does not stop executing. This can cause the rest of the learn sequence to take inappropriate editing actions.

Prekey and postkey procedures interact with learn sequences in the following order:

1. When you press the key or key sequence to which the learn sequence is bound, DECTPU executes the prekey procedure of that key if a prekey procedure has been set.
 2. For each key in the learn sequence, DECTPU executes procedures or programs in the following order:
 - a. DECTPU executes the prekey procedure of that key if a prekey procedure has been set.
 - b. DECTPU executes the code bound to the key itself.
 - c. DECTPU executes the postkey procedure of that key if a postkey procedure has been set.
 3. When all keys in the learn sequence have been processed, DECTPU executes the postkey procedure, if one has been set, for the key to which the entire learn sequence was bound.
-

Note

If, during the recording of a learn sequence, a margin action route is executed (for example EVE’s word wrap), the margin action routine is not executed when the learn sequence is replayed.

Signaled Errors

TPU\$_NOTLEARNING	WARNING	LEARN_BEGIN was not used since the last call to LEARN_END.
TPU\$_ONELEARN	WARNING	A learn sequence is already in progress.

TPU\$_TOOFEW	ERROR	LEARN_BEGIN requires one argument.
TPU\$_TOOMANY	ERROR	LEARN_BEGIN accepts only one argument.
TPU\$_INVPARAM	ERROR	The specified parameter has the wrong type.

Examples

The following example shows how to combine LEARN_BEGIN and LEARN_END so that all of the keystrokes that you enter between them are saved. The (EXACT) keyword specifies that if you use READ_LINE, READ_CHAR, or READ_KEY within the learn sequence, any input that you enter for these built-in procedures is repeated exactly when you replay the learn sequence.

```
LEARN_BEGIN (EXACT)
```

```
.  
.
.
```

This represents a typical editing session, in which you perform commands that are bound to keys.

```
.  
.
.
```

```
do_again := LEARN_END
```

LENGTH

Syntax

```
integer := LENGTH ( { range }  
                  { string } )
```

Parameters

buffer

The buffer whose length you want to determine. If you specify a buffer, line terminators are not counted as character positions.

range

The range whose length you want to determine. If you specify a range, line terminators are not counted as character positions.

string

The string whose length you want to determine.

Return Value

An integer that indicates the number of character positions in a buffer, range, or string you specify.

Description

The LENGTH procedure returns an integer that is the number of character positions in a buffer, range, or string.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	LENGTH must be on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	LENGTH requires one argument.
TPU\$_TOOMANY	ERROR	LENGTH accepts only one argument.
TPU\$_ARGMISMATCH	ERROR	The argument to LENGTH must be a string or a range.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C while LENGTH was executing.

Examples

1. The following example stores the number of characters in the string "Don Quixote" in the variable *str_len* . In this example, the integer value is 11:

```
str_len := LENGTH ("Don Quixote")
```

2. The following example puts a marker without any video attributes at the current position. The marker is assigned to a variable that begins with *user_mark_* and ends with the string you pass as a parameter. The procedure writes a message to the message area verifying the mark name that comes from the input parameter.

```
! Parameters:
!
!   mark_parameter is user-supplied string,
!   which is used as a mark name

PROCEDURE user_mark_ (mark_parameter)

! Local copy of mark_parameter

LOCAL mark_name;

ON_ERROR
  MESSAGE (FAO ("Cannot use !AS as a mark name", mark_name));
  RETURN;
ENDON_ERROR;

! 132 - length ("user_mark_")

IF LENGTH (mark_parameter) > 122
THEN
  mark_name := SUBSTR (mark_name, 1, 122);
ELSE
  mark_name := mark_parameter;
```

```
ENDIF;  
  
EXECUTE ("user_mark_" + mark_name + " := MARK (NONE)");  
MESSAGE (FAO ("Current position marked as !AS", mark_name));  
ENDPROCEDURE;
```

LINE_BEGIN

Syntax

LINE_BEGIN

Parameters

None.

Description

The LINE_BEGIN procedure matches the beginning of a line when used as part of a complex pattern or as an argument to SEARCH. Although LINE_BEGIN behaves much like a built-in, it is actually a keyword.

LINE_BEGIN lets you search for complex strings by creating patterns that match certain conditions. For example, if you want to find all occurrences of the exclamation point (!) when it is the first character in the line, use LINE_BEGIN to create the following pattern:

```
pat1 := LINE_BEGIN + "!";
```

For more information on patterns, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

LINE_BEGIN is a keyword and has no completion codes.

Examples

1. The following example stores the beginning-of-line condition in the variable *pat1* :

```
pat1 := LINE_BEGIN
```

2. The following example removes all lines that start with Compaq Standard Runoff (DSR) commands from a file by searching for a pattern that has a period (.) at the beginning of a line and then removing the lines that match this condition:

```
PROCEDURE user_remove_dsrlines  
  
LOCAL s1,  
      pat1;  
  
pat1 := LINE_BEGIN + ".";  
  
LOOP  
  s1 := SEARCH_QUIETLY (pat1, FORWARD);  
  EXITIF s1 = 0;  
  POSITION (s1);
```

```
        ERASE_LINE;  
    ENDLOOP;  
ENDPROCEDURE;
```

LINE_END

Syntax

LINE_END

Parameters

None

Description

The LINE_END procedure matches the end of a line when used as part of a complex pattern or as an argument to SEARCH. Although LINE_END behaves much like a built-in, it is actually a keyword.

The end-of-line condition is one character position to the right of the last character on a line.

For more information on patterns, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

LINE_END is a keyword and has no completion codes.

Examples

1. The following example stores the LINE_END keyword in the variable *pat1*. You can use *Pat1* as an argument to the SEARCH built-in or as part of a complex pattern.

```
pat1 := LINE_END
```

2. In the following example, if you are not already at the end of the current line, the preceding procedure moves the editing point to the end of the line:

2. PROCEDURE user_end_of_line

```
    LOCAL eol_range;  
  
    eol_range := SEARCH_QUIETLY (LINE_END, FORWARD);  
  
    IF eol_range <> 0  
    THEN  
        POSITION (eol_range);  
    ENDIF;  
ENDPROCEDURE;
```

LOCATE_MOUSE

Syntax

[[integer :=] LOCATE_MOUSE (window, x_integer, y_integer)

Parameters

buffer

The buffer whose length you want to determine. If you specify a buffer, line terminators are not counted as character positions.

range

The range whose length you want to determine. If you specify a range, line terminators are not counted as character positions.

string

The string whose length you want to determine.

Return Value

An integer that indicates whether the pointer was found in a window. The value is 1 if DECTPU finds a window position; otherwise it is 0.

Description

The `LOCATE_MOUSE` procedure locates the window position of the pointer at the time `LOCATE_MOUSE` is invoked. `LOCATE_MOUSE` returns the window name and the window position of the pointer and optionally returns a status that indicates whether the pointer was found in a window. When you press a mouse button, DECTPU determines the location of the mouse pointer and makes that information available while the code bound to the mouse button is being processed. Mouse pointer location information is not available at any other time.

In DECwindows DECTPU, you can use the `LOCATE_MOUSE` built-in procedure anytime after the first keyboard or mouse-button event. The built-in returns the location occupied by the pointer cursor at the time of the most recent keyboard or mouse button event.

If there is no mouse information available (because no mouse button has been pressed or if the mouse has been disabled using `SET (MOUSE)`), `LOCATE_MOUSE` signals the status `TPU$_MOUSEINV`.

Signaled Errors

<code>TPU\$_MOUSEINV</code>	WARNING	The mouse position is not currently valid.
<code>TPU\$_TOOFEW</code>	ERROR	<code>LOCATE_MOUSE</code> requires three parameters.
<code>TPU\$_TOOMANY</code>	ERROR	<code>LOCATE_MOUSE</code> accepts at most three parameters.
<code>TPU\$_BADDELETE</code>	ERROR	You specified a constant as one or more of the parameters.

Examples

1. The following example statement returns an integer in the variable *status* that indicates whether the pointer cursor was found in a window; returns the window in the parameter *new_window* where

the mouse was found; returns an integer in the parameter *x_value* that specifies the pointer cursor's location in the horizontal dimension; and returns an integer in the parameter *y_value* that specifies the pointer cursor's location in the vertical dimension.

```
status := LOCATE_MOUSE (new_window, x_value, y_value);
```

2. In the following example, binding the *user_move_to_mouse* procedure to a mouse button moves the cursor to the mouse location. The *user_move_to_mouse* procedure is essentially equivalent to POSITION (MOUSE).

```
PROCEDURE user_move_to_mouse

    LOCAL my_window,
           x_1,
           y1;

    IF (LOCATE_MOUSE (my_window, x_1, Y1) <> 0)
    THEN
        IF (CURRENT_WINDOW <> my_window)
        THEN
            POSITION (my_window);
            UPDATE (my_window);
        ENDIF;
        CURSOR_VERTICAL (y1 - (CURRENT_ROW - GET_INFO
                               (my_window, "visible_top") + 1));
        CURSOR_HORIZONTAL (CURRENT_COLUMN - x_1);
    ENDIF;
ENDPROCEDURE;
```

CURRENT_ROW and CURRENT_COLUMN return screen-relative location information, while LOCATE_MOUSE returns window-relative location information.

LOOKUP_KEY

Syntax

$$\left\{ \begin{array}{l} \text{integer} \\ \text{learn_sequence} \\ \text{program} \\ \text{string3} \end{array} \right\} := \text{LOOKUP_KEY } \text{key-name}, \left\{ \begin{array}{l} \text{COMMENT} \\ \text{KEY_MAP} \\ \text{PROGRAM} \end{array} \right\} \left[\left[\left\{ \begin{array}{l} \text{, string1} \\ \text{, string2} \end{array} \right\} \right] \right]$$

Parameters

key-name

A DECTPU key name for a key or a combination of keys. See the *Guide to the DEC Text Processing Utility* for a list of the DECTPU key names for the LK201, LK401, and VT100-series keyboards.

COMMENT

A keyword that specifies that the LOOKUP_KEY built-in procedure is to return the comment supplied when the key was defined. If no comment was supplied, the LOOKUP_KEY built-in returns the integer zero.

KEY_MAP

A keyword that specifies that the LOOKUP_KEY built-in procedure is to return the key map in which the key's definition is stored. If you specify a key that is not defined in any key map, LOOKUP_KEY returns a null string.

PROGRAM

A keyword that specifies that the LOOKUP_KEY built-in procedure is to return the program or learn sequence bound to the key specified. If the key is not defined, the LOOKUP_KEY built-in returns the integer 0.

string1

The name of the key map from which the LOOKUP_KEY built-in procedure is to get the key definition. Use this optional parameter if the key is defined in more than one key map. If you do not specify a key map or a key map list for the third parameter, the first definition found for the specified key in the key map list bound to the current buffer is returned.

string2

The name of the key map list from which the LOOKUP_KEY built-in procedure is to get the key definition. Use this optional parameter if the key is defined in more than one key map list. If you do not specify a key map or a key map list for the third parameter, the first definition found for the specified key in the key map list bound to the current buffer is returned.

Return Values**integer**

The integer 0. This value is returned if the key specified as a parameter has no definition.

learn_sequence

The learn sequence bound to the key specified as a parameter.

program

The program bound to the key specified as a parameter.

string3

If you specified COMMENT as the second parameter, *string3* is the comment bound to the key specified as the first parameter. If you specified KEY_MAP as the second parameter, *string3* is the string naming the key map in which the key definition was found.

Description

The LOOKUP_KEY procedure returns the executable code or the comment that is associated with the key that you specify. The code can be returned as a program or as a learn sequence. The comment is returned as a string. LOOKUP_KEY can return a program, a learn sequence, a string, or the integer 0 (0 means that the key has no definition).

LOOKUP_KEY is useful when you are defining keys temporarily during an editing session and you want to check the existing definitions of a key.

Signaled Errors

TPU\$_NOTDEFINABLE	WARNING	Argument is not a valid reference to a key.
TPU\$_NOKEYMAP	WARNING	Argument is not a defined key map.
TPU\$_NOKEYMAPLIST	WARNING	Argument is not a defined key map list.
TPU\$_KEYMAPNTFND	WARNING	The specified key map is not found.
TPU\$_EMPTYKMLIST	WARNING	The specified key map list contains no key maps.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the LOOKUP_KEY built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the LOOKUP_KEY built-in.
TPU\$_NEEDTOASSIGN	ERROR	LOOKUP_KEY must be on the right-hand side of an assignment statement.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the LOOKUP_KEY built-in.
TPU\$_BADKEY	ERROR	An unknown keyword was used as an argument. Only PROGRAM, COMMENT, and KEY_MAP are valid keywords.

Examples

1. The following example returns the executable code that is associated with *key1*. The second keyword, PROGRAM, indicates that the result is returned to a variable of type program or learn.

```
programx := LOOKUP_KEY (key1, PROGRAM)
```

2. The following example returns the comment associated with a particular key:

```
PROCEDURE user_get_key_info
  LOCAL key_to_interpret,
  key_info;

  MESSAGE ("Press the key you want information on: ");

  key_to_interpret := READ_KEY;
  key_info := LOOKUP_KEY (key_to_interpret, COMMENT);

  IF key_info <> ""
  THEN
    MESSAGE ("Comment: " + key_info);
  ELSE
    MESSAGE ("No comment is associated with that key.");
  ENDIF;
ENDPROCEDURE;
```

3. The following example implements multiple shift keys:

```

PROCEDURE shift_key_handler (key_map_list_name);

    LOCAL bound_program;

    bound_program := LOOKUP_KEY (READ_KEY, PROGRAM, "key_map_list_name");

    IF bound_program <> 0
    THEN
        EXECUTE (bound_program);
    ELSE
        MESSAGE ("Attempt to execute undefined key");
    ENDIF;
ENDPROCEDURE;

red_keys := CREATE_KEY_MAP ("red_keys");

red_key_map_list := CREATE_KEY_MAP_LIST ("red_key_map_list",
                                        red_keys);
DEFINE_KEY ("shift_key_handler (red_key_map_list)", PF3,
           "RED shift key");

```

LOWER_WIDGET

Syntax

LOWER_WIDGET (widget)

Parameters

widget

The widget you want DECTPU to lower. The specified widget must be a subclass of WindowObjClass.

Description

The LOWER_WIDGET procedure places the widget at the bottom of a viewing stack. This prevents the widget window associated with the widget from obscuring any sibling windows. LOWER_WIDGET calls the XLIB routine XLowerWindow.

Signaled Errors

TPU\$_INVPARAM	ERROR	The parameter to LOWER_WIDGET has the wrong data type.
TPU\$_NORETURNVALUE	ERROR	This built-in does not return a result.
TPU\$_NOTSUBCLASS	WARNING	The parameter to LOWER_WIDGET is not a

		widget that has an associated widget window.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.

MANAGE_WIDGET

Syntax

MANAGE_WIDGET (widget [, widget...])

Parameters

widget

The widget to be managed.

Description

The MANAGE_WIDGET procedure makes the specified widget or widgets visible, provided that their parent widget is also visible.

MANAGE_WIDGET allows the specified widget's parent to allocate space for the widget by laying out all its children for display. The parent ignores unmanaged children so that they do not take up space on the screen. If the parent widget is realized, and the specified widget's mappedWhenManaged resource is true (the default), then the widget becomes visible on the screen.

If you have multiple children of a single widget that you want to manage, include them in a single call to MANAGE_WIDGET. Managing several widgets at once is more efficient than managing one widget at a time.

All widgets passed in the same MANAGE_WIDGET operation must have the same parent.

Signaled Errors

TPU\$_INVPARAM	ERROR	You specified a parameter of the wrong type.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the MANAGE_WIDGET built-in.
TPU\$_NORETURNVALUE	ERROR	MANAGE_WIDGET cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the MANAGE_WIDGET built-in only if you are using DECwindows DECTPU.
TPU\$_WIDMISMATCH	ERROR	You have specified a widget whose class is not supported.

Example

For a sample procedure using the `MANAGE_WIDGET` built-in procedure, see Example A.1.

MAP

Syntax

```
MAP ( { window, buffer }  
      { widget
```

Parameters

window

The window you want to map to the screen.

buffer

The buffer you want to associate with the window.

widget

The widget you want to make visible.

Description

The `MAP` procedure associates a buffer with a window and causes the window or widget to become visible on the screen. Before using `MAP`, you must already have created the widget, buffer, and window that you specify as parameters. See `CREATE_WIDGET`, `CREATE_BUFFER`, and `CREATE_WINDOW`.

The window and buffer that you use as parameters become the current window and the current buffer, respectively. The map operation synchronizes the cursor position with the editing point in the buffer. If the window is not already mapped to the buffer when you use `MAP`, DECTPU puts the cursor back in the last position the cursor occupied the last time the window was the current window.

`MAP` may cause other windows that are mapped to the screen to be partially or completely occluded. If `MAP` causes the new window to segment another window into two pieces, only the upper part of the segmented window remains visible and continues to be updated. The lower part of the segmented window is erased on the next screen update. If you remove the window that is segmenting another window, DECTPU repaints the screen so that the window that was segmented regains its original size and position on the screen.

In DECwindows, `MAP` also maps the DECTPU main widget if it has not already been mapped.

If you execute `MAP` within a procedure, the screen is not updated to reflect such operations as window repainting, line erasure, or new mapping until the procedure has finished executing and control has returned to the screen manager. If you want the screen to reflect the changes before the entire program is executed, you can force the immediate update of a window by including the following statement in the procedure before any statements containing the `MAP` built-in:

```
UPDATE (WINDOW) ;
```

Signaled Errors

TPU\$_TOOFEW	ERROR	MAP requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_MAXMAPPEDBUF	WARNING	The buffer is already mapped to the maximum number of windows allowed by DECTPU.

Examples

- The following example associates the main buffer with the main window and maps the main window to the screen. You must have established the main buffer and the main window with CREATE_BUFFER and CREATE_WINDOW before you can use them as parameters for MAP.

```
MAP (main_window, main_buffer)
```

- The following example creates a message buffer and a message window. It then associates the message buffer with the message window and maps the message window to the screen.

```
PROCEDURE user_message_window

    message_buffer := CREATE_BUFFER ("message");
    SET (EOB_TEXT, message_buffer, "");
    SET (NO_WRITE, message_buffer);
    SET (SYSTEM, message_buffer);

    message_window := CREATE_WINDOW (23, 2, OFF);
    SET (VIDEO, message_window, NONE);
    MAP (message_window, message_buffer);
ENDPROCEDURE;
```

MARK

Syntax

$$\text{marker} := \text{MARK} \left(\left\{ \begin{array}{l} \text{BLINK} \\ \text{BOLD} \\ \text{FREE_CURSOR} \\ \text{REVERSE} \\ \text{UNDERLINE} \\ \text{NONE} \end{array} \right\} \right) \llbracket , \left\{ \begin{array}{l} \text{buffer} \\ \text{window} \end{array} \right\} \llbracket , \text{integer1} \llbracket , \text{integer2} \rrbracket \rrbracket$$

Parameters

BLINK

A keyword that directs DECTPU to display the marker in blinking rendition.

BOLD

A keyword that directs DECTPU to display the marker in bold rendition.

FREE_CURSOR

A keyword that directs DECTPU to create a free marker (that is, a marker not bound to a character).

A free marker has no video attribute.

REVERSE

A keyword that directs DECTPU to display the marker in reverse video.

UNDERLINE

A keyword that directs DECTPU to underline the marker.

NONE

A keyword that directs DECTPU to apply no video attributes to the marker.

buffer

The buffer in which the marker is to be located. By default, DECTPU locates markers in the current buffer.

window

The window that is mapped to the buffer in which the marker is to be located. You can specify a window variable only if the window is mapped to a buffer. By default, DECTPU locates markers in the current buffer.

integer1

An integer for the screen column where the marker is to be located. You can specify an integer from 1 to 32767. However, if you specify an integer smaller than the record's left margin or larger than the end of the record, DECTPU inserts padding spaces between the marker and the nearest text. The default is to locate the marker at the buffer offset that corresponds to the current screen column.

integer2

An integer for the record in the buffer where the marker is to be located. You can specify any integer greater than 1 but less than the maximum number of lines in the buffer, if it has been set with the SET (MAX_LINES) built-in procedure. The default is to locate the marker in the current record.

Return Value

A marker for the location in a buffer that you specify.

Description

The MARK procedure returns a marker for a specified location in a buffer. You must specify how the marker is to be displayed on the screen (no special video, reverse video, bolded, blinking, or underlined). You can use MARK to establish placeholders or “bookmarks.”

A marker can be either bound or free. For more information on how these markers differ, see the *Guide to the DEC Text Processing Utility*.

To create a bound marker, use the `MARK` built-in procedure 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 cursor in a location where there is no character, DECTPU fills the space between the marker and the nearest character with padding space characters.

A bound marker is tied to the character at which it is created. If the character tied to the marker moves, the marker moves also. If the character tied to the marker is deleted, the marker moves to the nearest character position. The nearest character position is determined in the following ways:

- If there is a character position on the same line and to the right, the marker moves to this position, even if the position is at the end of the line.
- If the line on which the marker is located is deleted, the marker moves to the first position on the following line.

You can move one column past the last character in a line and place a marker there. However, the video attribute for the marker is not visible unless a subsequent operation puts a character under the marker.

If you use a marker at the end of a line as part of a range, the end of line is included in the range even though the marker is not positioned on a character.

A marker is free if the following conditions are true:

- You used the `FREE_CURSOR` keyword to create the marker.
- There was no character in the position where you created the marker.

DECTPU 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 DECTPU measures distance to a free marker, the marker moves too. DECTPU preserves a uniform distance between the character and the marker. If you collapse white space that contains one or more free markers (for example, if you delete a tab or use the `APPEND_LINE` built-in procedure), DECTPU preserves the markers and binds them to the nearest character.

Unless you specify the parameter `FREE_CURSOR`, using the `MARK` built-in may result in the insertion of padding spaces or lines into the buffer if the new marker is one of the following:

- Before the beginning of a line
- In the middle of a tab
- Beyond the end of a line
- After the last line in the buffer

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>MARK</code> requires one parameter.
<code>TPU\$_TOOMANY</code>	ERROR	<code>MARK</code> accepts only one parameter.
<code>TPU\$_NEEDTOASSIGN</code>	ERROR	The <code>MARK</code> built-in must be on the right-hand side of an assignment statement.

TPU\$_NOCURRENTBUF	WARNING	You must be positioned in a buffer to set a marker.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be NONE, BOLD, BLINK, REVERSE, UNDERLINE, or FREE_CURSOR.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.
TPU\$_INSVIRMEM	FATAL	There is not enough memory to create the marker.

Examples

1. The following example places a marker at the editing point. There are no video attributes applied to the marker.

```
user_mark := MARK (NONE)
```

2. The following example marks a temporary position at the current character position, and then goes to the paste buffer and creates a range of the contents of the paste buffer. DECTPU then goes to *temp_pos* and copies the text from the paste buffer at the temporary position.

3. PROCEDURE user_paste

```
temp_pos := MARK (NONE);

POSITION (END_OF (paste_buffer));
MOVE_HORIZONTAL (-2);

paste_text := CREATE_RANGE (BEGINNING_OF (paste_buffer),
                           MARK (NONE), NONE);
POSITION (temp_pos);
COPY_TEXT (paste_text);
ENDPROCEDURE;
```

MATCH

Syntax

$$\text{pattern} := \text{MATCH} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \right)$$

Parameters

buffer

An expression that evaluates to a buffer. MATCH forms a string from the contents of the buffer and stops matching when it finds the resulting string.

range

An expression that evaluates to a range. MATCH forms a string from the contents of the range and stops matching when it finds the resulting string.

string

An expression that evaluates to a string. MATCH stops matching when it finds this string.

Return Value

A variable of type pattern that matches text from the editing point up to and including the characters specified in the parameter.

Description

The MATCH procedure returns a pattern that matches from the editing point up to and including the sequence of characters specified in the parameter. The matched string does not contain line terminators.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	MATCH must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	MATCH requires at least one argument.
TPU\$_TOOMANY	ERROR	MATCH requires no more than one argument.
TPU\$_ARGMISMATCH	ERROR	Argument to MATCH has the wrong type.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of MATCH.

Examples

1. The following example 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")
```

2. The following example finds text within double parentheses. It moves the editing point to the beginning of the parenthesized text, if it is found.

```
PROCEDURE user_double_parens
```

```
  paren_text    := "(" + MATCH (')')';
  found_range   := SEARCH_QUIETLY (paren_text, FORWARD, NO_EXACT);
```

```
  IF found_range = 0    ! No match
  THEN
    MESSAGE ("No match found.");
  ELSE
```

```

        POSITION (found_range);
    ENDIF;
ENDPROCEDURE;

```

MESSAGE

Syntax

```

MESSAGE ( { buffer
           range } [, integer1 ])

MESSAGE ( { integer2
           keyword
           string
           buffer } [, integer3
           [, FAO-parameter [, FAO-parameters... ] ] ])

```

Parameters

buffer

The buffer that contains the text that you want to include in the message buffer.

range

The range that contains the text that you want to include in the message buffer.

integer1

An integer that indicates the severity of the message placed in the message buffer. If you do not specify this parameter, no severity code is associated with the message. The allowable integer values and their meanings are as follows:

Integer	Meaning
0	Warning
1	Success
2	Error
3	Informational

integer2

The integer that represents the message code associated with the text to be fetched.

keyword

The DECTPU keyword that represents the message code associated with the text to be fetched. DECTPU provides keywords for all of the message codes used by DECTPU and EVE.

string

Either a quoted string or a variable that represents the text you want to include in the message buffer.

integer3

A bit-encoded integer that specifies what fields of the message text associated with the message code from the first parameter are to be fetched. If the message flags are not specified or the value is 0, then the message flags set by the SET (MESSAGE_FLAGS) built-in procedure are used. Table 2.7 shows the message flags.

Table 2.7. Message Flag Values for MESSAGE

Bit	Constant	Meaning
0	TPU\$K_MESSAGE_TEXT	Include text of message.
1	TPU\$K_MESSAGE_ID	Include message identifier.
2	TPU \$K_MESSAGE_SEVERITY	Include severity level indicator.
3	TPU \$K_MESSAGE_FACILITY	Include facility name.

FAO-parameter

One or more expressions that evaluate to an integer or string. The MESSAGE_TEXT built-in procedure uses these integers and strings as arguments to the \$FAO system service, substituting the values into the text associated with the message code to form the resultant string.

The FAO directives are listed in the description of \$FAO in the *VSI OpenVMS System Services Reference Manual*.

Description

The MESSAGE procedure, depending on the format you choose, either puts the characters that you specify into the message buffer or else fetches text associated with a message code, formats the text using FAO directives, and puts it in the message buffer.

If you use the first format, MESSAGE inserts the characters in the string, range, or buffer that you specify into the message buffer, if one exists. (By default, DECTPU looks for a buffer variable that is named MESSAGE_BUFFER.) If there is no message buffer, DECTPU displays the message at the current location on the device pointed to by SYS\$OUTPUT (usually your terminal).

If you use the first format, MESSAGE provides the user who is writing an editing interface with a method for displaying messages in a way that is consistent with the DECTPU language.

If you use the second format, MESSAGE fetches the text associated with a message code, uses FAO directives to format the text, and displays the formatted message in the message buffer. However, if there is no message buffer, DECTPU displays the message on SYS\$OUTPUT (usually your terminal).

If you use the second format, MESSAGE writes a formatted string in the message buffer. The difference between MESSAGE and MESSAGE_TEXT is that MESSAGE_TEXT returns the resulting string while MESSAGE places the resulting string in the message buffer. The string is specified by the message code passed as the first parameter and constructed according to the rules of the \$FAO system service. The control string associated with the message code directs the formatting process, and the optional arguments are values to be substituted into the control string.

MESSAGE accepts up to 127 parameters. This built-in can return strings of 65535 characters maximum.

If you have associated a message buffer with a message window, and if the message window is mapped to the screen, the range you specify appears immediately in the message window on the screen.

If you have not associated a message buffer with a message window, messages are written to the buffer but do not appear on the screen.

MESSAGE capitalizes the first character of the string placed in the message buffer. The MESSAGE_TEXT built-in procedure, on the other hand, does not capitalize the first character of the returned string.

You can include the following FAO directives as part of the message text:

!AS	Inserts a string as is
!OL	Converts an integer to octal notation
!XL	Converts an integer to hexadecimal notation
!ZL	Converts an integer to decimal notation
!UL	Converts an integer to decimal notation without adjusting for negative numbers
!SL	Converts an integer to decimal notation with negative numbers converted
!/	Inserts a new line character (carriage return/line feed)
!_	Inserts a tab
!}	Inserts a form feed
!!	Inserts an exclamation point
!%S	Inserts an s if the most recently converted number is not 1
!%T	Inserts the current time if you enter 0 as the parameter (you cannot pass a specific time because DECTPU does not use quadwords)
!%D	Inserts the current date and time if you enter 0 as the parameter (you cannot pass a specific date because DECTPU does not use quadwords)

Signaled Errors

TPU\$_TOOFEW	ERROR	MESSAGE requires at least one argument.
TPU\$_TOOMANY	ERROR	MESSAGE cannot accept as many arguments as you specified.
TPU\$_ARGMISMATCH	ERROR	You specified an argument of the wrong type.
TPU\$_INVFAOPARAM	WARNING	Argument was not a string or integer.
TPU\$_INVPARAM	ERROR	You specified an argument of the wrong type.
TPU\$_FLAGTRUNC	INFORMATIONAL	Message flag truncated to 4 bits.

TPU\$_SYSERROR	ERROR	Error fetching the message text.
TPU\$_ILLSEVERITY	WARNING	Illegal severity specified; DECTPU used the severity "error."
TPU\$_MSGNOTFND	WARNING	Message not found. DECTPU returned default message.

Examples

1. The following example writes the text "Hello" in the message area:

```
MESSAGE ("Hello")
```

2. The following example determines whether the cursor is at the end of the line. It sends a text message to the message area on the screen about the position of the cursor.

```
PROCEDURE user_on_eol

! test if at eol, return true or false

MOVE_HORIZONTAL (1);
IF CURRENT_OFFSET = 0           ! then we are on eol
THEN
    user_on_end_of_line := 1; ! return true
    MESSAGE ("Cursor at end of line");
ELSE
    user_on_end_of_line := 0; ! return false
    MESSAGE ("Cursor is not at the end of line");
ENDIF;
MOVE_HORIZONTAL (-1);           ! move back
ENDPROCEDURE;
```

3. The following example fetches the text associated with the message code TPU\$_OPENIN and substitutes the string "BAT.BAR" into the message:

```
MESSAGE (TPU$_OPENIN, TPU$K_MESSAGE_TEXT, "bat.bar");
```

All of the text of the message is fetched. The following string is displayed in the message buffer:

```
Error opening BAT.BAR as input
```

MESSAGE_TEXT

Syntax

```
string := MESSAGE_TEXT ( {integer1 } [, integer2 [, FAO-parameter
                        [, FAO-parameters... ] ] ] )
```

Parameters

integer1

The integer for the message code associated with the text that is to be fetched.

keyword

The keyword for the message code associated with the text that is to be fetched. DECTPU provides keywords for all of the message codes used by DECTPU and the EVE editor.

integer2

A bit-encoded integer that specifies what fields of the message text associated with the message code from the first parameter are to be fetched. If the message flags are not specified or the value is 0, then the message flags set by the SET (MESSAGE_FLAGS) built-in procedure are used. Table 2.8 shows the message flags.

Table 2.8. Message Flag Values for MESSAGE_TEXT

Bit	Constant	Meaning
0	TPU\$K_MESSAGE_TEXT	Include text of message.
1	TPU\$K_MESSAGE_ID	Include message identifier.
2	TPU \$K_MESSAGE_SEVERITY	Include severity level indicator.
3	TPU \$K_MESSAGE_FACILITY	Include facility name.

FAO-parameter

One or more expressions that evaluate to an integer or string. The MESSAGE_TEXT built-in procedure uses these integers and strings as arguments to the \$FAO system service. It then substitutes the resultant values into the text associated with the message code to form the returned string.

Return Value

The text associated with a message code that is fetched and formatted by MESSAGE_TEXT.

Description

The MESSAGE_TEXT procedure fetches the text associated with a message code. MESSAGE_TEXT uses FAO directives to specify how strings and integers should be substituted into the text.

MESSAGE_TEXT accepts up to 127 parameters. This built-in can return strings of 65535 characters maximum.

MESSAGE_TEXT returns a formatted string, specified by the message code passed as the first parameter and constructed according to the rules of the \$FAO system service. The control string associated with the message code directs the formatting process, and the optional arguments are values to be substituted into the control string.

MESSAGE_TEXT does not capitalize the first character of the returned string. The MESSAGE built-in procedure, on the other hand, does capitalize the first character.

You can include the following FAO directives as part of the message text:

!AS	Inserts a string as is
!OL	Converts an integer to octal notation

!XL	Converts an integer to hexadecimal notation
!ZL	Converts an integer to decimal notation
!UL	Converts an integer to decimal notation without adjusting for negative numbers
!SL	Converts an integer to decimal notation with negative numbers converted
!/	Inserts a new line character (carriage return/line feed)
!_	Inserts a tab
!}	Inserts a form feed
!!	Inserts an exclamation point
!%S	Inserts an s if the most recently converted number is not 1
!%T	Inserts the current time if you enter 0 as the parameter (you cannot pass a specific time because DECTPU does not use quadwords)
!%D	Inserts the current date and time if you enter 0 as the parameter (you cannot pass a specific date because DECTPU does not use quadwords)

For complete information on the \$FAO and \$GETMSG system services, see the *VSI OpenVMS System Services Reference Manual*.

Signaled Errors

TPU\$_INVFAOPARAM	WARNING	Argument was not a string or integer.
TPU\$_NEEDTOASSIGN	ERROR	MESSAGE_TEXT must appear on the right-hand side of an assignment statement.
TPU\$_INVPARAM	ERROR	You specified an argument of the wrong type.
TPU\$_TOOFEW	ERROR	MESSAGE_TEXT requires at least one parameter.
TPU\$_TOOMANY	ERROR	MESSAGE_TEXT accepts up to 20 FAO directives.
TPU\$_FLAGTRUNC	INFORMATIONAL	Message flag truncated to bits.
TPU\$_SYSERROR	ERROR	Error fetching the message text.

Example

The following example fetches the text associated with the message code TPU\$_OPENIN and substitutes the string "BAT.BAR" into the message:

```
all_message_flags := TPU$K_MESSAGE_TEXT OR
                    TPU$K_MESSAGE_ID OR
                    TPU$K_MESSAGE_SEVERITY OR
```

```

        TPU$K_MESSAGE_FACILITY;
openin_text := MESSAGE_TEXT (TPU$_OPENIN, all_message_flags,
        bat.bar");

```

All of the text of the message is fetched. The following string is stored in the variable *openin_text*:

```
%TPU-E-OPENIN, error opening BAT.BAR as input
```

MODIFY_RANGE

Syntax

```

MODIFY_RANGE (range, { marker1 } { marker2 }
              [, keyword1 ]
              [, keyword2 ])

```

Parameters

range

The range to be modified.

marker1

The starting mark for the range.

marker2

The ending mark for the range.

keyword1

A keyword that indicates the point in the buffer where you want the range to begin or end. Table 2.9 shows the valid keywords and their meanings. Use of the delimiting keywords is more efficient than the BEGINNING_OF and END_OF built-in procedures.

Table 2.9. MODIFY_RANGE Keyword Parameters

Keyword	Meaning
LINE_BEGIN	The beginning of the current buffer's current line.
LINE_END	The end of the current buffer's current line.
BUFFER_BEGIN	Line 1, offset 0 in the current buffer. This is the first position where a character could be inserted, regardless of whether there is a character there. This is the same as the point referred to by BEGINNING_OF (CURRENT_BUFFER).
BUFFER_END	The last position in the buffer where a character could be inserted, regardless of whether there is a character there. This is the same as the point referred to by END_OF (CURRENT_BUFFER).

keyword2

A keyword that specifies the new video attribute for the range. By default, the attribute is not modified. You can use the NONE, REVERSE, UNDERLINE, BLINK, or BOLD keywords to specify this parameter.

Description

The MODIFY_RANGE procedure dynamically modifies a range. You can use MODIFY_RANGE to specify a new starting mark and ending mark for an existing range.

MODIFY_RANGE can also change the characteristics of the range without deleting, re-creating, and repainting all the characters in the range. Using MODIFY_RANGE, you can direct DECTPU to apply or remove the range's video attribute to or from characters as you select and unselect text.

Ranges are limited to one video attribute at a time. Specifying a video attribute different from the present attribute causes DECTPU to apply the new attribute to the entire visible portion of the range.

If the video attribute stays the same and only the markers move, the only characters that are refreshed are those visible characters newly added to the range and those visible characters that are no longer part of the range.

Signaled Errors

TPU\$_NOTSAMEBUF	WARNING	The first and second marker are in different buffers.
TPU\$_ARGMISMATCH	ERROR	The data type of the indicated parameter is not supported by the MODIFY_RANGE built-in.
TPU\$_BADKEY	WARNING	You specified an illegal keyword.
TPU\$_INVPARAM	ERROR	You specified a parameter of the wrong type.
TPU\$_MODRANGEMARKS	ERROR	MODIFY_RANGE requires either two marker parameters or none.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the MODIFY_RANGE built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the MODIFY_RANGE built-in.
TPU\$_NORETURNVALUE	ERROR	MODIFY_RANGE cannot return a value.

Examples

1. The following example creates a range between the editing point and the pointer cursor location. At a point in the program after you might have moved the pointer cursor, the code fragment modifies the range to reflect the new pointer cursor location.

```
begin_mark := MARK (BOLD);
POSITION (MOUSE);
finish_mark := MARK (BOLD);
```

```
this_range := CREATE_RANGE (begin_mark, finish_mark, BOLD);
! .
! . (User may have moved mouse)
! .
POSITION (MOUSE);
new_mark := MARK (BOLD);
IF new_mark <> finish_mark
THEN
    MODIFY_RANGE (this_range, begin_mark, new_mark, BOLD);
ENDIF;
```

2. The following example causes the range *dynamic_range* to shrink to one character, then grow until it becomes as large as the range *remembered_range*:

```
PROCEDURE move_mark (place_to_start, direction);

    POSITION (place_to_start);

    IF direction = 1
    THEN
        MOVE_HORIZONTAL (1);
    ELSE
        MOVE_HORIZONTAL (-1);
    ENDIF;

    RETURN MARK (NONE);

ENDPROCEDURE;

PROCEDURE user_shrink_and_enlarge_range

    LOCAL start_mark,
           end_mark,
           direction,
           dynamic_range,
           rendition,
           remembered_range;

                                                    ! The following lines
                                                    ! create a range that
                                                    ! shrinks and grows and
                                                    ! a range that defines
                                                    ! the limits of the dynamic
                                                    ! range.

    POSITION (LINE_BEGIN);
    start_mark := MARK (NONE);
    POSITION (LINE_END);
    end_mark := MARK (NONE);
    rendition := REVERSE;
    remembered_range := CREATE_RANGE (start_mark, end_mark, NONE);
    dynamic_range := CREATE_RANGE (start_mark, end_mark, rendition);

                                                    ! The following lines
                                                    ! shrink and enlarge
                                                    ! the dynamic range.

    direction := 1;
```



```
LOOP
  UPDATE (CURRENT_WINDOW);

  start_mark := move_mark (BEGINNING_OF (dynamic_range),
direction);
  end_mark := move_mark (END_OF (dynamic_range), 1 - direction);

  MODIFY_RANGE (dynamic_range, start_mark, end_mark);

  IF start_mark > end_mark
  THEN
    EXITIF READ_KEY = Ctrl_Z_KEY;
    direction := 0;
    IF rendition = REVERSE
    THEN
      rendition := BOLD;
    ELSE
      rendition := REVERSE;
    ENDIF;
    MODIFY_RANGE (dynamic_range, , , rendition);

  ENDIF;

  IF (start_mark = BEGINNING_OF (remembered_range)) OR
    (end_mark = END_OF (remembered_range))
  THEN
    direction := 1;
  ENDIF;
ENDLOOP;

ENDPROCEDURE;
```

3. The following example aligns text that conforms to the pattern specified in the second parameter. For example, if you want to align all comments in a piece of DECTPU code, you would pass as the second parameter a pattern defined as an exclamation point followed by an arbitrary amount of text or white space and terminated by a line end.

The procedure is passed a range of text. As the procedure searches the range to identify the rightmost piece of text that matches the pattern, the procedure modifies the range to exclude any matching text. Next, the procedure searches the original range again and inserts padding spaces in front of each instance of matching text, making the text align with the rightmost instance of matching text.

```
PROCEDURE line_up_characters (text_range, lined_chars_pat)

LOCAL
  range_start,
  range_end,
  temp_range,
  max_cols;

range_end := END_OF (text_range);           ! These statements store
                                             ! the ends of the range
                                             ! containing the text operated on.

range_start := BEGINNING_OF (text_range);
```

```

! The following statements
! locate the portions of
! text that match the pattern
! and determine which is
! furthest to the right.
max_cols := 0;
LOOP
  temp_range := SEARCH_QUIETLY (lined_chars_pat, REVERSE, EXACT, text_range);
  EXITIF temp_range = 0;
  POSITION (temp_range);
  IF GET_INFO (MARK (NONE), "offset_column") > max_cols
  THEN
    max_cols := GET_INFO (MARK (NONE), "offset_column");
  ENDIF;
  MOVE_HORIZONTAL (-1);
  MODIFY_RANGE (text_range, BEGINNING_OF (text_range), MARK (NONE));
ENDLOOP;

! The following lines
! locate matches to the
! pattern and align them
! with the rightmost
! piece of matching text.
text_range := CREATE_RANGE (range_start, range_end);

LOOP
  temp_range := SEARCH_QUIETLY (lined_chars_pat, FORWARD, EXACT, text_range);
  EXITIF temp_range = 0;
  POSITION (temp_range);
  IF GET_INFO (MARK (NONE), "offset_column") < max_cols
  THEN
    COPY_TEXT (" " * (max_cols - GET_INFO (MARK (NONE), "offset_column")));
  ENDIF;
  MOVE_HORIZONTAL (1);
  MODIFY_RANGE (text_range, END_OF (text_range), MARK (NONE));
ENDLOOP;

!
! Restore the range to its original state, plus a reverse attribute.
!
text_range := CREATE_RANGE (range_start, range_end, REVERSE); ! This line
! restores the
! range to its
! original state
! and displays
! the contents
! in reverse
! video.

ENDPROCEDURE;
```

MOVE_HORIZONTAL

Syntax

MOVE_HORIZONTAL (integer)

Parameters

integer

The signed integer value that indicates the number of characters the editing point should be moved. A positive integer specifies movement toward the end of the buffer. A negative integer specifies movement toward the beginning of the buffer.

DECTPU does not count the column where the editing point is located when determining where to establish the new editing point. DECTPU does count the end-of-line (the column after the last text character on the line) when determining where to establish the new editing point.

Description

The `MOVE_HORIZONTAL` procedure changes the editing point in the current buffer by the number of characters you specify. The horizontal adjustment of the editing point is tied to text. `MOVE_HORIZONTAL` crosses line boundaries to adjust the current character position.

You cannot see the adjustment caused by `MOVE_HORIZONTAL` unless the current buffer is mapped to a visible window. If it is, DECTPU scrolls text in the window, if necessary, so that the editing point you establish with `MOVE_HORIZONTAL` is within the scrolling limits set for the window.

If you try to move past the beginning or the end of a buffer, DECTPU displays a warning message.

Using `MOVE_HORIZONTAL` may cause DECTPU to insert padding spaces or blank lines in the buffer. `MOVE_HORIZONTAL` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>MOVE_HORIZONTAL</code> requires one parameter.
<code>TPU\$_TOOMANY</code>	ERROR	You specified more than one parameter.
<code>TPU\$_INVPARAM</code>	ERROR	The specified parameter has the wrong type.
<code>TPU\$_NOCURRENTBUF</code>	WARNING	You are not positioned in a buffer.
<code>TPU\$_ENDOFBUF</code>	WARNING	You are trying to move forward past the last character of the buffer.
<code>TPU\$_BEGOFBUF</code>	WARNING	You are trying to move in reverse past the first character of the buffer.

Examples

1. The following example moves the editing point five characters toward the end of the current buffer:

```
MOVE_HORIZONTAL (+5)
```

2. The following example moves the editing point by sections that are eight lines long and uses `MOVE_HORIZONTAL` to put the editing point at the beginning of the line:

```
PROCEDURE user_move_by_lines
```

```
IF CURRENT_DIRECTION = FORWARD
THEN
    MOVE_VERTICAL (8)
ELSE
    MOVE_VERTICAL (- 8)
ENDIF;
MOVE_HORIZONTAL (-CURRENT_OFFSET);
ENDPROCEDURE;
```

MOVE_TEXT

Syntax

$$\llbracket \text{range2} := \rrbracket \text{MOVE_TEXT} \left(\begin{array}{l} \text{buffer} \\ \text{range1} \\ \text{string} \end{array} \right)$$

Parameters

buffer

The buffer from which text is moved.

range1

The range from which text is moved.

string

A string that represents the text you want to move. Text is not removed from its original location with this argument.

Return Value

The range where the copied text has been placed.

Description

The MOVE_TEXT procedure moves the text you specify and inserts or overwrites it in the current buffer, depending on the mode of the current buffer. When you move text with range and buffer parameters, you remove it from its original location. For information on how to copy text instead of removing it, see the description of the COPY_TEXT built-in procedure.

If the current buffer is in insert mode, the text you specify is inserted before the editing point in the current buffer. If the current buffer is in overstrike mode, the text you specify replaces text starting at the current position and continuing for the length of the string, range, or buffer.

Markers and ranges are not moved with the text. If the text of a marker or a range is moved, the marker or range structure and any video attribute that you specified for the marker or range are moved to the next closest character, which is always the character following the marker or range. To remove the marker or range structure, use the DELETE built-in procedure or set the variable to which the range is assigned to 0.

`MOVE_TEXT` is similar to `COPY_TEXT`. However, `MOVE_TEXT` erases the text from its original string, range, or buffer, while `COPY_TEXT` just makes a copy of the text and places the copy at the new location.

You cannot add a buffer or a range to itself. If you try to do so, DECTPU issues an error message. If you try to insert a range into itself, part of the range is copied before DECTPU signals an error. If you try to overstrike a range into itself, DECTPU may or may not signal an error.

`MOVE_TEXT` may cause DECTPU to insert padding spaces or blank lines in the buffer. `MOVE_TEXT` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

<code>TPU\$_NOCACHE</code>	ERROR	There is not enough memory to allocate a new cache.
<code>TPU\$_TOOFEW</code>	ERROR	<code>MOVE_TEXT</code> requires one argument.
<code>TPU\$_TOOMANY</code>	ERROR	<code>MOVE_TEXT</code> accepts only one argument.
<code>TPU\$_ARGMISMATCH</code>	ERROR	The argument to <code>MOVE_TEXT</code> must be a buffer, range, or string.
<code>TPU\$_NOTMODIFIABLE</code>	ERROR	You cannot copy text into an unmodifiable buffer.
<code>TPU\$_MOVETOCOPY</code>	WARNING	<code>MOVE_TEXT</code> was able to copy the text into the current buffer but could not delete it from the source buffer because the source buffer is unmodifiable.

Examples

1. If you are using insert mode for text entry, the following statement causes the text from *main_buffer* to be placed in front of the current position in the current buffer. The text is removed from *main_buffer*.

```
MOVE_TEXT (main_buffer)
```

2. The following example puts the text from the scratch buffer before the editing point in the main buffer. The text in the scratch buffer is removed; no copy of it is left there.

```
PROCEDURE user_move_text

    LOCAL this_mode;

    ! Save mode of current buffer in this_mode
    this_mode := GET_INFO (CURRENT_BUFFER, "mode");
```

```
! Set current buffer to insert mode
  SET (INSERT, CURRENT_BUFFER);

! Move the scratch buffer text to the current buffer
  MOVE_TEXT (scratch_buffer);

! Reset current buffer to original mode
  SET (this_mode, CURRENT_BUFFER);
ENDPROCEDURE;
```

MOVE_VERTICAL

Syntax

MOVE_VERTICAL (integer)

Parameters

integer

The signed integer value that indicates the number of lines that the editing point should be moved. A positive integer specifies movement toward the end of the buffer. A negative integer specifies movement toward the beginning of the buffer.

Description

The MOVE_VERTICAL procedure modifies the editing point in the current buffer by the number of lines you specify. The adjustment that MOVE_VERTICAL makes is tied to text. DECTPU tries to retain the same character offset relative to the beginning of the line when moving vertically. However, if there are tabs in the lines, or the lines have different margins, the editing point does not necessarily retain the same column position on the screen.

By default, DECTPU keeps the cursor at the same offset on each line. However, because DECTPU counts a tab as one character regardless of how wide the tab is, the cursor's column position may vary greatly even though the offset is the same.

To keep the cursor in approximately the same column on each line, use the following statement:

```
SET (COLUMN_MOVE_VERTICAL, ON)
```

This statement directs DECTPU to keep the cursor in the same column unless a tab character makes this impossible. If a tab occupies the column position, DECTPU moves the cursor to the beginning of the tab.

You cannot see the adjustment caused by MOVE_VERTICAL unless the current buffer is mapped to a visible window. If it is, DECTPU scrolls text in the window, if necessary, so that the editing point you establish with MOVE_VERTICAL is within the scrolling limits set for the window.

Using MOVE_VERTICAL may cause DECTPU to insert padding spaces or blank lines in the buffer. MOVE_VERTICAL causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

If you try to move past the beginning or end of a buffer, DECTPU signals a warning message.

Signaled Errors

TPU\$_TOOFEW	ERROR	MOVE_VERTICAL requires at least one parameter.
TPU\$_TOOMANY	ERROR	You specified more than one parameter.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BEGOFBUF	WARNING	You are trying to move backward past the first character of the buffer.
TPU\$_ENDOFBUF	WARNING	You are trying to move forward past the last character of the buffer.
TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.

Examples

1. The following example moves the editing point in the current buffer down five lines toward the end of the buffer:

```
MOVE_VERTICAL (+5)
```

2. The following example moves the editing point by sections that are eight lines long:

```
PROCEDURE user_move_8_lines
    IF CURRENT_DIRECTION = FORWARD
    THEN
        MOVE_VERTICAL (8);
    ELSE
        MOVE_VERTICAL (- 8);
    ENDIF;
    MOVE_HORIZONTAL (- CURRENT_OFFSET);
ENDPROCEDURE;
```

NOTANY

Syntax

$$\text{pattern} := \text{NOTANY} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \llbracket, \text{integer1} \rrbracket \right)$$

Parameters

buffer

An expression that evaluates to a buffer. NOTANY matches any character not in the resulting buffer.

range

An expression that evaluates to a range. NOTANY matches any character not in the resulting range.

string

An expression that evaluates to a string. NOTANY matches any character not in the resulting string.

integer1

This integer value indicates how many contiguous characters NOTANY matches. The default value for this integer is 1.

Return Value

A pattern that matches characters not in the string, buffer, or range used as a parameter.

Description

The NOTANY procedure returns a pattern that matches a specific number of contiguous characters not in the string, buffer, or range that is used as the first parameter. The second parameter determines the number of characters NOTANY must match. NOTANY does not match across line breaks.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	NOTANY must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	NOTANY requires at least one argument.
TPU\$_TOOMANY	ERROR	NOTANY accepts no more than two arguments.
TPU\$_ARGMISMATCH	ERROR	NOTANY was given an argument of the wrong type.
TPU\$_INVPARAM	ERROR	NOTANY was given an argument of the wrong type.
TPU\$_MINVALUE	WARNING	NOTANY was given an argument less than the minimum value.
TPU\$_CONTROL C	ERROR	You pressed Ctrl/C during the execution of NOTANY.

Examples

1. The following example creates a pattern that matches the first character that is not an X, a Y, or a Z. The match fails if no character other than X, Y, or Z is found.

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

2. The following example creates a pattern that matches any single character other than one of the characters a, b, c, x, and y:

```
a_buf := CREATE_BUFFER ("new buffer");
POSITION (a_buf);
COPY_TEXT ("xy");
```



```
SPLIT_LINE;  
COPY_TEXT ("abc");  
pat1 := NOTANY (a_buf);
```

3. The following example starts at the current location and looks for the first nonalphabetic, nonlowercase character. The variable *non_alpha_range* stores the character that matches these conditions.

```
!  
! The following procedure returns a marker pointing to  
! the next nonalphabetic character or the integer zero  
! if there are no more nonalphabetic characters. You  
! call the procedure in the following way:  
!  
!   non_alpha_marker := user_search_for_nonalpha;  
  
PROCEDURE user_search_for_nonalpha  
  
    LOCAL pat,  
           first_non_alpha;  
  
    pat := NOTANY ("abcdefghijklmnopqrstuvwxy");  
  
    first_non_alpha := SEARCH_QUIETLY (pat, FORWARD, NO_EXACT);  
  
    IF first_non_alpha <> 0  
  
    THEN  
        first_non_alpha := BEGINNING_OF (first_non_alpha);  
    ENDIF;  
  
    RETURN first_non_alpha;  
ENDPROCEDURE;
```

PAGE_BREAK

Syntax

PAGE_BREAK

Parameters

None.

Description

The PAGE_BREAK procedure specifies the form-feed character, ASCII(12), as a portion of a pattern to be matched. This character has an ASCII value of 12. Although PAGE_BREAK behaves much like a built-in, it is actually a keyword. If the form-feed character is the only character on a line, PAGE_BREAK matches the whole line. If the form-feed character is not the only character on a line, PAGE_BREAK matches only the form-feed character.

Signaled Error

PAGE_BREAK is a keyword and has no completion codes.

Example

The following example places the cursor on the next page in the current buffer. If you are already on the last page of a document, it places the cursor at the end of that document.

```
PROCEDURE user_next_page

  LOCAL next_page;

  next_page := SEARCH_QUIETLY (PAGE_BREAK, FORWARD);
  IF next_page <> 0
  THEN
    POSITION (next_page);
  ELSE
    POSITION (end_of (current_buffer));
  ENDIF;
ENDPROCEDURE;
```

POSITION

Syntax

```
POSITION ( {
  buffer
  BUFFER_BEGIN
  BUFFER_END
  integer
  LINE_BEGIN
  LINE_END
  marker
  MOUSE
  range
  TEXT
  window
} )
```

Parameters

buffer

The buffer in which you want to establish the editing point.

DECTPU maintains an editing point in each buffer even when the buffer is not the current buffer. When you position to a buffer, the editing point that DECTPU maintains becomes the active editing point. The location at which POSITION establishes the editing point is the last character that the cursor was on when the buffer was most recently current.

BUFFER_BEGIN

A keyword that directs DECTPU to establish the editing point at the beginning of the current buffer. This is more efficient than using POSITION (BEGINNING_OF (CURRENT_BUFFER)).

BUFFER_END

A keyword that directs DECTPU to establish the editing point at the end of the current buffer. This is more efficient than using POSITION (END_OF (CURRENT_BUFFER)).

integer

The number of the record where you want DECTPU to position the editing point.

A record number indicates the location of a record in a buffer. Record numbers are dynamic; as you add or delete records, DECTPU changes the number associated with a particular record, as appropriate. DECTPU counts each record in a buffer, regardless of whether the line is visible in a window, or whether the record contains text.

To position the editing point to a given record, specify the record number. The number can be in the range from 1 to the number of records in the buffer plus 1. For example, the following statement positions the editing point to record number 8 in the current buffer:

```
POSITION (8);
```

DECTPU places the editing point on the first character of the record.

Specifying a value of 0 has no effect. Specifying a negative number or a number greater than the number of records in the buffer plus 1 causes DECTPU to signal an error.

LINE_BEGIN

A keyword that directs DECTPU to establish the editing point at the beginning of the current line.

LINE_END

A keyword that directs DECTPU to establish the editing point at the end of the current line.

marker

The marker to which you want to tie the editing point. You can position to either a bound marker or a free marker. For more information on the distinction between bound and free markers, see the *Guide to the DEC Text Processing Utility*. Positioning to a free marker does not cause DECTPU to insert padding blanks between the nearest text and the free marker; such positioning establishes the editing point as free. For more information on the distinction between free and detached editing points, see Appendix C.

MOUSE

A keyword that directs DECTPU to associate the editing point with the location of the pointer cursor.

In DECwindows DECTPU, you can use the statement `POSITION (MOUSE)` at any point after the first keyboard or mouse button event. The statement positions the editing point to the location occupied by the pointer cursor at the time of the most recent keyboard or mouse-button event.

If the pointer cursor is on a window's status line when `POSITION (MOUSE)` is executed, DECTPU positions the editing point at the line just above the status line.

Note

Be sure that you do not have scroll margins active when you execute `POSITION (MOUSE)`. If scroll margins are active, the result can be unpredictable. Use the `SET (CROSS_WINDOW_BOUNDS, OFF)` command prior to using `POSITION (MOUSE)`.

If the pointer cursor is not located in a DECTPU window at the time of the most recent keyboard or mouse-button event, `POSITION (MOUSE)` returns the status `TPU$_NOWINDOW`.

In non-DECwindows DECTPU, POSITION (MOUSE) is valid only during a procedure that is executed as the result of a mouse click. At all other times, the mouse position is not updated.

The statement POSITION (MOUSE) makes the window in which the pointer cursor is located the current window, and the buffer in which the pointer cursor is located the current buffer.

range

The range in which you want to place the editing point. The editing point is established at the beginning of the range. To establish the editing point at the end of the range, use the statement POSITION (END_OF (range)).

TEXT

A keyword that indicates that if the editing point is at a free-cursor location (a portion of the screen where there is no text), the POSITION built-in procedure is to establish the editing point at the nearest location that has a text character in it. The character may be a space or an end of line. If you use POSITION (TEXT) when the editing point is already bound to a character, the built-in has no effect.

window

The window in which you want to establish the editing point. The window must be mapped to the screen.

The location at which POSITION establishes the editing point is the last character that the cursor was on when the window was most recently current. If that character has been deleted, the editing point is the character closest to the last character that the cursor was on when the window was current.

Positioning to a window causes the buffer associated with the window to become the current buffer. This is true whether you directly position to a window, or a new window is mapped as the result of a POSITION (MOUSE) statement.

Description

The POSITION procedure ties the editing point to a specific character in a specific buffer, and moves the editing point to a specified record in the current buffer. The character and buffer in which POSITION establishes the editing point depend on which parameter you pass to POSITION.

The editing point is the location in the current buffer where most editing operations are carried out. DECTPU maintains a marker pointing to an editing point in each buffer, but only the editing point in the current buffer is active. An editing point, whose location is always tied to a character in a buffer, is not necessarily the same as the cursor position, whose location is always tied to a position in a window.

The POSITION built-in procedure synchronizes the editing point and the cursor position if the current buffer is mapped to a visible window. POSITION also moves the editing point to the specified record in the current buffer.

When you pass the MOUSE keyword to POSITION, the built-in establishes the mouse pointer's location as the cursor position. POSITION also establishes the window in which the mouse pointer is located as the current window, and establishes the buffer mapped to that window as the current buffer.

Positioning to a buffer, a marker, or a range does not necessarily move the cursor. DECTPU does not change the cursor position unless the cursor is in a window that is mapped to the buffer specified or implied by the POSITION parameter. For example, if you use POSITION to establish the editing point

in a buffer that is not mapped to a window, the cursor is unaffected by the POSITION operation. If you want to do visible editing, you should position to a window rather than a buffer.

If you try to position to an invisible window, DECTPU issues a warning message.

For more information on the relationship between the editing point and the cursor position, see Appendix C.

Signaled Errors

TPU\$_TOOFEW	ERROR	POSITION requires one parameter.
TPU\$_TOOMANY	ERROR	You specified more than one parameter.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_ARGMISMATCH	ERROR	Wrong type of data sent to the built-in.
TPU\$_BADKEY	WARNING	You specified an invalid keyword.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.
TPU\$_BADVALUE	ERROR	You specified a record number less than 0 or greater than the length of the buffer plus 1.
TPU\$_MOUSEINV	WARNING	The mouse position is not currently valid.
TPU\$_NOWINDOW	WARNING	The pointer cursor was not located in a DECTPU window at the time of the most recent keyboard or mouse-button event.
TPU\$_WINDNOTMAPPED	WARNING	Window is not mapped to the screens
TPU\$_WINDNOTVIS	WARNING	Window is totally occluded.

Examples

1. The following example establishes the editing point in the message window. Your position in the window is the same character position you occupied when you were last positioned in the window.

```
POSITION (message_window)
```

2. The following example toggles the active editing point between two windows:

```
PROCEDURE user_change_windows

  IF CURRENT_WINDOW = main_window
  THEN
    POSITION (extra_window);
  ELSE
    POSITION (main_window);
```

```
ENDIF;
ENDPROCEDURE;
```

QUIT

Syntax

```
QUIT [( { ON
         OFF
         1
         0 } [, severity ] )]
```

Parameters

ON, 1

Either keyword, ON or 1, indicates that DECTPU should prompt you to find out whether you want to quit with modified buffers. This is the default value.

OFF, 0

Either keyword, OFF or 0, indicates that DECTPU should quit without asking you whether to quit with modified buffers.

severity

If present, the least significant two bits of this integer are used as the severity of the status DECTPU returns to whatever invoked it. The following shows the values and their severity:

Value	Severity
0	Warning
1	Success
2	Error
3	Informational

You cannot force DECTPU to return a fatal severity status.

Description

The QUIT procedure leaves the editor without writing to a file. If you modify any buffers that are not set to NO_WRITE and you do not specify OFF as the first parameter to the QUIT built-in procedure, DECTPU tells you that you have modified buffers and asks whether you want to quit. Enter Y (YES) if you want to quit without writing out any modified buffers. Enter N (NO) if you want to retain the modifications you have made and return to the editor. If you specify OFF as the first parameter to QUIT, DECTPU quits without informing you that you have modified buffers. All modifications are lost because DECTPU does not write out buffers when quitting.

Journal files (if any) are deleted upon quitting.

Use the EXIT built-in procedure when you have made changes and want to save them when you leave the editor. (For more information, see the description of EXIT.)

When DECTPU quits, it usually returns a status of TPU\$_QUITTING to whatever invoked it. This is a success status.

This feature is useful if you are using DECTPU to create an application in which quitting, especially before the end of a series of statements executing in batch mode, is an error.

A special use of QUIT is at the end of your section file when you are compiling it for the first time. See the *Guide to the DEC Text Processing Utility* for information on creating section files.

Signaled Errors

TPU\$_CANCELQUIT	WARNING	A NO response was received from the ". . . continue quitting?" prompt.
TPU\$_TOOMANY	ERROR	QUIT accepts no more than two arguments.
TPU\$_INVPARAM	ERROR	One of the arguments to QUIT has the wrong data type.
TPU\$_BADKEY	WARNING	QUIT accepts only the keywords ON and OFF.
TPU\$_NORETURNVALUE	ERROR	QUIT does not return a value.

Examples

1. The following example returns control of execution from an editor layered on DECTPU to the program, application, or operating system that called DECTPU:

```
QUIT;
```

If you have modified any buffers, you see the following prompt:

```
Buffer modifications will not be saved, continue quitting (Y or N)?
```

Enter YES if you want to quit and not save the modifications. Enter NO if you want to return to the editor.

2. The following example turns off the display of the success message "Editor successfully quitting" when you use QUIT to leave an editing interface:

```
PROCEDURE user_quit
```

```
    SET (SUCCESS, OFF);
    QUIT;
```

```
! Turn message back on in case user answers "No" to the
! prompt "Buffer modifications will not be saved, continue
! quitting (Y or N)?"
```

```
    SET (SUCCESS, ON);
ENDPROCEDURE;
```

RAISE_WIDGET

Syntax

```
RAISE_WIDGET (widget)
```

Parameters

widget

The widget you want DECTPU to raise. The specified widget must be a subclass of WindowObjClass.

Description

The RAISE_WIDGET procedure places the widget at the top of a viewing stack above all sibling widgets. This ensures that the widget window associated with the widget is not obscured by any sibling windows. It calls the XLIB routine XRaiseWindow. The widget window is mapped if it is not already mapped.

Signaled Errors

TPU\$_INVPARAM	ERROR	The parameter to LOWER_WIDGET has the wrong data type.
TPU\$_NORETURNVALUE	ERROR	This built-in does not return a result.
TPU\$_NOTSUBCLASS	WARNING	The parameter to LOWER_WIDGET is not a widget that has an associated widget window.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.

READ_CHAR

Syntax

string := READ_CHAR

Parameters

None.

Return Value

A variable of type string that contains a character entered from the keyboard.

Description

The READ_CHAR procedure stores the next character entered from the keyboard in a string variable. The character read by READ_CHAR is not echoed on the screen; therefore, the cursor position does not move.

READ_CHAR does not process escape sequences. If a DECTPU procedure uses READ_CHAR for an escape sequence, only part of the escape sequence is read. The remaining part of the escape sequence is

treated as text characters. If control then returns to DECTPU, or a `READ_KEY` or `READ_LINE` built-in procedure is executed, the results may be unpredictable.

In DECwindows DECTPU, `READ_CHAR` maps the main window if it is not already mapped.

In the DECwindows environment, `READ_CHAR` cannot read a keypad or function key. If a DECTPU procedure uses `READ_CHAR` and you press a keypad or function key, `READ_CHAR` returns a null string and signals the warning `TPU$_NOCHARREAD`.

DECwindows applications that execute `READ_CHAR` built-ins should use error handlers that contain the `TPU$_READABORTED` selector. DECTPU signals that error if a `READ_CHAR` built-in is aborted by any of the following events: resize, widget callback, loss of primary selection, and client message.

The code associated with that selector should return from the procedure by executing either an `ABORT` or `RETURN` statement. If instead of returning, the procedure executes another `READ_CHAR` built-in, DECTPU enters an infinite loop.

`READ_CHAR` does not abort for input focus events.

When you use the `/NODISPLAY` qualifier to invoke DECTPU, `READ_CHAR` signals `TPU$_REQUIRESTERM`, "Feature requires a terminal", if `SYS$INPUT` is not a terminal.

Signaled Errors

<code>TPU\$_NOCHARREAD</code>	WARNING	<code>READ_CHAR</code> did not read a character.
<code>TPU\$_NEEDTOASSIGN</code>	ERROR	<code>READ_CHAR</code> must be on the right-hand side of an assignment statement.
<code>TPU\$_TOOMANY</code>	ERROR	<code>READ_CHAR</code> takes no arguments.

Examples

1. The following example stores the next character that is entered on the keyboard in the string `new_char`:

```
new_char := READ_CHAR
```

2. The following example enters the next character that is entered from the keyboard in the current buffer. If a key that sends an escape sequence is pressed, the first character of the escape sequence is copied into the buffer. Subsequent keystrokes are interpreted as self-inserting characters, defined keys, or undefined keys, as appropriate.

```
PROCEDURE user_quote
    COPY_TEXT (READ_CHAR);
ENDPROCEDURE;
```

3. The following example uses a coding style that avoids an infinite loop. In this example, DECTPU aborts a `READ_CHAR` built-in when a widget callback occurs. The error handler then returns from that procedure.

```
procedure get_a_char (the_char)
on_error
    [TPU$_READABORTED]:
        message ("Prompt terminated.", 0);
```

```

        return;
endon_error;
loop
...
    the_key := read_char;
...
endloop;
endprocedure;

```

READ_CLIPBOARD

Syntax

```

[[ range
 unspecified := READ_CLIPBOARD
 ]]

```

Parameters

None.

Return Values

range

A range that contains the text copied into the current buffer.

unspecified

A data type that indicates that no data was obtained from the clipboard.

Description

The READ_CLIPBOARD procedure reads string format data from the clipboard and copies it into the current buffer, at the editing point, using the buffer's current text mode (insert or overstrike). If DECTPU finds a line-feed character in the data, it removes the line feed and any adjacent carriage returns and puts the data after the line feed on the next line of the buffer. If DECTPU must truncate the data from the clipboard, DECTPU copies the truncated text into the current buffer.

All text read from the clipboard is copied into the buffer starting at the editing point. If DECTPU must start a new line to fit all the text into the buffer, the new line starts at column 1, even if the current left margin is not set at column 1.

Signaled Errors

TPU\$_CLIPBOARDLOCKED	WARNING	DECTPU cannot read from the clipboard because some other application has locked it.
TPU\$_CLIPBOARDNODATA	WARNING	There is no string format data in the clipboard.
TPU\$_CLIPBOARDFAIL	WARNING	The clipboard did not return any data.
TPU\$_REQUIRESDECW	ERROR	You can use the READ_CLIPBOARD built-

		in only if you are using DECwindows TPU.
TPU\$_STRTOOLARGE	ERROR	The amount of data in the clipboard exceeds 65535 characters.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the READ_CLIPBOARD built-in.

Example

The following example shows one possible way that an application can use the READ_CLIPBOARD built-in procedure. This procedure is a modified version of the EVE procedure EVE\$_INSERT_CLIPBOARD. The original version is in SYS\$EXAMPLE S:EVE\$DECWINDOWS.TPU.

```
PROCEDURE eve$$insert_clipboard

ON_ERROR
  [TPU$_CLIPBOARDNODATA]:
    eve$message (EVE$_NOINSUSESEL);
    eve$learn_abort;
    RETURN (FALSE);
  [TPU$_CLIPBOARDLOCKED]:
    eve$message (EVE$_CLIPBDREADLOCK);
    eve$learn_abort;
    RETURN (FALSE);
  [TPU$_TRUNCATE]:
  [OTHERWISE]:
    eve$learn_abort;
  ENDON_ERROR;

IF eve$test_if_modifiable (CURRENT_BUFFER)
THEN
  READ_CLIPBOARD;                                ! This statement using
                                                  ! READ_CLIPBOARD reads
                                                  ! data from the clipboard
                                                  ! and copies it into the
                                                  ! current buffer.

  RETURN (TRUE);
ENDIF;

eve$learn_abort;
RETURN (FALSE);

ENDPROCEDURE;
```

EVE\$_INSERT_CLIPBOARD fetches the contents of the clipboard and places them in the current buffer.

READ_FILE

Syntax

```
[[string2 := ] READ_FILE (string1)
```

Parameter

string1

A string that is the name of the file you want to read and include in the current buffer.

Return Value

A string that is the specification of the file read.

Description

The READ_FILE procedure reads a file and inserts the contents of the file immediately before the current line in the current buffer. READ_FILE optionally returns a string that contains the file specification of the file read.

If the current buffer is mapped to a visible window, the READ_FILE built-in procedure causes the screen manager to synchronize the editing point (which is a buffer location) with the cursor position (which is a window location). This may result in the insertion of padding spaces or lines into the buffer if the cursor position is before the beginning of a line, in the middle of a tab, beyond the end of a line, or after the last line in the buffer.

DECTPU writes a message that indicates how many records (lines) were read.

If you try to read a file that contains lines longer than 32767 characters, DECTPU truncates the lines to the first 32767 characters and issues a warning.

Note

If you delete a file after using READ-FILE to insert the file into a buffer, you will not be able to recover the buffer. This is because DECTPU requires the original source file to recover when using a buffer-change journal file.

Signaled Errors

TPU\$_NOCURRENTBUF	WARNING	You are not positioned in a buffer.
TPU\$_CONTROLC	ERROR	The execution of the read terminated because you pressed Ctrl/C.
TPU\$_NOCACHE	ERROR	There is not enough memory to allocate a new cache.
TPU\$_TOOFEW	ERROR	READ_FILE requires at least one parameter.
TPU\$_TOOMANY	ERROR	READ_FILE accepts no more than one parameter.
TPU\$_INVPARAM	ERROR	The parameter to READ_FILE must be a string.
TPU\$_TRUNCATE	WARNING	One of the lines in the file was too long to fit in a DECTPU buffer.

The following errors, warnings, and messages can be signaled by DECTPU's file I/O routine. You can provide your own file I/O routine by using DECTPU's callable interface. If you do so, READ_FILE's signaled errors, warnings, and messages depend upon what status you signaled in your file I/O routine.

TPU\$_OPENIN	ERROR	READ_FILE could not open the file you specified.
TPU\$_READERR	ERROR	READ_FILE did not finish reading the file because it encountered a file system error.
TPU\$_CLOSEIN	ERROR	READ_FILE did not finish closing the file because it encountered a file system error.

Examples

1. The following example reads the file LOGIN.COM and adds it to your current buffer:

```
READ_FILE ("login.com")
```

2. The following example creates a second window and a second buffer and maps the window to the screen. The procedure also prompts you for a file name to include in the buffer and defines the key sequence Shift/W (the Shift key follow by W) as the sequence with which to move to the second window. (The default shift key is PF1.)

```
PROCEDURE user_two_windows

    w := CREATE_WINDOW (1, 10, ON);
    b := CREATE_BUFFER ("buf2");

    MAP (w, b);

    READ_FILE (READ_LINE ("Enter file name for 2nd window : "));

    POSITION (BEGINNING_OF (b));

    DEFINE_KEY ("POSITION (w)", KEY_NAME ("W", SHIFT_KEY));
ENDPROCEDURE;
```

READ_GLOBAL_SELECT

Syntax

$$\llbracket \left\{ \begin{array}{l} \text{range} \\ \text{unspecified} \end{array} \right\} := \rrbracket \text{ READ_GLOBAL_SELECT } \left(\left\{ \begin{array}{l} \text{PRIMARY} \\ \text{SECONDARY} \\ \text{selection_name} \end{array} \right\}, \right. \\ \left. \text{selection_property_name} \right)$$

Parameters

PRIMARY

A keyword that indicates that the application is requesting information about a property of the primary global selection.

SECONDARY

A keyword that indicates that the application is requesting information about a property of the secondary global selection.

selection_name

A string that identifies the global selection whose property is the subject of the application's information request. Specify the selection name as a string if the layered application needs information about a selection other than the primary or secondary global selection.

selection_property_name

A string that specifies the property whose value the application is requesting.

Return Values**range**

A range that contains the text copied into the current buffer.

unspecified

A data type that indicates that the information requested by the application was not available.

Description

The READ_GLOBAL_SELECT procedure requests information about the specified global selection from the owner of the global selection. For example, you can ask about the global selection's font, the number of lines it contains, or the string-formatted data it contains, if any. If the owner provides the information, READ_GLOBAL_SELECT reads it and copies it into the current buffer at the editing point, using the buffer's current text mode (insert or overstrike). The READ_GLOBAL_SELECT built-in procedure also puts line breaks in the text copied into the buffer.

All text read from the global selection is copied into the current buffer, starting at the editing point. If DECTPU must start a new line to fit all the text into the buffer, the new line starts at column 1, even if the current left margin is not set at column 1.

If the global selection information requested is an integer, the built-in converts the integer into a string before copying it into the current buffer. If the information requested is a string, the built-in copies the string into the buffer, replacing any line feeds with line breaks. Carriage returns adjacent to line feeds are not copied into the buffer.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_GBLSELOWNER	WARNING	DECTPU owns the global selection.
TPU\$_INVGBLSELDATA	WARNING	The global selection owner provided data that DECTPU cannot process.
TPU\$_NOGBLSELDATA	WARNING	The global selection owner indicated that it cannot provide the information requested.

TPU\$_NOGBLSELOWNER	WARNING	You requested information about an unowned global selection.
TPU\$_TIMEOUT	WARNING	The global selection owner did not respond before the timeout period expired.
TPU\$_ARGMISMATCH	ERROR	Wrong type of data sent to the READ_GLOBAL_SELECT built-in.
TPU\$_REQUIRESDECW	ERROR	You can use the READ_GLOBAL_SELECT built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the READ_GLOBAL_SELECT built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the READ_GLOBAL_SELECT built-in.

Examples

The following example reads the string-formatted contents of the primary global selection and copies it into the current buffer at the current location:

```
READ_GLOBAL_SELECTION (PRIMARY, "STRING");
```

READ_KEY

Syntax

keyword := READ_KEY

Parameters

None.

Return Value

A key name for the key just pressed.

Description

The READ_KEY procedure waits for you to press a key and then returns the key name for that key. READ_KEY should be used rather than READ_CHAR when you are entering escape sequences, control characters, or any characters other than text characters. READ_KEY processes escape sequences and DECTPU's shift key (PF1 by default).

The key that is read by READ_KEY is not echoed on the terminal screen.

When you invoke DECTPU with the /NODISPLAY qualifier, READ_KEY signals TPU\$_REQUIRESTERM, "Feature requires a terminal", if SY\$\$INPUT is not a terminal.

In DECwindows DECTPU, `READ_KEY` maps the main window if it is not already mapped.

DECwindows applications that execute `READ_KEY` built-ins should use error handlers that contain the `TPU$_READABORTED` selector. DECTPU signals that error if a `READ_KEY` built-in is aborted by any of the following events: resize, widget callback, loss of primary selection, and client message.

The code associated with that selector should return from the procedure by executing either an `ABORT` or `RETURN` statement. If instead of returning, the procedure executes another `READ_KEY` built-in, DECTPU enters an infinite loop.

`READ_KEY` does not abort for input focus events.

Signaled Errors

<code>TPU\$_NEEDTOASSIGN</code>	ERROR	<code>READ_KEY</code> must be on the right-hand side of an assignment statement.
<code>TPU\$_TOOMANY</code>	ERROR	<code>READ_KEY</code> accepts no arguments.
<code>TPU\$_CONTROLC</code>	ERROR	You pressed Ctrl/C during the execution of <code>READ_KEY</code> .
<code>TPU\$_REQUIRESTERM</code>	ERROR	You cannot use <code>READ_KEY</code> when DECTPU is in <code>NODISPLAY</code> mode.

Examples

1. The following example reads the next key that is entered and stores the keyword for that key in the variable `my_key`:

```
my_key := READ_KEY
```

2. The following example looks in the current key map list for the next key pressed. If the key is found, any comment associated with that key is put into the message buffer.

```
PROCEDURE user_help_on_key

    LOCAL key_pressed,
           key_comment;

    MESSAGE ("Press the key you want help on.");

    key_pressed := READ_KEY;
    key_comment := LOOKUP_KEY (key_pressed, COMMENT);

    IF key_comment = 0
    THEN MESSAGE
        ("That key is not defined.");
    ELSE
        IF key_comment = ""
        THEN
            MESSAGE ("There is no comment for that key.");
        ELSE
            MESSAGE (key_comment);
        END IF;
    END IF;
END PROCEDURE
```



```
        ENDIF;  
    ENDIF;  
ENDPROCEDURE;
```

3. The following example uses a coding style that avoids an infinite loop. In this example, DECTPU aborts a READ_KEY built-in when a widget callback occurs. The error handler then returns from that procedure.

```
procedure get_a_key (the_key)  
on_error  
    [TPU$_READABORTED]:  
        message ("Prompt terminated.", 0);  
        return;  
endon_error;  
loop  
    ...  
    the_key := read_key;  
    ...  
endloop;  
endprocedure;
```

READ_LINE

Syntax

```
string2 := READ_LINE [ (string1 [ ,integer ] ) ]
```

Parameters

string1

A string that is the text used as a prompt for input. The maximum length is 255 characters. This parameter is optional.

integer

The integer value that indicates how many characters to read from the input entered in response to the prompt. The maximum number is 132. This parameter is optional. If not present, control of execution passes from READ_LINE to DECTPU's main loop when you press Return, Ctrl/Z, or the one hundred thirty-second character.

Return Value

A string that stores your response to a prompt.

Description

The READ_LINE procedure displays the text that you specify as a prompt for input and reads the information entered in response to the prompt. You can optionally specify the maximum number of characters to be read. READ_LINE returns your data string response to the prompt.

The terminators for READ_LINE are the standard OpenVMS terminators such as Ctrl/Z and the Return key. READ_LINE is not affected by DECTPU key definitions; the built-in takes literally all keys except standard OpenVMS terminators.

By default, the text you specify as a prompt is written in the prompt area on the screen. The prompt area is established with the SET (PROMPT_AREA) built-in procedure. See SET (PROMPT_AREA) for more information.

If no prompt area is defined, the text specified as a prompt is displayed at the current location on the device pointed to by SYS\$OUTPUT (usually your terminal).

If READ_LINE terminates because it reaches the limit of characters specified as the second parameter, the last character read becomes the last key. See the example section for a procedure that tests for the last key entered in a prompt string.

In DECwindows DECTPU, READ_LINE maps the main widget if it is not already mapped.

When you invoke DECTPU with the /NODISPLAY qualifier, terminal functions such as screen display and key definitions are not used. The READ_LINE built-in procedure calls the LIB\$GET_INPUT routine to issue a prompt to SYS\$INPUT and accept input from you. A read done this way does not terminate when the number of keys you specified as the second parameter (*integer*) are entered. However, *string2* contains the number of characters specified by the integer parameter, and LAST_KEY contains the value of the key that corresponds to the integer specified as the last key to be read, except in the following cases: If the read is terminated by Ctrl/Z, LAST_KEY has the value Ctrl/Z; if the read is terminated by a carriage return before the specified integer limit is reached, LAST_KEY has the value of the Return key.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	READ_LINE must appear on the right-hand side of an assignment statement.
TPU\$_TOOMANY	ERROR	READ_LINE accepts no more than two arguments.
TPU\$_INVPARAM	ERROR	One of the arguments to READ_LINE has the wrong data type.

Examples

1. The following example displays the text "Enter key definition:" in the prompt area, and stores the first character of your response in the variable *my_prompt*:

```
my_prompt := READ_LINE ("Enter key definition:", 1)
```

2. The following example prompts for three characters and stores them in the variable *my_input*:

```
PROCEDURE user_test_lastkey

    LOCAL my_key,
           k;

    my_input := READ_LINE ("Enter 3 characters:", 3);

    ! Press the keys "ABC"

    my_key := LAST_KEY;
    IF my_key = KEY_NAME ("C")
```

```

THEN
    MESSAGE (" C key ");
ELSE
    MESSAGE (" Error ");
ENDIF;
ENDPROCEDURE;

```

It then tests for the last key entered.

3. The following example is used by commands that prompt for integers. The procedure returns true if prompting worked or was not needed; it returns false otherwise. The returned value is passed back as an output parameter.

```

! Parameters:
!
!   old_number           Old integer value - input
!   new_number          New integer value - output
!   prompt_string       Text of prompt - input
!   no_value_message    Message printed if user hits RETURN to
!                       get out of the command - input

PROCEDURE user_prompt_number (old_number, new_number,
                             prompt_string, no_value_message)

! String read after prompt

LOCAL read_line_string;

new_number := old_number;
IF old_number < 0
    THEN
        read_line_string := READ_LINE (prompt_string);
        EDIT (read_line_string, TRIM);
        IF read_line_string = "
        THEN
            MESSAGE (no_value_message);
            new_number := 0;
            RETURN (0);
        ELSE
            ! Change lowercase l to #1
            TRANSLATE (read_line_string, "l", "1");
            new_number := INT (read_line_string);
            IF (new_number = 0) and (read_line_string <> "0")
                THEN
                    MESSAGE (FAO ("Don't understand !AS",
                                   read_line_string));
                    RETURN (0);
                ELSE
                    RETURN (1);
            ENDIF;
        ENDIF;
    ELSE
        RETURN (1);
    ENDIF;
ENDIF;
ENDPROCEDURE;

```

REALIZE_WIDGET

Syntax

REALIZE_WIDGET (widget)

Parameters

widget

The widget you want DECTPU to realize.

Description

The REALIZE_WIDGET procedure creates a widget window for the specified widget and, if a composite widget, recursively realizes all the widget's managed children. REALIZE_WIDGET interacts with the widget's mapped_when_managed bit. The setting of this bit determines whether or not DECTPU maps the widget window to the display. See the *VMS DECwindows Toolkit Routines Reference Manual* for a complete explanation.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	REALIZE_WIDGET must return a value.
TPU\$_TOOMANY	ERROR	Too many arguments specified.
TPU\$_TOOFEW	ERROR	Too few arguments specified.
TPU\$_INVPARAM	ERROR	The argument to REALIZE_WIDGET has the wrong data type.
TPU\$_REQUIRESDECW	ERROR	Requires the DECTPU DECwindows screen updater.

RECOVER_BUFFER

Syntax

buffer1 := RECOVER_BUFFER (string1 { [,string2] }
 { [,buffer2] })

Parameters

string1

The name of the buffer you are trying to recover.

string2

The name of the journal file you want DECTPU to use to recover your buffer. If you did not use SET (JOURNALING) to set a journal file name, in most cases DECTPU will have created the journal file by using its default journal file naming algorithm. If the journal file was named by default, you need not specify a journal file name with RECOVER_BUFFER. If you specified a journal file name by using SET (JOURNALING), use the same name with RECOVER_BUFFER.

Do not specify any directory name in this string. Specify only the buffer name and the extension, if any.

buffer2

The buffer whose attributes you want applied to the newly created buffer. For more information on using a buffer as a template, see the description of the `CREATE_BUFFER` built-in procedure.

Return Value

The buffer that contains the recovered text. If the recovery failed, the integer 0 is returned.

Description

The `RECOVER_BUFFER` procedure reconstructs the work done in the buffer whose name you specify. DECTPU uses the specified buffer name to create a new buffer. It then uses the information in the original buffer's journal file to recover all the changes made to records in the original file. The resulting recovery is written to the newly created buffer.

Do not confuse the `RECOVER_BUFFER` built-in procedure with the `/RECOVER` qualifier. You use the `/RECOVER` qualifier to invoke DECTPU to recover a session or buffer. `RECOVER_BUFFER`, on the other hand, is used after DECTPU has been invoked. It uses a buffer-change journal file to recover the changes made to a specified buffer.

`RECOVER_BUFFER` works only with buffer-change journaling; you cannot recover a keystroke journal file with `RECOVER_BUFFER`.

Only the first parameter (the old buffer name) is required. If you want to specify the third parameter but not the second, you must use a comma as a placeholder, as follows:

```
RECOVER_BUFFER ("junk.txt", , template_buffer);
```

The third parameter is optional.

If some text is missing after recovery, the reason might be that the last few changes did not trigger a write operation. For more information on how DECTPU manages write operations to a journal file, see the description of the `SET (JOURNALING)` built-in procedure.

Buffer-change journaling does not journal changes in buffer attributes (such as modifiability of the buffer or visibility of tabs). Buffer-change journaling tracks only changes to records in the buffer, such as addition, deletion, or modification of a record or changes in a record's attributes.

If you press `Ctrl/C` during a recovery, DECTPU terminates the recovery, closes the journal file, and deletes the newly created buffer.

If possible, after a successful recovery, DECTPU continues journaling new changes into the journal file that was used during the recovery. However, it is likely that the journal file contains partial records at the end. In this case, DECTPU cannot continue journaling to the same file. DECTPU closes the journal file, marks the buffer unsafe for journaling, and signals an error.

Note

Be careful when using the default naming algorithm while editing multiple buffers. If DECTPU created your journal file as a result of editing multiple buffers with the same or similar names, `RECOVER_BUFFER` might not recover the buffer you intend.

For more information on the default journal file naming algorithm, see the *Guide to the DEC Text Processing Utility*. For example, suppose you were editing two buffers, one called TEST! and the other called TEST?. The default journal file naming algorithm creates for each buffer a journal file named TEST_ .TPU\$JOURNAL. The journal file for the buffer created first has the lower version number. If there were a system interruption while you were editing both buffers, and if the journal file for TEST! had the lower version number, then RECOVER_BUFFER would recover the journal file created for the buffer TEST?.

When you write the contents of a buffer to a file, DECTPU erases the journal file. If you write the contents of the buffer to a file other than the default output file, the journal file contains a pointer to the file to which you last wrote the buffer. For example, if the buffer is called MAIN but you write the contents of the buffer to a file called OPUS.TXT, the journal file contains a pointer to the file OPUS.TXT. OPUS.TXT is known as the “source file” because during a recovery DECTPU uses OPUS.TXT as the source for the contents of the buffer as they were when the write operation was performed.

Caution

If you delete the source file, or any of the files read into the buffer, the buffer becomes unrecoverable.

Similarly, if you have changed the name of the original files required for a recovery, the buffer will be unrecoverable. DECTPU prompts for a new file name if it cannot find the original file. Be careful to specify the correct file name in this case.

You must use the same major version of DECTPU to recover the journal that you used to create it.

Signaled Errors

TPU\$_JRNLNOTSAFE	WARNING	The buffer is not safe for journaling.
TPU\$_NOTJOURNAL	ERROR	The file specified is not a valid journal file.
TPU\$_RECOVERABORT	WARNING	An inconsistency was found between the journal file and the currently executing procedure. Recovery is aborted and the journal file closed.
TPU\$_RECOVERFAIL	ERROR	Recovery was terminated abnormally due to journal file inconsistency.
TPU\$_RECOVERQUIT	WARNING	You did not specify a valid source file name.

Examples

1. The following example directs DECTPU to find the buffer-change journal file associated with the original buffer JUNK.TXT and to create a new buffer called JUNK.TXT. Also, it uses the information from the journal file to recover the changes made in the original JUNK.TXT buffer. The results of the recovery are placed in the new JUNK.TXT buffer.

```
RECOVER_BUFFER ("JUNK.TXT");
```

2. The following example creates a default buffer, changes an attribute of the default buffer, and creates a user buffer. The fourth statement turns on buffer-change journaling and designates the file named USER1_JOURNAL.TPU\$JOURNAL as the journaling file. At some later point in the session (represented by the ellipsis) the RECOVER_BUFFER statement is used to recover the contents of the old USER1.TXT by using the journal file USER1_JOURNAL.TPU\$JOURNAL. The attributes of the default buffer are applied to the newly created buffer USER1.TXT. In this case, the new buffer has the end-of-buffer text "[That's all, folks!]".

```
defaults_buffer := CREATE_BUFFER ("Defaults");  
  
SET (EOB_TEST, defaults_buffer, "[That's all, folks!]);  
  
user_buffer := CREATE_BUFFER ("User1.txt", "", defaults_buffer);  
  
SET (JOURNALING, user_buffer, ON, "user1_journal.tpu$journal");  
.  
.  
.  
RECOVER_BUFFER ("User1.txt", "user1_journal.tpu$journal",  
                defaults_buffer);
```

REFRESH

Syntax

REFRESH

Parameters

None.

Description

The REFRESH procedure repaints the whole screen. REFRESH erases any extraneous characters, such as those caused by noise on a communication line, and repositions the text so that the screen represents the last known state of the editing context.

REFRESH causes a redrawing of every line of every window that is mapped to the screen. The prompt area is erased. This built-in procedure causes the screen to change immediately. Even if REFRESH is issued from within a procedure, the action takes place immediately; DECTPU does not wait until the entire procedure is completed to execute REFRESH.

If screen updating is disabled when DECTPU executes the REFRESH command, DECTPU performs the refresh operation when updating is enabled again.

DECTPU reissues escape sequences as appropriate to do any of the following:

- To set the width of the terminal
- To set the scrolling region
- To set the keypad to applications mode
- To set the video attributes to a known state

- To clear the screen of a Compaq-supported terminal
- To reset the nonalphanumeric character sets

REFRESH repaints the whole screen. See UPDATE for a description of how to update a single window to make it reflect the current state of its associated buffer. If you want to update every visible window without erasing the screen, use the UPDATE (ALL) built-in procedure.

See Appendix C for an explanation of how the screen is updated under various circumstances.

Signaled Errors

TPU\$_TOOMANY	ERROR	REFRESH takes no parameters.
---------------	-------	------------------------------

Examples

1. The following example causes the screen manager to repaint the whole screen so that it reflects the current internal state of the editor:

```
REFRESH
```

2. The following example removes the contents of the message buffer and then repaints the whole screen:

```
PROCEDURE user_repaint
    ERASE (message_buffer);
    REFRESH;
ENDPROCEDURE;
```

REMAIN

Syntax

REMAIN

Parameters

None.

Description

The REMAIN procedure specifies that all characters from the current position to the end of the line should be included in a pattern. When used as part of a complex pattern or as an argument to SEARCH, REMAIN matches the rest of the characters on a line. REMAIN matches successfully even if there are no more characters on the line. Although REMAIN behaves much like a built-in, it is actually a keyword.

Signaled Errors

REMAIN is a keyword and has no completion codes.

Examples

1. The following example stores in the variable *pat1* a pattern that matches all lines that have an exclamation point at the beginning of the line:


```
pat1 := LINE_BEGIN + "!" + REMAIN
```

2. The following example removes all comments from the current buffer. It does not correctly handle quoted strings that contain exclamation points.

```
PROCEDURE remove_comments

    LOCAL pat1,
           here,
           comment_range;
    here := MARK (NONE);    ! Remember our location
    pat1 := "!" + REMAIN;

    POSITION (BEGINNING_OF (CURRENT_BUFFER));
    LOOP
        comment_range := SEARCH_QUIETLY (pat1, FORWARD);
        EXITIF comment_range = 0;

        ERASE (comment_range);
        POSITION (comment_range);
    ENDLLOOP;

    POSITION (here);
ENDPROCEDURE;
```

REMOVE_KEY_MAP

Syntax

```
REMOVE_KEY_MAP (string1, string2 [, ALL ])
```

Parameters

string1

A quoted string, or a variable name representing a string constant, that specifies the name of the key map list containing the key map to be removed.

string2

A quoted string, or a variable name representing a string constant, that specifies the name of the key map to be removed from the key map list.

ALL

This keyword is an optional argument. It specifies that all the key maps with the name specified by *string2* in the key map list are to be removed.

Description

The REMOVE_KEY_MAP procedure removes one or more key maps from a key map list. If the optional ALL keyword is specified, all of the key maps with the specified name in the key map list are removed from the list; otherwise, only the first entry with the specified name is removed.

Signaled Errors

TPU\$_NOKEYMAP	WARNING	You specified an argument that is not a defined key map.
TPU\$_NOKEYMAPLIST	WARNING	You specified an argument that is not a defined key map list.
TPU\$_KEYMAPNOTFND	WARNING	The key map you specified is not found.
TPU\$_EMPTYKMLIST	WARNING	The key map list you specified contains no key maps.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the REMOVE_KEY_MAP built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the REMOVE_KEY_MAP built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the REMOVE_KEY_MAP built-in.
TPU\$_UNKKEYWORD	ERROR	An unknown keyword was used as an argument. Only the ALL keyword is allowed.
TPU\$_BADKEY	ERROR	An unknown keyword was used as an argument. Only the ALL keyword is allowed.

Example

In the following example, a key map list named KEYMAP_LIST is created. The call to SHOW (KEY_MAP_LISTS) shows that the key map list contains three key maps: KEYMAP_1, KEYMAP_2, and KEYMAP_1 again. After the call to REMOVE_KEY_MAP, the call to SHOW (KEY_MAP_LISTS) shows that the key map list contains only KEYMAP_2.

```

user$keymap_1 := CREATE_KEY_MAP ("keymap_1");
user$keymap_2 := CREATE_KEY_MAP ("keymap_2");
user$keymap_list := CREATE_KEY_MAP_LIST ("keymap_list", user$keymap_1,
                                         user$keymap_2);

ADD_KEY_MAP (user$keymap_list, "last", user$keymap_1);
.
.
.
SHOW (KEY_MAP_LISTS);
.
.
.
REMOVE_KEY_MAP (user$keymap_list, user$keymap_1, ALL);
.
.
.
SHOW (KEY_MAP_LISTS);

```

RETURN

Syntax

RETURN [expression]

Parameters

expression

This expression can be any DECTPU expression, variable, or built-in. It specifies what the current procedure should return to its caller. If not specified, RETURN returns 0. The expression can be enclosed within optional parentheses.

Description

The RETURN procedure is a DECTPU language element; it returns control from the current procedure to its caller, optionally specifying the value the current procedure returns to the caller.

RETURN is evaluated for correct syntax at compile time. In contrast, DECTPU procedures are usually evaluated for a correct parameter count and parameter types at execution time.

Signaled Errors

RETURN is a language element and signals no errors or warnings.

Examples

1. The following example erases the message buffer. If the current buffer is the message buffer, it returns without erasing it.

```
PROCEDURE user_erase_message_buffer
  IF CURRENT_BUFFER = message_buffer
  THEN
    RETURN;
  ENDIF;

  ERASE (message_buffer);
ENDPROCEDURE;
```

2. The following example searches for a string. If it does not find the string, it returns the string "String not found"; otherwise, it returns the range containing the found string.

```
PROCEDURE user_find_string (look_for)
  ON_ERROR
    RETURN "String not found";
  ENDON_ERROR;

  RETURN SEARCH (look_for, FORWARD);
ENDPROCEDURE;
```

SAVE

Syntax

```
SAVE (string1 [,"NO_DEBUG_NAMES"]  
      [,"NO_PROCEDURE_NAMES"]  
      [,"IDENT", string2])
```

Parameters

string1

A string that is a valid file specification. If you supply only a file name, DECTPU uses the *current* device and directory, not necessarily the SYSS\$LOGIN device and directory, in the file specification.

"NO_DEBUG_NAMES"

A string that prevents DECTPU from writing debugging information to the section file. When you use "NO_DEBUG_NAMES", DECTPU does not write procedure parameter names or local variable names. You can reduce the size of the section file by specifying this string. Do not specify this string if you intend to use the DECTPU debugger on the section file.

"NO_PROCEDURE_NAMES"

A string, or a variable or constant name representing this string, that prevents DECTPU from writing procedure names to the section file. You can reduce the size of the section file by specifying this string. However, the procedure names are required to display a meaningful traceback when an error occurs. Therefore, do not specify this string if you want to use the application created by the section file with the TRACEBACK or LINE_NUMBER function set to ON.

"IDENT"

A string that specifies that you want to assign an identifying string, such as a version number, to the section file.

string2

The string (usually a version number) that you want to assign to the section file.

Description

The SAVE procedure writes the binary forms of all currently defined procedures, variables, key definitions, key maps, and key map lists to the section file you specify. Use SAVE to create DECTPU section files. If you are adding to an existing section file, the new section file contains all of the items from the original section file and the new items from the current editing session.

Section files enable DECTPU interfaces to start up quickly because they contain the following items in binary form:

- All compiled PROCEDURE . . . ENDPROCEDURE statements
- Every variable created (only the variable's name is saved, not its contents)
- Every key definition that binds a statement, procedure, program, or learn sequence to a key, including the comments that you add to key definitions
- Every key map and key map list created

- All defined constants

When you use the SAVE built-in procedure during an editing session to add items to an existing section file, SAVE does not keep items that were established interactively with the SET built-in procedure (for example, margin settings for buffers, or setting the editor's shift key to something other than the PF1 key).

If you do not specify a device and directory in the string parameter, DECTPU uses your current device and directory. The default file type is TPU\$SECTION.

File backups are automatically provided by RMS.

When you use the SAVE built-in procedure, informational messages are generated for any undefined procedures or ambiguous symbols as they are written to the section file. If the display of informational messages has been disabled, these messages are not displayed.

Signaled Errors

TPU\$_SAVEERROR	ERROR	The section cannot be created because of errors in the context being saved.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SAVE built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SAVE built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the SAVE built-in.
TPU\$_SECTUNDEFPROC	WARNING	Undefined procedures or ambiguous symbols were found while the section file was being written.
TPU\$_BADSYMTAB	ERROR	DECTPU's symbol tables are inconsistent.
TPU\$_SAVEUNDEFPROC	INFORMATIONAL	An undefined procedure is being written to the section file.
TPU\$_SAVEAMBIGSYM	INFORMATIONAL	An ambiguous symbol is being written to the section file.

Examples

1. The following example procedure, issued just before exiting from the editor, adds all of the procedure definitions, key definitions, and variables from your current editing session to the OpenVMS section file with which you invoked DECTPU. The new file that you specify, SYS\$LOGIN:mysection.TPU\$SECTION, contains initialization items from the original section file and from your editing session.

```
SAVE ("SYS$LOGIN:mysection.TPU$SECTION")
```

To invoke DECTPU with the new section file, enter the following command at the DCL level:

```
$ EDIT/TPU/SECTION=sys$login:mysection
```

2. The following example shows how you can use SAVE in a command file to extend an application. The first procedure moves the cursor to the beginning of the next paragraph. The second procedure defines a shift key and binds the procedure *eve_next_paragraph* to the period key on the keypad. The SAVE statement directs DECTPU to write the binary form of *eve_next_paragraph* and the key definition to a section file called MY_SECTION.TPU\$SECTION. The second and third parameters to the SAVE statement direct DECTPU to assign the string "V1.5" to the section file. The QUIT statement terminates the DECTPU session.

```
PROCEDURE eve_next_paragraph

    LOCAL pat1,
           the_range;

    pat1 := LINE_BEGIN + LINE_BEGIN + ARB (1);
    the_range := SEARCH_QUIETLY (pat1, FORWARD, EXACT);

    IF the_range <> 0
    THEN
        POSITION (END_OF (the_range));
    ENDIF;
ENDPROCEDURE;

PROCEDURE tpu$local_init
    SET (SHIFT_KEY, KP0);
    DEFINE_KEY ("eve_next_paragraph", PERIOD, "Next Para");
ENDPROCEDURE;

SAVE ("my_section", "ident", "V1.5");
QUIT;
```

SCAN

Syntax

$$\text{pattern} := \text{SCAN} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \llbracket, \left\{ \begin{array}{l} \text{FORWARD} \\ \text{REVERSE} \end{array} \right\} \rrbracket \right)$$

Parameters

buffer

An expression that evaluates to a buffer. SCAN does not match any of the characters that appear in the buffer.

range

An expression that evaluates to a range. SCAN does not match any of the characters that appear in the range.

string

An expression that evaluates to a string. SCAN does not match any of the characters that appear in the string.

FORWARD

A keyword that directs DECTPU to match characters in the forward direction. This is the default.

REVERSE

A keyword that directs DECTPU to match characters as follows: first, match characters in the forward direction until DECTPU finds a character that is a member of the set of characters. Next, return to the first character matched and start matching characters in the reverse direction until DECTPU finds a character that is in the set.

You can specify REVERSE only if you are using SCAN in the first element of a pattern being used in a reverse search. In all other contexts, specifying REVERSE has no effect.

The behavior enabled by REVERSE allows an alternate form of reverse search. By default, a reverse search stops as soon as a successful match occurs, even if there might have been a longer successful match in the reverse direction. By specifying REVERSE, you direct DECTPU not to stop matching in either direction until it has matched as many characters as possible.

Return Value

A pattern matching only characters that do not appear in the buffer, range, or string used as the parameter.

Description

The SCAN procedure returns a pattern that matches only characters that do not appear in the string, buffer, or range used as its parameter. SCAN matches as many characters as possible, stopping only if it finds a character that is present in its parameter or if it reaches the end of a line. If SCAN is part of a larger pattern, SCAN does not match a character if doing so prevents the rest of the pattern from matching.

SCAN does not cross line boundaries. To match a string of characters that may cross one or more line boundaries, use SCANL.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	SCAN must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	SCAN requires at least one argument.
TPU\$_TOOMANY	ERROR	SCAN accepts no more than one argument.
TPU\$_ARGMISMATCH	ERROR	SCAN was given an argument of the wrong type.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of SCAN.

Examples

1. The following example stores a pattern that matches the longest string of characters that does not contain a, b, or c in *pat1*:

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

2. The following example identifies parenthesized text within a single line. It moves the editing point to the beginning of the parenthesized text, if it is found.

```
PROCEDURE user_find_parens

    paren_text      := ANY("(" + SCAN (')");
    found_range     := SEARCH (paren_text, FORWARD, NO_EXACT);

    IF found_range = 0 ! No parentheses.
    THEN
        MESSAGE ("No parentheses found.");
    ELSE
        POSITION (found_range);
    ENDIF;
ENDPROCEDURE;
```

3. The following example goes through the current file, deleting all characters that are not numbers, letters, or spaces:

```
PROCEDURE user_remove_odd_characters
    LOCAL pat1,
           odd_text;

    pat1 := SCAN ("abcdefghijklmnopqrstuvwxy 0123456789");
    POSITION (BEGINNING_OF (CURRENT_BUFFER));
    LOOP
        odd_text := SEARCH_QUIETLY (pat1, FORWARD);
        EXITIF odd_text = 0;
        ERASE (odd_text);
        POSITION (odd_text);
    ENDLOOP;
    POSITION (END_OF (CURRENT_BUFFER));
ENDPROCEDURE;
```

SCANL

Syntax

$$\text{pattern} := \text{SCANL} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \llbracket, \left\{ \begin{array}{l} \text{FORWARD} \\ \text{REVERSE} \end{array} \right\} \rrbracket \right)$$

Parameters

buffer

An expression that evaluates to a buffer. SCANL does not match any of the characters that appear in the buffer.

range

An expression that evaluates to a range. SCANL does not match any of the characters that appear in the range.

string

An expression that evaluates to a string. SCANL does not match any of the characters that appear in the string.

FORWARD

A keyword that directs DECTPU to match characters in the forward direction. This is the default.

REVERSE

A keyword that directs DECTPU to match characters as follows: first, match characters in the forward direction until DECTPU finds a character that is a member of the set of characters. Next, return to the first character matched and start matching characters in the reverse direction until DECTPU finds a character that is in the set.

You can specify REVERSE only if you are using SCANL in the first element of a pattern being used in a reverse search. In all other contexts, specifying REVERSE has no effect.

The behavior enabled by REVERSE allows an alternate form of reverse search. By default, a reverse search stops as soon as a successful match occurs, even if there might have been a longer successful match in the reverse direction. By specifying REVERSE, you direct DECTPU not to stop matching in either direction until it has matched as many characters as possible.

Return Value

A pattern that may contain line breaks and that matches only characters that do not appear in the buffer, range, or string used as the parameter.

Description

The SCANL procedure returns a pattern that matches a string of characters, including line breaks, none of which appear in the buffer, range, or string used as its parameter. The returned pattern contains as many characters and line breaks as possible.

SCANL is similar to SCAN in that it matches one or more characters that do not appear in the string, buffer, or range used as its parameter. Unlike SCAN, however, SCANL does not stop matching when it reaches the end of a line. Rather, it successfully matches the line end and continues trying to match characters on the next line.

If SCANL is part of a larger pattern, it does not match a character or line boundary if doing so prevents the rest of the pattern from matching.

SCANL must match at least one character.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	SCANL must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	SCANL requires at least one argument.
TPU\$_TOOMANY	ERROR	SCANL requires no more than one argument.

TPU\$_ARGMISMATCH	ERROR	Argument to SCANL has the wrong type.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of SCANL.

Examples

1. The following example creates a pattern that matches a sentence. It assumes that a sentence ends in one of the following characters: a period (.), an exclamation point (!), or a question mark (?). The matched text does not include the punctuation mark ending the sentence.

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

2. The following example goes through the current buffer, erasing anything that is not a number. The only line breaks it leaves in the file are those between a line ending with a number and one beginning with a number.

```
PROCEDURE user_remove_non_numbers
  LOCAL pat1,
         non_number_region;

  pat1 := SCANL ("0123456789");

  POSITION (BEGINNING_OF (CURRENT_BUFFER));

  LOOP
    non_number_region := SEARCH_QUIETLY (pat1, FORWARD);
    EXITIF non_number_region = 0;
    ERASE (non_number_region);
    POSITION (non_number_region);
  ENDLLOOP;

  POSITION (BEGINNING_OF (CURRENT_BUFFER));
ENDPROCEDURE;
```

SCROLL

Syntax

```
[[integer2 := ]SCROLL (window [,integer1])
```

Parameters

window

The window associated with the buffer whose text you want to scroll.

integer1

The signed integer value that indicates how many lines you want the text to scroll. If you supply a negative value for the second parameter, the lines of text scroll off the top of the screen, leaving the cursor closer to the beginning of the buffer. If you supply a positive value for the second parameter, the lines of text scroll off the bottom of the screen, leaving the cursor closer to the end of the buffer. If you specify 0 as the integer value, no scrolling occurs.

This parameter is optional. If you omit the second parameter, the text scrolls continuously until it reaches a buffer boundary or until you press a key. If the current direction of the buffer is forward, the text scrolls to the end of the buffer. If the current direction of the buffer is reverse, the text scrolls to the beginning of the buffer. If you press a key that has commands bound to it, the scrolling stops and DECTPU executes the commands bound to the key.

Return Value

An integer that indicates the number and direction of lines actually scrolled as a result of using `SCROLL`.

Description

The `SCROLL` procedure moves the lines of text in the buffer up or down on the screen by the number of lines you specify. You can scroll text only in a visible window. If the window is not currently visible on the screen, DECTPU issues an error message.

During scrolling, the cursor does not move but stays positioned at the same relative screen location. The current editing point is different from the editing point that was current before you issued the `SCROLL` built-in procedure.

`SCROLL` optionally returns an integer that indicates the number and direction of lines actually scrolled. If you supply a negative value for the second parameter, the lines of text scroll off the bottom of the screen, leaving the cursor closer to the beginning of the buffer. If you supply a positive value for the second parameter, the lines of text scroll off the top of the screen, leaving the cursor closer to the end of the buffer. The value of *integer2* may differ from what was specified in *integer1*.

`SCROLL` causes the screen to scroll immediately. Unlike screen updates, `SCROLL` does not wait to take effect until after the completion of a procedure.

If the buffer has been modified or the window display has altered since the last update, the window is updated before the scrolling operation begins.

`SCROLL` does not work if:

- you have turned off the screen update flag with `SET (SCREEN_UPDATE, OFF)`;
- you used the `/NODISPLAY` qualifier when invoking DECTPU on an unsupported device;
- the window that you specify is not visible on the screen.

When the scrolling is complete, the editing point (record and offset) is set to match the cursor position (screen line and column position).

After the scrolling stops, the cursor may be located to the right of the last character in the new current record, to the left of the left margin, or in the middle of a tab. In this instance, any DECTPU built-in procedure that requires a record offset (for example, `CURRENT_OFFSET`, `MOVE_HORIZONTAL`, `MOVE_VERTICAL`, `MARK`, and so on) causes the record to be padded with blanks to the cursor location.

If the screen you are using does not have hardware scrolling regions, the window being scrolled is repainted for each scroll that would have occurred. For instance, the statement `SCROLL (my_window,3)` repaints the window three times.

If you use `SCROLL` while positioned after the end of the buffer, `SCROLL` completes successfully and returns 0 as the amount scrolled.

Signaled Errors

TPU\$_CONTROLC	ERROR	You pressed Ctrl/C to stop scrolling.
TPU\$_WINDNOTMAPPED	WARNING	You are trying to scroll an unmapped window.
TPU\$_TOOFEW	ERROR	SCROLL requires at least one parameter.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Examples

1. The following example causes the text of the buffer that is mapped to the main window to scroll forward 10 lines:

```
SCROLL (main_window,+10)
```

2. The following example scrolls the buffer MAIN until you press a key. The procedure returns the number of lines scrolled.

```
PROCEDURE user_scroll_buffer

    LOCAL scrolled_lines;

    MESSAGE ("Press any key to stop scrolling...");
    scrolled_lines := SCROLL (main_window);
    dummy_key := READ_KEY;
    RETURN scrolled_lines;
ENDPROCEDURE;
```

SEARCH

Syntax

$$[[\text{range2} :=]\text{SEARCH} (\left\{ \begin{array}{l} \text{ANCHOR} \\ \text{BUFFER_BEGIN} \\ \text{BUFFER_END} \\ \text{LINE_BEGIN} \\ \text{LINE_END} \\ \text{PAGE_BREAK} \\ \text{pattern} \\ \text{REMAIN} \\ \text{string} \\ \text{UNAHCNOR} \end{array} \right\} , \left\{ \begin{array}{l} \text{FORWARD} \\ \text{REVERSE} \end{array} \right\} \llbracket , \left\{ \begin{array}{l} \text{EXACT} \\ \text{NO_EXACT} \end{array} \right\} \llbracket , \left\{ \begin{array}{l} \text{buffer} \\ \text{range1} \end{array} \right\} \llbracket \rrbracket)$$

Parameters

ANCHOR

A keyword that directs **SEARCH** to start a search at the current character position. Use this keyword as part of a complex pattern.

BUFFER_BEGIN

A keyword used to match the beginning of a buffer.

BUFFER_END

A keyword used to match the end of a buffer.

LINE_BEGIN

A keyword used to match the beginning of a line.

LINE_END

A keyword used to match the end of a line.

PAGE_BREAK

A keyword used to match a form-feed character.

pattern

The pattern that you want to match.

REMAIN

A keyword that specifies a match starting at the current character and continuing to the end of the current line.

string

The string that you want to match.

UNANCHOR

A keyword that specifies that the next pattern element can match anywhere after the previous pattern element. Use this keyword as part of a complex pattern. For more information on these keywords, refer to the individual descriptions of them in this chapter.

FORWARD

Indicates a search in the forward direction.

REVERSE

Indicates a search in the reverse direction.

EXACT

Indicates that the characters **SEARCH** is trying to match must be the same case and have the same diacritical markings as those in the string or pattern used as the first parameter to **SEARCH**.

NO_EXACT

Indicates that the characters `SEARCH` is trying to match need not be the same case nor have the same diacritical markings as those in the string or pattern used as the first parameter to `SEARCH`. `NO_EXACT` is the default value for the optional third parameter.

integer

Specifies how `SEARCH` should handle case and diacritical information if you want to match one attribute and ignore the other. VSI recommends that you use the defined constants available for specifying this integer. The defined constants are as follows:

- `TPU$K_SEARCH_CASE`—Equivalent to the integer 1. This specifies that the search should match the case of the first parameter but be insensitive to the diacritical markings of the first parameter.
- `TPU$K_SEARCH_DIACRITICAL`—Equivalent to the integer 2. This specifies that the search should match the diacritical markings of the first parameter but be insensitive to the case of the first parameter.

buffer

The buffer in which to search. `SEARCH` starts at the beginning of the buffer when doing a forward search and at the end of the buffer when doing a reverse search.

range1

The range in which to search. `SEARCH` starts at the beginning of the range when doing a forward search and at the end of the range when doing a reverse search.

To search a range for all occurrences of a pattern, you must define the range dynamically after each successful match; otherwise, `SEARCH` positions to the beginning of the range and finds the same occurrence over and over. See the example section for a procedure that searches for all occurrences of a pattern in a range.

Return Value

The range that contains characters that match the pattern or string specified as a parameter.

Description

The `SEARCH` procedure looks for a particular arrangement of characters in a buffer or range and returns a range that contains those characters. `SEARCH` looks for text that matches the string, pattern, or keyword specified as its first parameter. If it finds such text, it creates a range that contains this text and returns it. If `SEARCH` does not find a match, `SEARCH` returns 0 and signals the error `TPU $ _STRNOTFOUND`. To perform a search that does not signal an error when there is no match, use the `SEARCH_QUIETLY` built-in procedure.

The starting position for the search depends on the optional fourth parameter and the search direction. If you do not specify the fourth parameter, the search starts at the editing point.

If you specify a range for the fourth parameter, the search starts at the beginning of the range for a forward search or the end of the range for a reverse search. When searching a range, `SEARCH` matches only text inside the range. It does not look at text outside the range.

If you specify a buffer for the fourth parameter, the search starts at the beginning of the buffer for a forward search or the end of the buffer for a reverse search.

To determine whether the searched text contains a match, `SEARCH` examines the character at the starting position and attempts to match the character against the pattern, text, or keyword specified. By default, the search is unanchored. This allows `SEARCH` to move one character in the direction of the search if the character at the start position does not match. `SEARCH` continues in this manner until it finds a match or reaches the bounds of the buffer or range.

To prevent `SEARCH` from moving the starting position in the direction of the search, use the `ANCHOR` keyword when you define the pattern to be matched.

`SEARCH` does not change the current buffer or the editing point in that buffer. For more information about searching, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

<code>TPU\$_STRNOTFOUND</code>	WARNING	Search for a string or pattern was unsuccessful.
<code>TPU\$_TOOFEW</code>	ERROR	<code>SEARCH</code> requires at least two arguments.
<code>TPU\$_TOOMANY</code>	ERROR	<code>SEARCH</code> accepts no more than four arguments.
<code>TPU\$_ARGMISMATCH</code>	ERROR	One of the parameters to <code>SEARCH</code> is of the wrong type.
<code>TPU\$_INVPARAM</code>	ERROR	One of the parameters to <code>SEARCH</code> is of the wrong type.
<code>TPU\$_BADKEY</code>	WARNING	You specified an incorrect keyword to <code>SEARCH</code> .
<code>TPU\$_MINVALUE</code>	WARNING	The integer parameter to <code>SEARCH</code> must be greater than or equal to <code>-1</code> .
<code>TPU\$_MAXVALUE</code>	WARNING	The integer parameter to <code>SEARCH</code> must be less than or equal to <code>3</code> .
<code>TPU\$_NOCURRENTBUF</code>	ERROR	If you do not specify a buffer or range to search, you must position to a buffer before searching.
<code>TPU\$_CONTROLC</code>	ERROR	You pressed <code>Ctrl/C</code> while <code>SEARCH</code> was executing.
<code>TPU\$_ILLPATAS</code>	ERROR	The pattern to <code>SEARCH</code> contained a partial pattern assignment to a variable not defined in the current context.

Examples

1. In the following example, if you search a buffer in which the string "Reflections of Monet" appears, this assignment statement stores the characters "Reflections of Monet" in the range `user_range`. The search finds a successful match even though the characters in the word "Monet" do not match in case because you specified `NO_EXACT`.

```
user_range := SEARCH ("Reflections of MONET", FORWARD, NO_EXACT)
```

2. The following example searches the range *the_range* for all occurrences of the pattern "blue skies". If SEARCH finds the pattern, the procedure redefines *the_range* to begin after the end of the pattern just found. If the procedure did not redefine the range, SEARCH would keep finding the first occurrence over and over. The procedure reports the number of occurrences of the pattern.

```
PROCEDURE user_search_range

  LOCAL found_count;

  ON_ERROR
    [TPU$_STRNOTFOUND, TPU$_CONTROL C]:
    MESSAGE ( FAO ("Found !SL occurrences.", found_count));
    RETURN;
    [OTHERWISE]:ABORT;
  ENDON_ERROR;

  found_count := 0;
  the_pattern := "blue skies";
  the_range := CREATE_RANGE (BEGINNING_OF (CURRENT_BUFFER),
                           END_OF (CURRENT_BUFFER),
                           NONE);
  found_range := CREATE_RANGE (BEGINNING_OF (CURRENT_BUFFER),
                              BEGINNING_OF (CURRENT_BUFFER),
                              NONE);

  LOOP
    the_range := CREATE_RANGE (END_OF (found_range),
                              END_OF (the_range), NONE);
    found_range := SEARCH (the_pattern, FORWARD, NO_EXACT,
                          the_range);
    found_count := found_count + 1;
  ENDOLOOP;
ENDPROCEDURE;
```

SEARCH_QUIETLY

Syntax

```
[[range2 := ] SEARCH_QUIETLY ( { ANCHOR
                                BUFFER_BEGIN
                                BUFFER_END
                                LINE_BEGIN
                                LINE_END
                                PAGE_BREAK
                                pattern
                                REMAIN
                                string
                                UNANCHOR
                                }
, { FORWARD } || { REVERSE } ||, { EXACT
                                NO_EXACT
                                integer
                                } ||, { buffer
                                range1
                                } || )
```


Parameters

ANCHOR

A keyword that directs SEARCH_QUIETLY to start a search at the current character position. Use this keyword as part of a complex pattern.

BUFFER_BEGIN

A keyword used to match the beginning of a buffer.

BUFFER_END

A keyword used to match the end of a buffer.

LINE_BEGIN

A keyword used to match the beginning of a line.

LINE_END

A keyword used to match the end of a line.

PAGE_BREAK

A keyword used to match a form-feed character.

pattern

The pattern that you want to match.

REMAIN

A keyword that specifies a match starting at the current character and continuing to the end of the current line.

string

The string that you want to match.

UNANCHOR

A keyword that specifies that the next pattern element can match anywhere after the previous pattern element. Use this keyword as part of a complex pattern. For more information on these keywords, refer to the individual descriptions of them in this chapter.

FORWARD

Indicates a search in the forward direction.

REVERSE

Indicates a search in the reverse direction.

EXACT

Indicates that the characters SEARCH_QUIETLY is trying to match must be the same case and have the same diacritical markings as those in the string or pattern used as the first parameter to SEARCH_QUIETLY.

NO_EXACT

Indicates that the characters SEARCH_QUIETLY is trying to match need not be the same case nor have the same diacritical markings as those in the string or pattern used as the first parameter to SEARCH. NO_EXACT is the default value for the optional third parameter.

integer

Specifies how SEARCH_QUIETLY should handle case and diacritical information if you want to match one attribute and ignore the other. VSI recommends that you use the defined constants available for specifying this integer. The defined constants are as follows:

- TPU\$K_SEARCH_CASE—Equivalent to the integer 1. This specifies that the search should match the case of the first parameter but be insensitive to the diacritical markings of the first parameter.
- TPU\$K_SEARCH_DIACRITICAL—Equivalent to the integer 2. This specifies that the search should match the diacritical markings of the first parameter but be insensitive to the case of the first parameter.

buffer

The buffer in which to search. SEARCH_QUIETLY starts at the beginning of the buffer when doing a forward search and at the end of the buffer when doing a reverse search.

range1

The range in which to search. SEARCH_QUIETLY starts at the beginning of the range when doing a forward search and at the end of the range when doing a reverse search.

To search a range for all occurrences of a pattern, you must define the range dynamically after each successful match; otherwise, SEARCH_QUIETLY positions to the beginning of the range and finds the same occurrence over and over. See the example section for a procedure that searches for all occurrences of a pattern in a range.

Return Value

The range that contains characters that match the pattern or string specified as a parameter.

Description

The SEARCH_QUIETLY procedure looks for a particular arrangement of characters in a buffer or range and returns a range that contains those characters. Unlike the SEARCH built-in procedure, SEARCH_QUIETLY does not signal TPU\$_STRNOTFOUND when it fails to find a string.

SEARCH_QUIETLY looks for text that matches the string, pattern, or keyword specified as its first parameter. If it finds such text, it creates a range that contains this text and returns it. If SEARCH_QUIETLY does not find a match, the built-in returns 0 without signaling TPU\$_STRNOTFOUND.

The starting position for the search depends on the optional fourth parameter and the search direction. If you do not specify the fourth parameter, the search starts at the editing point.

If you specify a range for the fourth parameter, the search starts at the beginning of the range for a forward search or the end of the range for a reverse search. When searching a range, `SEARCH_QUIETLY` matches only text inside the range. It does not look at text outside the range.

If you specify a buffer for the fourth parameter, the search starts at the beginning of the buffer for a forward search or the end of the buffer for a reverse search.

To determine whether the searched text contains a match, `SEARCH_QUIETLY` examines the character at the starting position and attempts to match the character against the pattern, text, or keyword specified. By default, the search is unanchored. This allows `SEARCH_QUIETLY` to move one character in the direction of the search if the character at the start position does not match. `SEARCH_QUIETLY` continues in this manner until it finds a match or reaches the bounds of the buffer or range.

To prevent `SEARCH_QUIETLY` from moving the starting position in the direction of the search, use the `ANCHOR` keyword when you define the pattern to be matched.

`SEARCH_QUIETLY` does not change the current buffer or the editing point in that buffer. For more information about searching, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>SEARCH_QUIETLY</code> requires at least two arguments.
<code>TPU\$_TOOMANY</code>	ERROR	<code>SEARCH_QUIETLY</code> accepts no more than four arguments.
<code>TPU\$_ARGMISMATCH</code>	ERROR	One of the parameters to <code>SEARCH_QUIETLY</code> is of the wrong type.
<code>TPU\$_INVPARAM</code>	ERROR	One of the parameters to <code>SEARCH_QUIETLY</code> is of the wrong type.
<code>TPU\$_BADKEY</code>	WARNING	You specified an incorrect keyword to <code>SEARCH_QUIETLY</code> .
<code>TPU\$_MINVALUE</code>	WARNING	The integer parameter to <code>SEARCH_QUIETLY</code> must be greater than or equal to <code>-1</code> .
<code>TPU\$_MAXVALUE</code>	WARNING	The integer parameter to <code>SEARCH_QUIETLY</code> must be less than or equal to <code>3</code> .
<code>TPU\$_NOCURRENTBUF</code>	ERROR	If you do not specify a buffer or range to search, you must position to a buffer before searching.
<code>TPU\$_CONTROLC</code>	ERROR	You pressed <code>Ctrl/C</code> while <code>SEARCH_QUIETLY</code> was executing.

TPU\$_ILLPATAS	ERROR	The pattern to SEARCH_QUIETLY contained a partial pattern assignment to a variable not defined in the current context.
----------------	-------	--

Example

The following example searches the range *the_range* for all occurrences of the pattern "blue skies". If SEARCH_QUIETLY finds the pattern, the procedure redefines *the_range* to begin after the end of the pattern just found. If the procedure did not redefine the range, SEARCH_QUIETLY would keep finding the first occurrence over and over. The procedure reports the number of occurrences of the pattern. A procedure that uses SEARCH_QUIETLY does not trap the TPU\$_STRNOTFOUND error because SEARCH_QUIETLY does not signal this error.

```

PROCEDURE user_search_range
  LOCAL found_count;

  ON_ERROR
    [TPU$_CONTROL]:
      MESSAGE ( FAO ("Found !SL occurrences.", found_count));
      RETURN;
    [OTHERWISE]:
      ABORT;
  ENDON_ERROR;

  found_count := 0;
  the_pattern := "blue skies";
  the_range := CREATE_RANGE (BEGINNING_OF (CURRENT_BUFFER),
                             END_OF (CURRENT_BUFFER), NONE);

  found_range := CREATE_RANGE (BEGINNING_OF (CURRENT_BUFFER),
                               BEGINNING_OF (CURRENT_BUFFER), NONE);

  LOOP
    the_range := CREATE_RANGE (END_OF (found_range),
                              END_OF (the_range), NONE);

    found_range := SEARCH_QUIETLY (the_pattern, FORWARD,
                                   NO_EXACT, the_range);
    found_count := found_count + 1;
  ENDLOOP;
ENDPROCEDURE;

```

SELECT

Syntax

$$\text{marker} := \text{SELECT} \left(\begin{array}{l} \text{BLINK} \\ \text{BOLD} \\ \text{NONE} \\ \text{REVERSE} \\ \text{UNDERLINE} \end{array} \right)$$

BLINK

Specifies that the selected characters are to blink.

BOLD

Specifies that the selected characters are to be bolded.

NONE

Applies no video attributes to selected characters.

REVERSE

Specifies that the selected characters are to be displayed in reverse video.

UNDERLINE

Specifies that the selected characters are to be underlined.

Return Value

A marker for the editing point in the current buffer.

Description

The **SELECT** procedure returns a marker for the editing point in the current buffer. You must specify how the marker is to be displayed on the screen (no special video, reverse video, bolded, blinking, or underlined).

The marker returned by **SELECT** indicates the first character position in a select range. The video attribute that you specify for the marker applies to all the characters in a select range. For information on creating a select range, see **SELECT_RANGE**.

SELECT returns a special marker that establishes the beginning of a select range. The marker is positioned at the character position that is the editing point when the **SELECT** built-in procedure is executed. (The marker is actually positioned between character positions, rather than on a character position.) A select range includes all the characters between the select marker and the current position, but not the character at the current position.

Using **SELECT** may cause DECTPU to insert padding spaces or blank lines in the buffer. **SELECT** causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Only one select marker for a buffer can be active at any one time. If a buffer is associated with more than one visible window, the select range is displayed in only one window (the current or most recent window).

If the buffer in which you are selecting text is associated with the current window, as you move from the select marker to another character position in the same buffer, all the characters over which you move the cursor are included in the select range, and the video attribute that you specify for the select marker is applied to the characters in the range. The video attribute of a selected character is not visible when

you are positioned on the character, but once you move beyond the character, the character is displayed with the attribute you specify.

If two or more windows are mapped to the same buffer and one of the windows is the current window, only the current window displays the select area. If two or more windows are mapped to different buffers, you can have more than one visible select area on the screen at the same time. If none of the windows on the screen is the current window, the visible window that was most recently current displays the select area.

If the current character is deleted, the marker moves to the nearest character position. The nearest character position is determined in the following way:

1. If there is a character position on the same line to the right, the marker moves to this position, even if the position is at the end of the line.
2. If the line on which the marker is located is deleted, the marker moves to the first position on the following line.

If you are positioned at the select marker and you insert text, the select marker moves to the first character of the inserted text. You can move one column past the last character in a line. (With free cursor motion, you can move even further beyond the last character of a line.) However, if you establish a select marker beyond the last character in a line, no video attribute is visible for the marker.

Signaled Errors

TPU\$_ONESELECT	WARNING	SELECT is already active in the current buffer.
TPU\$_TOOFEW	ERROR	SELECT requires one argument.
TPU\$_TOOMANY	ERROR	SELECT accepts only one argument.
TPU\$_NEEDTOASSIGN	ERROR	SELECT must be on the right-hand side of an assignment statement.
TPU\$_NOCURRENTBUF	ERROR	You must position to a buffer before using SELECT.
TPU\$_BADKEY	WARNING	You specified the wrong keyword to SELECT.
TPU\$_INVPARAM	ERROR	SELECT's parameter is not a keyword.

Examples

1. The following example places a marker at the editing point. Because NONE is specified, no video attributes are applied to a select range that has this marker as its beginning.

```
select_mark := SELECT (NONE)
```

2. The following example creates a bold marker that is used as the beginning of a select region. As you move the cursor, the characters that you select are bolded. To turn off the selection of characters, set the variable *user_v_beginning_of_select* to 0.

```
! Bold selected text
```

```
PROCEDURE user_start_select
    user_v_beginning_of_select := SELECT (BOLD);
ENDPROCEDURE;
```

SELECT_RANGE

Syntax

```
range := SELECT_RANGE
```

Parameters

None.

Return Value

A range that contains all the characters between the marker established with SELECT and the editing point.

Description

The SELECT_RANGE procedure returns a range that contains all the characters between the marker established with the SELECT built-in procedure and the editing point. SELECT_RANGE does not include the current character.

If you select text in a forward direction, the select range includes the marked character and all characters up to but not including the current character. If you select text in a reverse direction from the marker, the select range includes the current character and all characters up to but not including the marked character.

You use SELECT_RANGE in conjunction with SELECT to let you mark a section of text for treatment as an entity.

Using SELECT_RANGE may cause DECTPU to insert padding spaces or blank lines in the buffer. SELECT_RANGE causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

TPU\$_NOSELECT	WARNING	There is no active select range in the current buffer.
TPU\$_SELRANGEZERO	WARNING	The select range contains no selected characters.
TPU\$_NEEDTOASSIGN	ERROR	SELECT_RANGE must be on the right-hand side of an assignment statement.

TPU\$_TOOMANY	ERROR	SELECT_RANGE takes no arguments.
TPU\$_NOCURRENTBUF	WARNING	There is no current buffer.

Examples

1. The following example puts the range for the currently selected characters in the variable *select_1*:

```
select_1 := SELECT_RANGE
```

2. The following example shows the use of SELECT_RANGE multiple times in the same procedure:

```
PROCEDURE user_select

! Start a select region

    user_select_position := SELECT (REVERSE);
    MESSAGE ("Selection started.");

! Move 5 lines and create a select region

    MOVE_VERTICAL (5);

    SR1 := SELECT_RANGE;

! Move 5 lines and create another select region

    MOVE_VERTICAL (5);
    SR2 := SELECT_RANGE;

! Stop the selection by setting the select marker to 0.

    user_select_position := 0;

ENDPROCEDURE;
```

SEND

Syntax

$$\text{SEND} \left(\left. \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\}, \text{process} \right)$$

Parameters

buffer

The buffer whose contents you want to send to the subprocess.

range

The range whose contents you want to send to the subprocess.

string

The string that you want to send to the subprocess.

process

The process to which you want to send data.

Description

The SEND procedure passes data to an OpenVMS subprocess. All output from the process is stored in the buffer that was associated with the process when you created it. See the CREATE_PROCESS built-in procedure. Your editing stops until the process responds to what is sent.

If you specify a buffer or a range as the data to pass to a process, the lines of the buffer or range are sent as separate records.

The maximum length of data that you can send in one SEND statement is 1024 characters. If you are constructing a command line for DCL processing that is longer than the maximum limit of a DCL line, the subprocess outputs a message stating that the record you are sending is too large for DCL to process. The process then terminates.

Note

At this time, the DCL line limit is 256 characters. This is less than the character length that the SEND built-in procedure is capable of sending to a process.

Before you send a string to a process that is running DCL, you should verify that the length of the string does not exceed the limit specified in your OpenVMS documentation. If your string is longer than that limit, you must break the string into two or more strings, use the DCL line continuation character, and send each string separately.

Signaled Errors

TPU\$_NOPROCESS	WARNING	Subprocess that you tried to send to has already terminated.
TPU\$_SENDFAIL	WARNING	Unable to send input to a subprocess.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SEND built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SEND built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the SEND built-in.
TPU\$_NOTMODIFIABLE	WARNING	Attempt to change unmodifiable buffer. The buffer to which a subprocess writes output must be modifiable.
TPU\$_DELETEFAIL	WARNING	Unable to terminate the subprocess.
TPU\$_NOSENBUFF	WARNING	Input buffer is the same as the output buffer.

TPU\$_CONTROLC	ERROR	The execution of the command you sent terminated because you pressed Ctrl/C.
----------------	-------	--

Example

The following example sends the DCL DIRECTORY command to the OpenVMS process named *user_process*. The process must already be created with the CREATE_PROCESS built-in procedure so that the output can be stored in the buffer associated with the process.

```
SEND ("directory", user_process)
```

SEND_CLIENT_MESSAGE

Syntax

```
SEND_CLIENT_MESSAGE ( { STUFF_SELECTION }  
                     { KILL_SELECTION } )
```

Parameters

STUFF_SELECTION

A keyword that indicates that the client message to be sent is the STUFF_SELECTION client message.

KILL_SELECTION

A keyword that indicates that the client message to be sent is the KILL_SELECTION client message.

Description

The SEND_CLIENT_MESSAGE procedure sends either of two client messages—STUFF_SELECTION or KILL_SELECTION—to other DECwindows applications. The EVE layered application cannot designate the application that is to receive the client message. DECTPU handles sending the message to the correct DECwindows application.

Signaled Errors

TPU\$_NORETURNVALUE	ERROR	Does not return a value.
TPU\$_TOOFEW	ERROR	SEND_CLIENT_MESSAGE requires one argument.
TPU\$_TOOMANY	ERROR	SEND_CLIENT_MESSAGE accepts only one argument.
TPU\$_BADKEY	WARNING	Keyword must be either KILL_SELECTION or STUFF_SELECTION.
TPU\$_INVPARAM	ERROR	The parameter must be a keyword.

TPU\$_NOGBLSELDATA	WARNING	There is no owner of the PRIMARY global selection to send a client message to.
TPU\$_NOFOCUSOWNER	WARNING	There is no owner of the input focus to send a client message to.

SEND_EOF

Syntax

SEND_EOF (process)

Parameters

process

The process to which the end-of-file message is being sent.

Description

The SEND_EOF procedure uses features of the mailbox driver to send an end-of-file message (IO \$_WRITEOF) to a subprocess. The end-of-file message causes a pending read from a subprocess to be completed with an SS\$_ENDOFFILE status.

Signaled Errors

TPU\$_SENDFAIL	WARNING	Unable to send input to a subprocess.
TPU\$_NOPROCESS	WARNING	No subprocess to send to.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SEND_EOF built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SEND_EOF built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the SEND_EOF built-in.
TPU\$_NOTMODIFIABLE	WARNING	Attempt to change unmodifiable buffer. The buffer to which a subprocess writes output must be modifiable.
TPU\$_DELETEFAIL	WARNING	Unable to terminate the subprocess.

Example

The following example sends an end-of-file to *sub_proc1*:

```
SEND_EOF (sub_proc1)
```

SET

Syntax

SET (keyword, parameter [,...])

Parameters

keyword

The keyword used as the first parameter specifies which feature is being established or changed. Following are the valid keywords for SET:

ACTIVE_AREA
AUTO_REPEAT
BELL
CLIENT_MESSAGE
COLUMN_MOVE_VERTICAL
CROSS_WINDOW_BOUNDS
DEBUG
DEFAULT_DIRECTORY
DEFAULT_FILE
DETACHED_ACTION
DISPLAY_VALUE
DRM_HIERARCHY
ENABLE_RESIZE
EOB_TEXT
ERASE_UNMODIFIABLE
FACILITY_NAME
FIRST_INPUT_ACTION
FORWARD
GLOBAL_SELECT
GLOBAL_SELECT_GRAB
GLOBAL_SELECT_READ
GLOBAL_SELECT_TIME
GLOBAL_SELECT_UNGRAB
HEIGHT
ICON_NAME
ICON_PIXMAP
INFORMATIONAL
INPUT_FOCUS
INPUT_FOCUS_GRAB
INPUT_FOCUS_UNGRAB
INSERT
JOURNALING
KEY_MAP_LIST
KEYSTROKE_RECOVERY
LEFT_MARGIN
LEFT_MARGIN_ACTION
LINE_NUMBER
MAPPED_WHEN_MANAGED
MARGINS
MAX_LINES
MENU_POSITION

MESSAGE_ACTION_LEVEL
MESSAGE_ACTION_TYPE
MESSAGE_FLAGS
MODIFIABLE
MODIFIED
MOUSE
MOVE_VERTICAL_CONTEXT
NO_WRITE
OUTPUT_FILE
OVERSTRIKE
PAD
PAD_OVERSTRUCK_TABS
PERMANENT
POST_KEY_PROCEDURE
PRE_KEY_PROCEDURE
PROMPT_AREA
RECORD_ATTRIBUTE
RECORD_MODE
RESIZE_ACTION
REVERSE
RIGHT_MARGIN
RIGHT_MARGIN_ACTION
SCREEN_LIMITS
SCREEN_UPDATE
SCROLL_BAR
SCROLL_BAR_AUTO_THUMB
SCROLLING
SELF_INSERT
SHIFT_KEY
SPECIAL_ERROR_SYMBOL
STATUS_LINE
SUCCESS
SYSTEM
TAB_STOPS
TEXT
TIMER
TRACEBACK
UID
UNDEFINED_KEY
VIDEO
WIDGET
WIDGET_CALLBACK
WIDGET_CALL_DATA
WIDGET_CONTEXT_HELP
WIDGET_RESOURCE_TYPES
WIDTH

These keywords and the parameters that follow them are described on the following pages. The descriptions of the keywords are organized alphabetically.

parameter [[, ...]]

The number of parameters following the first parameter varies according to the keyword you use. The parameters are listed in the format section of the applicable keyword description.

Description

With the SET procedure, you can establish or change certain features of a DECTPU session. SET requires a keyword as its first parameter. The keyword indicates which feature of the editor is being set. You can set the mode for entering text, the text that is to be displayed on certain parts of the screen, the direction of a buffer, the status of a buffer, and so on.

Both the programmer creating an editing interface and the person using the interface can use SET. The programmer can establish certain default behavior and screen displays for an editing interface. The user can change the default behavior and do some simple customizing of an existing DECTPU interface.

SET (ACTIVE_AREA)

Syntax

SET (ACTIVE_AREA, window, column, row [, width, height])

Parameters

ACTIVE_AREA

A keyword that directs DECTPU to set an attribute of the active area.

window

The window in which you want to define the active region.

column

An integer that specifies the leftmost column of the active region.

row

An integer that specifies the topmost row of the active region. If you use 0, the active row is the status line.

width

An integer that specifies the width in columns of the active region. Defaults to 1.

height

An integer that specifies the height in rows of the active region. Defaults to 1.

Description

The SET (ACTIVE_AREA) procedure designates the specified area as the active area in a DECTPU window. The active area is the region in a window in which DECTPU ignores movements of the mouse pointer for purposes of distinguishing clicks from drags. When you press down a mouse button, DECTPU interprets the event as a click if the upstroke occurs in the active area with the downstroke. If the upstroke occurs outside the active area, DECTPU interprets the event as a drag operation. A DECTPU layered application can have only one active area at a time, even if the application has more than one window visible on the screen. An active area is valid only if you are pressing a mouse button.

The default active area occupies one character cell. By default, the active area is located on the character cell that contains the cursor. For information on mouse button clicks, see the *OSF / Motif Style Guide*.

Signaled Errors

TPU\$_BADVALUE	ERROR	An integer parameter was specified with a value outside the valid range.
TPU\$_EXTRANEOUSARGS	ERROR	One or more extraneous arguments were specified for a DECwindows built-in.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (ACTIVE_AREA) cannot return a value.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (ACTIVE_AREA) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (ACTIVE_AREA) built-in.

Examples

The following example creates a rectangular active area from the upperleft character position to the character in column 10 of row 2:

```
SET (ACTIVE_AREA, CURRENT_WINDOW, 1, 1, 10, 2);
```

SET (AUTO_REPEAT)

Syntax

$$\text{SET (AUTO_REPEAT, } \left. \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

AUTO_REPEAT

A keyword that indicates that SET is to control whether DECTPU repeats keystrokes as long as you hold down a key. By default, AUTO_REPEAT is set ON (1).

ON, 1

Specifies that a key press should continue to generate characters until the key is released.

OFF, 0

Requires a separate keystroke for each character generated.

Description

The SET (AUTO_REPEAT) procedure controls whether DECTPU repeats keystrokes as long as you hold down a key. DECTPU sends an escape sequence to the terminal to set AUTO_REPEAT on or off.

The autorepeat feature affects all keyboard keys on the VT100 series of terminals except the following keys:

- Set-up
- Esc
- No Scroll
- Tab
- Return
- Ctrl and another key

The autorepeat feature affects all keyboard keys on the VT400, VT300, and VT200 series of terminals except the following keys:

- F1, F2, F3, F4, F5
- Return

If you want to slow down the movement of the cursor, you can use SET (AUTO_REPEAT) within a procedure that causes cursor motion. Because of the time the terminal requires to process the escape sequence that DECTPU sends, if you turn AUTO_REPEAT off before moving the cursor and on after moving the cursor, you slow down the cursor movement. You may find it useful to slow the cursor motion at the top or bottom of a window. The second example in the example section shows how to do this.

SET (AUTO_REPEAT) has no effect if you use it in DECwindows DECTPU.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (AUTO_REPEAT) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be either ON or OFF.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.

Examples

1. The following example turns AUTO_REPEAT off:

```
SET (AUTO_REPEAT, OFF)
```


2. The following example shows how to turn `AUTO_REPEAT` off and on to slow the cursor movement:

```
! Two procedures that slow the scrolling action

PROCEDURE user_slow_up_arrow
  SET (AUTO_REPEAT, OFF);
  MOVE_VERTICAL (-1);
  SET (AUTO_REPEAT, ON);
ENDPROCEDURE;

PROCEDURE user_slow_down_arrow
  SET (AUTO_REPEAT, OFF);
  MOVE_VERTICAL (1);
  SET (AUTO_REPEAT, ON);
ENDPROCEDURE;
```

SET (BELL)

Syntax

$$\text{SET (BELL, } \left\{ \begin{array}{l} \text{ALL} \\ \text{BROADCAST} \end{array} \right\}, \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

BELL

A keyword that indicates that SET is to control whether DECTPU rings the terminal bell when a message is written to the message window.

ALL

Indicates that the second parameter (ON or OFF) applies to all messages.

BROADCAST

Indicates that the second parameter applies to broadcast messages only.

ON, 1

Causes the terminal bell to ring when a message is written to the message window.

OFF, 0

Turns off the audible signal of the terminal bell.

Description

The SET (BELL) procedure controls whether DECTPU rings the terminal bell when a message is written to the message window. When the bell is on, the terminal bell rings to signal the fact that a message is being written to the message window. When you use ALL, internal DECTPU messages

as well as broadcast messages cause the terminal bell to ring. To cause DECTPU messages of success and informational severity level to be written to the message buffer, you must have used the SET ({INFORMATIONAL | SUCCESS}, ON) built-in procedure. When you use BROADCAST, only broadcast messages such as mail notifications and REPLY messages cause the bell to ring.

SET (BELL, ALL, {ON | OFF}) affects the setting of SET (BELL, BROADCAST, {ON | OFF}). If you want the behavior of broadcast messages to be different from other messages, use the SET (BELL, BROADCAST, {ON | OFF}) built-in procedure after using SET (BELL, ALL, {ON | OFF}).

DECTPU causes the bell to ring as a signal that a message is being written to the message window, not as an interpretation of a bell character in the message text. Bell characters in the message text are not interpreted; they are displayed. Positioning to the message window and moving the cursor to a bell character in the message text do not cause the terminal bell to ring.

You can also use DCL commands to affect the display of broadcast messages within DECTPU. If you use the SET TERMINAL/NOBROADCAST command at the DCL level, no broadcast messages are sent to your terminal. With the DCL SET BROADCAST command, you can enable or disable certain classifications of broadcast messages.

The bell is off by default.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (BELL) requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Examples

1. The following example causes the terminal bell to ring when a broadcast message is written to the message window:

```
SET (BELL, BROADCAST, ON)
```

2. The following example uses SET (BELL, ALL, ON) to cause the bell to ring for the message that is being sent in the second statement. After the message is written, the bell is turned off. You use SET (BELL, BROADCAST, ON) to cause broadcast messages to ring the terminal bell.

```
PROCEDURE user_ring_bell (msg_string)

    SET (BELL, ALL, ON);          ! Turn bell on
    MESSAGE (msg_string);        ! Write message text to message buffer
    SET (BELL, ALL, OFF);        ! Turn bell off
    SET (BELL, BROADCAST, ON);    ! Turn bell on for broadcast messages
ENDPROCEDURE;
```

SET (CLIENT_MESSAGE)

Syntax

```
SET (CLIENT_MESSAGE, SCREEN, { buffer  
                               learn_sequence  
                               program  
                               range  
                               string  
                               NONE })
```

Parameters

CLIENT_MESSAGE

A keyword that indicates that SET is being used to designate a client message action routine.

SCREEN

A keyword used to preserve compatibility with future versions of DECTPU.

buffer

The buffer that contains the code to be executed when DECTPU receives a client message.

learn_sequence

The learn sequence to be executed when DECTPU receives a client message.

program

The program to be executed when DECTPU receives a client message.

range

The range that contains the code to be executed when DECTPU receives a client message.

string

The string that contains the code to be executed when DECTPU receives a client message.

NONE

A keyword that directs DECTPU to delete the current client message routine. This is the default if you do not specify the optional third parameter.

Description

The SET (CLIENT_MESSAGE) procedure designates the action routine to be executed when DECwindows DECTPU receives a client message from another DECwindows application. A client message is a communication from one DECwindows application to another. The message enables the sending application to generate an event on the queue of the receiving application.

If the optional parameter is not specified, NONE is the default. When NONE is specified or used by default, DECTPU deletes the current client message routine. When no client message routine is defined, your application is not informed when DECTPU receives a client message.

Signaled Errors

TPU\$_COMPILEFAIL	WARNING	Compilation failed.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.
TPU\$_ARGMISMATCH	ERROR	Argument has the wrong type.
TPU\$_REQUIRESDECW	ERROR	SET(CLI...) is valid only in DECwindows DECTPU.

SET (COLUMN_MOVE_VERTICAL)

Syntax

$$\text{SET (CLIENT_MOVE_VERTICAL, } \left. \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

COLUMN_MOVE_VERTICAL

Specifies that you want to use SET to control how the MOVE_VERTICAL built-in procedure moves the cursor.

ON, 1

Directs the MOVE_VERTICAL built-in procedure to place the cursor in the same column on each new line unless doing so would put the cursor in the middle of a tab. If the cursor would be placed in a tab, MOVE_VERTICAL places the cursor at the beginning of the tab.

OFF, 0

Directs the MOVE_VERTICAL built-in procedure to place the cursor at the same offset in each new record to which the cursor moves. This behavior is the default for SET (COLUMN_MOVE_VERTICAL). Because DECTPU counts a tab as one character when determining the offset, the cursor's column location can change dramatically after you use MOVE_VERTICAL.

Description

The SET (COLUMN_MOVE_VERTICAL) procedure controls how the cursor moves when the MOVE_VERTICAL built-in procedure moves the cursor. When SET (COLUMN_MOVE_VERTICAL) is set to ON, you can get a different result from using MOVE_VERTICAL (n) than from using MOVE_VERTICAL (1) n times. When you use MOVE_VERTICAL (3), for example, the built-in tries to keep the cursor in the column the cursor occupied just before execution of MOVE_VERTICAL (3). When you use MOVE_VERTICAL (1) three times, the built-in resets the column where DECTPU

is trying to keep the cursor. Thus, if the first MOVE_VERTICAL (1) moves the cursor left to the beginning of a tab, the second MOVE_VERTICAL (1) does not move the cursor to the right again.

When SET (COLUMN_MOVE_VERTICAL) is set to OFF, MOVE_VERTICAL (n) produces the same results as MOVE_VERTICAL (1) n times.

To determine whether COLUMN_MOVE_VERTICAL is set to ON or OFF, use the following statement:

```
boolean := GET_INFO (SYSTEM, "COLUMN_MOVE_VERTICAL")
```

This GET_INFO call returns 1 if COLUMN_MOVE_VERTICAL is set to ON; 0 if it is set to OFF.

If you have previously written extensions to EVE and want to layer the extensions on EVE, you may have to rewrite some procedures because EVE sets COLUMN_MOVE_VERTICAL to ON.

For instance, if your extension contains the following code and if the first line has a left margin further to the right than the second line, the code may not work as intended:

```
MOVE_HORIZONTAL (-CURRENT_OFFSET); ! Go to beginning of line
MOVE_VERTICAL          (1); ! Move down a line
```

To compensate for the fact that EVE sets COLUMN_MOVE_VERTICAL to ON, you can substitute the following code for the previous code:

```
POSITION (LINE_END);           ! Go to end of existing line
MOVE_HORIZONTAL (1);           ! Advance to start of next line
```

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (COLUMN_MOVE_VERTICAL) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be either ON or OFF.

Examples

In the following example, the symbol > represents a tab character. The underscore shows the cursor location. Suppose you have the following two lines of text in a buffer, with the cursor on the c in the first line:

```
abcdefg
a>.....bcdefg
```

If you use the following code, the cursor ends up pointing to the b on the second line:

```
SET (COLUMN_MOVE_VERTICAL, OFF);
MOVE_VERTICAL (1);
```

After the MOVE_VERTICAL (1) statement, the cursor location is as follows:

```
abcdefg a>.....bcdefg
```

On the other hand, suppose you have the same text, as follows:

```
abcdefg
a>.....bcdefg
```

If you use the following code, the cursor ends up pointing to the beginning of the tab on the second line:

```
SET (COLUMN_MOVE_VERTICAL, ON);
MOVE_VERTICAL (1);
```

After the MOVE_VERTICAL (1) statement, the cursor location is as follows:

```
abcdefg
a≥.....bcdefg
```

SET (CROSS_WINDOW_BOUNDS)

Syntax

```
SET (CROSS_WINDOW_BOUNDS, { ON
                           OFF
                           1
                           0 })
```

Parameters

CROSS_WINDOW_BOUNDS

A keyword that specifies that SET is to control the way the CURSOR_VERTICAL built-in procedure behaves at a window boundary. The default setting for CROSS_WINDOW_BOUNDS is ON (preserving the behavior from previous versions of DECTPU).

ON, 1

Causes the CURSOR_VERTICAL built-in procedure to cross window boundaries and to ignore scrolling regions. However, even when crossing of window bounds is enabled, the CURSOR_VERTICAL built-in procedure still obeys screen boundaries. That is, if CURSOR_VERTICAL brings the cursor to the edge of the screen, DECTPU scrolls text into the window rather than making the cursor invisible.

OFF, 0

Prevents the CURSOR_VERTICAL built-in procedure from crossing window boundaries and causes CURSOR_VERTICAL to obey scrolling regions.

Description

The SET (CROSS_WINDOW_BOUNDS) procedure controls how the cursor behaves when the CURSOR_VERTICAL built-in executes at a window boundary.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (CROSS_WINDOW_BOUNDS) requires two parameters.
--------------	-------	--

TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example prevents subsequent invocations of the `CURSOR_VERTICAL` built-in from crossing window boundaries and causes the screen to scroll if the cursor moves into a scrolling region. This is the EVE default.

```
SET (CROSS_WINDOW_BOUNDS, OFF)
```

SET (DEBUG)

`SET (DEBUG)` — This built-in has five valid syntax permutations. You cannot use any combinations of parameters not shown in this description.

Syntax

$$\text{SET (DEBUG, PROGRAM, } \left. \begin{array}{l} \text{buffer} \\ \text{program} \\ \text{range} \\ \text{string1} \end{array} \right\})$$

Parameters

DEBUG

A keyword that indicates that `SET` is to control various attributes of a debugging program that helps locate DECTPU programming errors.

PROGRAM

A keyword that indicates that DECTPU is to use a user-written debugger.

buffer

An expression evaluating to a buffer that contains a procedure or program. The statement `SET (DEBUG, PROGRAM, buffer)` directs DECTPU to use the user-written debugger contained in the specified buffer during the current debugging session.

program

A variable of type program. The statement `SET (DEBUG, PROGRAM, program)` directs DECTPU to use the user-written debugger contained in the specified program during the current debugging session.

range

An expression evaluating to a range that contains a procedure or program. The statement `SET (DEBUG, PROGRAM, range)` directs DECTPU to use the user-written debugger contained in the specified range during the current debugging session.

string1

A string that contains executable DECTPU statements. The statement SET (DEBUG, PROGRAM, *string1*) directs DECTPU to use the DECTPU statements in the specified string during the current debugging session.

Syntax
$$\text{SET (DEBUG, } \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$
Parameters**DEBUG**

A keyword that indicates that SET is to control various attributes of a debugging program that helps locate DECTPU programming errors.

ON, 1

Enables single-stepping. The statement SET (DEBUG, ON) directs DECTPU to execute just one line of code and then return control to the debugger.

OFF, 0

Disables single-stepping. The statement SET (DEBUG, OFF) disables single-step execution. Because single-stepping is off by default, this format is useful only to turn off single-stepping after single-stepping has been turned on.

Syntax
$$\text{SET (DEBUG, } \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\}, \text{ string2)}$$
Parameters**DEBUG**

A keyword that indicates that SET is to control various attributes of a debugging program that helps locate DECTPU programming errors.

ON, 1

Sets a breakpoint. The statement SET (DEBUG, ON, *string2*) directs DECTPU to set a breakpoint at the procedure named by *string2*.

OFF, 0

Cancels one or more breakpoints. The statement SET (DEBUG, OFF, *string2*) cancels a breakpoint previously set at the procedure named by *string2*.

string2

The name of a procedure. The format SET (DEBUG, ON, *string2*) or SET (DEBUG, OFF, *string2*) sets or cancels a breakpoint at the procedure specified by *string2*.

Syntax

SET (DEBUG, OFF, ALL)

Parameters**DEBUG**

A keyword that indicates that SET is to control various attributes of a debugging program that helps locate DECTPU programming errors.

OFF, 0

Cancels breakpoints. The statement SET (DEBUG, OFF, ALL) cancels all breakpoints set during the debugging session.

ALL

A keyword that indicates that all breakpoints are to be canceled. The statement SET (DEBUG, OFF, ALL) clears all breakpoints.

Syntax

SET (DEBUG, *string3*, *value*)

Parameters**DEBUG**

A keyword that indicates that SET is to control various attributes of a debugging program that helps locate DECTPU programming errors.

string3

The name of a global variable, local variable, or parameter. When you use *string3* to specify a local variable or a parameter, the variable or parameter must be in the procedure you are currently debugging. The statement SET (DEBUG, *string3*, *value*) deposits the specified value in the variable or parameter specified by *string3*.

value

A value of any data type in DECTPU. The statement SET (DEBUG, *string*, *value*) deposits the specified value in the global variable, local variable, or parameter named by the string.

Description

The SET (DEBUG) procedure controls various attributes of a debugging program that helps locate DECTPU programming errors.

You use SET (DEBUG) to write or use user-written debuggers. You cannot freely mix parameters when using SET (DEBUG). The only valid usages are those shown in the format sections of this built-in.

Signaled Errors

TPU\$_NOCURRENTBUF	WARNING	There is no current buffer.
TPU\$_NONAMES	WARNING	No names match the one requested.
TPU\$_BADKEY	ERROR	An unknown keyword was used as an argument.
TPU\$_ARGMISMATCH	ERROR	You specified an unsupported data type.

Examples

1. The following example causes the debugger to be invoked each time the procedure "user_remove" is called:

```
SET (DEBUG, ON, "user_remove")
```

2. The following example causes the user-written program "user_debugger" to be called as the program to help locate programming errors:

```
SET (DEBUG, PROGRAM, "user_debugger")
```

SET (DEFAULT_DIRECTORY)

Syntax

```
[[old_default_string := ]] SET (DEFAULT_DIRECTORY, new_default_string)
```

Parameters

DEFAULT_DIRECTORY

A keyword that indicates that the SET built-in procedure is being used to control which directory is used as the default.

new_default_string

A string that names the directory to which you want the default changed.

Return Value

The string that you optionally specify returns the current default directory.

Description

The SET (DEFAULT_DIRECTORY) procedure determines the directory that will be used as the default for file operations. When you exit from DECTPU, the default directory is not restored to the default that was set when you invoked DECTPU.

When you issue the EVE DCL SHOW DEFAULT command, the default shown is not always the new default directory, even though the setting has actually been changed. To update DCL tracking of the current default directory, use the EVE DCL SET DEFAULT command instead of calling this built-in procedure directly.

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_TOOFEW	ERROR	SET (DEFAULT_DIRECTORY) requires two parameters.
TPU\$_SYSERROR	ERROR	One of the system routines used has failed. The system routine's error message will be in the message buffer.
TPU\$_INVPARAM	ERROR	The second parameter must be a string.
TPU\$_PARSEFAIL	WARNING	Parameter is not a valid RMS file specification.

Example

The following example sets the default directory to [WALSH.PINK] on the device DISK1. The variable *prev_dir* contains the string that names the previous default directory.

```
prev_dir := SET (DEFAULT_DIRECTORY, "DISK1:[WALSH.PINK]");
```

SET (DEFAULT_FILE)

Syntax

```
SET (DEFAULT_FILE, string)
```

Parameters

DEFAULT_FILE

A keyword that indicates that the SET built-in procedure is being used to merge a new X resource file into the display's resource database.

string

A file specification for the X resource file.

Description

The SET (DEFAULT_FILE) procedure sets a new file specification as the X resource file to merge into the display's resource database. The current database, merged during editor initialization or by a previous SET (DEFAULT_FILE), is lost. The new resource file will affect values returned from the GET_DEFAULT built-in procedure.

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified more than two parameters.
---------------	-------	---

TPU\$_TOOFEW	ERROR	SET (DEFAULT_FILE) requires two parameters.
TPU\$_INVPARAM	ERROR	The second parameter must be a string.

SET (DETACHED_ACTION)

Syntax

SET (DETACHED_ACTION, SCREEN

[, {
 buffer
 learn
 program
 range
 string
 }])

Parameters

DETACHED_ACTION

A keyword that indicates that the SET built-in procedure is being used to designate the detached cursor action routine.

SCREEN

A keyword that indicates that the detached action routine is being set for all buffers and windows used during the session.

buffer

The buffer that contains the detached cursor action routine.

learn

The learn sequence that is executed as the detached cursor action routine.

program

The program that contains the detached cursor action routine.

range

The range that contains the detached cursor action routine.

string

The string that contains the detached cursor action routine.

Description

The SET (DETACHED_ACTION) procedure specifies the code to be executed when the DECTPU main input loop detects that the current cursor position is detached (that is, that the cursor position cannot accurately represent the editing point in the current window).

If DECTPU determines that the current editing point is on a record that is not visible in the current window, the screen updater positions the cursor on the next visible record, placing the cursor in the comparable screen column. This condition is known as a *detached cursor*. Use SET (DETACHED_ACTION) to designate code to be executed when the cursor is detached.

There are five reasons for a detached cursor. The following table shows these reasons, along with their constants and values.

Constant	Value	Reason
TPU\$K_OFF_LEFT	1	The editing point is off the left side of the current window.
TPU\$K_OFF_RIGHT	2	The editing point is off the right side of the current window.
TPU\$K_INVISIBLE	4	The editing point is on a record that is invisible in the current window.
TPU\$K_DISJOINT	8	The current buffer is not mapped to the current window.
TPU\$K_UNMAPPED	16	No current window exists.

If you do not specify the optional third parameter, SET (DETACHED_ACTION) deletes the current detached action routine.

To fetch the current detached action routine, use GET_INFO (SCREEN, "detached_action"). To find out which of the five possible detached states the cursor is in, use GET_INFO (SCREEN, "detached_reason").

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_INVPARAM	ERROR	The second parameter must be a keyword.
TPU\$_ARGMISMATCH	ERROR	The third parameter must be a program or a learn key sequence.
TPU\$_BADKEY	WARNING	The second parameter must be SCREEN.
TPU\$_COMPILEFAIL	WARNING	The third parameter did not compile successfully.
TPU\$_COMPILED	SUCCESS	The third parameter successfully compiled.

Examples

- The following example designates the procedure as an application's detached action routine:

```
SET (DETACHED_ACTION, SCREEN, "detached_routine");
```

2. In the following example, the detached action routine shifts the current window to the right if the editing point is to the right of the last displayed column:

```
PROCEDURE detached_routine

LOCAL rightmost_column,
    the_offset;

IF GET_INFO (SCREEN, "detached_reason") < > tpu$k_off_right

    THEN RETURN;

ENDIF;

rightmost_column := GET_INFO (CURRENT_WINDOW, "right", VISIBLE_TEXT);

the_offset := GET_INFO (CURRENT_BUFFER, "offset_column");

IF the_offset > rightmost_column

    THEN SHIFT (CURRENT_WINDOW, the_offset - rightmost_column + 2)

ENDIF;

UPDATE (CURRENT_WINDOW);

ENDPROCEDURE;
```

SET (DISPLAY_VALUE)

Syntax

```
SET (DISPLAY_VALUE,window,display_value_integer)
```

Parameters

DISPLAY_VALUE

A keyword that indicates that the SET built-in procedure is being used to set the display value for a window.

window

The window whose display value you want to set.

display_value_integer

An integer from -127 to +127.

Description

The SET (DISPLAY_VALUE) procedure sets the display value of the specified window. DECTPU uses a window's display value, which is an integer value, to determine if a given record in a buffer should be made visible in the window mapped to the buffer. If the record's display value is greater than or equal to the window's setting, DECTPU makes the record visible in that window; otherwise, DECTPU makes the record invisible.

You use SET (RECORD_ATTRIBUTE S) to set the record's display values.

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_INVPARAM	ERROR	The second parameter must be a window.
TPU\$_BADDISPVAL	WARNING	Display values must be between -127 and +127.

Example

The following example gives the current window a display value of 10. This means that any record whose display value is less than 10 is invisible in the specified window.

```
SET (DISPLAY_VALUE, CURRENT_WINDOW, 10);
```

SET (DRM_HIERARCHY)

Syntax

```
integer := SET (DRM_HIERARCHY, filespec  
              [, filespec... ])
```

Parameter

filespec

A string that specifies the UID file to be used. DECTPU does not apply a default file specification to the UID file specification. You must specify at least one file name.

Return Value

An integer that is the identification number for the Motif Resource Manager hierarchy.

Description

The SET (DRM_HIERARCHY) procedure sets the user interface definition (UID) file or files to be used with DECTPU. However, the preferred built-in for UID files is SET (UID). Using UID files to specify hierarchies makes it easy to translate the product into other languages and to modify an application's interface without recompiling all the code implementing the application. For more information about UID files, see the *VMS DECwindows Guide to Application Programming*.

Signaled Errors

TPU\$_ARGMISMATCH	ERROR	The data type of the indicated parameter is not supported by the SET (DRM_HIERARCHY) built-in.
-------------------	-------	--

TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (DRM_HIERARCHY) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (DRM_HIERARCHY) built-in.
TPU\$_FAILURE_STATUS	ERROR	The Compaq Resource Manager returned an error status.
TPU\$_INVPARAM	ERROR	You specified an invalid parameter.
TPU\$_REQUIRESDECW	ERROR	Requires the DECTPU DECwindows screen updater.

Example

The following example designates the OpenVMS user interface definition (UID) file MYNODE\$DUA0:[SMIT H]EXAMPLE.UID as a file to be used with DECTPU to create widgets needed by the layered application:

```
example_hierarchy := SET (DRM_HIERARCHY, "mynode$dua0:[smith]example.uid");
```

SET (ENABLE_RESIZE)

Syntax

$$\text{SET (ENABLE_RESIZE, } \left. \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

ENABLE_RESIZE

A keyword that directs DECTPU to enable or disable screen resizing.

ON, 1

Enables screen resizing.

OFF, 0

Disables screen resizing.

Description

The SET (ENABLE_RESIZE) procedure enables or disables resizing of the DECTPU screen. If you specify the ON or 1 keyword, DECTPU gives the DECwindows window manager hints (parameters that the window manager is free to use or ignore) on the allowable maximum and minimum sizes for the DECTPU screen. The hints are set by the SET (SCREEN_LIMITS, array) built-in procedure. If you specify the OFF or 0 keyword, DECTPU uses the screen's current width and length as the maximum and minimum size.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (ENABLE_RESIZE) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (ENABLE_RESIZE) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (ENABLE_RESIZE) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (ENABLE_RESIZE) built-in.

Examples

The following example enables screen resizing. For a sample of this statement used in an initializing procedure, see the example in the description of the SET (SCREEN_LIMITS) built-in procedure.

```
SET (ENABLE_RESIZE, ON);
```

SET (EOB_TEXT)

Syntax

```
SET (EOB_TEXT, buffer, string)
```

Parameters

EOB_TEXT

A keyword that indicates that SET is to determine the text displayed at the end of a buffer. This text is merely a visual marker in a buffer and does not become part of the file that is written when a buffer is saved. The default end-of-buffer text is [EOB].

buffer

The buffer in which the text for the end-of-buffer is being set.

Description

The SET (EOB_TEXT) procedure sets the end-of-buffer text for the specified buffer. You may specify ranges that include the end-of-buffer text, but you cannot set the record_attributes of the end-of-buffer

“record.” Therefore, the end-of-buffer text is always visible, is left-justified on the screen, and cannot be modified using normal editing operations.

Setting a blank EOB_TEXT is the only way to remove the end-of-buffer text. Note, however, that a blank line will still remain.

Signaled Errors

TPU\$_TOOFEW	ERROR	This SET built-in requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_FAILURE	FATAL	DECTPU could not create the record for the EOB text.

Example

The following example causes [END OF MAIN EDITING BUFFER] to be displayed as the end-of-buffer text for the main buffer:

```
SET (EOB_TEXT, main_buffer, "[END OF MAIN EDITING BUFFER]")
```

SET (ERASE_UNMODIFIABLE)

Syntax

$$\left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \end{array} \right\} := \text{SET} (\text{ERASE_UNMODIFIABLE}, \text{buffer} \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

ERASE_UNMODIFIABLE

A keyword that indicates that the SET built-in procedure is being used to control whether unmodifiable records are deleted in response to built-ins that erase lines in a buffer.

buffer

The buffer for which you want to turn on or turn off erasing of unmodifiable records.

ON, 1

Enables erasing of unmodifiable records.

OFF, 0

Disables erasing of unmodifiable records.

Return Value

The keyword ON (1) or OFF (0), indicating the previous setting of ERASE_UNMODIFIABLE.

Description

The SET (ERASE_UNMODIFIABLE) procedure controls whether DECTPU erases unmodifiable records in response to built-ins that delete lines from a buffer.

The default setting lets you use built-ins such as ERASE_LINE to delete unmodifiable records. For example, ERASE_LINE deletes an unmodifiable record only if ERASE_UNMODIFIABLE is turned on. If ERASE_UNMODIFIABLE is turned off when ERASE_LINE or a similar built-in encounters an unmodifiable record, the built-in returns an error and does not delete the record.

Some built-ins delete records as a side effect of their normal action. Table 2.10 shows the built-ins that can delete records as a side effect and shows what these built-ins do instead when the ERASE_UNMODIFIABLE setting is turned off.

The SET (ERASE_UNMODIFIABLE) built-in procedure prevents these built-ins from unintentionally deleting unmodifiable records.

Table 2.10. Selected Built-In Actions with ERASE_UNMODIFIABLE Turned Off

Built-In	Action
APPEND_LINE	Signals a warning if an attempt is made to append to an unmodifiable line.
CHANGE_CASE	Signals a warning if any of the lines in the range or buffer are unmodifiable.
COPY_TEXT	Copies all records, preserving modifiability attribute while in insert mode. Signals a warning if the current editing position is in an unmodifiable line. Signals a warning if in overstrike mode and any of the lines to be overstruck are unmodifiable.
EDIT	Signals a warning if any of the lines in the range or buffer are unmodifiable.
EDIT	Signals a warning if any of the lines in the range or buffer are unmodifiable.
ERASE (buffer)	Signals a warning if any line in the buffer is unmodifiable.
ERASE (range)	Signals a warning if the start or the end of the range is in the middle of an unmodifiable line. Signals a warning if any of the lines in the range are unmodifiable.
ERASE_CHARACTER	Signals a warning if the current character is unmodifiable.
ERASE_LINE	Signals a warning if the current line is unmodifiable.
FILL	Signals a warning if any of the lines in the range or buffer are unmodifiable.

Built-In	Action
MOVE_TEXT	<p>Moves all records, preserving modifiability attribute while in insert mode.</p> <p>Signals a warning if the current editing point is in an unmodifiable line.</p> <p>Signals a warning if in overstrike mode and any of the lines to be overstruck are unmodifiable.</p> <p>If the start or the end of the range is in the middle of an unmodifiable line, the MOVE_TEXT is turned into a COPY_TEXT and a warning is issued.</p> <p>If any of the lines in the buffer or range are unmodifiable, the MOVE_TEXT is turned into a COPY_TEXT and a warning is issued.</p>
SPLIT_LINE	<p>Signals a warning if the current editing position is in the middle of an unmodifiable line.</p> <p>If the current editing position is at the beginning of an unmodifiable line, a new modifiable line is created before it.</p> <p>If the current editing position is at the end of an unmodifiable line, a new modifiable line is created after it.</p> <p>If the current editing position is on an empty unmodifiable line, then a new modifiable line is created after it.</p>
TRANSLATE	<p>Signals a warning if any of the lines in the range or buffer are unmodifiable.</p>

SET (ERASE_UNMODIFIABLE) optionally returns an integer (0 or 1) indicating whether ERASE_UNMODIFIABLE was turned on before the current call was executed. This makes it easier to return to the previous setting later in the program.

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	WARNING	The third parameter must be ON or OFF.

Example

The following example turns off erasing of unmodifiable records in the current buffer and returns the previous setting of ERASE_UNMODIFIABLE:

```
old_setting := SET (ERASE_UNMODIFIABLE, CURRENT_BUFFER, OFF);
```

SET (FACILITY_NAME)

Syntax

SET (FACILITY_NAME, string)

Parameters

FACILITY_NAME

A keyword that indicates that the SET built-in procedure is being used to set the first item (the facility name) in a message generated by DECTPU.

string

The string that you specify as the facility name for messages. The maximum length of this name is 10 characters.

Description

The SET (FACILITY_NAME) procedure sets the facility name for messages. The facility name appears in messages if you have used the SET (MESSAGE_FLAGS) built-in procedure either to explicitly include the facility name in messages or to include the facility name only if enabled by the default message flags for your OpenVMS process.

Signaled Errors

TPU\$_FACTOOLONG	WARNING	Name specified is longer than maximum allowed.
TPU\$_MINVALUE	WARNING	Argument specified is less than the minimum allowed.
TPU\$_ARGMISMATCH	ERROR	The second parameter must be a string.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Example

The following example causes "new_editor" to be used as the facility name in messages:

```
SET (FACILITY_NAME, "new_editor")
```

SET (FIRST_INPUT_ACTION)

Syntax

```
SET (FIRST_INPUT_ACTION, { NONE  
program_source })
```

Parameters

FIRST_INPUT_ACTION

A keyword that specifies that DECTPU is to implement the application's first input action routine.

NONE

A keyword that disables the current action routine.

program_source

Specifies the program or learn sequence that DECTPU executes when it gets the first key or button event.

Description

The SET (FIRST_INPUT_ACTION) procedure specifies the program or learn sequence that DECwindows DECTPU executes when it gets the first key or button event. DECwindows EVE uses FIRST_INPUT_ACTION to set an action routine that removes the copyright notice from the title bar when you first press a key or mouse button.

This built-in is valid only until DECTPU gets the first key or mouse event. After that first event, DECTPU signals TPU\$_BUILTININV, "Built-in is invalid at this time."

Signaled Errors

TPU\$_BUILTININV	ERROR	This indicates that the first key or button event has already occurred.
TPU\$_REQUIRESDECW	ERROR	This built-in is valid on DECwindows only.
TPU\$_TOOFEW	ERROR	This SET built-in requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	The second parameter is not of type string.

Example

The following example causes the procedure REMOVE_COPYRIGHT to be executed when you press the first key or mouse button in the DECwindows application. That procedure typically sets the "title" resource of the application's shell widget to be the name of the application, removing the original copyright notice in the title bar.

```
SET (FIRST_INPUT_ACTION, "remove_copyright")
```

SET (FORWARD)**Syntax**

```
SET (FORWARD, buffer)
```

Parameters**FORWARD**

A keyword that specifies the direction of the buffer. FORWARD means to go toward the end of the buffer. The default direction for a buffer is forward.

buffer

The buffer whose direction you want to set.

Description

The SET (FORWARD) procedure sets the specified buffer's direction to forward. The editor uses this feature to keep track of direction for searching or movement.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (FORWARD) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example causes the direction of the buffer to be toward the end of the buffer:

```
SET (FORWARD, my_buffer)
```

SET (GLOBAL_SELECT)**Syntax**

```
[ integer := ] SET (GLOBAL_SELECT, SCREEN, { PRIMARY  
SECONDARY  
selection_name } ),  
{ GLOBAL_SELECT_GRAB  
GLOBAL_SELECT_UNGRAB }
```

Parameters**GLOBAL_SELECT**

A keyword that indicates that the SET built-in procedure is being used to request a global selection property.

SCREEN

A keyword used to preserve compatibility with future versions of DECTPU.

PRIMARY

A keyword that directs DECTPU to request ownership of the primary global selection.

SECONDARY

A keyword that directs DECTPU to request ownership of the secondary global selection.

selection_name

A string that names the global selection whose ownership DECTPU is to request.

GLOBAL_SELECT_GRAB

A keyword that tells DECTPU to grab the selection. This is the default.

GLOBAL_SELECT_UNGRAB

A keyword that tells DECTPU to relinquish the grab of the selection.

Return Value

Returns 1 if the global selection ownership request was granted; otherwise, returns 0.

Description

The SET (GLOBAL_SELECT) procedure requests ownership of the specified global selection property. SET (GLOBAL_SELECT) returns the integer 1 if the request for ownership of a global selection was granted; otherwise 0.

DECTPU is notified immediately if its request is granted. Therefore, DECTPU does not automatically execute the global selection grab routine when it encounters SET (GLOBAL_SELECT). DECTPU executes the routine only when it automatically grabs the primary selection after it receives input focus.

Applications can voluntarily relinquish global selections through the optional GLOBAL_SELECT_UNGRAB parameter.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (GLOBAL_SELECT) builtin only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (GLOBAL_SELECT) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (GLOBAL_SELECT) built-in.

Examples

1. The following example requests ownership of the primary global selection:

```
SET (GLOBAL_SELECT, SCREEN, PRIMARY);
```

2. The following example shows the use of the GLOBAL_SELECT_UNGRAB parameter. In this statement, global selection is relinquished.

```
SET (GLOBAL_SELECT, SCREEN, PRIMARY, GLOBAL_SELECT_UNGRAB);
```


SET (GLOBAL_SELECT_GRAB)

Syntax

```
SET (GLOBAL_SELECT_GRAB, SCREEN
    [ , {
        buffer
        learn_sequence
        program
        range
        string
        NONE
    } ] )
```

Parameters

GLOBAL_SELECT_GRAB

A keyword that indicates that the SET built-in procedure is being used to set the global select grab routine.

SCREEN

A keyword used to preserve compatibility with future versions of DECTPU.

buffer

The buffer that contains the grab routine.

learn_sequence

The learn sequence that specifies the grab routine.

program

The program that specifies the grab routine.

range

The range that contains the grab routine.

string

The string that contains the grab routine.

NONE

A keyword that directs DECTPU to delete the current global selection grab routine. This is the default if you do not specify the optional third parameter.

Description

The SET (GLOBAL_SELECT_GRAB) procedure specifies the program or learn sequence that DECTPU should execute whenever it automatically grabs ownership of the primary selection. For more information about DECTPU global selection support, see the *Guide to the DEC Text Processing Utility*.

If the optional parameter is not specified, NONE is the default. When NONE is specified or used by default, DECTPU deletes the current global selection grab routine. When no global selection grab routine is defined, your application is not informed when DECTPU grabs the primary global selection.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (GLOBAL_SELECT_GRAB) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (GLOBAL_SELECT_GRAB) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (GLOBAL_SELECT_GRAB) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (GLOBAL_SELECT_GRAB) built-in.

Example

The following example designates the procedure *user_grab_global* as a global selection read routine:

```
SET (GLOBAL_SELECT_GRAB, SCREEN, "user_grab_global");
```

SET (GLOBAL_SELECT_READ)

Syntax

```
SET (GLOBAL_SELECT_READ, { buffer1
                           SCREEN
                           }
     [, { buffer2
          learn_sequence
          program
          range
          string
          NONE
        } ])
```

Parameters

GLOBAL_SELECT_READ

A keyword that indicates that the SET built-in procedure is being used to set the global select read routine.

buffer1

The buffer with which the global selection read routine is to be associated.

SCREEN

A keyword that indicates that the specified routine is to be the application's default global selection read routine.

buffer2

The buffer that contains the global selection read routine.

learn_sequence

The learn sequence that specifies the global selection read routine.

program

The program that specifies the global selection read routine.

range

The range that contains the global selection read routine.

string

The string that contains the global selection read routine.

NONE

A keyword that indicates that the global selection read routine should be deleted. If you do not specify the optional third parameter, NONE is the default.

Description

The SET (GLOBAL_SELECT_READ) procedure specifies the program or learn sequence that DECTPU should execute whenever it receives a selection request event on a global selection it owns. To specify a buffer-specific global selection read routine, use the buffer1 parameter. To specify a global selection read routine for the entire application, use the SCREEN keyword.

When DECTPU receives a request for information about a global selection it owns, it checks to see if the current buffer has a global selection read routine. If so, it executes that routine. If not, it checks to see if there is an application-wide global selection read routine. If so, it executes that routine. If not, it tries to respond to the request itself.

If the optional parameter is not specified, NONE is the default. When NONE is specified or used by default, DECTPU deletes the current global selection read routine.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (GLOBAL_SELECT_READ) cannot return a value.

TPU\$_REQUIRESDECW	ERROR	You can use the SET (GLOBAL_SELECT_READ) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (GLOBAL_SELECT_READ) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (GLOBAL_SELECT_READ) built-in.

Example

The following example designates the procedure *user_read_global* as a global selection read routine:

```
SET (GLOBAL_SELECT_READ, SCREEN, "user_read_global");
```

SET (GLOBAL_SELECT_TIME)

Syntax

```
SET (GLOBAL_SELECT_TIME, SCREEN, { integer }  
                                     { string })
```

Parameters

GLOBAL_SELECT_TIME

A keyword that indicates that the SET built-in procedure is directing DECTPU to set the expiration time for a global selection information request.

SCREEN

A keyword used to maintain compatibility with future versions of DECTPU.

integer

The number of seconds that DECTPU should wait.

string

A string that indicates how long DECTPU should wait. The format of the string is "dd hh:mm:ss.cc" where dd is the number of days (0-24), hh is the number of hours (0-23), mm is the number of minutes (0-59), ss is the number of seconds (0-59), and cc is the number of hundredths of seconds (0-99).

Description

The SET (GLOBAL_SELECT_TIME) procedure specifies how long DECTPU should wait before it assumes that a request for information about a global selection will not be satisfied. The default waiting time is set by DECwindows. The maximum waiting time you can set is 24 days, 20 hours.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
--------------	---------	--

TPU\$_INVTIME	WARNING	You specified an invalid time interval.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	The SET (GLOBAL_SELECT_TIME) built-in cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (GLOBAL_SELECT_TIME) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (GLOBAL_SELECT_TIME) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (GLOBAL_SELECT_TIME) built-in.

Example

The following example sets the waiting time for a global selection response to 3 seconds:

```
SET (GLOBAL_SELECT_TIME, SCREEN, 3);
```

SET (GLOBAL_SELECT_UNGRAB)

Syntax

```
SET (GLOBAL_SELECT_UNGRAB, SCREEN [, ( {
    buffer
    learn_sequence
    program
    range
    string
    NONE
} ] )
```

Parameters

GLOBAL_SELECT_UNGRAB

A keyword that indicates that the SET built-in procedure is being used to set the global select ungrab routine.

SCREEN

A keyword used to preserve compatibility with future versions of DECTPU.

buffer

The buffer that contains the global selection ungrab routine.

learn_sequence

The learn sequence that specifies the global selection ungrab routine.

program

The program that specifies the global selection ungrab routine.

range

The range that contains the global selection ungrab routine.

string

The string that contains the global selection ungrab routine.

NONE

A keyword that directs DECTPU to delete the current global selection ungrab routine. This is the default if you do not specify the optional third parameter.

Description

The SET (GLOBAL_SELECT_UNGRAB) procedure specifies the program or learn sequence that DECTPU should execute whenever it loses ownership of a selection. For more information about DECTPU global selection support, see the *Guide to the DEC Text Processing Utility*.

If the optional parameter is not specified, NONE is the default. When NONE is specified or used by default, DECTPU deletes the current global selection ungrab routine. When no global selection ungrab routine is defined, your application is not informed when DECTPU loses ownership of the primary global selection.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (GLOBAL_SELECT_UNGRAB) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (GLOBAL_SELECT_UNGRAB) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (GLOBAL_SELECT_UNGRAB) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (GLOBAL_SELECT_UNGRAB) built-in.

Example

The following example designates the procedure user_ungrab_global as a global selection ungrab routine:

```
SET (GLOBAL_SELECT_UNGRAB, SCREEN, "user_ungrab_global");
```

SET (HEIGHT)**Syntax**

```
SET (HEIGHT, SCREEN, length)
```

Parameters

HEIGHT

A keyword that indicates that the SET built-in procedure is setting the height of the DECTPU main window.

SCREEN

A keyword that indicates that the screen is being resized.

length

The length (in lines) that you want the screen to have. The value must be an integer between 1 and 255.

Description

The SET (HEIGHT) procedure sets the height of the DECTPU screen without modifying the height or location of any DECTPU window. SET (HEIGHT) does not alter any DECTPU windows. However, the default EVE behavior when the screen is made smaller is to unmap windows from the screen, starting with the bottom-most window and working upward until there is room on the screen for the remaining windows. If the screen is subsequently made larger, the unmapped windows are not remapped by default.

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_TOOFEW	ERROR	SET (HEIGHT) requires three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	WARNING	The second parameter must be SCREEN.
TPU\$_BADLENGTHCHANGE	WARNING	The terminal's characteristics will not allow the height of the screen to change.
TPU\$_BADVALUE	ERROR	The terminal cannot be set to the requested height.

Example

The following example causes the screen to have a height of 20 lines:

```
SET (HEIGHT, SCREEN, 20);
```

SET (ICON_NAME)

Syntax

```
SET (ICON_NAME, string)
```

Parameters

ICON_NAME

A keyword that instructs DECTPU to set the text of an icon.

string

The text you want to appear in the icon.

Description

The SET (ICON_NAME) procedure designates the string used as the layered application's name in the DECwindows icon box.

Signaled Errors

TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (ICON_NAME) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (ICON_NAME) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (ICON_NAME) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (ICON_NAME) built-in.

Example

The following example sets the text naming the layered application to be the string *WordMonger*:

```
SET (ICON_NAME, "WordMonger");
```

SET (ICON_PIXMAP)**Syntax**

SET (ICON_PIXMAP, integer, icon_pixmap [, widget]) or SET (ICON_PIXMAP, bitmap_file_name [, widget])

Parameters**ICON_PIXMAP**

A keyword that indicates that the SET built-in procedure is being used to determine the pixmap that the application uses to create its icon in the DECwindows icon box.

integer

The hierarchy identifier returned by the SET (UID) built-in procedure. This identifier is passed to the Motif Resource Manager, which uses the identifier to find the hierarchy's resource name in the resource database.

icon_pixmap

A case-sensitive string that is the name assigned to the icon in the User Interface Language (UIL) file defining the icon pixmap. The icon must be declared EXPORTED in the UIL file.

Applications must have three icons in their UIL file, as supported by the Motif window manager:

- A small icon that is 32 pixels by 32 pixels
- A medium icon that is 50 pixels by 50 pixels
- A large icon that is 75 pixels by 75 pixels

The icon name that you pass to this built-in must match the root name of the three icon names in the UIL file.

The UIL names start with the root name and end with the dimension `_nXn`. For example, EVE's root name is `EVE_ICON`. The three icon names in EVE's Motif UIL file are therefore `EVE_ICON_32X32`, `EVE_ICON_50X50`, and `EVE_ICON_75X75`.

When you use the `SET (ICON_PIXMAP)` built-in procedure, or change the window manager icon size and restart the Motif Window Manager, DECTPU automatically selects the application's largest icon currently allowed by the Motif Window Manager. Thus, the icon pixmap can correctly fill the image area decoration of the icon. This means that a Motif application cannot specify which icon to display; DECTPU decides for it.

widget

The widget whose icon pixmap is to be set. By default, DECTPU sets the icon pixmap of its top-level widget.

bitmap_file_name

The file specification of a bitmap file. `SET (ICON_PIXMAP)` requires these files to be in the format created by the Xlib routine `WRITE BITMAP FILE`. To create a file with the correct format, you can use the OpenVMS program `SYSSYSTEM:DECW$PAINT.EXE` (the DECpaint application) or the program `DECW$EXAMPLES:BITMAP.EXE`. If you use the paint application, use the `Customize Picture Size` option to set the picture size to nonstandard. Use the `Zoom` option to manipulate this small image. Choose the X11 format when you save the file.

On DECwindows, set the height and width to 75 pixels for large icons, to 50 pixels for medium icons, and to 32 pixels for small icons.

Description

The `SET (ICON_PIXMAP)` procedure determines the pixmap the application uses to create its icon in the DECwindows icon box. To specify an icon pixmap defined in a UIL file, use the first format variant shown in the format section. To specify an icon created in a bitmap file, use the second format variant shown in the format section.

Signaled Errors

<code>TPU\$_FAILURE_STATUS</code>	ERROR	The Resource Manager returned an error status.
<code>TPU\$_INVPARAM</code>	ERROR	One of the parameters was specified with data of the wrong type.
<code>TPU\$_REQUIRESDECW</code>	ERROR	Requires the DECTPU DECwindows screen updater.
<code>TPU\$_TOOFEW</code>	ERROR	You specified too few parameters.
<code>TPU\$_TOOMANY</code>	ERROR	You specified too many parameters.
<code>TPU\$_BADHIERARCHY</code>	ERROR	You specified an invalid hierarchy identifier.

Example

The following example causes the icon pixmap stored in the file `ICON_FLAMINGO.X11` to be displayed in the application's icon:

```
SET (ICON_PIXMAP, "DISK1:[SMITH]ICON_FLAMINGO.X11")
```

SET (INFORMATIONAL)

Syntax

```
SET (INFORMATIONAL, { ON  
OFF  
1  
0 })
```

Parameters

INFORMATIONAL

Informational messages that DECTPU writes.

ON, 1

Causes the informational messages to be displayed.

OFF, 0

Suppresses the display of informational messages.

Description

The `SET (INFORMATIONAL)` procedure sets whether or not you see informational messages. If you specify a section file when invoking DECTPU (either by default or by using the `/NOSECTION` qualifier), DECTPU may not display informational messages. You can cause informational messages to be written by using `SET (INFORMATIONAL, ON)`.

If you use the `/NOSECTION` qualifier when invoking DECTPU, informational messages are written by default.

When you are developing DECTPU programs, the informational messages help you find errors in your program, so it is a good idea to use the `SET (INFORMATIONAL)` built-in procedure to cause the messages to be displayed.

See Appendix B for a list of the DECTPU informational messages.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (INFORMATIONAL) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example causes the display of informational messages to be turned off:

```
SET (INFORMATIONAL, OFF)
```

SET (INPUT_FOCUS)

Syntax

```
SET (INPUT_FOCUS [ [ SCREEN  
                  widget _ _ ] ] )
```

Parameters

INPUT_FOCUS

A keyword that directs DECTPU to request the input focus.

SCREEN

An optional keyword that indicates that the top-level widget associated with DECTPU's screen is to receive the input focus. This keyword is the default.

widget

The widget that is to receive the input focus. If you specify a widget for this parameter, the DECTPU key bindings are not available to process keyboard input into the specified widget.

You can set or get the input-focus state of a widget only if that widget is a shell widget, that is, if it has a resource named XtNinput.

Description

The SET (INPUT_FOCUS) procedure requests ownership of the input focus. Ownership of the input focus determines which application or widget processes user input from the keyboard. It does not guarantee that DECTPU or the widget gets the input focus. If DECTPU or the widget receives the input focus, it gets a focus-in event. When DECTPU gets this event, it calls the input focus grab routine. For more information about the role of events in DECwindows applications, see the *VMS DECwindows Guide to Application Programming*.

When the top-level widget for DECTPU's screen has the input focus, DECTPU processes keystrokes normally. That is, undefined printable keys insert characters in the current buffer, and defined keys execute the code bound to them.

If a child widget in the widget hierarchy has the input focus, keystrokes are processed by that widget. For example, when a text widget in an EVE dialog box has the input focus, keystrokes are processed by the text widget, not by DECTPU. No DECTPU key bindings are in effect.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
--------------	---------	--

TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (INPUT_FOCUS) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (INPUT_FOCUS) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (INPUT_FOCUS) built-in.

Example

The following example is a modified version of the EVE procedure EVE\$ \$WIDGET_REPLACE_EACH_OK. The original version is in SYS\$EXAMPLES:EVE\$MENUS.TPU. For more information about using the files in SYS\$EXAMPLES as examples, see Appendix A.

```
PROCEDURE eve$$widget_replace_each_ok
SET (INPUT_FOCUS);           ! This statement grabs input focus
                             ! so Ctrl/C events will be detected.
eve$$replace_loop (0, eve$x_yes);
ENDPROCEDURE;
```

Procedure EVE\$\$WIDGET_REPLACE_EACH_OK responds when you press the OK button in the REPLACE dialog box that asks if an occurrence of the old string should be replaced with the new string.

SET (INPUT_FOCUS_GRAB)

Syntax

```
SET (INPUT_FOCUS_GRAB , SCREEN [ ( {
    buffer
    learn_sequence
    program
    range
    string
    NONE
} ] )
```

Parameters

INPUT_FOCUS_GRAB

A keyword that indicates that the SET built-in procedure is being used to set the input focus grab routine.

SCREEN

A keyword used for compatibility with future versions of DECTPU.

buffer

The buffer that specifies the actions that DECTPU should take whenever it processes a focus-in event.

learn_sequence

The learn sequence that specifies the actions that DECTPU should take whenever it processes a focus-in event.

program

The program that specifies the actions that DECTPU should take whenever it processes a focus-in event.

range

The range that specifies the actions that DECTPU should take whenever it processes a focus-in event.

string

The string that specifies the actions that DECTPU should take whenever it processes a focus-in event.

NONE

A keyword that directs DECTPU to delete the input focus grab routine. If you specify this keyword or do not specify the parameter at all, the application is not notified when input focus is received.

Description

The SET (INPUT_FOCUS_GRAB) procedure specifies the program or learn sequence that DECTPU should execute whenever it processes a focus-in event. For more information about DECTPU input focus support, see the *Guide to the DEC Text Processing Utility*

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (INPUT_FOCUS_GRAB) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (INPUT_FOCUS_GRAB) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (INPUT_FOCUS_GRAB) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (INPUT_FOCUS_GRAB) built-in.

Example

The following example designates the procedure *user_grab_focus* as an input focus grab routine:

```
SET (INPUT_FOCUS_GRAB, SCREEN, "user_grab_focus");
```

SET (INPUT_FOCUS_UNGRAB)

Syntax

```
SET (INPUT_FOCUS_UNGRAB , SCREEN  $\llbracket$  ( { buffer  
learn_sequence  
program  
range  
string  
NONE }  $\rrbracket$  )
```

Parameters

INPUT_FOCUS_UNGRAB

A keyword that indicates that the SET built-in procedure is being used to set the input focus ungrab routine.

SCREEN

A keyword used for compatibility with future versions of DECTPU.

buffer

The buffer that specifies the actions that DECTPU should take whenever it processes a focus-out event.

learn_sequence

The learn sequence that specifies the actions that DECTPU should take whenever it processes a focus-out event.

program

The program that specifies the actions that DECTPU should take whenever it processes a focus-out event.

range

The range that specifies the actions that DECTPU should take whenever it processes a focus-out event.

string

The string that specifies the actions that DECTPU should take whenever it processes a focus-out event.

NONE

A keyword that directs DECTPU to delete the input focus ungrab routine. If you specify this keyword or do not specify the parameter at all, the application is not notified when input focus is received.

Description

The SET (INPUT_FOCUS_UNGRAB) procedure specifies the program or learn sequence that DECTPU should execute whenever it processes a focus-out event. For more information about DECTPU input focus support, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (INPUT_FOCUS_UNGRAB) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (INPUT_FOCUS_UNGRAB) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (INPUT_FOCUS_UNGRAB) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (INPUT_FOCUS_UNGRAB) built-in.

Example

The following example designates the procedure *user_ungrab_focus* as an input focus grab routine:

```
SET (INPUT_FOCUS_UNGRAB, SCREEN, "user_ungrab_focus");
```

SET (INSERT)

Syntax

```
SET (INSERT, buffer)
```

Parameters

INSERT

A keyword that specifies the mode of entering text. INSERT means that characters are added to the buffer immediately before the editing point. See also the description of the SET (OVERSTRIKE) built-in procedure. The default mode for text entry is insert.

buffer

The buffer whose mode of text entry you want to set.

Description

The SET (INSERT) procedure sets the specified buffer to insert mode.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (INSERT) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Example

The following example causes the characters that you add to the buffer to be added immediately before the editing point:

```
SET (INSERT, my_buffer)
```

SET (JOURNALING)

Syntax

SET (JOURNALING, buffer $\left\{ \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\}$, integer [,file_name_string]) or SET (JOURNALING, integer

Parameters

JOURNALING

A keyword that indicates that the SET built-in procedure is being used to enable or disable buffer-change journaling or set the frequency of journaling.

buffer

The buffer for which you want to enable or disable buffer-change journaling.

ON, 1

A keyword that turns on buffer-change journaling.

OFF, 0

A keyword that turns off buffer-change journaling.

integer

The integer that you specify that determines how frequently records are written to the journal file. The value of this integer must be between 1 and 10.

file_name_string

The string that names the file you want to use as the buffer's journal file. If the file does not exist, DECTPU automatically creates it. You cannot specify this parameter if you have specified the OFF keyword. If you do not specify any file name when you turn journaling on, DECTPU creates a journal file for you and names the file by using the default naming algorithm.

You can set a default journaling directory.

Description

The SET (JOURNALING) procedure turns buffer-change journaling on or off, sets the journaling frequency, and specifies a journal file name. You can turn on journaling only if the buffer is safe for journaling. For a buffer to be safe for journaling, it must either be empty, have never been modified, or be unmodified since the last time it was written to a file. (Whether the buffer has been modified is not the same as whether the buffer is marked as modified. The modified flag can be set or cleared by the application or by you.)

Once a buffer that is being journaled is written to a file, the journal file is closed and deleted and a new journal file is started that references the newly created file. Similarly, reading a file into an empty buffer does not copy the file into the journal—it simply inserts a reference to the file in the journal. This behavior must be taken into account when you perform operations that use temporary files. For example, writing a buffer to a temporary file (which is modified by an external program), then erasing the buffer and rereading a (modified) temporary file will point the journal file at the temporary file. If you then delete the temporary file, the buffer will be unrecoverable.

You can supply a journal file name only if journaling is turned on. If a journal file name is supplied, DECTPU creates a journal file that uses the name you specified. If this parameter is omitted, DECTPU creates a journal file name based on the buffer's name.

If journaling is turned off for the specified buffer, DECTPU closes the journal file but does not delete it.

DECTPU signals a warning or error if any of the following conditions apply:

- Journaling is turned on and one or more of the following is also true:
 - The specified buffer is not safe for journaling.
 - The specified buffer is already being journaled.
 - An RMS error was returned when DECTPU tried to create the journal file.
- Journaling is turned off and a journal file name is specified in the built-in call.

Caution

This setting is determined by how your computer accesses the rest of your network from the VM. In the example described here, the target machines are connected via VPN, which requires the NAT adapter type. If you are using a local network, change the adapter type to Bridged.

The integer parameter specifies the journaling frequency. DECTPU provides a 500-byte buffer for journaling keystrokes. If journaling is enabled, a write to the journal file occurs when the buffer is full. With this built-in procedure, you can determine the frequency with which records are written to the journal file; the lower the integer you specify, the more often journal records are written to disk.

A value of 1 causes a record to be written for approximately every 10 keys pressed. A value of 10 causes a record to be written for approximately every 125 keys. If you are entering only text (rather than procedures that are bound to keys), the number of keystrokes included in a record is greater. For a value of 1, a record is written for approximately every 30 to 35 keystrokes; for a value of 10, a record is written for approximately every 400 keystrokes.

Signaled Errors

TPU\$_MINVALUE	WARNING	Argument is less than minimum allowed.
----------------	---------	--

TPU\$_MAXVALUE	WARNING	Argument is greater than maximum allowed.
TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_INVPARAM	ERROR	You specified a parameter with the wrong data type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.
TPU\$_JRNLOPEN	ERROR	A journal file for this buffer is already open.

Examples

1. The following example turns on buffer-change journaling for the current buffer and directs DECTPU to use the file JOURNAL.JNL in the directory [JONES] as the journal file:

```
SET (JOURNALING, CURRENT_BUFFER, ON, "disk1:[jones]journal.jnl");
```

2. The following example causes a record to be written from the buffer to a journal file at intervals of approximately 10 user keystrokes. If all or most of the keys pressed have procedures bound to them, DECTPU may write out the contents of the buffer after fewer than 10 keystrokes. The journaling interval shown in this statement is the shortest that you can specify.

```
SET (JOURNALING, 1)
```

SET (KEY_MAP_LIST)

Syntax

```
SET (KEY_MAP_LIST, string ( { [ , buffer ] } )
```

Parameters

KEY_MAP_LIST

The key map list that you bind to a buffer or window. The default key map list is TPU \$KEY_MAP_LIST.

string

A quoted string, or a variable name representing a string constant, that specifies the key map list that you bind to a buffer or window.

buffer

The buffer to which you bind the specified key map list. The default is the buffer to which you are positioned.

window

The window with which you want to associate the key map list.

You use the key map list manipulated by SET (KEY_MAP_LIST) only to process mouse events in the specified window. Keystrokes are processed using the key map list associated with the buffer.

Description

The SET (KEY_MAP_LIST) procedure binds a specified key map list to a buffer or window. If the buffer or window is not specified, the default is to bind the key map list to the current buffer. A buffer or window can be associated with only one key map list at a time. A key map list can be associated with many buffers or windows simultaneously.

Signaled Errors

TPU\$_NOKEYMAPLIST	WARNING	Attempt to access an undefined key map list.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (KEY_MAP_LIST) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (KEY_MAP_LIST) built-in.
TPU\$_NOCURRENTBUF	ERROR	You are not positioned in a buffer.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the SET (KEY_MAP_LIST) built-in.

Examples

1. The following example binds the key map list called TPU\$_KEY_MAP_LIST to the current buffer:

```
SET (KEY_MAP_LIST, "tpu$_key_map_list")
```

2. The following example creates a small “scratch pad” window and maps it to a scratch buffer called *junk1.txt*. The procedure defines a key map list that consists of a user-defined key map redefining M1DRAG plus the standard EVE mouse key map. By setting the scratch window’s key map list to be *user_scratch_list*, the procedure invokes *sample_m1_drag* when you drag the mouse in the scratch window. By not declaring the variables as local variables, they are global variables that exists after the procedure executes.

```
PROCEDURE user_scratch_window
scratch_window := CREATE_WINDOW (20, 3, ON);
scratch_buffer := CREATE_BUFFER ("test", "junk.txt");
scratch_map := CREATE_KEY_MAP ("user_scratch_map");
DEFINE_KEY (eve$$kt_return + "sample_M1_DRAG", M1DRAG, "mouse_button_1",
            "user_scratch_map");
scratch_list := CREATE_KEY_MAP_LIST ("user_scratch_list",
            "user_scratch_map",
                                     eve$x_mouse_keys);
SET (KEY_MAP_LIST, "user_scratch_list", scratch_window);
MAP (scratch_window, scratch_buffer);
ENDPROCEDURE;
```

SET (KEYSTROKE_RECOVERY)

Syntax

$$\text{SET (KEYSTROKE_RECOVERY } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

KEYSTROKE_RECOVERY

A keyword that directs DECTPU to enable or disable keystroke recovery, depending on whether the /RECOVER or /NORECOVER qualifier was specified on the command line.

ON, 1

A keyword that enables keystroke recovery. (This has the same effect as specifying the /RECOVER qualifier.)

OFF, 0

A keyword that disables keystroke recovery. (This has the same effect as specifying the /NORECOVER qualifier.)

Description

The SET (KEYSTROKE_RECOVERY) procedure turns keystroke journal recovery on or off. If you specify the /RECOVER qualifier when you invoke DECTPU, DECTPU checks whether the application calls the JOURNAL_OPEN builtin procedure to open a keystroke journal file. If the application does not call the JOURNAL_OPEN built-in, by default DECTPU signals an error when the application starts accepting keyboard input.

In some circumstances, you may want your application to accept the /RECOVER qualifier without error, even though the application does not call the JOURNAL_OPEN built-in. For example, if your application uses only buffer-change journaling, you can use the /RECOVER qualifier when DECTPU is invoked, but the JOURNAL_OPEN built-in is not used.

Use SET (KEYSTROKE_RECOVERY, OFF) to disable the error caused by the lack of a call to JOURNAL_OPEN and concurrently to prevent DECTPU from performing keystroke recovery (even if the /RECOVER qualifier is specified and you use JOURNAL_OPEN). Conversely, use SET (KEYSTROKE_RECOVERY, ON) to direct DECTPU to perform keystroke recovery (even if the /NORECOVER qualifier is specified and you use JOURNAL_OPEN). SET (KEYSTROKE_RECOVERY) signals an error if the application code or the user calls the built-in after DECTPU has started accepting keyboard input.

To determine whether a recovery using a keystroke journal file is currently in progress, use GET_INFO (SYSTEM, "recover"). This GET_INFO call returns FALSE (0) if no keystroke recovery is currently in progress, and returns TRUE (1) if a keystroke recovery is currently in progress. SET (KEYSTROKE_RECOVERY) can determine the value returned by GET_INFO (SYSTEM, "recover") but cannot affect the value returned by GET_INFO (COMMAND_LINE, "recover"). GET_INFO (COMMAND_LINE, "recover") returns a value that indicates whether you specified the /RECOVER qualifier when you invoked DECTPU.

Signaled Errors

TPU\$_JNLNOTOPEN	ERROR	No keystroke journal file is open from which to recover.
TPU\$_RECJNLOPEN	ERROR	A keystroke journal file is already open.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.

TPU\$_INVPARAM	ERROR	You specified a parameter with the wrong data type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example directs DECTPU to do keystroke journal recovery even if the /NORECOVER qualifier was specified on the command line that invoked DECTPU:

```
SET (KEYSTROKE_RECOVERY, ON)
```

SET (LEFT_MARGIN)

Syntax

```
SET (LEFT_MARGIN, buffer, integer)
```

Parameters

LEFT_MARGIN

The left margin of a buffer.

buffer

The buffer in which the left margin is being set.

integer

The column at which the left margin is set.

Description

With the SET (LEFT_MARGIN) procedure, you can change only the left margin of a buffer.

Newly created buffers receive a left margin of 1 (that is, the margin is set in column 1) if a template buffer is not specified in the call to the CREATE_BUFFER built-in procedure. If a template buffer is used, that buffer sets the left margin for all newly created buffers.

Use SET (LEFT_MARGIN) to override the default left margin.

The buffer margin settings are independent of the terminal width or window width settings. The FILL built-in procedure uses these margin settings when it fills the text of a buffer.

When DECTPU creates a new line, the line gets its left margin value from the left margin of the current buffer setting. However, changing the left margin setting for the buffer does not change the left margin for any existing lines.

The value of the left margin must be at least 1 and less than the right margin value.

If you want to use the margin settings of an existing buffer in a user-written procedure, GET_INFO (buffer, "left_margin") and GET_INFO (buffer, "right_margin") return the values of the margin settings in the specified buffer.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (LEFT_MARGIN) built-in requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADMARGINS	WARNING	The left margin setting must be less than the right; both must be greater than zero.

Examples

1. The following example causes the left margin of the buffer represented by the variable *my_buffer* to be changed. The left margin of the buffer is set to 1. The right margin is unchanged.

```
SET (LEFT_MARGIN, my_buffer, 1)
```

2. The following example causes the left margin of the current buffer to be changed to 10; the right margin is unchanged:

```
SET (LEFT_MARGIN, CURRENT_BUFFER, 10)
```

SET (LEFT_MARGIN_ACTION)

Syntax

```
SET (LEFT_MARGIN_ACTION, buffer1 { , buffer2
                                , learn_sequence
                                , program
                                , range
                                , string } )
```

Parameters

LEFT_MARGIN_ACTION

Refers to the action taken when you press a self-inserting key while the cursor is to the left of a line's left margin. A self-inserting key is one that is associated with a printable character.

buffer1

The buffer in which the left margin action routine is being set.

buffer2

A buffer that contains the DECTPU statements to be executed when you press a self-inserting key while the cursor is to the left of a buffer's left margin.

learn_sequence

A learn sequence that is to be replayed when you press a self-inserting key while the cursor is to the left of a buffer's left margin.

program

A program that is to be executed when you press a self-inserting key while the cursor is to the left of a buffer's left margin.

range

A range that contains DECTPU statements that are to be executed when you press a self-inserting key while the cursor is to the left of a buffer's left margin.

string

A string that contains DECTPU statements that are to be executed when you press a self-inserting key while the cursor is to the left of a buffer's left margin.

Description

With the SET (LEFT_MARGIN_ACTION) procedure, you can specify an action to be taken when you attempt to insert text to the left of the left margin of a line. If the third parameter is not specified, the left margin action routine is deleted. If no left margin action routine has been specified, the text is inserted at the current position before any necessary padding spaces, and the left margin of the line becomes the current position.

Newly created buffers do not receive a left margin action routine if a template buffer is not specified on the call to the CREATE_BUFFER built-in procedure. If a template buffer is specified, the left margin action routine of the template buffer is used.

The left margin action routine affects only text entered from the keyboard or a learn sequence. Using the COPY_TEXT or MOVE_TEXT built-in procedure to insert text into a buffer to the left of the left margin does not trigger the left margin action routine.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (LEFT_MARGIN_ACTION) built-in requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_COMPILEFAIL	ERROR	Compilation aborted because of syntax errors.

Examples

1. The following example causes the procedure PUSH_TO_LEFT_MARGIN to be executed when you attempt to type a character to the left of the left margin of the current line. A typical left margin action routine moves the editing point to the left margin and inserts an appropriate number of spaces starting at the left margin.

```
SET (LEFT_MARGIN_ACTION, CURRENT_BUFFER, "push_to_left_margin")
```

2. The following example deletes any left margin action routine that may be defined for the current buffer. When there is no user-defined left margin action routine, if you type a character to the left of the current line's left margin, the text is inserted with spaces padding the text to the old left margin. The leftmost character on the line establishes the line's new left margin.

```
SET (LEFT_MARGIN_ACTION, CURRENT_BUFFER)
```

SET (LINE_NUMBER)

Syntax

$$\text{SET (LINE_NUMBER, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

LINE_NUMBER

Refers to the DECTPU display of the procedure and line number at which an error occurred.

ON, 1

Turns on display of the line number and procedure at which an error occurred.

OFF, 0

Turns off display of the line number and procedure at which an error occurred.

Description

The SET (LINE_NUMBER) procedure is related to the SET (TRACEBACK) procedure. SET (TRACEBACK, ON) turns on both traceback and line numbers. SET (LINE_NUMBER, OFF) turns off both traceback and line numbers. You can also set traceback off and line numbers on. Line numbers are useful for programmers debugging DECTPU programs, but they do not have much meaning to users who do not have the source code available to them.

After a compilation, the line numbers displayed for procedures are relative to the beginning of the procedure. For DECTPU statements compiled outside a procedure, the line numbers displayed are relative to the beginning of the buffer, range, or string being compiled. If there are no procedure declarations before the executable statements, line numbering starts at the beginning of the buffer or range that is being compiled. For strings, the line number is always 1.

You can change line numbers when you use the SAVE built-in to write a sectionfile. If you specify the parameter NO_PROCEDURE_NAMES, the line numbers displayed are relative to the beginning of the buffer or range that was compiled, not relative to the beginning of a procedure.

The default setting for LINE_NUMBER depends on whether a section file was loaded by DECTPU. If a section file was loaded, the default is OFF. If a section file was not loaded, the default is ON.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (LINE_NUMBER) built-in requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.

TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	Only the keywords ON and OFF are allowed.

Example

The following example displays the line number at which the error occurred:

```
PROCEDURE line_number_example
    SET (LINE_NUMBER, ON);
    SET (LINE_NUMBER, BELL);
ENDPROCEDURE;
```

Executing this procedure displays the following in the message buffer:

```
BELL is an invalid keyword
At line 3
```

SET (MAPPED_WHEN_MANAGED)

Syntax

$$\text{SET (MAPPED_WHEN_MANAGED, widget, } \left. \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

MAPPED_WHEN_MANAGED

A keyword that indicates that SET is being used to control whether the specified widget should become visible when it is managed.

widget

The widget whose mapped status you want to set.

ON, 1

A keyword that directs DECTPU to make the specified widget visible when it is managed. This is the default value.

OFF, 0

A keyword that directs DECTPU not to make the specified widget visible when it is managed.

Description

The SET (MAPPED_WHEN_MANAGED) procedure controls whether a widget is mapped to the screen when it is managed. SET (MAPPED_WHEN_MANAGED) does not return the previous state of the modified widget. For more information on managing widgets, see the *VMS DECwindows Guide to Application Programming* and the *VMS DECwindows Toolkit Routines Reference Manual*.

Signaled Errors

TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	Built-in does not return a value.
TPU\$_REQUIRESDECW	ERROR	Requires the DECTPU DECwindows screen updater.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.

Example

The following example directs DECTPU to make the widget contained in *example_widget* invisible when the widget is managed:

```
SET (MAPPED_WHEN_MANAGED, example_widget, OFF);
```

SET (MARGINS)

Syntax

```
SET (MARGINS, buffer, integer1, integer2)
```

Parameters

MARGINS

A keyword that indicates that SET is to determine the left and right margins of a buffer. The default left margin is 1 and the default right margin is 80.

buffer

The buffer in which the margins are being set.

integer1

The column at which the left margin is set.

integer2

The column at which the right margin is set.

Description

With the SET (MARGINS) procedure, you can change the left and right margins of a buffer. The default margins for a buffer are set to 1 for the left margin and 80 for the right margin when you use the CREATE_BUFFER built-in. The FILL built-in procedure uses these margin settings when it fills the text of a buffer.

This built-in procedure controls the buffer margin settings even if the terminal width or window width is set to something else.

The value of the left margin must be at least 1 and less than the right margin value. The value of the right margin must be less than the maximum record size for the buffer. You can use the GET_INFO (buffer, "record_size") built-in procedure to find out the maximum record size of a buffer.

If you want to use the margin settings of an existing buffer in a user-written procedure, the statements GET_INFO (buffer, "left_margin") and GET_INFO (buffer, "right_margin") return the values of the margin settings.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (MARGINS) built-in requires four parameters.
TPU\$_TOOMANY	ERROR	You specified more than four parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADMARGINS	WARNING	Left margin must be smaller than right; both must be greater than zero.

Examples

1. The following example causes the margins of the buffer represented by the variable *my_buffer* to be changed. The left margin of the buffer is set to 1 and the right margin is set to 132.

```
SET (MARGINS, my_buffer, 1, 132)
```

2. The following example causes the margins of the current buffer to be changed to left margin 10 and right margin 70:

```
SET (MARGINS, CURRENT_BUFFER, 10, 70)
```

SET (MAX_LINES)

Syntax

```
SET (MAX_LINES, buffer, integer)
```

Parameters

MAX_LINES

The maximum number of lines a buffer can contain.

buffer

The buffer for which you are setting the maximum number of lines.

integer

The maximum number of lines for the buffer. The valid values are 0, 2, or an integer greater than 2. The maximum value depends on the memory capacity of your system. The default maximum number of lines is 0 (in other words, this feature is turned off).

Description

With the SET (MAX_LINES) procedure, if you exceed the maximum number of lines for a buffer, DECTPU deletes lines from the beginning of the buffer to make room for any lines that exceed the maximum.

SET (MAX_LINES) does not consider the end-of-buffer text to be a record. For example, if you set the maximum number of lines to be 1000, the buffer can contain 1000 records plus the end-of-buffer text.

If you specify a value of 0 for integer, this feature is turned off and DECTPU does not check for the maximum number of lines.

Signaled Errors

TPU\$_MINVALUE	WARNING	Argument less than minimum allowed.
TPU\$_MAXVALUE	WARNING	Argument greater than maximum allowed.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_TOOMANY	ERROR	SET (MAX_LINES) accepts only three parameters.
TPU\$_TOOFEW	ERROR	SET (MAX_LINES) requires three parameters.

Example

The following example causes the maximum number of lines for the message buffer to be 20. Only the most recent lines of messages are kept.

```
SET (MAX_LINES, message_buffer, 20)
```

SET (MENU_POSITION)

Syntax

$$\left\{ \begin{array}{l} \text{array1} \\ \text{NONE} \end{array} \right\} := \text{SET} (\text{MENU_POSITION}, \text{mouse_down_button}, \left\{ \begin{array}{l} \text{array2} \\ \text{NONE} \\ \text{widget} \end{array} \right\})$$

Parameters

MENU_POSITION

A keyword that indicates that the SET built-in procedure is being used to set the menu position of a pop-up widget or widgets.

mouse_down_button

A keyword (M1DOWN, M2DOWN, M3DOWN, M4DOWN, or M5DOWN) that indicates the mouse button associated with the pop-up menus.

array2

An integer-indexed array of pop-up menu widgets to be set for automatic menu positioning.

NONE

A keyword that requests that DECTPU stop automatic positioning of pop-up menu widgets for the specified mouse button.

widget

The pop-up menu widget to be set for automatic menu positioning.

Return Values**array1**

An integer-indexed array of all pop-up menu widgets that were set for automatic positioning for the specified mouse button before this built-in call.

NONE

A keyword that indicates that no pop-up menu widgets were set for the specified mouse button before this built-in call.

Description

The SET (MENU_POSITION) procedure sets menu positioning for one or more pop-up widgets. DECwindows systems do not require pop-up menus to position the last menu item chosen under the mouse pointer. However, this built-in is required to position the pop-up menu below and to the right of the pointer. If you do not use this built-in, DECTPU positions the pop-up widget at the upper left corner of the display.

Signaled Errors

TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_REQUIRESDECW	ERROR	Requires the DECTPU DECwindows screen updater.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_BADKEY	WARNING	You specified an invalid keyword.
TPU\$_NEEDTOASSIGN	ERROR	Built-in must return a value.
TPU\$_EXTRANEOUSARGS	ERROR	The array of widgets parameter had a nonwidget element.
TPU\$_REQARGSMISSING	ERROR	The array of widgets parameter was empty.

SET (MESSAGE_ACTION_LEVEL)**Syntax**

```
SET (MESSAGE_ACTION_LEVEL, { integer }
    { keyword })
```

Parameters

MESSAGE_ACTION_LEVEL

A keyword that indicates that SET is to determine the severity level at which DECTPU sounds the terminal bell or highlights a message.

integer

A value between 0 and 3 that specifies the severity level at which DECTPU is to take the action you designate. The default value is 2. The severity levels and corresponding values, in ascending order of severity, are as follows:

1	Success
3	Informational
0	Warning
2	Error

DECTPU performs the action you specify on all completion messages at the severity level you designate and on all messages of greater severity.

keyword

The keyword associated with a DECTPU completion message. DECTPU uses the keyword to determine the severity level of the associated completion message and performs the action you specify on all completion messages of that severity level or greater.

Description

With the SET (MESSAGE_ACTION_LEVEL) procedure, you can set the action that is taken when DECTPU returns a completion status of the specified severity.

The action you specify using SET (MESSAGE_ACTION_LEVEL) is taken for all completion messages of the specified severity or greater severity.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (MESSAGE_ACTION_LEVEL) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.
TPU\$_ILLSEVERITY	WARNING	Illegal severity specified; DECTPU used the severity "error".

Examples

1. The following example directs DECTPU to display informational, warning, and error messages in reverse video for 1/2 second, then in ordinary video:

```
SET (MESSAGE_ACTION_TYPE, REVERSE);
SET (MESSAGE_ACTION_LEVEL, 3);
```

2. The following example directs DECTPU to ring the terminal's bell whenever a completion status occurs with a severity equal to or greater than the severity of TPU\$_SUCCESS:

```
SET (MESSAGE_ACTION_TYPE, BELL);
SET (MESSAGE_ACTION_LEVEL, TPU$_SUCCESS);
```

SET (MESSAGE_ACTION_TYPE)

Syntax

```
SET (MESSAGE_ACTION_TYPE, { NONE
                           BELL
                           REVERSE })
```

Parameters

MESSAGE_ACTION_TYPE

A keyword that indicates the action to be taken when DECTPU generates a completion status of the severity you specify.

NONE

A keyword that directs DECTPU to take no action. This is the default.

BELL

A keyword that directs DECTPU to ring the terminal's bell when a completion status of the specified severity is returned.

REVERSE

A keyword that directs DECTPU to display the completion status in reverse video for 1/2 second, then display the status in ordinary video.

Description

With the SET (MESSAGE_ACTION_TYPE) procedure, you can set the severity at which the action is taken. The action you specify using SET (MESSAGE_ACTION_TYPE) is taken for all completion messages of the specified severity or greater severity.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (MESSAGE_ACTION_TYPE) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example directs DECTPU to display informational, warning, and error messages in reverse video for 1/2 second, then in ordinary video:

```
SET (MESSAGE_ACTION_TYPE, REVERSE);
SET (MESSAGE_ACTION_LEVEL, 3);
```

SET (MESSAGE_FLAGS)

Syntax

```
SET (MESSAGE_FLAGS, integer)
```

Parameters

MESSAGE_FLAGS

A keyword that indicates that the SET built-in procedure is being used to specify which parts of messages are displayed.

integer

A bit-encoded value for the message code.

Description

The SET (MESSAGE_FLAGS) procedure specifies which items of a message DECTPU displays. Table 2.11 shows the message codes.

Table 2.11. Message Codes

Bit	Value	Meaning
0	1	Include text of message.
	0	Do not include text of message.
1	1	Include message identifier.
	0	Do not include message identifier.
2	1	Include severity level indicator.
	0	Do not include severity level indicator.
3	1	Include facility name.
	0	Do not include facility name.

If you do not set a value for the message flags, the default message flags for your process are used. Setting the message flags to 0 does not turn off the message text; it causes DECTPU to use the default message flags for your process.

In addition to setting the message flags from within DECTPU, you can set them at the DCL level with the SET MESSAGE command. The DCL SET MESSAGE command is the only way you can turn off all message text. See the *VSI OpenVMS DCL Dictionary* for information on the DCL SET MESSAGE command.

Table 2.12 shows the predefined constants available for use with SET (MESSAGE_FLAGS).

Table 2.12. Message Flag Values for SET (MESSAGE_FLAGS)

Bit	Constant	Meaning
0	TPU\$K_MESSAGE_TEXT	Include text of message.
1	TPU\$K_MESSAGE_ID	Include message identifier.
2	TPU\$K_MESSAGE_SEVERITY	Include severity level indicator.
3	TPU\$K_MESSAGE_FACILITY	Include facility name.

Signaled Errors

TPU\$_FLAGTRUNC	WARNING	Message flag values must be less than or equal to 15.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_TOOFEW	ERROR	SET (MESSAGE_FLAGS) requires at least two parameters.
TPU\$_TOOMANY	ERROR	SET (MESSAGE_FLAGS) accepts no more than two parameters.

Examples

1. The following example causes the message identifier to be the only item included in DECTPU messages. The integer 2 sets bit 1.

```
SET (MESSAGE_FLAGS, 2)
```

2. In the following example, the SET (MESSAGE_FLAGS) statement directs DECTPU to include only the message severity level in messages identified by keywords or integers. Because TPU\$_TOOFEW is an error-level message, the MESSAGE statement above causes DECTPU to display "%E" in the message buffer. DECTPU does not display the text associated with the TPU\$_TOOFEW keyword because the statement does not contain an integer or constant directing DECTPU to display the text.

```
SET (MESSAGE_FLAGS, TPU$K_MESSAGE_SEVERITY);
MESSAGE (TPU$_TOOFEW);
```

For more information on using constants to specify message format, see the description of the MESSAGE_TEXT built-in procedure.

SET (MODIFIABLE)

Syntax

```
SET (MODIFIABLE, buffer, { ON  
OFF  
1  
0 })
```

Parameters

MODIFIABLE

The ability to modify a buffer.

buffer

The buffer that will either be unmodifiable or able to be edited.

ON, 1

Makes the buffer modifiable.

OFF, 0

Makes the buffer unmodifiable, allowing only deletion of the buffer and setting of marks and ranges. Any attempt to change the buffer results in a warning message.

Description

With the SET (MODIFIABLE) procedure, you can set whether the buffer is modifiable or not. When a buffer is not modifiable, any attempt to insert, delete, or otherwise modify the contents of the buffer results in a warning message. This affects only the text within the buffer. You can still delete the buffer and you can still create or delete marks and ranges in the text within the buffer.

Newly created buffers are modifiable by default if a template buffer was not used on the call to the CREATE_BUFFER procedure. The modifiability status is taken from the template buffer if one was specified.

You cannot make the messages buffer unmodifiable.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (MODIFIABLE) built-in requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_MSGBUFSET	ERROR	You cannot force the message buffer to be nonmodifiable.
TPU\$_BADKEY	ERROR	Only the ON and OFF keywords are valid.

Example

The following example makes the current buffer unmodifiable. Any attempt to change the buffer fails with a warning message.

```
SET (MODIFIABLE, CURRENT_BUFFER, OFF)
```

SET (MODIFIED)

Syntax

$$\text{SET (MODIFIED, buffer, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

MODIFIED

A keyword that directs DECTPU to turn on or turn off the indicator designating a buffer as modified.

buffer

The buffer whose indicator you want to control.

ON, 1

A keyword that directs DECTPU to mark a buffer as modified.

OFF, 0

A keyword that directs DECTPU to mark a buffer as unmodified.

Description

The SET (MODIFIED) procedure turns on or turns off the flag that indicates that the specified buffer has been modified. Use SET (MODIFIED) with caution. When you turn off the flag indicating that the buffer is modified, you could exit from an application layered on DECTPU without writing out the contents of a modified buffer. Be sure your extension or layered application handles this possibility.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (MODIFIED) cannot return a value.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (MODIFIED) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (MODIFIED) built-in.

Example

The following example marks the current buffer as modified:

```
SET (MODIFIED, CURRENT_BUFFER, ON);
```

SET (MOUSE)

Syntax

$$\llbracket \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \end{array} \right\} := \rrbracket \text{ SET (MOUSE, } \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

MOUSE

Indicates that you are using SET to enable or disable DECTPU's mouse support. The default mouse setting depends on the terminal you are using. If the DECTPU statement GET_INFO (SCREEN, "dec_crt2") returns true on your terminal, mouse support is turned on by default; otherwise, mouse support is turned off by default.

ON, 1

Causes DECTPU to recognize mouse buttons when they are pressed, and lets you bind programs or procedures to mouse buttons. Enables the LOCATE_MOUSE and POSITION (MOUSE) built-in procedures.

OFF, 0

Disables DECTPU mouse support. Pressing a mouse button when the mouse is set to OFF has no effect.

Return Value

Optionally, returns the previous setting.

Description

With the SET (MOUSE) procedure, you can turn mouse support on or off. Because DECTPU mouse support disables the terminal emulator's cut and paste feature in non-DECwindows DECTPU, you must turn off DECTPU mouse support to use the non-DECTPU cut and paste capability while DECTPU is running.

The optional return value specifies whether DECTPU mouse support was enabled or disabled before the current SET (MOUSE) statement was executed. Thus, you can enable or disable mouse support and then reset the support to its previous setting without having to make a separate call.

Signaled Errors

TPU\$_BADKEY	WARNING	The keyword must be either ON or OFF.
TPU\$_MOUSEINV	WARNING	You tried to enable mouse support on an incompatible terminal.
TPU\$_TOOFEW	ERROR	SET (MOUSE) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

SET (MOVE_VERTICAL_CONTEXT)

Syntax

SET (MOVE_VERTICAL_CONTEXT, buffer, integer)

Parameters

MOVE_VERTICAL_CONTEXT

A keyword that indicates that the SET built-in procedure is being used to set the target column where the cursor should remain during MOVE_VERTICAL operations.

buffer

The buffer where cursor motion in the horizontal dimension will be restricted when the cursor moves vertically.

integer

An encoded integer that represents the column where the cursor will be restricted. This value is not a simple column number. It should be specified only with the value returned from the GET_INFO (buffer_variable, "move_vertical_context") built-in procedure.

Description

The SET (MOVE_VERTICAL_CONTEXT) procedure sets the specified buffer's target column for MOVE_VERTICAL operations when the COLUMN_MOVE_VERTICAL setting is on. This attempts to restrict the cursor to a column (in the horizontal dimension) during MOVE_VERTICAL operations.

When the COLUMN_MOVE_VERTICAL setting is on, DECTPU tries to keep the cursor in a single column during MOVE_VERTICAL operations. DECTPU saves the current vertical "context", which is more than just the column number, in order to provide this cursor behavior. The POSITION built-in procedure, however, can interfere with cursor positioning by resetting the column to the one that DECTPU tries to position to during subsequent MOVE_VERTICAL operations.

To avoid this problem, applications should save the current vertical context, use the MOVE_VERTICAL and POSITION built-in procedures, and then restore the vertical context. Applications save the vertical context by getting the value from the GET_INFO (buffer_variable, "move_vertical_context") built-in procedure. Applications restore the saved vertical context by using the SET (MOVE_VERTICAL_CONTEXT) built-in procedure and specifying the saved value for the integer parameter.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (MOVE_VERTICAL_CONTEXT) requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Example

The following example saves the value of the current buffer's vertical context before DECTPU positions the editing point to another buffer. After repositioning to the first buffer, the code sets the buffer's context back to its previous value.

```
saved_context := GET_INFO (CURRENT_BUFFER, "move vertical_context");
saved_location := MARK (FREE_CURSOR);
POSITION (message_buffer);
COPY_TEXT ("Unless you save the context before you use POSITION,");
COPY_TEXT ("you cannot restore the context after POSITION changes it.");
POSITION (saved_location);
SET (MOVE_VERTICAL_CONTEXT, CURRENT_BUFFER, saved_context);
```

SET (NO_WRITE)

Syntax

$$\text{SET (NO_WRITE, buffer } \left[\left[\left\{ \begin{array}{l} , \text{ ON} \\ , \text{ OFF} \\ , 1 \\ , 0 \end{array} \right\} \right] \right])$$

Parameters

NO_WRITE

Specifies that DECTPU should not create an output file from the contents of a buffer after executing a QUIT or EXIT statement even if the buffer contents have been modified. By default, a buffer is written out if it has been modified.

buffer

The buffer whose contents you do not want written out.

ON, 1

Causes the buffer you name not to be written out.

OFF, 0

Lets you change a buffer from the no-write state to the default state. By default, any modified buffers are written out after execution of a QUIT or EXIT statement.

Description

With the SET (NO_WRITE) procedure, you can set whether an output file is written or not written.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (NO_WRITE) requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.

TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Examples

1. The following example causes *my_buffer* not to be saved in a file after execution of a QUIT or EXIT statement:

```
SET (NO_WRITE, my_buffer)
```

2. The following example turns off the no-write state of *my_buffer*. The contents of the buffer are written out after execution of a QUIT or EXIT statement if the buffer has been modified.

```
SET (NO_WRITE, my_buffer, OFF)
```

SET (OUTPUT_FILE)

Syntax

```
SET (OUTPUT_FILE, buffer, string)
```

Parameters

OUTPUT_FILE

A keyword that indicates that SET is to control creation of an output file for the contents of a buffer after execution of a QUIT or EXIT statement.

buffer

The buffer whose contents are written to the specified file.

string

The file specification for the file being written out. The default output file is the input file name and the highest existing version number for that file plus 1.

Description

The SET (OUTPUT_FILE) procedure specifies a file to be written out for the contents of the buffer. DECTPU does not write out the contents of a buffer after execution of a QUIT or EXIT statement if the buffer has not been modified.

If a buffer is set to NO_WRITE, a file is not written out after execution of a QUIT or EXIT statement even though you specified a file specification for the contents of the buffer with SET (OUTPUT_FILE).

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (OUTPUT_FILE) requires three parameters.
--------------	-------	--

TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example causes the output file for *paste_buffer* to be NEWFILE.TXT:

```
SET (OUTPUT_FILE, paste_buffer, "NEWFILE.TXT")
```

SET (OVERSTRIKE)

Syntax

```
SET (OVERSTRIKE, buffer)
```

Parameters

OVERSTRIKE

A keyword that specifies that SET is to control the mode of text entry. OVERSTRIKE means that the characters that you add to the buffer replace the characters in the buffer starting at the editing point and continuing for the length of the text that you enter. The default mode of text entry is INSERT.

See also the description of the SET (INSERT) built-in procedure. For information on how to control overstrike behavior in tabs, see SET (PAD_OVERSTRUCK_TABS).

buffer

The buffer whose mode of text entry you want to set.

Description

The SET (OVERSTRIKE) procedure sets the mode of text entry to OVERSTRIKE.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (OVERSTRIKE) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example sets the mode for text entry in *my_buffer* to overstrike. Characters that you enter replace characters already in the buffer, starting at the editing point and continuing for the length of the text that you enter.


```
SET (OVERSTRIKE, my_buffer)
```

SET (PAD)

Syntax

```
SET (PAD, window, { ON  
OFF  
1  
0 })
```

Parameters

PAD

A keyword that indicates that SET is to control whether screen lines are padded with blanks. This keyword determines whether SET pads out the left and right ends of lines beyond the text on the line. When video attributes are applied to a padded window, the window has an even or “boxed” appearance.

window

The window in which lines are padded.

ON, 1

Causes DECTPU to display blanks after the last character of a record so that the screen line extends to the right side of the window. If there are not enough lines in a buffer to fill an entire window, DECTPU displays blank lines (according to the video setting of the window) from the end-of-buffer line to the end of the window.

OFF, 0

Causes the display of lines on the screen to stop at the last character of a record. When video attributes are applied to the window, the window has a ragged appearance on the sides.

Description

The SET (PAD) procedure pads the number of display lines on the screen. By default, DECTPU ends a line on the screen at the end of a record, without adding padding blanks. The default behavior of not padding the screen gives maximum editing performance. You can change the default with SET (PAD) for special visual effects. The records in the buffer are not padded; only the display lines have the padding.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (PAD) requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be ON or OFF.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.

Example

In the following example, the first statement causes *second_window* to be padded with blanks. The second statement causes *second_window* to be displayed in reverse video. The window has an even right and left margin when displayed.

```
SET (PAD, second_window, ON);  
SET (VIDEO, second_window, REVERSE);
```

SET (PAD_OVERSTRUCK_TABS)

Syntax

$$\text{SET (PAD_OVERSTRUCK_TABS, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

PAD_OVERSTRUCK_TABS

Determines what happens when you overstrike a tab and when you overstrike with a tab.

ON, 1

SET PAD_OVERSTRUCK_TABS ON functions differently depending on whether you are overstriking a tab character with another character or overstriking another character with a tab character.

OFF, 0

SET PAD_OVERSTRUCK_TABS OFF functions differently depending on whether you are overstriking a tab character with another character or overstriking another character with a tab character.

Description

The SET (PAD_OVERSTRUCK_TABS) procedure controls how DECTPU handles tabs in overstrike mode. When earlier versions of DECTPU overstruck a tab, DECTPU inserted spaces, if necessary, to preserve the cursor position within the tab, and then replaced the tab with the character that was being entered. This behavior is preserved when PAD_OVERSTRUCK_TABS is set OFF.

When PAD_OVERSTRUCK_TABS is set ON, DECTPU inserts spaces as necessary to preserve the cursor position within the tab of the first character of the text, and then inserts the text. The tab is replaced only when it occupies a single column.

When SET (PAD_OVERSTRUCK_TABS) is set to ON and you overstrike a tab, DECTPU does the following:

- Inserts as many spaces as necessary to fill the column to where the cursor is
- Inserts with the characters you are overstriking with

- If it fills to the last column, deletes the tab
- Inserts spaces to keep the text after the cursor in the same column

When SET (PAD_OVERSTRUCK_TABS) is set to ON and you overstrike with a tab, DECTPU replaces multiple characters. The characters to the right of the tab do not move their column position. They are overstruck.

When SET (PAD_OVERSTRUCK_TABS) is set to OFF and you overstrike a tab, DECTPU does the following:

- Inserts as many spaces as are necessary to fill the column to where the cursor is.
- Replaces the tab character with the new character. The tab character always is replaced.
- Inserts spaces to keep text after the tab character in the same column.

When SET (PAD_OVERSTRUCK_TABS) is set to OFF and you overstrike with a tab, the characters in the text get replaced by the tab character. This may move characters to the right on the screen.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (PAD_OVERSTRUCK_TABS) built-in requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	WARNING	Only ON and OFF are allowed.

Examples

The following example shows what happens when PAD_OVERSTRUCK_TABS is set to OFF. In this example, a right angle (>) represents the tab, a period (.) represents one column of white space, and an underscore (_) represents the cursor.

Suppose a buffer contains the following text, with the cursor in the middle of white space created by a tab:

```
abc>...def
```

If you insert the asterisk (*) while PAD_OVERSTRUCK_TABS is set to OFF, the white space to the left of the * is preserved. The tab character is removed, and the white space to the right of the * is not preserved. The text to the right of the collapsed white space moves leftward. The result is as follows:

```
abc..*def
```

The cursor is on the d. Given the same initial text, if you type the string "xyzy" while PAD_OVERSTRUCK_TABS is set to OFF, the tab is removed. The text to the right of the tab moves to the left. Your new string, xyzy, is written over the old text. The result is as follows:

```
abc..xyzy
```

When PAD_OVERSTRUCK_TABS is set to ON, the text to the right of the tab does not move to the left when text is inserted within the tab. Instead of removing the tab, DECTPU places the tab to the right

of the inserted text if the inserted text is shorter than the length of the tab. The newly placed tab creates only enough white space to preserve the original column position of the text to the right of the tab.

The following example shows what happens when `PAD_OVERSTRUCK_TABS` is set to `ON`. In this example, a right angle (`>`) represents the tab, a period (`.`) represents one column of white space, and an underscore (`_`) represents the cursor.

Suppose a buffer contains the following text, with the cursor in the middle of white space created by a tab:

```
abc>._.def
```

If you insert an asterisk (`*`) while `PAD_OVERSTRUCK_TABS` is set to `ON`, the white space to the left of the `*` is preserved. The tab is inserted after the `*` character. The result is as follows:

```
abc..*≥.def
```

Given the same initial text, if you insert the string "xyzyz" while `PAD_OVERSTRUCK_TABS` is set to `ON`, to preserve the original position of the text to the right of the tab, DECTPU fills the white space created by the tab with characters from the new string. When the white space is filled, DECTPU writes the new characters over the old characters. Thus, the old text does not move left or right, but rather is overwritten by the new text. The result is as follows:

```
abc..xyzyzf
```

SET (PERMANENT)

Syntax

```
SET (PERMANENT, buffer)
```

Parameters

PERMANENT

Specifies that a buffer cannot be deleted. By default, buffers can be deleted; they are not permanent.

buffer

The buffer that is not to be deleted.

Description

With the `SET (PERMANENT)` procedure, you can make a buffer permanent. Once you use `SET (PERMANENT)`, you cannot reset the buffer so that it can be deleted.

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>SET (PERMANENT)</code> requires two parameters.
<code>TPU\$_TOOMANY</code>	ERROR	You specified more than two parameters.
<code>TPU\$_INVPARAM</code>	ERROR	One or more of the specified parameters have the wrong type.

TPU\$_BADKEY	WARNING	You specified an invalid keyword.
--------------	---------	-----------------------------------

Example

The following example causes *master_buffer* to become a permanent buffer:

```
SET (PERMANENT, master_buffer)
```

SET (POST_KEY_PROCEDURE)

Syntax

```
SET (POST_KEY_PROCEDURE, string1 { , buffer
                                   , learn_sequence
                                   , program
                                   , range
                                   , string2 } )
```

Parameters

POST_KEY_PROCEDURE

The action taken after the code or learn sequence bound to a key is executed.

string1

A quoted string, or a variable name representing a string constant, that specifies the key map list for which this procedure is called.

buffer

The buffer that contains DECTPU statements specifying the action to be taken after the code or learn sequence bound to a key is executed. SET (POST_KEY_PROCEDURE) compiles the statements in the buffer and stores the resulting program in the specified key map list.

learn_sequence

The learn sequence that specifies the action to be taken after the code or learn sequence bound to a key is executed. The contents of a variable of type learn do not require compilation. SET (POST_KEY_PROCEDURE) stores the learn sequence in the specified key map list.

program

The program that specifies the action to be taken after the code or learn sequence bound to a key is executed. The contents of a variable of type program do not require compilation. SET (POST_KEY_PROCEDURE) stores the program in the specified key map list.

range

The range that contains DECTPU statements specifying the action to be taken after the code or learn sequence bound to a key is executed. SET (POST_KEY_PROCEDURE) compiles the statements in the range and stores the resulting program in the specified key map list.

string2

The string that contains DECTPU statements specifying the action to be taken after the code or learn sequence bound to a key is executed. SET (POST_KEY_PROCEDURE) compiles the statements in the string and stores the resulting program in the specified key map list.

Description

The SET (POST_KEY_PROCEDURE) procedure enables an editor to perform some specified action before and after execution of code bound to a key. If you do not specify the third parameter, the postkey procedure for the specified key map list is deleted.

Prekey and postkey procedures interact with learn sequences in the following order:

1. When you press the key or key sequence to which the learn sequence is bound, DECTPU executes the prekey procedure of that key if a prekey procedure has been set.
2. For each key in the learn sequence, DECTPU executes procedures or programs in the following order:
 - a. DECTPU executes the prekey procedure of that key if a prekey procedure has been set.
 - b. DECTPU executes the code bound to the key itself.
 - c. DECTPU executes the postkey procedure of that key if a postkey procedure has been set.
3. When all keys in the learn sequence have been processed, DECTPU executes the postkey procedure, if one has been set, for the key to which the entire learn sequence was bound.

You can use the following calls to the GET_INFO built-in procedure to find the prekey and postkey procedures bound to a key map list:

```
GET_INFO (key_map_list_name, "pre_key_procedure")
GET_INFO (key_map_list_name, "post_key_procedure")
```

By default, newly created key map lists do not have postkey procedures.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (POST_KEY_PROCEDURE) built-in requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_COMPILEFAIL	ERROR	Compilation aborted because of syntax errors.
TPU\$_NOKEYMAPLIST	WARNING	Attempt to access an undefined key map list.

Example

The following example displays a message after the code bound to a key is executed:

```
SET (POST_KEY_PROCEDURE, "tpu$key_map_list",
'MESSAGE ("Key " + GET_INFO (LAST_KEY, "name") + " Executed")');
```

SET (PRE_KEY_PROCEDURE)

Syntax

```
SET (PRE_KEY_PROCEDURE, string1 { , buffer
                                , learn_sequence
                                , program
                                , range
                                , string2 } )
```

Parameters

PRE_KEY_PROCEDURE

The action taken before the code or learn sequence bound to a key is executed.

string1

A quoted string, or a variable name representing a string constant, that specifies the key map list for which this procedure is called.

buffer

The buffer that contains DECTPU statements specifying the action to be taken before the code or learn sequence bound to a key is executed. SET (PRE_KEY_PROCEDURE) compiles the statements in the buffer and stores the resulting program in the specified key map list.

learn_sequence

The learn sequence that specifies the action to be taken before the code or learn sequence bound to a key is executed. The contents of a variable of type learn do not require compilation. SET (PRE_KEY_PROCEDURE) stores the learn sequence in the specified key map list.

program

The program that specifies the action to be taken before the code or learn sequence bound to a key is executed. The contents of a variable of type program do not require compilation. SET (PRE_KEY_PROCEDURE) stores the program in the specified key map list.

range

The range that contains DECTPU statements specifying the action to be taken before the code or learn sequence bound to a key is executed. SET (PRE_KEY_PROCEDURE) compiles the statements in the range and stores the resulting program in the specified key map list.

string2

The string that contains DECTPU statements specifying the action to be taken before the code or learn sequence bound to a key is executed. SET (PRE_KEY_PROCEDURE) compiles the statements in the string and stores the resulting program in the specified key map list.

Description

The SET (PRE_KEY_PROCEDURE) procedure enables an editor to perform some specified action before the execution of code bound to a key. If you do not specify the third parameter, the prekey procedure for the specified key map list is deleted.

Prekey and postkey procedures interact with learn sequences in the following order:

1. When you press the key or key sequence to which the learn sequence is bound, DECTPU executes the prekey procedure of that key if a prekey procedure has been set.
2. For each key in the learn sequence, DECTPU executes procedures or programs in the following order:
 - a. DECTPU executes the prekey procedure of that key if a prekey procedure has been set.
 - b. DECTPU executes the code bound to the key itself.
 - c. DECTPU executes the postkey procedure of that key if a postkey procedure has been set.
3. When all keys in the learn sequence have been processed, DECTPU executes the postkey procedure, if one has been set, for the key to which the entire learn sequence was bound.

You can use the following calls to the GET_INFO built-in procedure to find the prekey and postkey procedures bound to a key map list:

```
GET_INFO (key_map_list_name, "pre_key_procedure")
GET_INFO (key_map_list_name, "post_key_procedure")
```

By default, newly created key map lists do not have postkey procedures.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (PRE_KEY_PROCEDURE) built-in requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_COMPILEFAIL	ERROR	Compilation aborted because of syntax errors.
TPU\$_NOKEYMAPLIST	WARNING	Attempt to access an undefined key map list.

Example

The following example displays a message after the code bound to a key is executed:

```
SET (PRE_KEY_PROCEDURE, "tpu$key_map_list",
'MESSAGE ("Working...")');
```

SET (PROMPT_AREA)

Syntax

```
SET (PROMPT_AREA, integer1, integer2, { NONE
BOLD
BLINK
REVERSE
UNDERLINE })
```


Parameters

PROMPT_AREA

An area on the screen in which the prompts generated by the built-in procedure READ_LINE are displayed. By default, there is no prompt area.

integer1

The screen line number at which the prompt area starts.

integer2

The number of screen lines in the prompt area.

NONE

Applies no video attributes to the characters in the prompt area.

BOLD

Causes the characters in the prompt area to be bolded.

BLINK

Causes the characters in the prompt area to blink.

REVERSE

Causes the characters in the prompt area to be displayed in reverse video.

UNDERLINE

Causes the characters in the prompt area to be underlined.

Description

The SET (PROMPT_AREA) procedure sets an area on the screen where prompts generated by the READ_LINE procedure are displayed. Except in Motif DECwindows, if the prompt area overlaps a line of a window that is visible on the screen, the line is erased when the READ_LINE is executed. When the execution of READ_LINE is completed, the line is restored. If the prompt area does not overlap any windows, the prompt area continues to display the READ_LINE prompt and your input until new information is sent to the prompt area.

If you have a multiple-line prompt area and your terminal has hardware scrolling capabilities, the first prompt appears on the last line of the prompt area. As subsequent prompts are issued, the previous prompts scroll up to make room for new ones. If there are more prompts than there are prompt-area lines, the extra prompts are scrolled out of the window.

If your terminal does not have hardware scrolling capabilities, prompts are displayed starting at the first line in the prompt area. When the prompt area is filled, display starts again at the first line in the prompt area.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (PROMPT_AREA) requires four parameters.
--------------	-------	---

TPU\$_TOOMANY	ERROR	You specified more than four parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be NONE, BOLD, BLINK, REVERSE, or UNDERLINE.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.
TPU\$_BADFIRSTLINE	WARNING	Prompt area must not start off screen or be less than one line long.
TPU\$_BADPROMPTLEN	WARNING	Prompt area must not extend off screen.

Example

The following example causes the prompt area to be screen line number 24. It is one line and is displayed in reverse video.

```
SET (PROMPT_AREA, 24, 1, REVERSE)
```

SET (RECORD_ATTRIBUTE)

Syntax

$$\text{SET (RECORD_ATTRIBUTE, } \left\{ \begin{array}{l} \text{buffer} \\ \text{marker} \\ \text{range} \end{array} \right\}, \left\{ \begin{array}{l} \text{DISPLAY_VALUE} \\ \text{LEFT_MARGIN} \end{array} \right\}, \left\{ \begin{array}{l} \text{display_setting_integer} \\ \text{margin_setting_integer} \end{array} \right\})$$

$$\text{SET (RECORD_ATTRIBUTE, } \left\{ \begin{array}{l} \text{buffer} \\ \text{marker} \\ \text{range} \end{array} \right\}, \text{MODIFIABLE, } \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

PROMPT_AREA

An area on the screen in which the prompts generated by the built-in procedure READ_LINE are displayed. By default, there is no prompt area.

integer1

The screen line number at which the prompt area starts.

integer2

The number of screen lines in the prompt area.

NONE

Applies no video attributes to the characters in the prompt area.

BOLD

Causes the characters in the prompt area to be bolded.

BLINK

Causes the characters in the prompt area to blink.

REVERSE

Causes the characters in the prompt area to be displayed in reverse video.

UNDERLINE

Causes the characters in the prompt area to be underlined.

Description

The SET (RECORD_ATTRIBUTE) procedure sets or alters any of three possible attributes for the specified record or records. The attributes you can set for a record are its left margin, its modifiability, and its visibility. With each call to SET (RECORD_ATTRIBUTE), you can set only one attribute. For example, you cannot change visibility and modifiability by using just one call. To set more than one record attribute, use multiple calls to SET (RECORD_ATTRIBUTE).

When you set an attribute for multiple records, each record gets the same value. For example, if you specify a range of records and a value for the left margin attribute, all records in the range receive the same left margin value.

You cannot change the left margin of an unmodifiable record. You can change the display value of a record at any time.

Signaled Errors

TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_INVPARAM	ERROR	The third parameter must be a keyword.
TPU\$_ARGMISMATCH	ERROR	The second or fourth parameter has an incorrect type.
TPU\$_BADKEY	WARNING	You specified an invalid keyword.
TPU\$_BADDISPVAL	WARNING	Display values must be between -127 and +127.
TPU\$_BADMARGINS	WARNING	You specified an illegal left margin value.

Examples

1. The following example uses statements that change buffer modifiability and record modifiability independently. You can turn on the modifiability of a record or range of records even when the buffer's modifiability is turned off.

```
SET (MODIFIABLE, buf1, OFF);
r1:= CREATE_RANGE (BEGINNING_OF(buf1), END_OF(buf1), REVERSE);
SET (RECORD_ATTRIBUTE, r1, MODIFIABLE, OFF);
SET (RECORD_ATTRIBUTE, r1, MODIFIABLE, ON);
SET (MODIFIABLE, buf1, ON);
```

2. The following example makes the records in the range *select_range* invisible in the current window:

```
SET (DISPLAY_VALUE, CURRENT_WINDOW, 0);
```

```
SET (RECORD_ATTRIBUTE, SELECT_RANGE, DISPLAY_VALUE, -1);
Descriptions
```

SET (RECORD_MODE)

Syntax

```
[keyword1 :=] SET (RECORD_MODE), { buffer
                                SYSTEM } , keyword2 )
```

Parameters

buffer

The buffer whose output record mode should be changed.

SYSTEM

A keyword that indicates that all new buffers created with no input file have the new record mode.

keyword2

The keyword that specifies the new record mode. It can be any of the following:

Keyword2	Record Syntax	Record Attributes
VARIABLE_NONE	fab\$c_var	0
VARIABLE_FTN	fab\$c_var	fab\$m_ftn
VARIABLE_CR	fab\$c_var	fab\$m_cr
STREAM	fab\$c_stm	fab\$m_cr
STREAM_CR	fab\$c_stmcr	fab\$m_cr
SYSTEM_DEFAULT	fab\$c_var	fab\$m_cr
SYSTEM_DEFAULT	fab\$c_s tmlf	fab\$m_cr
UNSPECIFIED	Use the record mode of the input file if supported; otherwise use the current system default. Valid only for buffers.	Use the record mode of the input file if supported; otherwise use the current system default. Valid only for buffers.

Return Value

Optionally returns a keyword for the previous record mode setting or the UNSPECIFIED keyword, if none.

Description

The SET (RECORD_MODE) procedure sets the record mode for a buffer or for all new buffers created without an associated input file. Record mode specifies the record format and record attributes for files written from the buffer.

This built-in does not affect journal files, work files, or section files. A buffer created with no input file gets the current system default record mode. A buffer created with an input file gets the record mode

from the input file if it is supported. If not supported, the buffer's record mode is left unspecified, and the output file takes the input file record mode.

Record modes are specific to file systems. Setting the record mode to a value not supported by your file system may result in your buffer being written to the disk in an unusable format.

Signaled Errors

TPU\$_INVPARAM	ERROR	The third parameter to the built-in has the wrong data type.
TPU\$_TOOFEW	ERROR	You specified too few parameters.
TPU\$_TOOMANY	ERROR	You specified too many parameters.
TPU\$_BADKEY	ERROR	You specified an invalid keyword for the second or third parameter.
TPU\$_BADDISPVAL	ERROR	The second parameter must be a buffer of the keyword SYSTEM.
TPU\$_BADMARGINS	ERROR	You cannot specify the keyword UNSPECIFIED for the system default record mode.

Examples

1. The following example sets the record mode of buffer *my_buffer* to STREAM_LF. Writing *my_buffer* to a file creates a file with STREAM_LF record format and carriage return record attributes.

```
SET (RECORD_MODE, my_buffer, STREAM_LF);
```

2. The following example sets the default record mode for all new buffers created with no input file. Files written from these buffers will have variable length record format and carriage return record attributes.

```
SET (RECORD_MODE, SYSTEM, VARIABLE_CR);
```

SET (RESIZE_ACTION)

Syntax

```
SET (RESIZE_ACTION  $\left[ \begin{array}{l} \text{buffer} \\ \text{learn\_sequence} \\ \text{program} \\ \text{range} \\ \text{string} \\ \text{NONE} \end{array} \right]$  )
```

Parameters

RESIZE_ACTION

A keyword that directs DECTPU to set an attribute related to a resize action routine.

buffer

The buffer that specifies the actions that DECTPU should take whenever it is notified of a resize event.

learn_sequence

The learn sequence that specifies the actions that DECTPU should take whenever it is notified of a resize event.

program

The program that specifies the actions that DECTPU should take whenever it is notified of a resize event.

range

The range that specifies the actions that DECTPU should take whenever it is notified of a resize event.

string

The string that specifies the actions that DECTPU should take whenever it is notified of a resize event.

NONE

A keyword that directs DECTPU to delete the resize action routine. If you specify this keyword or do not specify the parameter at all, the application is not notified when a resize event occurs.

Description

The SET (RESIZE_ACTION) procedure specifies code to be executed when a resize event has occurred. Specifying a resize action routine overrides any previous resize action routines that have been defined.

Signaled Errors

TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (RESIZE_ACTION) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (RESIZE_ACTION) builtin only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (RESIZE_ACTION) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (RESIZE_ACTION) built-in.

Example

The following example specifies the procedure EVE\$\$RESIZE_ACTION as the resize routine. To see this statement used in an initializing procedure, see the example in the description of the SET (SCREEN_LIMITS) built-in procedure.

```
SET (RESIZE_ACTION, "eve$$resize_action");
```

SET (REVERSE)

Syntax

```
SET (REVERSE, buffer)
```

Parameters

REVERSE

The direction of the buffer. REVERSE means to go toward the beginning of the buffer. The default direction for a buffer is forward.

buffer

The buffer whose direction you want to set.

Description

Interfaces use the SET (REVERSE) procedure to keep track of direction for searching or movement.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (REVERSE) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example causes the direction of the buffer to be toward the beginning of the buffer:

```
SET (REVERSE, my_buffer)
```

SET (RIGHT_MARGIN)

Syntax

```
SET (RIGHT_MARGIN, buffer, integer)
```

Parameters

RIGHT_MARGIN

The right margin of a buffer.

buffer

The buffer in which the right margin is being set.

integer

The column at which the right margin is set.

Description

With the SET (RIGHT_MARGIN) procedure, you can change only the right margin of a buffer.

Newly created buffers receive a right margin of 80 if a template buffer is not specified on the call to the CREATE_BUFFER built-in procedure. If a template buffer is specified, the right margin of the template buffer is used.

Use SET (RIGHT_MARGIN) to override the default right margin.

The buffer margin settings are independent of the terminal width or window width settings. The FILL built-in procedure uses these margin settings when it fills the text of a buffer.

The SET (RIGHT_MARGIN) built-in procedure controls the buffer margin setting even if the terminal width or window width is set to something else.

The value of the right margin must be less than the maximum record size for the buffer and greater than the left margin value. You can use the GET_INFO (buffer, "record_size") built-in procedure to find out the maximum record size of a buffer.

If you want to use the margin settings of an existing buffer, in a user-written procedure, the statements GET_INFO (buffer, "left_margin") and GET_INFO (buffer, "right_margin") return the values of the margin settings in the specified buffer.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (RIGHT_MARGIN) built-in requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADMARGINS	WARNING	Right must be greater than left; both must be greater than zero.

Examples

1. The following example causes the right margin of the buffer represented by the variable *my_buffer* to be changed. The right margin of the buffer is set to 132. The left margin is unchanged.

```
SET (RIGHT_MARGIN, my_buffer, 132)
```

2. The following example causes the right margin of the current buffer to be changed to 70. The left margin is unchanged.

```
SET (RIGHT_MARGIN, CURRENT_BUFFER, 70)
```

SET (RIGHT_MARGIN_ACTION)

Syntax

SET (RIGHT_MARGIN_ACTION, buffer1 $\left[\left[\left\{ \begin{array}{l} \text{buffer2} \\ \text{learn_sequence} \\ \text{program} \\ \text{range} \\ \text{string} \end{array} \right\} \right] \right] \right)$)

Parameters

RIGHT_MARGIN_ACTION

Refers to the action taken when you press a self-inserting key while the cursor is to the right of a buffer's right margin. A self-inserting key is one that is associated with a printable character.

buffer1

The buffer in which the right margin action routine is being set.

buffer2

A buffer that contains the DECTPU statements to be executed when you press a self-inserting key while the cursor is to the right of a buffer's right margin.

learn_sequence

A learn sequence that is to be replayed when you press a self-inserting key while the cursor is to the right of a buffer's right margin.

program

A program that is to be executed when you press a self-inserting key while the cursor is to the right of a buffer's right margin.

range

A range that contains DECTPU statements that are to be executed when you press a self-inserting key while the cursor is to the right of a buffer's right margin.

string

A string that contains DECTPU statements that are to be executed when you press a self-inserting key while the cursor is to the right of a buffer's right margin.

Description

With the SET (RIGHT_MARGIN_ACTION) procedure, you can specify an action to be taken when you attempt to insert text to the right of the right margin of a line. If the third parameter is not specified, the right margin action routine is deleted. If no right margin action routine has been specified, the text is inserted at the current position after any necessary padding spaces.

Newly created buffers do not receive a right margin action routine if a template buffer is not specified on the call to the CREATE_BUFFER built-in procedure. If a template buffer is specified, the right margin action routine of the template buffer is used.

The right margin action routine affects only text entered from the keyboard or a learn sequence. Using the COPY_TEXT or MOVE_TEXT built-in procedures to insert text into a buffer to the right of the right margin does not trigger the right margin action routine.

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (RIGHT_MARGIN_ACTION) built-in requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_COMPILEFAIL	WARNING	Compilation aborted because of syntax errors.

Examples

1. The following example causes the procedure FILL_CURRENT_LINE to be executed when you attempt to type a character to the right of the right margin of the current line. A typical right margin action routine invokes the FILL built-in procedure to fill the current line and force text to the right of the right margin to a new line.

```
SET (RIGHT_MARGIN_ACTION, CURRENT_BUFFER, "fill_current_line")
```

2. The following example deletes any right margin action routine that may be defined for the current buffer. If you attempt to type a character to the right of the right margin of the current line, the text is inserted with spaces padding the text from the end of the line.

```
SET (RIGHT_MARGIN_ACTION, CURRENT_BUFFER)
```

SET (SCREEN_LIMITS)

Syntax

```
SET (SCREEN_LIMITS, array)
```

Parameters

SCREEN_LIMITS

A keyword that directs DECTPU to pass hints to the DECwindows window manager about screen size.

array

An integer-indexed array that specifies hints for the minimum and maximum screen width and length. The second pair of elements are optional. The array indices and their corresponding elements are as follows:

- The minimum screen width, in columns. This value must be at least 0 and less than or equal to the maximum screen width. The default value is 0.
- The minimum screen length, in lines. This value must be at least 0 and less than or equal to the maximum screen length. The default value is 0.

- The maximum screen width, in columns. This value must be greater than or equal to the minimum screen width and less than or equal to 255. The default value is 255. This element is optional, but if present, must be accompanied by the fourth element.
- The maximum screen length, in lines. This value must be greater than or equal to the minimum screen length and less than or equal to 255. The default value is 255. This element is optional.

Description

The SET (SCREEN_LIMITS) procedure specifies the minimum and maximum allowable sizes for the DECTPU screen during resize operations. DECTPU passes these limits to the DECwindows window manager, which is free to use or ignore the limits.

Signaled Errors

TPU\$_BADVALUE	WARNING	An integer parameter was specified with a value outside the valid range.
TPU\$_MAXVALUE	WARNING	You specified a value higher than the maximum allowable value.
TPU\$_MINVALUE	WARNING	You specified a value lower than the minimum allowable value.
TPU\$_EXTRANEOUSARGS	ERROR	One or more extraneous arguments have been specified for a DECwindows built-in.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (SCREEN_LIMITS) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (SCREEN_LIMITS) builtin only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (SCREEN_LIMITS) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (SCREEN_LIMITS) built-in.
TPU\$_REQARGSMISSING	ERROR	One or more required arguments are missing.

Example

The following procedure sets up screen size limits. It is part of the EVE\$ \$DECWINDOWS_MODULE_INIT procedure. The original version is in SYS\$EXAMPLES:EVE \$DECWINDOWS.TPU.

```
! Module Initialization
LOCAL temp_array;
eve$x_decwindows_active := GET_INFO (SCREEN, "decwindows");
! .
! .
! .
IF NOT eve$x_decwindows_active
THEN
```

```

RETURN (FALSE)
ENDIF;
! The following statements set up to handle resize actions.
temp_array := CREATE_ARRAY (4);
temp_array {1} := 20; ! Minimum width.
temp_array {2} := 6; ! Minimum height.
! Don't set max for Motif DECwindows so the maximize button will
! make the window fill the screen.
SET (SCREEN_LIMITS, temp_array);
SET (RESIZE_ACTION, "eve$$resize_action");
SET (ENABLE_RESIZE, ON);
! .
! .
! .
ENDMODULE;

```

SET (SCREEN_UPDATE)

Syntax

$$\left[\left\{ \begin{array}{c} \text{ON} \\ \text{OFF} \end{array} \right\} : = \right] \text{SET (SCREEN_UPDATE, } \left\{ \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\} \left[, \text{window_variable} \right])$$

Parameters

SCREEN_UPDATE

A keyword that directs DECTPU to set an attribute of screen updating.

ON, 1

A keyword that indicates that screen updating is enabled.

OFF, 0

A keyword that indicates that screen updating is disabled.

window_variable

The window for which you want screen updating turned on or off. Windows are set to update by default.

If you set a window to “no update,” the screen updater ignores the window. Applications that modify part of the screen through some external means may map a “no-update” window to that portion or portions of the screen to prevent DECTPU from overwriting the screen. EVE does not support the use of “no-update” windows.

Return Value

A variable that contains the keyword value ON or OFF. The keyword specifies whether DECTPU screen updating support was enabled or disabled before the current SET (SCREEN_UPDATE) statement was executed. Using the returned variable, you can enable or disable screen updating and then reset the support to its previous setting without having to make a separate call to fetch the previous setting.

Description

The SET (SCREEN_UPDATE) procedure turns on or turns off support for screen updating. When you set SCREEN_UPDATE on, the screen manager is immediately called to update the screen. The extent of the update depends on the built-ins that have been used since the last screen update. The update may range from a complete screen refresh to an updating of the existing text on the screen.

For more information on screen updating, see the *Guide to the DEC Text Processing Utility*

Signaled Errors

TPU\$_BADKEY	WARNING	The keyword must be ON or OFF.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_TOOFEW	ERROR	SET (SCREEN_UPDATE) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_UNKKEYWORD	ERROR	You have specified an unknown keyword.

Example

The following example causes screen updating to be turned off. When you design an editing interface, you can use this statement to prevent some intermediate processing steps from appearing on the screen.

```
SET (SCREEN_UPDATE, OFF)
```

SET (SCROLL_BAR)

Syntax

$$\llbracket \left\{ \begin{array}{l} \text{integer} \\ \text{widget} \end{array} \right\} : = \rrbracket \text{SET} (\text{SCROLL_BAR}, \text{window}, \left\{ \begin{array}{l} \text{HORIZONTAL} \\ \text{VERTICAL} \end{array} \right\}, \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

SCROLL_BAR

A keyword that directs DECTPU to enable or disable a scroll bar in a DECTPU window.

window

The window in which the scroll bar does or does not appear.

HORIZONTAL

A keyword that directs DECTPU to enable or disable a horizontal scroll bar.

VERTICAL

A keyword that directs DECTPU to enable or disable a vertical scroll bar.

ON, 1

A keyword that indicates that the scroll bar is to be visible in the specified window.

OFF, 0

A keyword that indicates that the scroll bar is not to be visible in the specified window.

Return Values**integer**

The value 0 if an error prevents DECTPU from associating a widget with the window.

widget

The widget that implements the vertical or horizontal scroll bar associated with a window.

Description

The SET (SCROLL_BAR) procedure enables a horizontal or vertical scroll bar for the specified window. Scroll bars represent the location of the editing point in the buffer. By dragging the scroll bar's slider, you can reposition the editing point in the buffer mapped to the window.

Scroll bars are unique among DECTPU widgets in the following respects:

- Each scroll bar widget is associated with a specific DECTPU window.
- Instead of handling scroll widgets at the application level, you can direct DECTPU to handle resizing and repositioning of the scroll bar slider. DECTPU always handles sizing and positioning of the scroll bar itself.

Windows having fewer than four lines of text cannot display a vertical scroll bar. Similarly, a window less than four columns wide cannot display a horizontal scroll bar. When the size of a DECTPU window changes, DECTPU automatically adjusts the scroll bar to fit the new window size. If a window becomes too small to support a scroll bar, DECTPU turns off the scroll bar. However, if the window subsequently becomes larger, DECTPU automatically turns the scroll bar back on.

SET (SCROLL_BAR) returns the scroll bar widget, or 0 if an error prevents DECTPU from associating a widget with the window.

By default, DECTPU creates its windows without any scroll bars; using SET (SCROLL_BAR) with the ON keyword overrides the default. To make a scroll bar invisible after it has been placed in a window (for example, to allow the user of a layered application to turn off scroll bars), use SET (SCROLL_BAR) with the OFF keyword.

The height of a vertical scroll bar represents the total number of lines in the buffer mapped to the window.

The width of a horizontal scroll bar represents the greater of the following:

- The width of the widest line in the set of lines visible in the window. “Width” means the distance from the first character on the line to the last character, regardless of whether all characters on the line are visible.

- The width of the widest line from the first character on the line to the rightmost window column, when none of the lines in the set of lines visible in the window has text extending all the way to the rightmost window column.

The horizontal scroll bar represents only the lines that are visible in the window, not all the lines in the buffer mapped to the window.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (SCROLL_BAR) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (SCROLL_BAR) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (SCROLL_BAR) built-in.

Example

The following example turns on a vertical scroll bar in the current window:

```
vertical_bar := SET (SCROLL_BAR, CURRENT_WINDOW, VERTICAL, ON);
```

SET (SCROLL_BAR_AUTO_THUMB)

Syntax

$$\text{SET (SCROLL_BAR_AUTO_THUMB, window, } \left\{ \begin{array}{l} \text{HORIZONTAL} \\ \text{VERTICAL} \end{array} \right\}, \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

SCROLL_BAR_AUTO_THUMB

A keyword that directs DECTPU to enable or disable automatic adjustment of the scroll bar slider in a DECTPU window.

window

The window whose scroll bar slider you want DECTPU to adjust.

HORIZONTAL

A keyword that directs DECTPU to set the slider on a horizontal scroll bar.

VERTICAL

A keyword that directs DECTPU to set the slider on a vertical scroll bar.

ON, 1

A keyword that directs DECTPU to enable automatic adjustment of the scroll bar slider.

OFF, 0

A keyword that directs DECTPU to disable automatic adjustment of the scroll bar slider.

Description

The SET (SCROLL_BAR_AUTO_THUMB) procedure enables or disables automatic adjustment of the scroll bar slider. By default, SET (SCROLL_BAR_AUTO_THUMB) is set to ON and DECTPU automatically manages a window's scroll bar slider in the following ways:

- Adjusts the size of the slider as you add, delete, or move text, so that the slider size represents the amount of visible text in relation to the total amount of text
- Adjusts the size of the slider whenever the size of the window and the size of the scroll bar change, so that the slider size remains proportional to the scroll bar size
- Adjusts the position of the slider as you add, delete, or move text, so that the slider shows whether the current buffer or line contains text not visible on the screen and, if so, where the invisible text is in relation to the visible text

When the scroll bar slider is adjusted automatically, the width of the slider in a horizontal scroll bar represents the width of the window. For example, the size of the slider changes when the window width is changed from 80 to 132 columns or the reverse. The position of the slider changes when the window is shifted left or right. The height of the slider in a vertical scroll bar represents the height of the window.

If you do not want DECTPU to adjust the scroll bar slider automatically or if you want to change the size or position of the slider, specify the OFF keyword. For more information about calculating the size and position of the slider, see the description of the SET (SCROLL_BAR) built-in procedure.

You cannot disable DECTPU's automatic adjustment of the scroll bar itself. DECTPU always adjusts the scroll bar to the size of the window.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (SCROLL_BAR_AUTO_THUMB) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (SCROLL_BAR_AUTO_THUMB) built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (SCROLL_BAR_AUTO_THUMB) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (SCROLL_BAR_AUTO_THUMB) built-in.

Example

The following example turns on automatic adjustment of the vertical scroll bar's slider in the current window:

```
vertical_bar := SET (SCROLL_BAR_AUTO_THUMB, CURRENT_WINDOW, VERTICAL, ON);
```

SET (SCROLLING)

Syntax

$$\text{SET (SCROLLING, window, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\}, \text{ integer1, integer2, integer3) or SET (SCROLLING, window, integer1, integer2, integer3)}$$

Parameters

SCROLLING

A keyword that refers to the upward or downward movement of existing lines in a window to make room for new lines at the bottom or top of the window. When a window is scrolled, the cursor position remains in the same column, but the screen line that the cursor is on may change.

window

The window in which the scrolling limits are being set.

ON, 1

Causes scrolling of the text in a window to be turned on. This is the default value for the third parameter if the terminal supports scrolling.

OFF, 0

Causes scrolling of the text in a window to be turned off. The screen is completely repainted each time a scroll would otherwise take place. This is the default value for the third parameter if the terminal does not support scrolling.

integer1

The offset from the top screen line of a window. The offset identifies the top limit of an area in which the cursor can move as it tracks the editing point. If the cursor is forced to move above this screen line to track the editing point, lines in the window move downward so that the cursor stays within the limits of the scroll margins. If you reach the beginning of the buffer, the text is no longer scrolled.

The value you specify for this parameter must be greater than or equal to zero and less than or equal to the number of lines in the window.

integer2

The offset from the bottom screen line of a window. The offset identifies the bottom limit of an area in which the cursor can move as it tracks the editing point. If the cursor is forced to move

below this screen line to track the editing point, lines in the window move upward so that the cursor stays within the limits of the scroll margins. If you reach the end of the buffer, the text is no longer scrolled.

The value you specify for this parameter must be greater than or equal to zero and less than or equal to the number of lines in the window.

integer3

The number that indicates how many lines from the top or the bottom scroll margin the cursor should be positioned after a window is scrolled. For example, if the bottom scroll margin is screen line 14 an *integer3* has a value of 0, the cursor is positioned on screen line 14 after text is scrolled upward. However, if *integer3* has a value of 3, the cursor is positioned on screen line 11.

The value you specify for this parameter must be greater than or equal to zero and less than or equal to the number of lines in the window.

You cannot specify a value that would position the cursor outside the window. That is, *integer1* + *integer3* or *integer2* + *integer3* must be less than the height of the window. For example, if the window is 10 lines long and *integer1* is set at 3, you cannot specify a value of 7 or more for *integer3*. Such a specification would place the cursor outside the window.

If you use the SET (SCROLLING) built-in procedure from within EVE by way of the TPU command, EVE may override the value you specify for this parameter.

JUMP

Directs DECTPU to repaint new text instead of scrolling new text into your window. When scrolling is set to jump mode, DECTPU first scrolls the text that will remain in your window, leaving part of the window empty. DECTPU then displays the new text in the empty region in a single repaint operation. Scrolling is faster in jump mode than in smooth mode because only part of your window is scrolled. To determine if the scrolling mode is set to the jump setting, use the GET-INFO (SCREEN, "jump_scroll") built-in procedure. A return value of 1 indicates that the jump setting is in effect.

SMOOTH

Directs DECTPU to repaint each new line of text as it is brought into the window. When scrolling is set to smooth mode, the text appears to slide smoothly in and out of the window. This setting is the default. To determine if the scrolling mode is set to the smooth setting, use the GET-INFO (SCREEN, "jump_scroll") built-in procedure. A return value of 0 indicates that the smooth setting is in effect.

Description

The SET (SCROLLING) procedure modifies the scrolling action of a window.

If the terminal on which you are running DECTPU supports scrolling, you can use the SET (SCROLLING) built-in procedure to turn scrolling on or off. If the terminal does not support scrolling, scrolling will always be off. If scrolling is off, the window is repainted every time a scroll would otherwise occur.

The SET (SCROLLING) built-in procedure also defines scroll margins by using *integer1* and *integer2*. If you use CURSOR_VERTICAL, MOVE_HORIZONTAL, MOVE_VERTICAL, POSITION, or a text

manipulation built-in to move the cursor above the top scroll margin or below the bottom scroll margin, then SET (SCROLLING) moves the cursor by the number of lines specified in *integer3*.

You must provide values for *integer1* and *integer2* that leave at least one line in the window unaffected by either scroll margin. That is, *integer1* + *integer2* must be less than the height of the window. For example, if you have a window that is ten lines tall, you cannot specify a value of 5 for the top scroll margin and a value of 5 for the bottom scroll margin. Such a specification leaves no area of the window that is not within a scroll margin.

You can move the cursor above or below a scroll margin under certain circumstances. If CROSS_WINDOW_BOUNDS is set to off, CURSOR_VERTICAL does not cause scrolling when the cursor reaches a scroll margin. If you are moving backward through the file and the top line of the buffer is already visible on the screen, the top scroll margin is ignored. If you are moving forward through the file and the bottom line of the buffer is already visible on the screen, the bottom scroll margin is ignored.

If using the ADJUST_WINDOW built-in procedure makes the window so much smaller that the scroll margins overlap, DECTPU automatically reduces the scroll margins proportionally to fit the new window. If you use ADJUST_WINDOW to make a window larger, DECTPU does not adjust the scroll margins.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (SCROLLING) requires at least six parameters.
TPU\$_TOOMANY	ERROR	You specified more than six parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters has the wrong type.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.
TPU\$_BADKEY	ERROR	Keyword must be either ON or OFF.
TPU\$_BADMARGINS	ERROR	You specified values for the top margin, bottom margin, and cursor movement that exceed the dimensions of the window.
TPU\$_BADVALUE	ERROR	Integer values must be from 0 to 255.

Examples

1. The following example turns on scrolling in the window `new_window`. The statement sets the top and bottom scroll margins to 0. This means that you can move the cursor all the way to the top or bottom of the window before new text is scrolled into the window. Finally, the statement causes DECTPU to place the cursor two lines down from the top or up from the bottom of the window when scrolling is completed.

```
SET (SCROLLING, new_window, ON, 0, 0, 2)
```

2. The following example causes DECTPU to repaint new text in your window in one operation instead of scrolling each line of text:

```
SET (SCROLLING, JUMP);
```

SET (SELF_INSERT)

Syntax

$$\text{SET (SELF_INSERT, string, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

SELF_INSERT

A keyword that specifies whether a character is inserted into the buffer when you press a key with the following characteristics:

- Associated with a printable character
- Not bound to a procedure or program

string

A string that specifies the key map list in which the behavior of undefined keys associated with printing characters is to be set.

ON, 1

Causes the printable characters to be inserted when no procedures are bound to them while the specified key map list is active. This is the default.

OFF, 0

Causes the UNDEFINED_KEY procedure to be called when you enter these undefined characters. If an undefined key procedure has not been specified, DECTPU displays a warning message when you press an undefined, printable key. You can specify an undefined key procedure by using the SET (UNDEFINED_KEY) built-in procedure.

Description

With the SET (SELF_INSERT) procedure, you can control what happens when you press an undefined key associated with a printable character. If SELF_INSERT is set ON and you press an undefined key associated with a printable character, the character is inserted into the current buffer at the current cursor position. If SELF_INSERT is turned off, printable characters whose keys are not defined in any key maps in the key map list bound to the current buffer are considered undefined. These undefined keys cause either the message “key has no definition” to be displayed or some user-defined action to occur.

The default result for pressing an undefined key associated with a printable character procedure is that the character is inserted. The default condition for SET (SELF_INSERT) is ON. The default behavior, if SET (SELF_INSERT) is OFF, is to call the UNDEFINED_KEY procedure.

For more information on how to define what happens when SET (SELF_INSERT) is turned off, see the description of the SET (UNDEFINED_KEY) built-in procedure.

Signaled Errors

TPU\$_NOKEYMAPLIST	WARNING	You attempted to access an
--------------------	---------	----------------------------

		undefined key map list.
TPU\$_TOOFEW	ERROR	SET (SELF_INSERT) requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example toggles the ON and OFF setting of SELF_INSERT for the key map list bound to the current buffer:

```
PROCEDURE toggle_self_insert
  LOCAL current_key_map_list;
  current_key_map_list := GET_INFO (CURRENT_BUFFER, "key_map_list");
  IF GET_INFO (current_key_map_list, "self_insert")
  THEN
    SET (SELF_INSERT, current_key_map_list, OFF)
  ELSE
    SET (SELF_INSERT, current_key_map_list, ON)
  ENDIF;
ENDPROCEDURE;
```

SET (SHIFT_KEY)

Syntax

SET (SHIFT_KEY, keyword [, string])

Parameters

SHIFT_KEY

A keyword that refers to DECTPU's shift key (by default PF1), not the key marked Shift on the keyboard.

keyword

A DECTPU key name for a key.

string

A string that is a key map list name. This optional argument specifies the key map list in which the shift key is used. If the key map list is not specified, the key map list associated with the current buffer is used.

Description

With the SET (SHIFT_KEY) procedure, you can assign two commands to one key: one is used when the key is pressed by itself, and the other is used when the key is pressed after the defined shift key. The DECTPU shift key is similar to the GOLD key in EDT editor.

Only one DECTPU shift key can be active at a time. The DECTPU shift key can be any key other than the following keys:

- Shift
- Escape
- SCROLL on the VT100 keyboard
- F1, F2, F3, F4, and F5 on the LK201 or LK401 keyboards
- Compose Character on the LK201 or LK401 keyboards

By default, PF1 is the DECTPU shift key.

You cannot make DECTPU execute a procedure or learn sequence bound to the shift key. However, designating a defined key as the shift key does not undefine the key; it merely disables the definition when the key is designated as the shift key. If you define another key as the shift key, DECTPU reenables the first key's definition.

If you want to use PF1 for another purpose, use SET (SHIFT_KEY) to define a key other than PF1 as DECTPU's shift key.

If you use SET (SHIFT_KEY) to define a GOLD key in EVE, EVE does not undefine the GOLD key correctly. When you use the EVE SET NOGOLD command, EVE returns the error message "There is no user GOLD key currently set." Although this message appears to say that the GOLD key has successfully been undefined, what it really means is that EVE does not recognize that a GOLD key was ever defined.

To redefine a GOLD key in these circumstances, you can do either of the following:

- Use the EVE SET GOLD KEY command.
- Undefine the GOLD key by using the DECTPU statement SET (SHIFT_KEY, KEY_NAME (PF1, SHIFT_KEY)). Then set the GOLD key by using the SET GOLD KEY command.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (SHIFT_KEY) requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.
TPU\$_NOKEYMAPLIST	WARNING	You specified an undefined key map list.

Examples

1. The following example causes the keypad key PF4 to be defined as the shift key for the editor. The definition is stored in the default key map list, TPU\$KEY_MAP_LIST. PF4 operates as the shift key only in buffers to which TPU\$KEY_MAP_LIST is bound.

```
SET (SHIFT_KEY, PF4, "tpu$key_map_list")
```

2. The following example disables the shift key by making the shift key itself a shifted key:

```
SET (SHIFT_KEY, KEY_NAME (PF1, SHIFT_KEY))
```

You can substitute the key name of whatever key is the shift key. This technique works regardless of what key is defined as the shift key. You might want to use such a statement if you are creating an editor that does not support user-defined shift key sequences.

SET (SPECIAL_ERROR_SYMBOL)

Syntax

```
SET (SPECIAL_ERROR_SYMBOL, string)
```

Parameters

SPECIAL_ERROR_SYMBOL

A keyword that specifies that you want to use SET to designate a global variable to be set to 0 when a case-style error handler does not return from a Ctrl/C or other error.

string

The name of the global variable that you want DECTPU to set to 0.

Description

The SET (SPECIAL_ERROR_SYMBOL) procedure designates a global variable to be set to 0 when a case-style error handler does not return from a Ctrl/C or other error. Once you designate the variable that is to be the special error symbol, DECTPU sets the variable to 0 if any of the following events occurs:

- DECTPU executes the TPU\$_CONTROL_C selector in a case-style error handler and does not encounter a RETURN statement.
- DECTPU executes the OTHERWISE clause in a case-style error handler and does not encounter a RETURN statement.
- DECTPU generates an error that is not handled by any clause in a case-style error handler.

You can use SET (SPECIAL_ERROR_SYMBOL) only once in a program. You usually use this built-in during initialization. You must declare or create the variable before you use it in the SET statement. DECTPU does not clear the variable in response to noncase-style error handlers.

You can use the variable specified by SET (SPECIAL_ERROR_SYMBOL) to determine whether DECTPU has exited from current procedures and returned to the main loop to wait for a new keystroke.

Signaled Errors

TPU\$_ERRSYMACTIVE	ERROR	A special error symbol has already been declared.
TPU\$_TOOFEW	ERROR	SET (SPECIAL_ERROR_SYMBOL) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.

TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
----------------	-------	--

Example

The following example designates the global variable *back_to_main* as the variable to be cleared if a procedure or program with a case-style error handler fails to handle a Ctrl/C error or other error:

```
SET (SPECIAL_ERROR_SYMBOL "back_to_main")
```

SET (STATUS_LINE)

Syntax

```
SET (STATUS_LINE, window, {
    BLINK
    BOLD
    REVERSE
    SPECIAL_GRAPHICS
    UNDERLINE
    NONE
}, string )
```

Parameters

STATUS_LINE

The last line in a window. You can use the status line to display regular text or you can use it to display status information about the window.

window

The window whose status line you want to modify.

BLINK

Causes the characters on the status line to blink.

BOLD

Causes the characters on the status line to be bolded.

REVERSE

Causes the characters on the status line to be displayed in reverse video.

SPECIAL_GRAPHICS

Causes the characters on the status line to display graphic characters, such as a solid line. These characters are from the DEC Special Graphics Set (also known as the VT100 Line Drawing Character Set). For more information on the special graphics that are available, see the appropriate programming manual for your terminal.

UNDERLINE

Causes the characters on the status line to be underlined.

NONE

Applies no video attributes to the characters on the status line.

string

A string that is the text to be displayed on the status line. To remove a status line, use a null string ("") for this parameter.

Description

With the SET (STATUS_LINE) procedure, you can set the last line in a window. To have a status line, a window must be at least two lines high. You can establish a status line for a window when you create a window. CREATE_WINDOW requires you to specify whether the status line is ON (used for status information) or OFF (used as a regular text line). When you specify ON, the default status line is displayed in reverse video.

The algorithm for determining whether a window is tall enough to be given a status line depends on whether the window is visible or invisible.

If the window to which you want to add a status line is visible, DECTPU checks the length of the visible portion of the window. A visible window can have an invisible portion if the window is partially occluded by another window. The visible portion of the visible window must have at least one text line; that is, at least one line not occupied by a scroll bar.

If the window is invisible, DECTPU checks the full length of the window. The window must have at least one text line.

If the window that you use as a parameter for SET (STATUS_LINE) already has a status line, either because you specified ON for the status line parameter in the CREATE_WINDOW built-in procedure, or because you used a previous SET (STATUS_LINE) for the window, the video attribute that you specify is added to the video attribute of the existing status line unless you specify NONE. NONE overrides the other video keywords and specifies that there are to be no video attributes for the status line. The string you specify as the last parameter replaces the text of an existing status line. Adding a status line to a window that already has a status line does not cause an error.

If there is no status line for a window, the SET (STATUS_LINE) built-in procedure establishes a status line on the last visible screen line of the window. The status line has the video attribute and the text you specify. Adding a status line reduces the number of screen lines available for text by one line.

To remove a status line, use a null string ("") as the last parameter. The status line is removed even if the window is not two lines high at that time.

The default setting for the status line (ON or OFF) is determined by the CREATE_WINDOW built-in procedure.

If a window has a status line, by default the status line contains the name of the buffer associated with the window and the name of the file associated with the buffer, if there is one.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (STATUS_LINE) requires four parameters.
TPU\$_TOOMANY	ERROR	You specified more than four parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

See Appendix B for a table of the DECTPU messages and their severity levels.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (SUCCESS) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example turns off the display of success messages:

```
SET (SUCCESS, OFF)
```

SET (SYSTEM)

Syntax

```
SET (SYSTEM, buffer)
```

Parameters

SYSTEM

The status of a buffer. SYSTEM means that it is a system buffer rather than a user buffer.

By default, newly created buffers are user buffers.

buffer

The buffer that is being set as a system buffer.

Description

With the SET (SYSTEM) procedure, programmers who are building an editing interface can distinguish their system buffers from user-created buffers. Once you make a buffer a system buffer, you cannot reset the buffer to be a user buffer. DECTPU does not handle system buffers differently from user buffers. Any distinction between the two kinds of buffers must be implemented by the application programmer.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (SYSTEM)) requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.

Example

The following example makes the message buffer a system buffer:

```
SET (SYSTEM, message_buffer)
```

SET (TAB_STOPS)

Syntax

```
SET (TAB_STOPS, buffer, { integer } )
```

Parameters

TAB_STOPS

A keyword that indicates that SET is to control placement of tab stops in a buffer.

buffer

The buffer in which the tab stops are being set.

integer

An integer that specifies the interval between tab stops, measured in column widths. The minimum value for the integer is 1. The maximum value is 65535.

string

A string of numbers that specifies the tab stops. The string represents column numbers at which the tab stops are placed. The minimum value for a tab stop is 1. The maximum value is 65535. The maximum number of tab stops that you can include in the string is 100. The quoted string must list tab stops in ascending order, separating values with a single space (for example, "3 6 9 12").

Description

With the SET (TAB_STOPS) procedure, you can set the tab stops at positions you specify or to establish equal intervals other than the default eight. When a buffer is created, the tabs are set at every eight columns, unless, when the buffer is created, a template buffer with different tab settings is specified.

Tab stops are not saved when you write a file. When you create a buffer, the tabs are set to the default unless, when you create the buffer, you specify a template buffer with different tab settings. For information on creating buffers, see the CREATE_BUFFER procedure.

SET (TAB_STOPS) does not affect the hardware tab settings of your terminal. On any terminals or printers that have tab settings different from those you specify with this built-in, the file does not appear the same as it does when viewed using DECTPU. In addition, if you invoke DECTPU with the /NODISPLAY qualifier, any values you enter for SET (TAB_STOPS) are ignored, and a SHOW (BUFFER) command will show no tab stops for the buffer.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (TAB_STOPS) requires at least three parameters.
--------------	-------	---

TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.
TPU\$_ARGMISMATCH	ERROR	The third parameter must be a string or an integer.
TPU\$_INVTABSPEC	WARNING	You specified a bad third argument.

Examples

1. The following example causes the tab stops in the current buffer to be set at intervals of four columns:

```
SET (TAB_STOPS, CURRENT_BUFFER, 4);
```

2. The following example causes the tab stops in the current buffer to be set at 4, 8, 12, and 16 columns:

```
SET (TAB_STOPS, CURRENT_BUFFER, "4 8 12 16");
```

SET (TEXT)

Syntax

$$\text{SET (TEXT, } \left\{ \begin{array}{l} \text{widget, string} \\ \text{key-window, } \left\{ \begin{array}{l} \text{BLANK_TABS} \\ \text{GRAPHIC_TABS} \\ \text{NO_TRANSLATE} \end{array} \right\} \end{array} \right\})$$

Parameters

TEXT

A keyword that indicates that SET is to control the way text is displayed in a window or to determine the text that is to appear in a widget.

widget

The widget whose text you want to set. SET (TEXT, widget, string) is equivalent to the Toolkit routine *S TEXT SET STRING*.

You can use *widget* as the second parameter only if you are using DECwindows DECTPU.

string

The text you want to assign to the simple text widget.

window

The window in which the mode of display is being set.

BLANK_TABS

Displays tabs as blank spaces. This is the default keyword.

GRAPHIC_TABS

Displays tabs as special graphic characters so that the width of each tab is visible.

NO_TRANSLATE

Sends every keystroke from the keyboard to the terminal without any translation. In this mode, the terminal settings, not DECTPU, determine the effect of characters typed from the keyboard.

VSI recommends that you use this mode for sending directives to the terminal but not for editing. DECTPU does not manage margins or window shifts while NO_TRANSLATE mode is enabled. Furthermore, DECTPU does not necessarily update lines of text in the order in which they appear while NO_TRANSLATE mode is enabled.

To send escape sequences from within a DECTPU procedure, you can use SET (TEXT) with the NO_TRANSLATE keyword followed by statements that use the MESSAGE and UPDATE built-in procedures.

For more information on the effect of using various characters and sequences in NO_TRANSLATE mode, see your terminal manual.

Description

With the SET (TEXT) procedure, you control the way text is displayed in a window or determine the text that is to appear in a widget.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	SET (TEXT) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You have specified <i>widget</i> as the second parameter to SET (TEXT) while using non-DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (TEXT) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (TEXT) built-in.
TPU\$_WIDMISMATCH	ERROR	The specified widget is not of class SText.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.

Examples

1. In the following example, assuming that the variable *user_text_widget* has been assigned a text widget, this statement causes the widget to display the text *No default string available*:

```
SET (TEXT, user_text_widget, "No default string available.");
```

2. The following example shows one possible way that a layered application can use the SET (TEXT) widget. The variable *eve\$x_target* stores the string (if one exists) that you specified as the wildcard search string the last time you invoked the wildcard find dialog box. The SET (TEXT) statement directs EVE's wildcard find dialog box widget to display the string assigned to *eve\$x_target*.

```
wildcard_dialog_box := GET_INFO (WIDGET, "widget_id",  
eve$x_wildcard_find_dialog,  
"WILDCARD_FIND_DIALOG.WILDCARD_FIND_TEXT");  
status := SET (TEXT, wildcard_dialog_box, eve$x_target);
```

SET (TIMER)

Syntax

$$\text{SET (TIMER, } \left. \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\} \llbracket \text{, string } \rrbracket)$$

Parameters

TIMER

Controls the displaying of timed messages in the prompt area.

ON, 1

Causes the message that you specify to be written to the prompt area and displayed at 1-second intervals. By default, the display of the timed message is turned off.

OFF, 0

Turns off the display of timed messages in the prompt area.

string

A string that is displayed in the prompt area. The maximum length of the message is 15 characters. If you specify a string longer than 15 characters, DECTPU truncates the string but does not signal an error. The message is displayed in the last 15 character positions of the prompt area. If ON is specified and a string was never specified for the last argument, the timer puts out the message "working." If ON is specified and a string was specified previously, the saved string is used as the default.

Description

When the SET (TIMER) procedure is set to ON, the timer puts out messages at 1-second intervals while you are executing procedures or editing actions that are bound to a key. The message is written out to the prompt area and then erased to clear the prompt area for the next message. The first display of the timed message does not occur until 3 seconds after you press the key. This eliminates unnecessary timed messages during short editing transactions.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (TIMER) requires at least two parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be ON or OFF.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.

Example

The following example causes the message “Executing” to be written to the prompt area at 1-second intervals while you are executing a DECTPU procedure:

```
SET (TIMER, ON, "Executing")
```

SET (TRACEBACK)

Syntax

$$\text{SET (TRACEBACK, } \left. \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\})$$

Parameters

TRACEBACK

Determines whether DECTPU displays the sequence of procedures called after an error occurs.

ON, 1

Causes DECTPU to display the procedure calling sequence after an error occurs.

OFF, 0

Prevents DECTPU from displaying the procedure calling sequence after an error occurs.

Description

With the SET (TRACEBACK) procedure, you can determine whether DECTPU provides information on the context in which an error occurs. Turning on the traceback setting can be helpful to a programmer debugging a DECTPU program. The traceback setting is usually turned off during normal editing because end users of editors do not often use the traceback information.

The default setting for TRACEBACK depends on whether a section file was loaded by DECTPU. If a section file was loaded, the default is OFF. If a section file was not loaded, the default is ON.

SET (TRACEBACK) is related to SET (LINE_NUMBER). SET (TRACEBACK, ON) turns on both traceback and line numbers because both are needed for debugging. SET (LINE_NUMBER, OFF) turns off both traceback and line numbers because one feature is not useful without the other.

Allowable settings are as follows:

- Both off
- Both on
- Traceback off
- Line number on

Signaled Errors

TPU\$_TOOFEW	ERROR	The SET (TRACEBACK) built-in requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	WARNING	Only ON and OFF are allowed.

Examples

1. The following example prevents DECTPU from displaying the procedure calling sequence after an error occurs:

```
SET (TRACEBACK, OFF)
```

2. The following example results in a traceback display when the procedure is executed and traceback is enabled:

```
PROCEDURE traceback_example
SET (TRACEBACK, ON);
SET (TRACEBACK, BELL);
RETURN 5;
ENDPROCEDURE;
PROCEDURE call_example
traceback_example;
ENDPROCEDURE;
```

Invoking the procedure CALL_EXAMPLE results in the following traceback:

```
BELL is an invalid keyword
Occurred in builtin SET
At line 2
Called from builtin EXECUTE
Called from line 22 of procedure EVE_TPU
Called from line 1
Called from builtin EXECUTE
Called from line 96 of procedure EVE$PROCESS_COMMAND
Called from line 3 of procedure EVE$PARSER_DISPATCH
Called from line 97 of procedure EVE$$EXIT_COMMAND_WINDOW
Called from line 2
```

SET (UID)

Syntax

```
integer := SET (UID, filespec [, filespec... ] )
```

Parameter

filespec

A string that specifies the UID file to be used. DECTPU does not apply a default file specification to the UID file specification. You must specify at least one file name.

Return Value

An integer that is the identification number for the DECwindows resource manager hierarchy.

Description

The SET (UID) procedure sets the User Interface Definition (UID) file or files to be used with DECTPU. This built-in is preferred over the SET (DRM_HIERARCHY) built-in procedure. Using UID files to specify hierarchies makes it easy to translate the product into other languages and to modify an application's interface without recompiling all the code that implements the application.

SET (UID) is similar to SET (DRM_HIERARCHY). However, SET (UID) is the preferred form.

DECTPU uses the Motif Version 1.1-3 compatibility libraries and requires that UID files are produced using the 1.1-3 UIL compiler and not the version 1.2 UIL compiler provided with new releases of DECwindows. Procedures for using the 1.1-3 UIL compiler are described in the DECwindows release notes.

For more information about UID files, see the *VMS DECwindows Guide to Application Programming*.

Signaled Errors

TPU\$_ARGMISMATCH	ERROR	The data type of the indicated parameter is not supported by the SET (UID) built-in.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (UID) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (UID) built-in.
TPU\$_FAILURE_STATUS	ERROR	The Resource Manager returned an error status.
TPU\$_INVPARAM	ERROR	You specified an invalid parameter.
TPU\$_REQUIRESDECW	ERROR	Requires the DECTPU DECwindows screen updater.

Example

The following example designates the User Interface Definition (UID) file MYNODE\$DUA0:[SMITH]EXAMPLE.UID as a file to be used with DECTPU to create widgets needed by the layered application:

```
example_hierarchy := SET (UID, "mynode$dual0:[smith]example.uid");
```

SET (UNDEFINED_KEY)

Syntax

$$\text{SET (UNDEFINED_KEY, string1} \left[\left[\left\{ \begin{array}{l} \text{buffer} \\ \text{learn_sequence} \\ \text{program} \\ \text{range} \\ \text{string2} \end{array} \right\} \right] \right] \right)$$

Parameters

UNDEFINED_KEY

A keyword that specifies that SET is to determine the action taken when an undefined key is pressed.

string1

A string that specifies the key map list for which this procedure is called.

buffer

The buffer that contains DECTPU statements that specify the action to be taken if you press an undefined key. SET (UNDEFINED_KEY) compiles the statements in the buffer and stores the resulting program in the specified key map list.

learn_sequence

The learn sequence that specifies the action to be taken if you press an undefined key. The contents of a variable of type learn do not require compilation. SET (UNDEFINED_KEY) stores the learn sequence in the specified key map list.

program

The program that specifies the action to be taken if you press an undefined key. The contents of a variable of type program do not require compilation. SET (UNDEFINED_KEY) stores the program in the specified key map list.

range

The range that contains DECTPU statements that specify the action to be taken if you press an undefined key. SET (UNDEFINED_KEY) compiles the statements in the range and stores the resulting program in the specified key map list.

string2

The string that contains DECTPU statements that specify the action to be taken if you press an undefined key. SET (UNDEFINED_KEY) compiles the statements in the string and stores the resulting program in the specified key map list.

Description

The SET (UNDEFINED_KEY) procedure determines the action taken when an undefined key is pressed.

If the third parameter is not specified, DECTPU displays the message "key has no definition" when you press an undefined key.

Signaled Errors

TPU\$_NOKEYMAPLIST	WARNING	You attempted to access an undefined key map list.
TPU\$_TOOFEW	ERROR	SET (UNDEFINED_KEY) requires at least two parameters.
TPU\$_TOOMANY	ERROR	SET (UNDEFINED_KEY) accepts no more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_ARGMISMATCH	ERROR	The second parameter must be a string.

Example

The following example causes the default undefined key message to be displayed when an undefined key is entered:

```
IF GET_INFO ("tpu$key_map_list", "undefined_key") <> 0
THEN
SET (UNDEFINED_KEY, "tpu$key_map_list");
ENDIF;
```

SET (VIDEO)

Syntax

```
SET (VIDEO, window, {
    BLINK
    BOLD
    REVERSE
    UNDERLINE
    NONE
})
```

Parameters

VIDEO

The video attributes of a window.

window

The window in which a video attribute is being set.

BLINK

Causes the characters in the window to blink.

BOLD

Causes the characters in the window to be bolded.

REVERSE

Causes the characters in the window to be displayed in reverse video.

UNDERLINE

Causes the characters in the window to be underlined.

NONE

Applies no video attributes to the characters in the window. This is the default.

Description

The SET (VIDEO) procedure sets the video attributes, which are cumulative, for a window. The window assumes the video attribute of each video keyword that you use with SET (VIDEO) during an editing session. If you want to change the video attribute of a window, and you do not want the cumulative effect of previous attributes, use SET (VIDEO, window, NONE) before specifying the new attribute. SET (VIDEO, window, NONE) turns off all video attributes for a window.

The video attribute is applied during the next screen update. The screen manager repaints the window to apply the video attributes, even if the cumulative effect of your changes has been to leave the video attributes the same.

SET (VIDEO) does not affect the status line of a window. You can specify a video attribute for a status line either with CREATE_WINDOW or with the SET (STATUS_LINE) built-in procedure. When the window and the status line have different video attributes, you can use the status line to separate multiple windows on the screen or to highlight status information.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (VIDEO) requires three parameters.
TPU\$_TOOMANY	ERROR	SET (VIDEO) accepts no more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADKEY	ERROR	You specified an invalid keyword.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.

Example

The following example causes the current window to be displayed in reverse video and with underlining:

```
SET (VIDEO, CURRENT_WINDOW, REVERSE);
SET (VIDEO, CURRENT_WINDOW, UNDERLINE);
```

SET (WIDGET)

Syntax

```
SET (WIDGET, widget,
    { widget_args [ , widget_args... ] } )
```

Parameters

WIDGET

A keyword that directs DECTPU to set an attribute of a widget.

widget

The widget whose values you want to set.

widget_args

One or more pairs of resource names and resource values.

You can specify a pair as an array or as a pair of separate parameters. If you use an array, you index the array with a string that is the name of the resource you want to set. Resource names are case sensitive. The corresponding array element contains the value you want to assign to that resource. The array can contain any number of elements. If you use a pair of separate parameters, use the following format:

resource_name_string, resource_value

Arrays and string/value pairs may be interspersed. Each array index and its corresponding element value, or each string and its corresponding value, must be valid widget arguments for the class of widget whose resources you are setting.

Description

With the SET (WIDGET) procedure, you can assign values to various resources of a widget. SET (WIDGET) is functionally equivalent to the X Toolkit routine XtSetValues.

If you specify the name of a resource that the widget does not support, DECTPU signals the error TPU \$ _NONAMES.

For more information about specifying resources, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_ARGMISMATCH	ERROR	You specified a value whose data type is not supported.
TPU\$_NONAMES	WARNING	You specified an invalid widget resource name.
TPU\$_NORETURNVALUE	ERROR	SET (WIDGET) cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the SET (WIDGET) built-in only if you are using DECwindows DECTPU.

TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (WIDGET) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (WIDGET) builtin.

Example

The following example sets the "value" resource of the current window's scroll bar widget to 100. This causes the scroll bar slider to be displayed as far toward the bottom of the scroll bar widget as possible.

```
scroll_bar_widget := SET (SCROLL_BAR, CURRENT_WINDOW, VERTICAL, ON);
SET (WIDGET, scroll_bar_widget, "value", 100);
```

SET (WIDGET_CALLBACK)

Syntax

```
SET (WIDGET_CALLBACK, widget, {
    buffer
    learn_sequence
    program
    range
    string
}, closure)
```

Parameters

WIDGET_CALLBACK

A keyword that directs DECTPU to set the application-level widget callback.

widget

The widget whose callback you want to set.

buffer

The buffer that contains the application-level callback routine. This code is executed when the widget performs a callback to DECTPU.

learn_sequence

The learn sequence that specifies the application-level callback routine. This code is executed when the widget performs a callback to DECTPU.

program

The program that specifies the application-level callback routine. This code is executed when the widget performs a callback to DECTPU.

range

The range that contains the application-level callback routine. This code is executed when the widget performs a callback to DECTPU.

string

The string that contains the application-level callback routine. This code is executed when the widget performs a callback to DECTPU.

closure

A string or integer. DECTPU passes the value to the application when the widget performs a callback to DECTPU. DECwindows documentation refers to closures as tags. For more information about using closures, see the *Guide to the DEC Text Processing Utility*.

Description

The SET (WIDGET_CALLBACK) procedure specifies the DECTPU program or learn sequence to be called by DECTPU when a widget callback occurs for the widget.

Signaled Errors

TPU\$_ARGMISMATCH	ERROR	The data type of the indicated parameter is not supported by the SET (WIDGET_CALLBACK) built-in.
TPU\$_BADDELETE	ERROR	You are attempting to modify an integer, a keyword, or a string constant.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (WIDGET_CALLBACK) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (WIDGET_CALLBACK) built-in.
TPU\$_COMPILEFAIL	WARNING	Program compilation has been terminated because of a syntax error.
TPU\$_REQUIRESDECW	ERROR	You can use SET (WIDGET_CALLBACK) only if you are using DECwindows DECTPU.

Example

The following example designates the EVE procedure EVE\$SCROLL_DISPATCH as the callback routine for the widget *scroll_bar_widget* and assigns to the callback the closure value 'h':

```
SET (WIDGET_CALLBACK, scroll_bar_widget, "eve$scroll_dispatch", 'h');
```

SET (WIDGET_CALL_DATA)**Syntax**

```
SET (WIDGET_CALL_DATA, widget, reason_code, request_string, keyword [, request_string, k
```

Parameters**WIDGET_CALL_DATA**

A keyword that indicates that the SET built-in procedure is being used to control how DECTPU interprets information in a widget's callback data structure.

widget

The specific widget for which you want to determine how the callback data are interpreted.

reason_code

The identifier for the reason code with which the callback data structure is associated. For example, if you are using SET (WIDGET_CALL_DATA) to set the format of the callback structure associated with the Help reason code of the File Selection widget, and if your program defines the VAX reason code bindings as constants, you could refer to the Help reason code by using the constant XmCR_HELP for DECwindows.

request_string

One of the six valid strings that describes the data type of a given field in a callback data structure. The valid strings are as follows:

"char"	"compound_string"
"int"	"short"
"void"	"widget"

keyword

One of the four valid keywords that indicates the DECTPU data type to which DECTPU should convert the data in a given field of a callback data structure. The valid keywords are as follows:

INTEGER	STRING
UNSPECIFIED	WIDGET

Use the *request_string* parameter with the *keyword* parameter to inform DECTPU, for each field of the structure, what data type the field had originally and what DECTPU data type corresponds to the original data type. The valid keywords corresponding to each request string are as follows:

Request String	Associated Keyword(s)
"widget"	WIDGET or UNSPECIFIED
"short"	INTEGER or UNSPECIFIED
"int"	INTEGER or UNSPECIFIED
"compound_string"	STRING or UNSPECIFIED
"char"	STRING or UNSPECIFIED
"void"	UNSPECIFIED

Description

With the SET (WIDGET_CALL_DATA) procedure, you can create a template that tells DECTPU how to interpret the information in the fields of a widget's callback data structure. You use SET (WIDGET_CALL_DATA) to tell DECTPU what data type to assign to each field in a callback data structure. You must specify the widget and the callback reason whose data structure you want DECTPU to process. During a callback generated by the specified widget for the specified reason, DECTPU interprets the data in the callback structure according to the description you create.

In an application layered on DECTPU, you can get the interpreted callback data by using the GET_INFO (WIDGET, "callback_parameters") built-in procedure.

You can create a different template for each of the reason codes associated with a given widget. To do so, make a separate call to the SET (WIDGET_CALL_DATA) built-in procedure for each reason code. If you specify the same widget and reason code in more than one call, DECTPU uses the most recently specified format.

In all callback data structures defined by the DECwindows Toolkit, the first field is the reason code field and the second field is the event field. For more information on the fields in each widget's callback structures, see the *VMS DECwindows Toolkit Routines Reference Manual*. If your application creates or uses a new kind of widget, the widget's callback structure must follow this convention.

Do not specify any request string or keyword for the reason field. In almost all cases, you specify the event field with the request string "void" and the UNSPECIFIED keyword. Specify all subsequent fields, if the callback structure has such fields, up to and including the last field you want to specify. The VAX longword data type corresponds to the "int" request string and the INTEGER data type in DECTPU.

Although you can skip trailing fields, you cannot skip intermediate fields even if they are unimportant to your application. To direct DECTPU to ignore the information in a given field, use the request string "void" and the UNSPECIFIED keyword when specifying that field.

If you specify an invalid request string, DECTPU signals TPU\$_ILLREQUEST. If you specify an invalid keyword, DECTPU signals TPU\$_BADKEY. If you use valid parameters but assign the wrong data type to a field and if DECTPU detects the error, DECTPU assigns the data type UNSPECIFIED to that field while processing a callback.

An application should use this built-in only if it needs access to callback information other than the reason code. For more information on how SET (WIDGET_CALL_DATA) affects GET_INFO (WIDGET, "callback_parameters"), see the online Help topic GET_INFO(WIDGET).

Signaled Errors

TPU\$_BADKEY	WARNING	You specified an invalid keyword as a parameter.
TPU\$_REQUIRESDECW	ERROR	You can use this built-in only if you are using DECwindows DECTPU.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the SET (WIDGET-CALL_DATA) built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the SET (WIDGET-CALL_DATA) built-in.

Example

The following example begins by defining the constant APP\$_CRSINGLE to be the integer value 23, which is the integer associated with the reason "user selected a single item":

```
CONSTANT APP$_CRSINGLE := 23;
SET (WIDGET_CALL_DATA, initial_list_box, APP$_CRSINGLE,
"void", UNSPECIFIED, ! event
"compound_string", STRING, ! item
```

```
"int", INTEGER, ! item length  
"int", INTEGER); ! item number
```

The file `DECW$INCLUDE:XM.H` contains constants defined for callback reason codes. If you layer an application, the values you assign to the reason code constants must match the values in this file.

The next statement tells DECTPU how to interpret the fields of the callback data structure associated with a List Box widget assigned to the variable `"initial_list_box"`. The statement directs DECTPU to ignore the data in the Event field and to treat the data in the Item field as type `STRING` (in the Item Length field as type `INTEGER`) and the Item Number field as type `INTEGER`.

SET (WIDGET_CONTEXT_HELP)

Syntax

$$\text{SET (WIDGET_CONTEXT_HELP, widget, } \left. \begin{array}{c} \text{ON} \\ 1 \\ \text{OFF} \\ 0 \end{array} \right\})$$

Parameters

WIDGET_CONTEXT_HELP

A keyword that directs DECTPU to enter the DECwindows context-sensitive help mode in which the mouse pointer changes to a question mark until you click MB1 on a widget. The mouse pointer then reverts to the default pointer shape, and a help callback is called on the selected widget (or its parent if the selected widget has no help callback).

widget

Specifies the widget within which modal help interaction will be limited.

Applications usually specify the top-level widget returned from the `GET_INFO (SCREEN, "widget")` built-in procedure. A help callback occurs only when the mouse is clicked on the specified widget or any of its children widgets.

ON, 1

Confines the question mark pointer to the specified widget. If any children widgets have been moved outside the specified widget's boundaries, then the question mark pointer cannot be moved to those children unless this parameter is `OFF` or `0`.

OFF, 0

The `SET (WIDGET_CONTEXT_HELP)` procedure specifies the widget within which context-sensitive help interaction will occur until you click MB1 on a widget.

Example

The following example enters context-sensitive help mode; it does not restrict the question mark mouse pointer to the boundaries of the top-level DECTPU widget. This lets the pointer be moved to the application dialog boxes that have been moved outside the DECTPU boundaries.

```
SET (WIDGET_CONTEXT_HELP, GET_INFO (SCREEN, "widget"), OFF);
```

SET (WIDGET_RESOURCE_TYPES)

Syntax

```
SET (WIDGET_RESOURCE_TYPES, widget_data_type, widget_resource_type)
```

Parameters

WIDGET_RESOURCE_TYPES

A keyword that directs DECTPU to add new DECwindows widget resource types to the list of supported resource types. If you redefine a resource type to be of another data type, DECTPU signals the warning `TPU$_TYPEREDEFINED`. You cannot save resources in section files. DECTPU does not verify that your third parameter specifies the name of a valid widget resource or resource type. If you misspell the third parameter, you will get an error when you try to use that resource with a widget. For the current list of supported resource types, see the `GET_INFO (WIDGET, "widget_resource_types")` built-in procedure.

widget_data_type

A string that is the data type of the widget resource types given by the third parameter. It tells DECTPU how to process the resource types. DECTPU supports the following data types: "boolean", "callback", "char", "compound_string", "compound_string_table", "int", "short", "unsigned_short", and "unsigned_char".

widget_resource_type

A series of names of widget resources or resource types that are of the data type specified by the second parameter. You can specify an array of strings or a comma-separated list of strings. If you use an array, you index the array with any valid DECTPU array indexes. The array elements contain the names of either widget resources (for example, "dialogStyle"), or of widget resource types (for example, "Int"). If you use a comma-separated list of strings, the strings are the names of the widget resources or resource types.

Description

The `SET (WIDGET_RESOURCE_TYPES)` procedure specifies additional DECwindows widget resource types or resources that DECTPU should support. This built-in is valid only with DECwindows DECTPU.

Example

The following example enables your application to use the "dialogStyle" resource of the DECwindows `XmBulletinBoard` widget, which is an `unsigned_char` data type. "dialogStyle" is the name of the resource.

```
SET (WIDGET_RESOURCE_TYPES, "unsigned_char", "dialogStyle")
```

SET (WIDTH)

Syntax

SET (WIDTH, { window
ALL
SCREEN }, integer)

Parameters

WIDTH

A keyword that indicates that the horizontal dimension is being set.

window

The window for which you want to set or change the width.

ALL

A keyword that indicates that DECTPU should set the screen and all windows, visible and invisible, to the specified width.

SCREEN

A keyword that indicates that DECTPU should set the screen to the specified width without altering the size of any DECTPU windows. By default, EVE resizes the windows to match the width of the screen. You cannot set the screen to be narrower than the widest DECTPU window.

integer

The width of the window in columns. You can specify any integer between 1 and 255. In non-DECwindows DECTPU, a value of 80 causes DECTPU to repaint the screen and depict the text in normal-width font, if the text is not already so depicted. A value of 132 causes DECTPU to repaint the screen and depict the text in narrow font, if the text is not already so depicted. Other values do not affect the font. By default, the width of a window is the same as the physical width of the terminal when the window is created.

Description

The SET (WIDTH) procedure sets the width of a window or the DECTPU screen. When you call SET (WIDTH), DECTPU determines the width of the widest visible window. If this width has changed, the effect of SET (WIDTH) depends on your terminal.

If you are using DECTPU with a DECwindows terminal emulator, the terminal emulator is resized to match the width of the widest visible window. You can specify any width between 1 column and 255 columns.

If you are using DECTPU on a VT400-series, VT300-series, VT200-series, or VT100-series terminal, setting the width of a window causes a change only if the widest visible window is 80 or 132 columns wide. When the new width is one of these numbers, DECTPU causes the terminal to switch from 80-column mode to 132-column mode, or the reverse.

If you are using DECwindows DECTPU (that is, not in a DECterm window), changing the width of the screen does not affect the font of the characters displayed. (There are no 80-column or 132-column modes.)

If the width of the widest visible window has changed, DECTPU redisplay all windows.

By default, the width of a window is the same as the number of columns on the screen of the terminal on which you are running DECTPU. If you exceed the value set for the width of the window when entering text, DECTPU displays a diamond symbol in the rightmost column of the screen to indicate that there is text beyond the diamond symbol that is not visible on the screen. You cannot force DECTPU to use multiple lines to display a line that is longer than the width of a window.

Signaled Errors

TPU\$_TOOFEW	ERROR	SET (WIDTH) requires three parameters.
TPU\$_TOOMANY	ERROR	You specified more than three parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
TPU\$_BADVALUE	ERROR	Arguments are out of minimum or maximum bounds.

Examples

1. The following example sets the width of the main window to 132 columns and changes the font from standard to narrow:

```
SET (WIDTH, main_window, 132);
```

2. The following example sets the width of the screen and all windows, visible and invisible, to 40 columns. The statement does not affect the font.

```
SET (WIDTH, ALL, 40);
```

SHIFT

Syntax

```
[[integer2 := ]] SHIFT (window, integer1)
```

Parameters

window

The window that is shifted.

integer1

The signed integer that specifies how many columns to shift the window. A positive integer causes the window to shift to the right so that you can see text that was previously beyond the right edge of the window.

A negative integer causes the window to shift to the left so that you can see text that was previously beyond the left edge of the window. If the first character in the line of text is already in column 1, then using a negative integer has no effect.

If you specify 0 as the value, no shift takes place. Furthermore, 0 as the value does *not* cause the window to be repainted.

By default, windows are not shifted.

Return Value

An integer representing the amount by which the window has been shifted to the right.

Description

The SHIFT procedure, for a buffer whose lines are too long to be displayed all at once, moves the window so the unseen parts of the lines can be viewed. SHIFT can move the window right to display text past the right edge of the window, or left (for a previously shifted window). SHIFT optionally returns an integer that specifies the number of columns in the buffer lying to the left of the left edge of the shifted window.

Because SHIFT commands are cumulative during an editing session, this built-in procedure optionally returns a value in *integer2*. This positive integer represents the absolute shift value.

The shift applies to any buffer associated with the window that you specify. For example, if you shift a window and then map another buffer to that window, you see the text of the newly mapped buffer in the shifted position. You must specify another shift to return the window to its unshifted position.

If you specify an integer value of 0, the window is not shifted left. Furthermore, when you attempt to shift left, the window is not repainted; otherwise, SHIFT causes the entire window to be repainted. If you execute the SHIFT built-in procedure within a procedure, the screen is not updated to reflect the shift until the procedure has finished executing and control has returned to the screen manager. If you want the screen to reflect changes before the entire program is executed, you can force the immediate update of a window by adding an UPDATE statement to the procedure.

Signaled Errors

TPU\$_TOOFEW	ERROR	SHIFT requires two parameters.
TPU\$_TOOMANY	ERROR	You specified more than two parameters.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Examples

1. The following example shifts the window *user_window* five columns to the right:

```
SHIFT (user_window, +5)
```

2. The following example shifts the current window five columns to the left. (If the window was not previously shifted, this statement has no effect.)

```
SHIFT (CURRENT_WINDOW, -5)
```

SHOW

Syntax

```
SHOW ( ( BUFFER[S]  
        KEY_MAP_LIST[S]  
        KEY_MAP[S]  
        KEYWORDS  
        PROCEDURES  
        SCREEN  
        SUMMARY  
        VARIABLES  
        WINDOW[S]  
        buffer  
        string  
        window ) )
```

Parameters

BUFFER[[S]]

Displays information about all buffers available to the editor. **BUFFER** is a synonym for **BUFFERS**.

KEY_MAP_LIST[[S]]

Displays the names of all defined key map lists, their key maps, and the number of keys defined in each key map. **KEY_MAP_LIST** is a synonym for **KEY_MAP_LISTS**.

KEY_MAP[[S]]

Displays the names of all defined key maps. **KEY_MAP** is a synonym for **KEY_MAPS**.

KEYWORDS

Displays the contents of the internal keyword table.

PROCEDURES

Displays the names of all defined procedures.

SCREEN

Displays information about the terminal.

SUMMARY

Displays statistics about DECTPU, including the current version number.

VARIABLES

Displays the names of all defined variables.

WINDOW[[S]]

Displays information about all windows available to the editor. **WINDOW** is a synonym for **WINDOWS**.

buffer

Shows information about the buffer variable you specify.

string

Shows information about the string variable you specify.

window

Shows information about the window variable you specify.

Description

The SHOW procedure displays information about DECTPU data types and the current settings of attributes that you can apply to certain data types. See also the description of the GET_INFO built-in procedure.

DECTPU looks for the variable *show_buffer* and checks to see if it refers to a buffer. DECTPU also looks for the variable *info_window* and checks to see if it refers to a window. If these two items exist when you call the SHOW builtin procedure, DECTPU writes information to *show_buffer* and displays the information on the screen in the window called *info_window*.

You, or the interface you are using, must create the buffer variable *show_buffer* when you initialize the interface to ensure that the SHOW built-in procedure works as expected.

If you create a window called *info_window*, DECTPU associates *show_buffer* with *info_window* and maps this window to the screen when there is information to be displayed. You can optionally create a different window in which to display the information from *show_buffer*. In this case, you must associate *show_buffer* with the window that you create and map the window to the screen when there is information to be displayed.

Because this built-in procedure maps *info_window* to the screen, any interfaces layered on DECTPU should provide a mechanism for unmapping *info_window* and returning you to the editing position that was current before SHOW was invoked.

DECTPU always deletes the current text in the show buffer before inserting the new information.

Signaled Errors

TPU\$_NOSHOWBUF	WARNING	The requested information cannot be stored because the buffer variable <i>show_buffer</i> does not exist.
TPU\$_TOOMANY	ERROR	SHOW accepts only one parameter.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.

Examples

1. The following example displays on the screen a list of all the DECTPU built-in procedures and the user-written procedures that are available to your editing interface:

```
SHOW (PROCEDURES)
```

2. The following example displays the names of all defined key map lists, their key maps, and the number of keys defined in each key map:

```
SHOW (KEY_MAP_LISTS)
```

When you use the default interface, EVE, the DECTPU SHOW (KEY_MAP_LISTS) command displays information similar to the following:

```
Defined key map lists:
TPU$KEY_MAP_LIST contains the following key maps:
    EVE$USER_KEYS           (0 keys defined)
    EVE$VT200_KEYS         (14 keys defined)
    EVE$STANDARD_KEYS      (29 keys defined)
Total of 1 key map list defined
```

SLEEP

Syntax

```
SLEEP ( { integer }
        { string  } )
```

Parameters

integer

The number of seconds to sleep.

string

An absolute or a delta time string indicating how long to sleep. The format of an absolute time string is "dd-mmm-yyyy hh:mm:ss.cc".

In the string above, values have the following meanings:

dd is the day of the month (1-31)

mmmm is the month (JAN, FEB, ..., DEC)

yyyy is the year (000-9999)

hh is the hour (0-23)

mm is the minute (0-59)

s is the second (0-59)

cc is the hundredth of a second (0-99)

For the format of a delta time, see the SET (GLOBAL_SELECT_TIME) built-in procedure.

Description

The SLEEP procedure causes DECTPU to pause for the amount of time you specify or until input is available. This built-in is useful if you want to display something for only a short period of time. In the character-cell interface, SLEEP ends immediately if input comes from the terminal before the time interval expires. In the DECwindows interface, SLEEP is not interrupted by terminal input.

Signaled Errors

TPU\$_TOOFEW	ERROR	SLEEP requires one argument.
TPU\$_TOOMANY	ERROR	SLEEP accepts only one argument.
TPU\$_ARGMISMATCH	ERROR	The argument to SLEEP must be an integer or string.
TPU\$_INVTIME	ERROR	The argument to SLEEP was an invalid sleep time.

Examples

1. The following example suspends DECTPU for 2 seconds:

```
SLEEP (2);
```

2. The following example puts a string into the message buffer. The procedure displays the string in reverse video for a third of a second. After a third of a second, reverse video is canceled and the string is displayed as usual.

```
PROCEDURE user_emphasize_message (user_message)
LOCAL here,
start_mark,
the_range;
here := MARK (NONE);
POSITION (END_OF (message_buffer));
COPY_TEXT (user_message);
MOVE_HORIZONTAL (-CURRENT_OFFSET);
start_mark := MARK (NONE);
MOVE_VERTICAL (1);
MOVE_HORIZONTAL (-1);
the_range := CREATE_RANGE (start_mark, MARK (NONE), REVERSE);
UPDATE (message_window);
SLEEP ("0 00:00:00.33");
the_range := 0;
UPDATE (message_window);
POSITION (here);
ENDPROCEDURE;
```

SPAN

Syntax

$$\text{pattern} := \text{SPAN} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \llbracket, \left\{ \begin{array}{l} \text{FORWARD} \\ \text{REVERSE} \end{array} \right\} \rrbracket \right)$$

Parameters

buffer

An expression that evaluates to a buffer. SPAN matches only those characters that appear in the buffer.

range

An expression that evaluates to a range. SPAN matches only those characters that appear in the range.

string

An expression that evaluates to a string. SPAN matches only those characters that appear in the string.

FORWARD

A keyword that directs DECTPU to match characters in the forward direction. This is the default.

REVERSE

A keyword that directs DECTPU to match characters as follows: first, match characters in the forward direction until DECTPU finds a character that is not a member of the set of characters in the specified buffer, range, or string.

Next, return to the first character matched and start matching characters in the reverse direction until DECTPU finds a character that is not in the specified buffer, range, or string.

You can specify REVERSE only if you are using SPAN in the first element of a pattern being used in a reverse search. In all other contexts, specifying REVERSE has no effect.

The behavior enabled by REVERSE allows an alternate form of reverse search.

By default, a reverse search stops as soon as a successful match occurs, even if there might have been a longer successful match in the reverse direction. By specifying REVERSE, you direct DECTPU not to stop matching in either direction until it has matched as many characters as possible.

Return Value

A pattern that matches a sequence of characters, each of which appears in the buffer, range, or string used in the parameter to SPAN.

Description

The SPAN procedure returns a pattern that matches a string of characters, each of which appears in the buffer, range, or string used as its parameter. SPAN matches as many characters as possible, stopping only if it finds a character not present in its parameter or if it reaches the end of a line. If SPAN is part of a larger pattern, SPAN does not match a character if doing so prevents the rest of the pattern from matching.

SPAN does not cross line boundaries. To match a string of characters that may cross one or more line boundaries, use SPANL.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	SPAN must appear on the right-hand side of an assignment statement.
--------------------	-------	---

TPU\$_TOOFEW	ERROR	SPAN requires at least one argument.
TPU\$_TOOMANY	ERROR	SPAN accepts no more than one argument.
TPU\$_ARGMISMATCH	ERROR	Argument passed to SPAN is of the wrong type.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of SPAN.

Examples

1. The following example creates a pattern that matches any word of two or more letters ending in the letter s. Given the word *dogs*, the SPAN part of the pattern matches *dog*. It does not also match the s because this would prevent the rest of the pattern from matching.

```
pat1 := span ("abcdefghijklmnopqrstuvwxyz") + "s";
```

2. The following example removes all lines that contain only the letters x, y, and z:

```
PROCEDURE user_remove_xyz
LOCAL pat1,
xyz_line;
pat1 := LINE_BEGIN + SPAN ("xyz") + LINE_END;
LOOP
xyz_line := SEARCH_QUIETLY (pat1, FORWARD);
EXITIF xyz_line = 0;
POSITION (xyz_line);
ERASE_LINE;
ENDLOOP;
ENDPROCEDURE;
```

SPANL

Syntax

$$\text{pattern} := \text{SPANL} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\} \llbracket, \left\{ \begin{array}{l} \text{FORWARD} \\ \text{REVERSE} \end{array} \right\} \rrbracket \right)$$

Parameters

buffer

An expression that evaluates to a buffer. SPANL matches only those characters that appear in the buffer.

range

An expression that evaluates to a range. SPANL matches only those characters that appear in the range.

string

An expression that evaluates to a string. SPANL matches only those characters that appear in the string.

FORWARD

A keyword that directs DECTPU to match characters in the forward direction. This is the default.

REVERSE

A keyword that directs DECTPU to match characters as follows: first, match characters in the forward direction until DECTPU finds a character that is not a member of the set of characters in the specified buffer, range, or string. Next, return to the first character matched and start matching characters in the reverse direction until DECTPU finds a character that is not in the specified buffer, range, or string.

You can specify REVERSE only if you are using SPANL in the first element of a pattern being used in a reverse search. In all other contexts, specifying REVERSE has no effect.

The behavior enabled by REVERSE allows an alternate form of reverse search. By default, a reverse search stops as soon as a successful match occurs, even if there might be a longer successful match in the reverse direction. By specifying REVERSE, you direct DECTPU not to stop matching in either direction until it has matched as many characters as possible.

Return Value

A pattern that matches a sequence of characters and line breaks.

Description

The SPANL procedure returns a pattern that matches a string of characters and line breaks, each of which appears in the buffer, range, or string used as its parameter. The pattern matches as many characters and line breaks as possible. SPANL is similar to SPAN; however, unlike SPAN, SPANL does not stop matching when it reaches the end of a line. It successfully matches the end of the line and continues trying to match characters on the next line. If SPANL is part of a larger pattern, it does not match a character or line boundary if doing so prevents the rest of the pattern from matching.

Usually, SPANL must match at least one character. However, if the argument to SPANL contains no characters, then SPANL matches one or more line breaks.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	SPANL must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	SPANL requires at least one argument.
TPU\$_TOOMANY	ERROR	SPANL accepts no more than one argument.
TPU\$_ARGMISMATCH	ERROR	Argument passed to SPANL is of the wrong type.
TPU\$_CONTROLC	ERROR	You pressed Ctrl/C during the execution of SPANL.

Examples

1. The following example stores a pattern in *pat1* that matches the longest sequence of blank characters, starting at the editing point and continuing until the search encounters a nonmatching character or the end of the buffer, range, or string:

```
pat1 := SPANL (" ")
```

2. The following example removes all parts of a document that contain only numbers:

```
PROCEDURE user_remove_numbers
  LOCAL pat1,
         number_region;
  pat1 := SPANL ("0123456789");
  POSITION (BEGINNING_OF (CURRENT_BUFFER));
  LOOP
    number_region := SEARCH_QUIETLY (pat1, FORWARD);
    EXITIF number_region = 0;
    ERASE (number_region);
    POSITION (number_region);
  ENDLOOP; POSITION (BEGINNING_OF (CURRENT_BUFFER));
ENDPROCEDURE;
```

3. The following example positions you to the next occurrence of the text *Mark Twain*, where *Mark* and *Twain* may be separated by any number of spaces, tabs, or line breaks:

```
PROCEDURE user_find_mark_twain
  LOCAL pat1,
         mark_twain;
  pat1 := "Mark" + (SPANL (" " + ASCII(9)) | SPANL (""))
         + "Twain";
  mark_twain := SEARCH_QUIETLY (pat1, FORWARD, NOEXACT);
  IF mark_twain = 0
  THEN
    MESSAGE ("String not found");
  ELSE
    POSITION (mark_twain);
  ENDIF;
ENDPROCEDURE;
```

SPAWN

Syntax

$$\text{SPAWN } \llbracket \text{string}, \left[\left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\} \right] \rrbracket$$

Parameters

string

The command string that you want to be executed in the context of the subprocess that is created with SPAWN.

ON, 1

A keyword that indicates that control is to be returned to DECTPU after the command has been executed. This is the default unless the value specified for the first parameter is the null string.

OFF, 0

A keyword that indicates that you are to be prompted for additional operating system commands after the specified command has been executed. If the value specified for the first parameter is the null string, the default value for the second parameter is OFF.

Description

The SPAWN procedure creates a subprocess that runs the command line interpreter. SPAWN suspends your DECTPU process and starts a subprocess. This built-in procedure is especially useful for running programs and utilities that take control of the screen (these programs cannot be run in a subprocess created with the CREATE_PROCESS) built-in procedure. See the *Guide to the DEC Text Processing Utility* for a list of restrictions for subprocesses.

If you are using DCL, you can return to your DECTPU session after finishing in a subprocess by using either the DCL ATTACH command or the DCL LOGOUT command. If you use the DCL ATTACH command, the subprocess is available for future use. If you use the DCL LOGOUT command, the subprocess is deleted. When you return to the DECTPU session, the screen is repainted.

If you specify a DCL command as the parameter for SPAWN, the command is executed after the subprocess is created. When the command completes, the subprocess terminates and control is returned to the DECTPU process. If you want to remain at the system prompt, add the OFF keyword as the second parameter.

Subprocesses created with SPAWN give you direct access to the command line interpreter. These subprocesses are different from the subprocesses created with the CREATE_PROCESS built-in procedure. CREATE_PROCESS creates a subprocess within a DECTPU session, and all of the output from the subprocess goes into a buffer.

SPAWN is not a valid built-in in DECwindows DECTPU. However, if you are running non-DECwindows DECTPU in a DECwindows terminal emulator, SPAWN works as described in this section.

SPAWN fails if you are running in an OpenVMS account with the CAPTIVE flag set in the authorization file.

See the description of the ATTACH built-in procedure for information on moving control from one subprocess to another. See the *VSI OpenVMS DCL Dictionary* for more information on the characteristics of an OpenVMS spawned subprocess.

If the current buffer is mapped to a visible window, the SPAWN built-in procedure causes the screen manager to synchronize the editing point (which is a buffer location) with the cursor position (which is a window location). This may result in the insertion of padding spaces or lines into the buffer if the cursor position is before the beginning of a line, in the middle of a tab, beyond the end of a line, or after the last line in the buffer.

Signaled Errors

TPU\$_TOOMANY	ERROR	Too many arguments passed to the SPAWN built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the SPAWN built-in.
TPU\$_REQUIRETERM	ERROR	SPAWN is not a valid built-in in DECwindows DECTPU.

TPU\$_UNKKEYWORD	ERROR	An unknown keyword was used as an argument. Only ON or OFF is allowed.
TPU\$_BADKEY	ERROR	An unknown keyword was used as an argument. Only ON or OFF is allowed.
TPU\$_CAPTIVE	WARNING	Unable to create a subprocess in a captive account.
TPU\$_CREATEFAIL	WARNING	Unable to activate the subprocess.

Example

The following example spawns a subprocess and puts your DECTPU process on hold. The DCL command is executed in the subprocess to show the translation of the logical name SYS\$LOGIN, and you are left at the DCL prompt. After completing work in the subprocess, you can return to your DECTPU session by using the DCL ATTACH command or the DCL LOGOUT command.

```
SPAWN ("SHOW LOGICAL SYS$LOGIN", OFF)
```

SPLIT_LINE

Syntax

SPLIT_LINE

Parameters

None.

Description

The SPLIT_LINE procedure breaks the current line before the editing point and creates two lines. The relative screen position of the line you are splitting may change as a result of this procedure. The first line contains any characters to the left of the editing point. The second line contains the rest of the characters. The new line that is created is inserted directly after the former current line.

When you use SPLIT_LINE, the editing point remains on the same character, but that character is now the first character on the newly created line.

If the editing point is not the first character in the line being split, the left margin of the old line is not changed. The new line, which contains the editing point and the characters to the right of the editing point, takes the buffer's left margin as its own left margin.

If the editing point is the first character of a line, SPLIT_LINE creates a blank line where the original line was. The left margin of this blank line is the buffer's left margin. SPLIT_LINE moves the original line, including the editing point, to the line below the blank line. If the original line had a left margin different from the buffer's current left margin, SPLIT_LINE preserves the original line's left margin when it moves the line down.

If the editing point is on a blank line, SPLIT_LINE creates a new blank line below the existing line. The editing point moves to the new blank line. The new blank line receives the buffer's left margin value. If the original blank line had a left margin different from the buffer's current left margin, the original blank line retains its margin.

Using `SPLIT_LINE` may cause DECTPU to insert padding spaces or blank lines in the buffer. `SPLIT_LINE` causes the screen manager to place the editing point at the cursor position if the current buffer is mapped to a visible window. For more information on the distinction between the cursor position and the editing point, see Appendix C.

If the cursor is not located on a character (that is, if the cursor is before the beginning of a line, beyond the end of a line, in the middle of a tab, or below the end of the buffer), DECTPU inserts padding spaces or blank lines into the buffer to fill the space between the cursor position and the nearest text.

Signaled Errors

<code>TPU\$_NOCURRENTBUF</code>	WARNING	You are not positioned in a buffer.
<code>TPU\$_NOCACHE</code>	ERROR	There is not enough memory to allocate a new cache.
<code>TPU\$_NOTMODIFIABLE</code>	WARNING	You cannot modify an unmodifiable buffer.
<code>TPU\$_TOOMANY</code>	ERROR	<code>SPLIT_LINE</code> takes no arguments.

Example

The following example splits a line at the editing point. If the editing point is row 1, column 1, the procedure causes the screen to scroll.

```
PROCEDURE user_split_line
  LOCAL old_position,
        new_position;
  SPLIT_LINE;
  IF (CURRENT_ROW = 1) AND (CURRENT_COLUMN = 1)
  THEN
    old_position := MARK (NONE);
    SCROLL (CURRENT_WINDOW, -1);
    new_position := MARK (NONE);
    !Make sure we scrolled before doing CURSOR_VERTICAL
    IF new_position <> old_position
    THEN
      CURSOR_VERTICAL (1);
    ENDIF;
  ENDIF;
ENDPROCEDURE;
```

STR

Syntax

$$\text{string3} := \text{STR} \left\{ \begin{array}{l} \text{integer1} \text{ [,integer2] } \\ \text{keyword} \\ \text{string} \end{array} \right\}$$

$$\text{string3} := \text{STR} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \end{array} \right\} \text{ [, string2] } \left[\begin{array}{l} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right] \right)$$

Parameters

integer1

The integer you want converted to a string.

integer2

The radix (base) you want DECTPU to use when converting the first integer parameter to a string. The default radix is 10. The other allowable values are 8 and 16.

keyword

The keyword whose string representation you want.

string1

Any string. STR now accepts a parameter of type string, so you need not check the type of the parameter you supply to the built-in.

buffer

The buffer whose contents you want returned as a string.

range

The range whose contents you want returned as a string.

string2

A string that specifies how you want line ends represented. The default is the null string. You can use *string2* only if you specify a range or buffer as the first parameter. If you want to specify the ON or OFF keyword but do not want to specify *string2*, you must use a comma before the keyword as a placeholder, as follows:

```
new_string := STR (old_buffer, , ON);
```

ON, 1

A keyword that directs DECTPU to insert spaces, thus preserving the white space created by the left margin of each record in the specified buffer or range. If you specify a buffer or range with a left margin greater than 1, the ON keyword directs DECTPU to insert a corresponding number of spaces after the line ends in the resulting string. For example, if the left margin of the specified lines is 10 and you use the ON keyword, DECTPU inserts 9 spaces after each line end in the resulting string. DECTPU does not insert any spaces after line beginnings that do not contain characters. If the first line of a buffer or range starts at the left margin, DECTPU inserts spaces before the text in the first line.

You can use this keyword only if you specify a buffer or range as a parameter.

OFF, 0

A keyword that directs DECTPU to ignore the left margin setting of the records in the specified buffer or range. This is the default. For example, if the left margin of the specified lines is 10 and

you use the OFF keyword, DECTPU does not insert any spaces after the line ends in the resulting string.

You can use this keyword only if you specify a buffer or range as a parameter.

Return Value

A string that is the equivalent of the parameter you specify.

Description

The STR procedure returns a string equivalent for an integer, a keyword, a string, or the contents of a range or buffer. If you use the first format, STR returns a string representation of an integer or a keyword. You can then use the variable that contains the returned string in operations that require string data types. For another method of generating a string representation of an integer, see the description of the FAO built-in procedure.

If you use the second format, STR returns a string equivalent for any string or for the contents of a range or buffer.

Signaled Errors

TPU\$_TRUNCATE	WARNING	You specified a buffer or range so large that converting it would exceed the maximum length for a string. DECTPU has truncated characters from the returned string.
TPU\$_NEEDTOASSIGN	ERROR	STR must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	STR requires at least one argument.
TPU\$_TOOMANY	ERROR	STR accepts only two arguments.
TPU\$_INVPARAM	ERROR	The argument to STR must be an integer, buffer, string, or range.
TPU\$_BADVALUE	ERROR	You specified a value other than 8, 10, or 16 for the radix parameter.

Examples

1. The following example creates a string that uses the text in the select range. Line breaks are marked with the string *CRLF*. The white space created by the margin is preserved by inserting spaces after the line breaks.

```
return_string := STR (SELECT_RANGE, "<CRLF>", ON);
```

2. The following example uses the STR built-in procedure to convert the integer variables *v1* and *v2* to strings so that your row and column position can be displayed in the message area:

```
PROCEDURE user_display_position
    v1 := GET_INFO (second_window, "current_column");
    MESSAGE ("Column: " + STR (v1));
    v2 := GET_INFO (second_window, "current_row");
    MESSAGE ("Row: " + STR (v2));
```

```
ENDPROCEDURE;
```

3. The following example forms a string that uses the text in the range "this_range":

```
this_string := STR (this_range, "EOL")
```

In the string, each end-of-line is represented by the letters EOL. For example, suppose the text in "this_range" is as follows:

```
Sufficient unto the day  
are the cares thereof
```

Given the text in "this_range", "this_string" contains the following:

```
Sufficient unto the dayEOLare the cares thereof
```

If "this_range" extends to the character after the "f" in "thereof", "this_string" contains the following:

```
Sufficient unto the dayEOLare the cares thereofEOL
```

SUBSTR

Syntax

$$\text{string2} := \text{SUBSTR} \left(\left\{ \begin{array}{l} \text{buffer} \\ \text{range} \\ \text{string} \end{array} \right\}, \text{integer1} [, \text{integer2}] \right)$$

Parameters

buffer

The buffer that contains the substring.

range

The range that contains the substring.

string

The string that contains the substring.

integer1

The character position at which the substring starts. The first character position is 1.

integer2

The number of characters to include in the substring. If you do not specify this parameter, DECTPU sets the returned string's end point to the end of the first parameter.

Return Value

A string that represents a substring of a string or range.

Description

The SUBSTR procedure returns a string that represents a substring of a buffer, range, or string. If you specify a larger number of characters for *integer2* than are present in the substring, only the characters present are returned in *string2*. No error is signaled.

Signaled Errors

TPU\$_NEEDTOASSIGN	ERROR	SUBSTR must appear on the right-hand side of an assignment statement.
TPU\$_TOOFEW	ERROR	SUBSTR requires three arguments.
TPU\$_TOOMANY	ERROR	SUBSTR accepts only three arguments.
TPU\$_INVPARAM	ERROR	One of the arguments to SUBSTR is of the wrong type.
TPU\$_ARGMISMATCH	ERROR	One of the arguments to SUBSTR is of the wrong type.
TPU\$_TRUNCATE	WARNING	You specified a buffer or range so large that returning the requested substring would exceed the maximum length for a string. DECTPU has truncated characters from the returned string.

Examples

1. The following example returns the string "com" in the variable *file_type*. The substring starts at the seventh character position (c) and contains three characters (com). If you use a larger number for *integer2*, for example, 7, the variable *file_type* still contains "com" and no error is signaled.

```
file_type := SUBSTR ("login.com", 7, 3)
```

2. The following example capitalizes the first character in a string. It does not affect any other characters in the string. It makes use of the fact that SUBSTR returns a null string if the second parameter points past the end of the string.

```
! Capitalize the first letter in a string.
!
PROCEDURE user_initial_cap (my_string)
LOCAL
first_part_of_string,
rest_of_string,
first_letter,
cur_loc;
cur_loc := 1;
first_part_of_string := "";
rest_of_string := "";
LOOP
first_letter := SUBSTR (my_string, cur_loc, 1);
EXITIF first_letter = "";
EXITIF (first_letter >= "a") AND (first_letter <= "z");
EXITIF (first_letter >= "A") AND (first_letter <= "Z");
cur_loc := cur_loc + 1;
ENDLOOP;
CHANGE_CASE (first_letter, UPPER);
first_part_of_string := SUBSTR (my_string, 1, cur_loc - 1);
```

```

rest_of_string := SUBSTR (my_string, cur_loc + 1,
LENGTH (my_string) - cur_loc);
my_string := first_part_of_string + first_letter
+ rest_of_string;
ENDPROCEDURE;

```

TRANSLATE

Syntax

$$\left\{ \begin{array}{l} \text{buffer1} \\ \text{range1} \\ \text{string1} \end{array} \right\} := \text{TRANSLATE} \left(\left\{ \begin{array}{l} \text{buffer2} \\ \text{range2} \\ \text{string2} \end{array} \right\} \text{string3}, \text{string4} \right) \left[\begin{array}{l} \text{IN_PLACE} \\ \text{NOT_IN_PLACE} \end{array} \right]$$

Parameters

buffer2

A buffer in which one or more characters are to be replaced. You cannot use the NOT_IN_PLACE keyword if you specify a buffer for the first parameter.

range2

A range in which one or more characters are to be replaced. You cannot use the NOT_IN_PLACE keyword if you specify a range for the first parameter.

string2

A string in which one or more characters are to be replaced. If a return value is specified, the substitution is performed in the returned string. If you specify IN_PLACE for the third parameter, TRANSLATE makes the specified change to the string specified in the first parameter. If string2 is a constant, IN_PLACE has no effect.

string3

The string of replacement characters.

string4

The literal characters within the text specified by parameter 1 that are to be replaced.

IN_PLACE

A keyword that directs DECTPU to make the indicated change in the buffer, range, or string specified. This is the default.

NOT_IN_PLACE

A keyword that directs DECTPU to leave the specified string unchanged and return a string that is the result of the specified translation. You cannot use NOT_IN_PLACE if the first parameter is specified as a range or buffer. To use NOT_IN_PLACE, you must specify a return value for TRANSLATE.

Return Values

buffer1

A variable of type `buffer` that points to the buffer that contains the modified text, if you specify a buffer for the first parameter. The variable `returned_buffer` points to the same buffer pointed to by the buffer variable specified as the first parameter.

range1

A range that contains the modified text, if you specify a range for the first parameter. The returned range spans the same text as the range specified as a parameter, but they are two separate ranges. If you subsequently change or delete one of the ranges, this has no effect on the other range.

string1

A string that contains the modified text, when you specify a string for the first parameter. `TRANSLATE` can return a string even if you specify `IN_PLACE`.

Description

The `TRANSLATE` procedure substitutes one set of specified characters for another set. `TRANSLATE` returns a value for the translated range or buffer or for the string representation of the translated text. `TRANSLATE` is based on the Run-Time Library (RTL) routine `STR$TRANSLATE`. For complete information on `STR$TRANSLATE`, see the *OpenVMS RTL String Manipulation (STR\$) Manual*.

`TRANSLATE` searches the text specified by the first parameter for the characters contained in the third parameter. When `DECTPU` finds the sequence specified by *string3*, `DECTPU` substitutes the first character in *string2* for the first character in *string3*, and so forth.

If the translate string, *string2*, is shorter than the match string, *string3*, and the number of matched character positions is greater than the number of character positions in the translate string, the translation character is a space.

The `IN_PLACE` and `NOT_IN_PLACE` keywords specify whether the source is to be changed. `IN_PLACE` means that the source is modified; `NOT_IN_PLACE` indicates that the source is not changed.

Signaled Errors

<code>TPU\$_TOOFEW</code>	ERROR	<code>TRANSLATE</code> requires three arguments.
<code>TPU\$_TOOMANY</code>	ERROR	<code>TRANSLATE</code> accepts no more than three arguments.
<code>TPU\$_ARGMISMATCH</code>	ERROR	One of your arguments to <code>TRANSLATE</code> is of the wrong data type.
<code>TPU\$_INVPARAM</code>	ERROR	One of your arguments to <code>TRANSLATE</code> is of the wrong data type.
<code>TPU\$_NOTMODIFIABLE</code>	WARNING	You cannot translate text in an unmodifiable buffer.
<code>TPU\$_CONTROLC</code>	ERROR	You pressed <code>Ctrl/C</code> during the execution of <code>TRANSLATE</code> .

Examples

1. The following example replaces any lowercase "I" in *second_buffer* with an uppercase "I":

```
TRANSLATE (second_buffer, "I","i")
```

2. The following example strips the eighth bit from all characters in the current buffer. You can use this kind of procedure for reading files from systems like TOPS-20 (on which the eighth bit is set) without using the DEC Multinational Character Set.

```
PROCEDURE user_strip_eighth
  LOCAL i,          ! Loop counter
        seven,     ! ASCII (0) through ASCII (127)
        eight;     ! ASCII (128) through ASCII (255)
! Build translate strings
  seven := "";
  eight := "";
  i := 0;
  LOOP
    seven := seven + ASCII (i);
    eight := eight + ASCII (i + 128);
    i := i + 1;
    EXITIF i = 128;
  ENDLOOP;
  TRANSLATE (CURRENT_BUFFER, seven, eight);
ENDPROCEDURE;
```

UNANCHOR

Syntax

UNANCHOR

Parameters

None.

Description

The UNANCHOR procedure specifies that the next pattern element may match anywhere after the previous pattern element. Usually, when a pattern contains several concatenated or linked pattern elements, the pattern matches only when the text that matches one particular pattern element immediately follows the text that matches the previous pattern element. If UNANCHOR appears between two pattern elements, the text that matches the second pattern element may appear anywhere after the text that matches the first pattern element.

Although UNANCHOR behaves much like a built-in, it is actually a keyword.

For more information on patterns or pattern searching, see the *Guide to the DEC Text Processing Utility*.

Signaled Errors

UNANCHOR is a keyword and has no completion codes.

Examples

1. The following example creates a pattern that matches any text beginning with the letter a and ending with the digits 123. Any amount of text may appear between the a and the 123.

```
pat1 := "a" + UNANCHOR + "123"
```

2. The following example removes all parenthesized text from a buffer. The text may span several lines. It does not handle multiple levels of parentheses.

```
PROCEDURE user_remove_paren_text (paren_buffer)
  LOCAL pat1,
         paren_text,
         searched_text;
pat1 := "(" + UNANCHOR + ")";
searched_text := paren_buffer;
LOOP
  paren_text := SEARCH_QUIETLY (pat1, FORWARD, EXACT,
                               searched_text);

  EXITIF paren_text = 0;
  ERASE (paren_text);
  searched_text := CREATE_RANGE (END_OF (paren_text),
                                END_OF (paren_buffer), NONE);
ENDLOOP;
ENDPROCEDURE;
```

UNDEFINE_KEY

Syntax

$$\text{UNDEFINE_KEY (keyword } \left[\left[\left\{ \begin{array}{l} \text{key-map-list-name} \\ \text{key-map-name} \end{array} \right\} \right] \right])$$

Parameters

keyword

The name of a key or key combination that you can define. See the *Guide to the DEC Text Processing Utility* for a list of the valid DECTPU key names.

key-map-list-name

Specifies a key map list in which the key is defined. The first definition of the key in the key maps that make up the key map list is deleted. If neither a key map nor a key map list is specified, the key map list bound to the current buffer is used.

key-map-name

Specifies a key map in which the key is defined. The first definition of the key in the key map is deleted. If neither a key map nor a key map list is specified, the key map list bound to the current buffer is used.

Description

The UNDEFINE_KEY procedure removes the current binding from the key that you specify. After you use UNDEFINE_KEY, the key you specify is no longer defined. DECTPU does not save any previous definitions that you may have associated with the key. However, any definitions of the specified key in key maps or key map lists other than the ones you specified are not removed.

DECTPU writes a message to the message buffer telling you that the key is undefined if you try to use it after you have undefined it.

Signaled Errors

TPU\$_NODEFINITION	WARNING	There is no definition for this key.
TPU\$_NOTDEFINABLE	WARNING	First argument is not a valid reference to a key.
TPU\$_NOKEYMAP	WARNING	Second argument is not a defined key map.
TPU\$_NOKEYMAPLIST	WARNING	Second argument is not a defined key map list.
TPU\$_KEYMAPNTFND	ERROR	The key map listed in the second argument is not found.
TPU\$_EMPTYKMLIST	WARNING	The key map list specified in the second argument contains no key maps.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the UNDEFINE_KEY built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the UNDEFINE_KEY built-in.
TPU\$_INVPARAM	ERROR	Wrong type of data sent to the UNDEFINE_KEY built-in.

Examples

1. The following example undefines a key. You can use this kind of procedure for keypad initialization procedures.

```

! Parameters:
!
!      Name           Function           Input or Output?
!      ----           -
!      which_key      Keyword for key to clear           input
PROCEDURE user_clear_key (which_key)
  IF (LOOKUP_KEY (which_key, PROGRAM) <> 0)
  THEN
    UNDEFINE_KEY (which_key);
  ELSE
    MESSAGE ("Key not defined");
  ENDIF;
ENDPROCEDURE;

```

2. The following example deletes all of the key definitions in the key map TPU\$KEY_MAP:

```

PROCEDURE delete_all_definitions
  LOCAL key;
  LOOP

```

```

key := GET_INFO (DEFINED_KEY, "first", "tpu$key_map");
EXITIF key = 0;
UNDEFINE_KEY (key, "tpu$key_map");
ENDLOOP;
ENDPROCEDURE;

```

UNMANAGE_WIDGET

Syntax

UNMANAGE_WIDGET (widget [, widget...])

Parameter

widget

The widget to be unmanaged.

Description

The UNMANAGE_WIDGET procedure makes the specified widget and all of its children invisible. If you want to unmanage several widgets that are children of the same parent, but you do not want to unmanage the parent, include all the children in a single call to UNMANAGE_WIDGET. Unmanaging several widgets at once is more efficient than unmanaging one widget at a time.

Signaled Errors

TPU\$_INVPARAM	ERROR	You specified a parameter of the wrong type.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the UNMANAGE_WIDGET built-in.
TPU\$_NORETURNVALUE	ERROR	UNMANAGE_WIDGET cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the UNMANAGE_WIDGET built-in only if you are using DECwindows DECTPU.
TPU\$_WIDMISMATCH	ERROR	You have specified a widget whose class is not supported.

Example

The following example removes EVE's FIND dialog box from the screen:

```
UNMANAGE_WIDGET (eve$x_find_dialog);
```

UNMAP

Syntax

```
UNMAP ( { widget
        }
        { window
        } )
```

Parameters

widget

The widget you want to make invisible.

window

The window you want to remove from the screen.

Description

The UNMAP procedure disassociates a window from its buffer and removes the window or widget from the screen. If you unmap the current window, DECTPU tries to move the cursor position to the window that was most recently the current window. The window in which DECTPU positions the cursor becomes the current window, and the buffer that is associated with this window becomes the current buffer.

The screen area of the window you unmap is either erased or returned to any windows that were occluded by the window you unmapped. DECTPU returns lines to adjacent windows if the size of the windows requires the lines that were used for the window you unmap. The size of a window is determined by the values you specified for the CREATE_WINDOW built-in procedure when you created the window, or by the values you specified for the ADJUST_WINDOW built-in procedure if you changed the size of the window. If adjacent windows do not require the lines that were used by the window you unmap, the lines that the window occupied on the screen remain blank.

The window that you unmap is not deleted from the list of available windows. You can cause the window to appear on the screen again with MAP. UNMAP does not have any effect on the buffer that was associated with the window being unmapped. Unmapping a widget does not delete the widget. Future MAP operations will make the widget visible again.

Signaled Errors

TPU\$_TOOFEW	ERROR	UNMAP requires one parameter.
TPU\$_TOOMANY	ERROR	UNMAP accepts only one parameter.
TPU\$_INVPARAM	ERROR	One or more of the specified parameters have the wrong type.
PU\$_WINDNOTMAPPED	WARNING	Window is not mapped to a buffer.

Examples

1. The following example removes the main window from the screen and disassociates from the main window the buffer that was mapped to it:

```
UNMAP (main_window)
```

2. The following example unmaps the current window and puts two new windows in its place. (If the window that you are replacing has a status line, this line is not included in the screen area used by the two new windows. This is because GET_INFO (window, "visible_bottom") does not take the status line into account.)

```
PROCEDURE user_one_window_to_two
    LOCAL wind_length,
```

```
        wind_half,
        first_line,
        last_line;
    cur_wind := CURRENT_WINDOW;
! If it exists
    IF (cur_wind <> 0)
    THEN
        first_line := GET_INFO (cur_wind, "visible_top");
        last_line := GET_INFO (cur_wind, "visible_bottom");
        wind_buf := GET_INFO (cur_wind, "buffer");
        UNMAP (cur_wind);
    ELSE
! If there is no current window then create an empty buffer
        first_line := 1;
        last_line := GET_INFO (SCREEN, "visible_length");
        wind_buf := CREATE_BUFFER ("Empty Buffer");
    ENDIF;
    wind_length := (last_line - first_line) + 1;
    wind_half := wind_length/2;
    new_window_1 := CREATE_WINDOW (first_line, wind_half, OFF);
    SET (VIDEO, new_window_1, UNDERLINE);
    new_window_2 := CREATE_WINDOW (wind_half+1,
        last_line-wind_half, OFF);
! Associate the same buffer with both windows
! and map the windows to the screen
    MAP (new_window_1, wind_buf);
    MAP (new_window_2, wind_buf);
ENDPROCEDURE;
```

UPDATE

Syntax

```
UPDATE ( { ALL
         { window } })
```

Parameters

ALL

A keyword that directs DECTPU to make all visible windows reflect the current state of the buffers mapped to them.

window

The window that you want updated. The window must be mapped to the screen for the update to occur.

Description

The UPDATE procedure causes the screen manager to make a window reflect the current internal state of the buffer that is associated with the window. One important task that UPDATE performs is to move the cursor to the editing point if the cursor and the editing point are not synchronized when the UPDATE built-in procedure is executed.

The screen manager updates windows after each keystroke. However, if a key has a procedure bound to it, DECTPU may execute many statements when that key is pressed. By default, UPDATE does not reflect the result of any statement in a procedure bound to a key until all the statements in the procedure have been executed. As a result, the screen may not reflect the current state of the buffer during execution of a procedure bound to a key. If you want the screen to reflect changes before the entire procedure is executed, you can force an immediate update by adding an UPDATE statement to the procedure.

UPDATE (window) affects a single window that is visible on the screen. If the buffer associated with the window you use as a parameter is associated with other windows that are mapped to the screen, all of these windows may be updated.

UPDATE (ALL) updates all visible windows. The difference between the UPDATE (ALL) built-in procedure and the REFRESH built-in procedure is that UPDATE (ALL) makes whatever changes are necessary on a window-by-window basis. REFRESH clears the screen and repaints everything from scratch, as well as reinitializing scrolling regions and other terminal-dependent settings.

For more information on how the DECTPU screen manager uses the UPDATE built-in in various circumstances, see Appendix C. For more information on the results of the REFRESH built-in, see the description of REFRESH in this chapter.

Signaled Errors

TPU\$_TOOFEW	ERROR	UPDATE requires one parameter.
TPU\$_TOOMANY	ERROR	You specified more than one parameter.
TPU\$_INVPCARAM	ERROR	The specified parameter has the wrong type.
TPU\$_BADKEY	ERROR	The keyword must be ALL.
TPU\$_UNKKEYWORD	ERROR	You specified an unknown keyword.
TPU\$_WINDNOTMAPPED	WARNING	You cannot update a window that is not on the screen.

Examples

1. The following example causes the screen manager to make *new_window* reflect the current internal state of the buffer associated with *new_window*:

```
UPDATE (new_window)
```

2. The following example updates the screen to display the new line of text that you are inserting before the top line of the window. (When you insert text in front of the top of a window, the included text is not visible on the screen unless you use a procedure such as this one to ensure that the text is displayed.)

```
PROCEDURE user_show_first_line
    LOCAL old_position,           ! Marker of position before scroll
          new_position;          ! Marker of position after scroll
    UPDATE (CURRENT_WINDOW);
    IF (GET_INFO (CURRENT_WINDOW, "current_row") =
        GET_INFO (CURRENT_WINDOW, "visible_top"))
        AND
        (CURRENT_COLUMN = 1)
    THEN
        old_position := MARK (NONE);
```

```
    SCROLL (CURRENT_WINDOW, -1);
    new_position := MARK (NONE);
! Make sure we scrolled before doing the CURSOR_VERTICAL
    IF new_position <> old_position
    THEN
        CURSOR_VERTICAL (1);
    ENDIF;
ENDIF;
ENDPROCEDURE;
```

WRITE_CLIPBOARD

Syntax

WRITE_CLIPBOARD (clipboard_label , { buffer
range
string })

Parameters

clipboard_label

The label for multiple entries in the clipboard. Because the clipboard does not currently support multiple labels, use any string, including the null string, to specify this parameter.

buffer

The buffer that contains text to be written to the clipboard. DECTPU represents line breaks by a line-feed character (ASCII (10)). If you specify a buffer, DECTPU converts the buffer to a string, replacing line breaks with line feeds and replacing the white space before the left margin with padding blanks.

The buffer must contain at least one character or line break. If it does not, DECTPU signals TPU `$_CLIPBOARDZERO`.

range

The range that contains text to be written to the clipboard. DECTPU represents line breaks by a line-feed character (ASCII (10)). If you specify a range, DECTPU converts the range to a string, replacing line breaks with line feeds and replacing the white space before the left margin with padding blanks.

The range must contain at least one character or line break. If it does not, DECTPU signals TPU `$_CLIPBOARDZERO`.

string

The string that contains text to be written to the clipboard. The string must contain at least one character. If it does not, DECTPU signals TPU `$_CLIPBOARDZERO`.

Description

The `WRITE_CLIPBOARD` procedure writes string format data to the clipboard. The *clipboard_label* parameter provides support for multiple entries on the clipboard; at present, however, the clipboard does not support multiple entries.

Signaled Errors

TPU\$_CLIPBOARDLOCKED	WARNING	The clipboard is locked by another process.
TPU\$_CLIPBOARDZERO	WARNING	The data to be written to the clipboard have zero length.
TPU\$_TRUNCATE	WARNING	DECTPU has truncated characters from the data written because you specified a buffer or range that contains more than 65535 characters.
TPU\$_INVPARAM	ERROR	One of the parameters was specified with data of the wrong type.
TPU\$_NORETURNVALUE	ERROR	WRITE_CLIPBOARD cannot return a value.
TPU\$_REQUIRESDECW	ERROR	You can use the WRITE_CLIPBOARD built-in only if you are using DECwindows DECTPU.
TPU\$_TOOFEW	ERROR	Too few arguments passed to the WRITE_CLIPBOARD built-in.
TPU\$_TOOMANY	ERROR	Too many arguments passed to the WRITE_CLIPBOARD built-in.

Example

The following example writes the contents of the range *this_range* to the clipboard:

```
WRITE_CLIPBOARD ("", this_range);
```

WRITE_FILE

Syntax

$$\llbracket \text{string2} := \rrbracket \text{WRITE_FILE} \left(\left\{ \begin{array}{c} \text{buffer} \\ \text{range} \end{array} \right\} \llbracket , \text{string1} \rrbracket \llbracket , \left\{ \begin{array}{c} \text{ON} \\ \text{OFF} \\ 1 \\ 0 \end{array} \right\} \rrbracket \right)$$

Parameters

buffer

The buffer whose contents you want to write to a file.

range

The range whose contents you want to write to a file.

If you use WRITE_FILE on a range that does not start at the left margin of a line, DECTPU does the following:

- Determines the left margin of the line in which the range starts
- Writes the range to the output file starting at the same left margin as the margin of the line where the range starts

For example, if you write a range that starts in column 30 of a line whose left margin is 10, `WRITE_FILE` writes the range in the output file starting at column 10.

string1

A string that specifies the file to which the contents of the buffer are to be written. If you do not specify a full file specification, DECTPU determines the output file specification by using the current device and directory as defaults. The string is case insensitive.

This parameter is optional. If you omit it, DECTPU uses the associated output file name for the buffer. If there is no associated file name, DECTPU prompts you for one. If you do not give a file name at the prompt, DECTPU does not write to a file. In that case, the optional *string2* that is returned is a null string.

ON

A keyword specifying that output is padded with spaces to keep the first character of each record at the same column as the text in the buffer. This is the default.

OFF

A keyword specifying that no padding spaces are inserted when writing to the file.

Return Value

A string that represents the file specification of the file created.

Description

The `WRITE_FILE` procedure writes data to the file that you specify. `WRITE_FILE` optionally returns a string that is the file specification of the file created. If you specify a result, `WRITE_FILE` returns a string that is the file specification of the file to which the data was written.

DECTPU uses a flag to mark a buffer as modified or not modified. When you write data from a buffer to an external file, DECTPU clears the modified flag for that buffer. If you do not make any further modifications to that buffer, DECTPU does not consider the buffer as being modified and does not write out the file by default when you exit. If an error occurs while DECTPU is writing a file, DECTPU does not clear the modified flag.

When the contents of a buffer are written to a file, the associated journal file (if any) is closed and deleted and a new journal file is created. The new file contains the name of the file to which the buffer was written.

Deleting the file that has been written out invalidates the buffer-change journal.

Signaled Errors

<code>TPU\$_CONTROL</code>	ERROR	The execution of the write operation terminated because you pressed Ctrl/C.
<code>TPU\$_TOOFEW</code>	ERROR	<code>WRITE_FILE</code> requires at least one parameter.
<code>TPU\$_TOOMANY</code>	ERROR	<code>WRITE_FILE</code> accepts no more than two parameters.
<code>TPU\$_ARGMISMATCH</code>	ERROR	One of the parameters to <code>WRITE_FILE</code> is of the wrong type.

TPU\$_INVPARAM	ERROR	One of the parameters to WRITE_FILE is of the wrong type.
----------------	-------	---

DECTPU's file I/O routine can signal the following completion codes. You can provide your own file I/O routine by using the DECTPU callable interface. If you do so, WRITE_FILE's completion status depends upon what status you signaled in your file I/O routine.

TPU\$_OPENOUT	ERROR	WRITE_FILE could not create the output file.
TPU\$_NOFILEACCESS	ERROR	WRITE_FILE could not connect to the newly created output file.
TPU\$_WRITEERR	ERROR	WRITE_FILE could not write the text to the file because it encountered a file system error during the operation.
TPU\$_CLOSEOUT	ERROR	WRITE_FILE encountered a file system error when closing the file.

Examples

1. The following example writes the contents of the paste buffer to the file named MYFILE.TXT:

```
WRITE_FILE (paste_buffer, "myfile.txt")
```

2. The following example writes the contents of a buffer called extra_buf to a file. (Because you do not specify a file name, the associated file for the buffer is used.) The procedure then removes the extra window and buffer from your editing context.

```
PROCEDURE user_write_file
    WRITE_FILE (extra_buf);
    DELETE (extra_window);
    DELETE (extra_buf);
    ! Return the lines from extra_window to the main window
    ADJUST_WINDOW (main_window, -11, 0);
ENDPROCEDURE;
```

WRITE_GLOBAL_SELECT

Syntax

```
WRITE_GLOBAL_SELECT ( {
    array
    buffer
    integer
    range
    string
    NONE
} )
```

Parameters

array

An array that passes information about a global selection whose contents describe information that is not of a data type supported by DECTPU. For example, the array could pass information about a pixmap, an icon, or a span.

DECTPU does not use or alter the information in the array; the application layered on DECTPU is responsible for determining how the information is used, if at all. Because the array is used to pass information to and from other DECwindows applications, all applications that send or receive information whose data type is not supported by DECTPU must agree on how the information is to be sent and used.

The application sending the information is responsible for creating the array and giving it the proper structure. The array's structure is as follows:

- The element *array {0}* contains a string naming the data type of the information being passed. For example, if the information being passed is a span, the element contains the string "SPAN".
- The element *array {1}* contains either the integer 8, indicating that the information is passed as a series of bytes, or the integer 32, indicating that the information is passed as a series of longwords.
- If *array {1}* contains the value 8, the element *array {2}* contains a string, and there are no array elements after *array {2}*. The string does not name anything; rather it is a series of bytes. The meaning and use of the information is agreed upon by convention among the DECwindows applications.
- If *array {1}* contains the value 32, the remaining elements of the array contain integer data. In this case, the array can have any number of elements after *array {2}*. These elements must be numbered sequentially, starting at *array {3}*. All the elements contain integers. Each integer represents a longword of data. To determine how many longwords are being passed, an application can determine the length of the array and subtract 2 to allow for elements *array {0}* and *array {1}*.

buffer

The buffer that contains the information to be sent to the requesting application as the response to the global selection information request. If you specify a buffer, DECTPU converts the buffer to a string, converts line breaks to line feeds, and inserts padding blanks before text to fill any unoccupied space before the left margin.

integer

An integer whose value is to be sent to the requesting application as the response to the global selection information request. DECTPU sends the information in integer format.

range

The range that contains the information to be sent to the requesting application as the response to the global selection information request. If you specify a range, DECTPU converts the buffer to a string, converts line breaks to line feeds, and inserts padding blanks before and after text to fill any unoccupied space before the left margin.

string

The string that contains the information to be sent to the requesting application as the response to the global selection information request. DECTPU sends the information in string format.

NONE

A keyword indicating that no information about the global selection is available.

Description

The `WRITE_GLOBAL_SELECT` procedure sends requested information about a global selection from the DECTPU layered application to the application that issued the information request. `WRITE_GLOBAL_SELECT` is valid only inside a routine that responds to requests for information about a global selection.

The parameter specifies the data to supply to the requesting application. If you specify `NONE`, DECTPU informs the requesting application that no information is available. However, for any case in which a routine omits a `WRITE_GLOBAL_SELECT` statement, by default DECTPU informs the requesting application that no information is available.

Call `WRITE_GLOBAL_SELECT` no more than once during the execution of a global selection read routine. DECTPU signals `TPU$_INVBUILTININV` if you attempt to call this routine more than once.

Signaled Errors

<code>TPU\$_BUILTININV</code>	WARNING	<code>WRITE_GLOBAL_SELECT</code> was used more than once in the same routine or the builtin is being called outside of a global section read routine.
<code>TPU\$_TRUNCATE</code>	WARNING	DECTPU truncated characters from the data written because you specified a buffer or range that contained more than 65535 characters.
<code>TPU\$_INVPARAM</code>	ERROR	One of the parameters was specified with data of the wrong type.
<code>TPU\$_NORETURNVALUE</code>	ERROR	<code>WRITE_GLOBAL_SELECT</code> cannot return a value.
<code>TPU\$_REQUIRESDECW</code>	ERROR	You can use the <code>WRITE_GLOBAL_SELECT</code> built-in only if you are using DECwindows DECTPU.
<code>TPU\$_TOOFEW</code>	ERROR	Too few arguments passed to the <code>WRITE_GLOBAL_SELECT</code> built-in.
<code>TPU\$_TOOMANY</code>	ERROR	Too many arguments passed to the <code>WRITE_GLOBAL_SELECT</code> built-in.

Example

The following example sends the contents of the range *this_range* to the requesting application:

```
WRITE_GLOBAL_SELECT (this_range);
```

For an example of a procedure that uses the `WRITE_GLOBAL_SELECT` built-in procedure, see Example A.6.

Appendix A. Sample DECwindows DECTPU Procedures

This appendix presents a number of procedures that use DECwindows built-ins.

The following examples demonstrate some of the ways in which you can use DECTPU procedures:

- Creating a mouse pad
- Implementing an EDT-style APPEND command
- Testing and returning a select range
- Handling callbacks from a scroll bar widget
- Reactivating a select range
- Implementing the DECwindows COPY SELECTION operation from EVE to another application

Most of the procedures are drawn from the code implementing the Extensible Versatile Editor (EVE). Some have been modified to make them easier to understand.

You can see all the code used to implement EVE by looking at the files in SYS\$EXAMPLES:EVE\$*.*.

A.1. Creating a Mouse Pad

Example A.1 shows how to use the variant of CREATE_WIDGET that calls the Toolkit low-level creation routine. The module in Example A.1 creates a screen that represents a keypad. Instead of pressing a keypad key, you can click on the widget that represents the key.

Example A.1. Procedure That Creates a Mouse Pad

```
! SAMPLE.TPU
!++
!
!                               Table of Contents
!
!                               SAMPLE.TPU
!
!      Procedure name           Description
!      -----
!
!      sample_sample_module_ident      Ident.
!      sample_sample_module_init      Initializes the module.
!      eve_mouse_pad                   Implements the user command DISPLAY MOUSE
PAD.
!      sample_key_def                  Creates a mouse pad "key" push button.
!      sample_key_dispatch             Handles push button widget callbacks.
!      sample_row_to_pix               Converts a row number to pixels.
!      sample_col_to_pix               Converts a column number to pixels.
!      sample_key_height               Converts y dimension from rows to pixels.
!
!      sample_key_width                Converts x dimension from columns to pixels.
!
!--
! This module layers a "mouse pad" on top of DECTPU. The mouse pad
! is implemented by creating a dialog box widget that is the parent of a group
! of push button widgets depicting keypad keys. The resulting
! "mouse pad" is a screen representation of a keypad. The user can
! click on a push button to execute the same function that would be
! executed by pressing the corresponding keypad key. The module uses
! the key map list mapped to the current buffer to determine what
```

```

! code to execute when the user clicks on a given push button. To
! use a different key map, substitute a string naming the desired
! key map for the null string assigned to "sample_k_keymap".
! This module can be used with the EVE section file
! or with a non-EVE section file.
!
! This module uses the variant of CREATE_WIDGET that calls
! X Toolkit widget creation routines to create instances
! of widgets of specific classes.

! Widget class records for the DECwindows widgets for DEFINE_WIDGET built-in.
CONSTANT
sample_k_labelwidgetclass      := "xmLabelWidgetClassRec",
sample_k_dialogwidgetclass     := "xmBulletinBoardClassRec",
sample_k_pushbuttonwidgetclass := "xmPushButtonClassRec";
! Motif Toolkit resource name strings, callback reasons, resource values, or
! arguments to the CREATE_WIDGET built-in.
CONSTANT
    sample_k_cstyle := "dialogStyle",
    sample_k_modeless := 0,      ! = XmDIALOG_MODELESS
    sample_k_nunits := "unitType",
    sample_k_pixelunits := 0,    ! = XmPIXELS
    sample_k_ntitle := "dialogTitle",
    sample_k_nx := "x",
    sample_k_ny := "y",
    sample_k_nautounmanage := "autoUnmanage",
    sample_k_nheight := "height",
    sample_k_nwidth := "width",
    sample_k_nlabel := "labelString",
    sample_k_nactivate_callback := "activateCallback",
    sample_k_nborderwidth := "borderWidth",
    sample_k_t_nrecomputeSize := "recomputeSize",
    sample_k_cractivate := 10;

! These constants are intended for use only in this sample module
! because their values are specific to the mouse pad application.
CONSTANT
    sample_k_x_pos := 500,          ! Screen position for mouse pad.
    sample_k_y_pos := 500,
    sample_k_keypad_border := 5,    ! Width of border between keys and edge.
    sample_k_key_height := 30,     ! Key dimensions.
    sample_k_key_width := 60,
    sample_k_button_border_frac := 3, ! Determines spacing between keys.
    sample_k_overall_height := (sample_k_key_height * 5)
        + ((sample_k_key_height
            / sample_k_button_border_frac) * 5)
        + sample_k_keypad_border,
    sample_k_overall_width := (sample_k_key_width * 4)
        + ((sample_k_key_width
            / sample_k_button_border_frac) * 4)
        + sample_k_keypad_border,
    sample_k_keymap := '',          ! If this constant has a null string
        ! as its value, the program uses the
        ! current key map list to determine what
        ! code to execute when the user
        ! clicks on a given push button.
    sample_k_pad_title := "Sample mouse pad", ! Title of the mouse pad.
    sample_k_closure := '';          ! Not currently used.
PROCEDURE sample_sample_module_ident      ! This procedure returns
RETURN "V01-001";                        ! the Ident.
ENDPROCEDURE;
PROCEDURE sample_sample_module_init      ! Module initialization.
ENDPROCEDURE;
PROCEDURE eve_mouse_pad                  ! Implements a user-created command MOUSE
    PAD
! that the user can invoke from within EVE.
ON_ERROR
    [TPU$_CONTROL]:
        eve$learn_abort;
        ABORT;

```



```
ENDON_ERROR
```

```
! Checks whether the dialog box widget class has already been defined.
! If not, defines the dialog box widget class and creates a widget
! instance to be used as the "container" for the mouse pad.
IF GET_INFO (sample_x_dialog_class, 'type') <> INTEGER
THEN
sample_x_dialog_class
❶! := DEFINE_WIDGET_CLASS (sample_k_dialogwidgetclass,
"XmCreateBulletinBoardDialog");
! Tell TPU about new resource types
set (WIDGET_RESOURCE_TYPES, "unsigned_char", "dialogStyle", "unitType");
ENDIF;
! Create the dialog box
❷"sample_x_keypad := CREATE_WIDGET (sample_x_dialog_class, "Keypad", SCREEN,
"MESSAGE('CALLBACK activated')",
"sample_k_closure",
sample_k_cstyle, sample_k_modeless,
sample_k_nunits, sample_k_pixelunits,
sample_k_ntitle, sample_k_pad_title,
sample_k_nautounmanage, FALSE,
sample_k_nheight, sample_k_overall_height,
sample_k_nwidth, sample_k_overall_width,
sample_k_nx, sample_k_x_pos,
sample_k_ny, sample_k_y_pos);

! Checks whether the push button widget class has already been defined
! and, if not, defines the class.
IF GET_INFO (sample_x_pushbutton_class, 'type') <> INTEGER
THEN
sample_x_pushbutton_class
:= DEFINE_WIDGET_CLASS (sample_k_pushbuttonwidgetclass,
"XmCreatePushButton");
ENDIF;
! Initializes the array that the program passes repeatedly
! to the procedure "sample_key_def".

sample_x_attributes := CREATE_ARRAY;
sample_x_attributes {sample_k_nactivate_callback} := 0; ! must be 0 (non-UII)
sample_x_attributes {sample_k_nborderwidth} := 2;
sample_x_pad_program := COMPILE ("sample_key_dispatch");

! Creates and manages all the "keys" in the mouse pad. The procedure
! "sample_key_def" returns a variable of type widget, so you can use the
! returned value as an argument to the built-in MANAGE_WIDGET.
❸#MANAGE_WIDGET (sample_key_def ("PF1", 0, 0, 1, 1, sample_x_pad_program),
sample_key_def ("PF2", 1, 0, 1, 1, sample_x_pad_program),
sample_key_def ("PF3", 2, 0, 1, 1, sample_x_pad_program),
sample_key_def ("PF4", 3, 0, 1, 1, sample_x_pad_program),
sample_key_def ("KP7", 0, 1, 1, 1, sample_x_pad_program),
sample_key_def ("KP8", 1, 1, 1, 1, sample_x_pad_program),
sample_key_def ("KP9", 2, 1, 1, 1, sample_x_pad_program),
sample_key_def ("-", 3, 1, 1, 1, sample_x_pad_program, "minus"),
sample_key_def ("KP4", 0, 2, 1, 1, sample_x_pad_program),
sample_key_def ("KP5", 1, 2, 1, 1, sample_x_pad_program),
sample_key_def ("KP6", 2, 2, 1, 1, sample_x_pad_program),
sample_key_def ("", 3, 2, 1, 1, sample_x_pad_program, "comma"),
sample_key_def ("KP1", 0, 3, 1, 1, sample_x_pad_program),
sample_key_def ("KP2", 1, 3, 1, 1, sample_x_pad_program),
sample_key_def ("KP3", 2, 3, 1, 1, sample_x_pad_program),
sample_key_def ("Enter", 3, 3, 2, 1, sample_x_pad_program, "enter"),
sample_key_def ("KP0", 0, 4, 1, 2, sample_x_pad_program),
sample_key_def (".", 2, 4, 1, 1, sample_x_pad_program, "period"));
sample_shift_was_last := FALSE; ! The program starts out assuming that
! no GOLD key has been pressed.

❹$MANAGE_WIDGET (sample_x_keypad); ! This statement displays the
! resulting mouse pad.

RETURN (TRUE);
ENDPROCEDURE ! End of procedure eve_mouse_pad.
```

```

PROCEDURE sample_key_def                                ! Creates a mouse pad "key" push button
                                                    ! widget.
(the_legend,                                          ! What characters to show on the push button label.
the_row, the_col,                                    ! Location of the key in relation to the parent
                                                    ! widget's upper left corner.
the_width, the_height,                              ! Dimensions of the key.
the_pgm;                                             ! Program to use as the callback routine; used
                                                    ! as a parameter to the CREATE_WIDGET built-in.
the_string);                                         ! The string representation of the name
                                                    ! of a key if the key name is not going
                                                    ! to be the same as the legend (as in
                                                    ! the case of the comma). Specify the null
                                                    ! string if the key name and the legend are
                                                    ! the same.
IF GET_INFO (the_string, 'type') = UNSPECIFIED
THEN
    the_string := the_legend;                        ! Determines whether the optional parameter
                                                    ! the_string is provided.
ENDIF;

RETURN CREATE_WIDGET (sample_x_pushbutton_class,
    "Key",                                           ! name
    sample_x_keypad,                                 ! parent
    the_pgm,                                         ! program
    (sample_k_keymap + ' ' + the_string),           ! closure
    sample_x_attributes,                             ! attributes...
    sample_k_t_nrecomputesize, 1,
    sample_k_nlabel, the_legend,
    sample_k_nheight, sample_key_height (the_width),
    sample_k_nwidth, sample_key_width (the_height),
    sample_k_nx, sample_col_to_pix (the_row),
    sample_k_ny, sample_row_to_pix (the_col));
ENDPROCEDURE ! End of the procedure "sample_key_def".

PROCEDURE sample_key_dispatch                        ! Handles push button widget callbacks.
LOCAL status,                                       ! Variable to contain the return value from
                                                    ! GET_INFO (WIDGET, "callback_parameters",).
blank_index,                                       ! Position of the blank space in the tag string.
temp_array,                                        ! Holds callback parameters.
a_shift_key,                                       ! The SHIFT key in the current key map list.
the_key,                                           ! A string naming a key.
gold_key;                                          ! Name of the GOLD key.
ON_ERROR
    [TPU$_CONTROL]:
        eve$learn_abort;
        ABORT;
ENDON_ERROR

⑤%status := GET_INFO (widget, "callback_parameters", temp_array);
$widget := temp_array {'widget'};
$widget_tag := temp_array {'closure'};
$widget_reason := temp_array {'reason_code'};

⑥&the_key := EXECUTE ("RETURN(KEY_NAME (" + $widget_tag + "))");
gold_key := GET_INFO (eve$current_key_map_list, "shift_key");
IF the_key = gold_key
THEN
    sample_shift_was_last := TRUE; ! User pressed Gold Key
ELSE
    IF sample_shift_was_last
    THEN
        the_key := KEY_NAME (the_key, SHIFT_KEY);
    ENDIF;
    CASE $widget_reason
    [sample_k_cractivate]:
        EXECUTE (the_key);
    [OTHERWISE]:
        eve_show_key (the_key)
    ENDCASE;
    sample_shift_was_last := FALSE;

```

```

ENDIF;
RETURN;
ENDPROCEDURE ! End of the procedure "sample_key_dispatch".

! These procedures implement position and
! size calculations for the push button widgets.

PROCEDURE sample_row_to_pix (row)          ! Converts a row number to the
                                           ! pixel-based measuring system.
RETURN sample_k_keypad_border +
(row * (sample_k_key_height + (sample_k_key_height
                               / sample_k_button_border_frac)));
ENDPROCEDURE ! End of the procedure "sample_row_to_pix".

PROCEDURE sample_col_to_pix (col)         ! Converts a column number to the
                                           ! pixel-based measuring system.
RETURN sample_k_keypad_border +
col * (sample_k_key_width + ( 2* sample_k_button_border_frac) );
ENDPROCEDURE ! End of the procedure "sample_col_to_pix".

PROCEDURE sample_key_height (given_height) ! Converts the y dimension
                                           ! from rows to pixels.
IF given_height = 1
THEN
RETURN sample_k_key_height;
ELSE
RETURN ((sample_k_key_height * given_height)
        + (sample_k_key_height / sample_k_button_border_frac)
        * (given_height - 1));
ENDIF;
ENDPROCEDURE ! End of the procedure "sample_key_height".

PROCEDURE sample_key_width (given_width)   ! Converts the x dimension
                                           ! from rows to pixels.
IF given_width = 1
THEN
RETURN sample_k_key_width;
ELSE
RETURN ((sample_k_key_width * given_width)
        + (sample_k_key_width / sample_k_button_border_frac)
        * (given_width - 1));
ENDIF;
ENDPROCEDURE ! End of the procedure "sample_key_width".

```

- ❶ • When you create widgets directly in DECTPU (that is, without using the Resource Manager to manipulate widgets defined in a UID file), you must define each class of widget. For example, a widget can belong to a push button, dialog box, menu, or another similar class of widget. The DEFINE_WIDGET_CLASS built-in procedure tells DECTPU the widget class name and creation entry point for the class of widget. DEFINE_WIDGET_CLASS also returns an ID for that widget class. Define a widget class for each widget only once in a DECTPU session.
- ❷ • With CREATE_WIDGET, you can create an instance of a widget for which you have a widget class ID. An instance is one occurrence of a widget of a given class. For example, EVE has many menu widgets, each of which is an instance of a menu widget.

This example creates a dialog box widget to contain the mouse pad.

- ❸ • Each of the keys of the mouse pad is managed. However, they do not become visible until their parent, the dialog box widget in variable SAMPLE_X_KEYPAD, is managed.

This example creates a dialog box widget to contain the mouse pad.

- ❹ • Managing a widget whose parent is visible causes that widget and all its managed children to become visible.
- ❺ • GET_INFO (WIDGET, "callback_parameters", array) returns the callback information in the array parameter. For more information about using this built-in, see the built-in's description in GET_INFO.

- When each key widget of the mouse pad is created, the closure value for the widget is set to the string that corresponds to the name of the key that the widget represents. This statement uses the EXECUTE built-in to translate the string into a key name.

A.2. Implementing an EDT-Style APPEND Command

Example A.2 shows one of the ways an application can use the GET_CLIPBOARD built-in. This procedure is based on the EVE procedure EVE\$EDT_APPEND. The original version is in SYS\$EXAMPLES:EVE\$EDT.TPU.

The procedure EVE\$EDT_APPEND appends the currently selected text to the contents of the clipboard if you have activated the clipboard; otherwise, the procedure appends the current selection to the contents of the Insert Here buffer.

This example uses the following global variables and procedures from EVE:

- EVE\$MESSAGE—A procedure that translates the specified message code into text and displays the text in the message buffer.
- EVE\$\$RESTORE_POSITION—A procedure that repositions the editing point to the location indicated by the specified window and marker.
- EVE\$LEARN_ABORT—A procedure that aborts a learn sequence.
- EVE\$SELECTION—A procedure that returns a range containing the current selection. This can be the select range, the found range, or the text of the global selection.
- EVE\$\$TEST_IF_MODIFIABLE—A procedure that checks whether a buffer can be modified.
- EVE\$X_DECWINDOWS_ACTIVE—A Boolean global variable that is true if DECTPU is using the DECwindows screen updater. If DECTPU is not using DECwindows, the DECwindows features are not available.
- EVE\$\$X_STATE_ARRAY—A global variable of type array describing various EVE flags and data. This variable is private to EVE and should not be used by user routines.
- EVE\$\$EDT_APPEND_PASTE—Procedure that appends text to the INSERT HERE buffer.

Example A.2. EVE Procedure That Implements a Variant of the EDT APPEND Command

```
PROCEDURE eve$edt_append                                ! Implements EVE's version of
                                                         ! the EDT APPEND command.

LOCAL saved_mark,                                     ! Marks the editing point at the
                                                         ! beginning of the procedure.

    remove_range,                                     ! Stores the currently selected text.

    old_string,                                       ! Stores the text that was in the clipboard.

    new_string,                                       ! Stores the old contents of the clipboard
                                                         ! plus the currently selected text.
remove_status;                                       ! Indicates whether the selected text
                                                         ! should be removed.

ON_ERROR
    [TPU$_CLIPBOARDNODATA]:
```

```

eve$message (EVE$_NOINSUSESEL);
eve$$restore_position (saved_mark);
eve$learn_abort;
RETURN (FALSE);
[TPU$_CLIPBOARDLOCKED]:
eve$message (EVE$_CLIPBDREADLOCK);
eve$$restore_position (saved_mark);
eve$learn_abort;
RETURN (FALSE);
[TPU$_CONTROL]:
eve$$restore_position (saved_mark);
eve$learn_abort;
ABORT;
[OTHERWISE]: eve$$restore_position (saved_mark);
eve$learn_abort;
ENDON_ERROR;

remove_range := eve$selection (TRUE,,, eve$x_select_remove_flag);
IF remove_range <> 0
THEN
    saved_mark := MARK (NONE);
    remove_status := eve$test_if_modifiable (GET_INFO (saved_mark, "buffer"));
    IF eve$x_decwindows_active
    THEN
        IF eve$$x_state_array {eve$$k_clipboard}
        THEN
            ❶ ! old_string := GET_CLIPBOARD;
              new_string := old_string + str (remove_range);
            ❷ WRITE_CLIPBOARD ("", new_string);

        IF remove_status
        THEN
            ERASE (remove_range);
            eve$message (EVE$_REMCLIPBOARD);
        ENDIF;
        ELSE
            eve$$edt_append_paste (remove_range, remove_status);
        ENDIF;
        ELSE
            eve$$edt_append_paste (remove_range, remove_status);
        ENDIF;

        POSITION (saved_mark);
        remove_range := 0;
        RETURN (TRUE);
    ENDIF;
    eve$learn_abort;
    RETURN (FALSE);
ENDPROCEDURE;

```

- ❶ • The GET_CLIPBOARD built-in procedure returns a copy of the text stored in the clipboard. Only data of type string can be retrieved from the clipboard. Any other data type causes DECTPU to signal an error.
- ❷ • The WRITE_CLIPBOARD built-in procedure stores data in the clipboard. The first parameter lets you specify the label for this data. However, the clipboard currently supports only one entry at a time, so you can use any string for the first parameter.

A.3. Testing and Returning a Select Range

The code fragment in Example A.3 shows how a layered application can use GET_GLOBAL_SELECT. This code fragment is based on the EVE procedure EVE\$SELECTION. The original version is in SYS\$EXAMPLES:EVE\$CORE.TPU.

The procedure EVE\$SELECTION returns a select range, found range, or global selection for use with EVE commands that operate on the select range.

This example uses the following global variables and procedures from EVE:

- `EVE$MESSAGE`—A procedure that translates the specified message code into text and displays the text in the message buffer.
- `EVE$LEARN_ABORT`—A procedure that aborts a learn sequence.
- `EVE$X_DECWINDOWS_ACTIVE`—A Boolean global variable that is true if DECTPU is using DECwindows. If DECTPU is not using DECwindows, the DECwindows features are not available.

Example A.3. EVE Procedure That Returns a Select Range

```

PROCEDURE eve$selection (
    do_messages;                ! Display error messages?
    found_range_arg,           ! Use found range? (D=TRUE).
    global_arg,                ! Use global select? (D=FALSE).
    null_range_arg,           ! Extend null ranges? (D=TRUE).
    cancel_arg)                ! Cancel selection? (D=TRUE).
! Return Values:      range      The selected range.
!                    0           There was no select range.
!                    NONE        There was a null range and
!                               null_range_arg is FALSE.
!                    string      Text of the global selection
!                               if "global_arg" is TRUE.

LOCAL      possible_selection,
           use_found_range,
           use_global,
           extend_null_range,
           cancel_range;

ON_ERROR
    [TPU$_SELRANGEZERO]:
    [TPU$_GBLSELOWNER]:
        eve$message (EVE$_NOSELECT);
        eve$learn_abort;
        RETURN (FALSE);
    [OTHERWISE]:
ENDON_ERROR;

! The procedure first tests whether it
! has received a parameter directing
! it to return a found range or global
! selection if no select range has been
! created by the user.
IF GET_INFO (found_range_arg, "type") = INTEGER
THEN
    use_found_range := found_range_arg;
ELSE
    use_found_range := TRUE;
ENDIF;
IF GET_INFO (global_arg, "type") = INTEGER
THEN
    use_global := global_arg;
ELSE
    use_global := FALSE;
ENDIF;
! .
! .
! .

!In the code omitted from this example,
! eve$selection returns the appropriate
! range if the calling procedure has
! requested the user's select range
! or a found range.
! .
! .
! .

! If there is no found range or select

```

```

                                ! range, the procedure returns
                                ! the primary global selection
                                ! if it exists.
IF use_global and eve$x_decwindows_active
THEN
❶    ! possible_selection := GET_GLOBAL_SELECT (PRIMARY,
"STRING");
    IF GET_INFO (possible_selection, "type") = STRING
    THEN
        RETURN (possible_selection);
    ENDIF;
ENDIF;
.
.
.
RETURN (0); ! Indicates failure.
ENDPROCEDURE;

```

- ❶ • With DECwindows, you can designate more than one global selection. The two most common global selections are the primary and secondary selections. A global selection can be owned by only one DECwindows application at a time.

The `GET_GLOBAL_SELECT` built-in returns the data for the requested selection in the requested format. If the requested selection is not currently owned by any application, or if the owner cannot return it in the requested format, then `GET_GLOBAL_SELECT` returns TPU `$K_UNSPECIFIED`.

If the selected information contains multiple records, the records are separated by the line-feed character (ASCII (10)).

A.4. Handling Callbacks from a Scroll Bar Widget

Example A.4 shows one of the ways an application can get values for widget resources. The procedure is based on the EVE procedure `eve$scroll_dispatch`. The original version is in `SYSS$EXAMPLES:EVE$DECWINDOWS.TPU`.

The procedure `eve$scroll_dispatch` is the callback routine that handles callbacks from scroll bar widgets. The portion of the procedure shown here determines where to position the editing point based on how you have changed the scroll bar slider. The procedure fetches the position of the slider with the built-in `GET_INFO` (`widget_variable`, `"widget_info"`) and positions the editing point to the line in the buffer equivalent to the slider's position in the scroll bar. For more information about the resource names used with the scroll bar widget, see the *VMS DECwindows Toolkit Routines Reference Manual*.

EVE uses the following constants in this procedure:

- `"increment"`—The resource name for the amount that the scroll bar slider position is to be incremented or decremented when a scroll bar button is pressed.
- `"pageIncrement"`—The resource name for the amount that the scroll bar slider position is to be incremented or decremented when a click occurs within the scroll bar above or below the slider.
- `"maximum"`—The resource name for the maximum value of the scroll bar slider position.
- `"minimum"`—The resource name for the minimum value of the scroll bar slider position.
- `"value"`—The resource name for the top of the scroll bar slider position.

- "sliderSize"—The resource name for the size of the slider.
- 2—A constant for the callback reason code XmCR_VALUE_CHANGED. This reason code indicates that you changed the value of the scroll bar slider.
- "closure"—An index for the array returned by GET_INFO (WIDGET, "callback_parameters", array).
- "reason_code"—An index for the array returned by GET_INFO (WIDGET, "callback_parameters", array).
- "widget"—An index for the array returned by GET_INFO (WIDGET, "callback_parameters", array).

Example A.4. EVE Procedure That Handles Callbacks from a Scroll Bar Widget

```

PROCEDURE eve$scroll_dispatch
LOCAL
    status,
    widget_called,
    widget_tag,
    widget_reason,
    scroll_bar_values,
    linenum,
    temp_array,
    .
    .
    .;
ON_ERROR
    [TPU$_CONTROL]:
        eve$learn_abort;
        ABORT;
ENDON_ERROR

❶!status := GET_INFO (WIDGET, "callback_parameters", temp_array);
widget_called := temp_array {"widget";
widget_tag := temp_array {"closure";
widget_reason := temp_array {"reason_code";
POSITION (eve$$scroll_bar_window {widget_called});
.
.
.
scroll_bar_values := CREATE_ARRAY;
scroll_bar_values {"increment"} := 0;
scroll_bar_values {"pageIncrement"} := 0;
scroll_bar_values {"maximum"} := 0;
scroll_bar_values {"minimum"} := 0;
scroll_bar_values {"value"} := 0;
scroll_bar_values {"sliderSize"} := 0;
❷"status := GET_INFO (widget_called, "widget_info", scroll_bar_values);

! The deleted statements scroll the window as dictated
! by the callback reason.
.
.
.
CASE widget_reason
.
.
.
[2]:
    IF (scroll_bar_values {"value"} =
        scroll_bar_values {"minimum"})
    THEN
        POSITION (beginning_of (current_buffer));
    ELSE

```



```

③          POSITION (scroll_bar_values {"value"});
          ENDIF;
          .
          .
          .
ENDCASE;
!          .
!          .
!          .
!          .
ENDPROCEDURE;

```

- ② • GET_INFO (WIDGET, "callback_parameters", array) returns an array that contains the values for the current callback. The array elements are indexed by the strings "widget", "closure", and "reason_code" that reference the widget that is calling back, the widget's closure value, and the reason code for the callback.
- ② • With GET_INFO (WIDGET, "widget_info", array), you can fetch information from a widget. The array parameter is indexed by the resource names associated with the specified widget. Resource names are case sensitive and the set of supported resources varies from one widget type to another. When you use GET_INFO (widget, "widget_info", array), DECTPU queries the widget for the requested information and puts the returned information in the array elements. Any previous values in the array are lost.
- ③ • With POSITION (integer), you can move the editing point to the record specified by the parameter *integer*. DECTPU interprets this parameter as a record number.

A.5. Reactivating a Select Range

Example A.5 shows one of the ways an application can use the SET (GLOBAL_SELECT) built-in. The procedure is based on the EVE procedure EVE\$RESTORE_PRIMARY_SELECTION. The original version is in SYS\$EXAMPLES:EVE\$MOUSE.TPU.

The procedure *eve\$restore_primary_selection* reactivates EVE's select range when you press the Ctrl/Shift/E4 key. The select range was deactivated when EVE lost the primary selection to another DECwindows application.

Example A.5. EVE Procedure That Reactivates a Select Range

```

PROCEDURE eve$restore_primary_selection
LOCAL      saved_position;
ON_ERROR
    [TPU$CONTROL]:
        eve$restore_position (saved_position);
        eve$learn_abort;
        ABORT;
    [OTHERWISE]:
        eve$restore_position (saved_position);
ENDON_ERROR;
IF NOT eve$x_decwindows_active
THEN
    RETURN (FALSE);
ENDIF;
saved_position := MARK (FREE_CURSOR);
IF GET_INFO (eve$$x_save_select_array, "type") = ARRAY
THEN
    CASE eve$$x_save_select_array {"type"}
    [RANGE]:
        eve$select_a_range (eve$$x_save_select_array {"start"},
            eve$$x_save_select_array {"end"});
        eve$$x_state_array {eve$$k_select_all_active} :=
            eve$$x_save_select_array
                {"select_all"};
        POSITION (eve$$x_save_select_array {"current"});

```

```

    eve$start_pending_delete;
[MARKER]:
    POSITION (eve$$x_save_select_array {"start"});
    eve$x_select_position := select (eve$x_highlighting);
    POSITION (eve$$x_save_select_array {"end"});
    eve$start_pending_delete;
[OTHERWISE]:
    RETURN (FALSE);
ENDCASE;

eve$$restore_position (saved_position);
eve$$found_post_filter;          ! This is necessary if the
                                ! cursor is outside the selection.

eve$$x_save_select_array {"type"} := 0;
UPDATE (current_window);
SET (GLOBAL_SELECT, SCREEN, PRIMARY);    ! This statement
                                           ! requests ownership of
                                           ! the primary global selection.

    RETURN (TRUE);
ENDIF;
RETURN (FALSE);
ENDPROCEDURE;

```

A.6. Copying Selected Material from EVE to Another DECwindows Application

Example A.6 shows one of the ways a layered application can use the WRITE_GLOBAL_SELECT built-in. The procedure is based on the EVE procedure EVE\$WRITE_GLOBAL_SELECT. The original version is in SYS\$EXAMPLES:EVE\$MOUSE.TPU.

The procedure implements the operation of copying selected material from DECwindows EVE to another DECwindows application.

The procedure determines what property of the primary global selection is being requested, gets the value of the appropriate property by using a GET_INFO statement or an EVE procedure, and sends the information to the requesting application.

Example A.6. EVE Procedure That Implements COPY SELECTION

```

PROCEDURE eve$write_global_select          ! EVE uses this procedure
                                           ! to respond to requests
                                           ! for information about
                                           ! selections.

LOCAL      saved_position,
            the_data,
            temp_array,
            total_lines,
            the_line,
            status,
            eob_flag,
            percent;

ON_ERROR   [OTHERWISE]:
            eve$$restore_position (saved_position);
ENDON_ERROR;

saved_position := MARK (FREE_CURSOR);
IF NOT eve$x_decwindows_active
THEN
    RETURN (FALSE);
ENDIF;

```

```

the_data := NONE;
temp_array := GET_INFO (SCREEN, "event", GLOBAL_SELECT);
    ! Finds out which global selection and which property
    ! of the global selection are the subject of the
    ! information request.
CASE temp_array {2}      ! Determines the property requested by the other application.
["STRING", "TEXT"]: ! If one of these strings is requested, the
    ! procedure sends the text in the global
    ! selection to the requesting application.
CASE temp_array {1} ! Checks which global selection was specified.
[PRIMARY]:
    IF eve$x_select_position <> 0
    THEN
        POSITION (GET_INFO (eve$x_select_position, "buffer"));
        IF GET_INFO (eve$x_select_position, "type") = RANGE
        THEN
            the_data := STR (eve$x_select_position);
        ELSE
            IF GET_INFO (eve$x_select_position, "type") = MARKER
            THEN
                the_data := STR (eve$select_a_range (eve$x_select_position,
                                                    MARK (FREE_CURSOR)));
            ELSE
                the_data := NONE;
            ENDIF;
        ENDIF;
        eve$$restore_position (saved_position);
    ENDIF;
[OTHERWISE]:
    the_data := NONE;
ENDCASE;
[OTHERWISE]:
    the_data := NONE;      ! The procedure does not send data if
                          ! the requesting application has asked
                          ! for something other than the text,
                          ! the file name, or the line number.

ENDCASE;

WRITE_GLOBAL_SELECT (the_data);      ! This statement sends the
                                      ! requested information to
                                      ! the requesting application.

ENDPROCEDURE;

```


Appendix B. DECTPU Messages

This appendix presents the messages produced by DECTPU. The table below lists the messages alphabetically by their abbreviations. The text of the message and its severity level appears with each abbreviation.

Abbreviation	Message	Severity Level
ACCVIO	access violation, reason mask='xx', virtual address='xxxxxxxx',PC='xxxxxxxx', PSL='xxxxxxxx'	FATAL
ADJSCROLLREG	scrolling parameters altered to top: 'top', bottom: 'bottom', amount: 'amount'	INFORMATIONAL
AMBIGSYMUSED	ambiguous symbol 'name' used as procedure parameter	INFORMATIONAL
ARGMISMATCH	parameter 'number's' data type, 'type', unsupported	ERROR
ASYNCACTIVE	journal file prohibited with asynchronous handlers declared	
ATLINE	at line 'integer'	INFORMATIONAL
ATPROCLINE	at line 'integer' of procedure 'name'	INFORMATIONAL
BACKUPERR	error making backup copy of 'file-spec'	ERROR
BADASSIGN	target of the assignment cannot be a function/keyword	ERROR
BADBUFWRITE	error occurred writing buffer 'buffer name'	WARNING
BADCASELIMIT	CASE constant outside CASE limits	ERROR
BADCASERANGE	invalid CASE range	ERROR
BADCHAR	unrecognized character in input	ERROR
BADDELETE	cannot modify constant integer, keyword, or string	ERROR
BADDISPVAL	display value = 'integer', must be between 'integer' and 'integer'	WARNING
BADEXITIF	EXITIF occurs outside a LOOP	ERROR
BADFIRSTLINE	first line = 'integer', must be between 'integer' and 'integer'	WARNING
BADHIERARCHY	invalid hierarchy identifier specified	ERROR
BADJOUCHAR	expected character in journal file	WARNING
BADJOUCOM	journalled command file was 'string', recovering with 'string'	ERROR
BADJOUPOS	journalled starting character was 'integer', recovering with 'integer'	ERROR
BADJOUEDIT	journalled edit mode was 'string', recovering with 'string'	ERROR
BADJOUHEIGHT	journalled eightbit was 'string', recovering with 'string'	ERROR

Appendix B. DECTPU Messages

BADJOUFILE	operation terminated due to error in journal file access	ERROR
BADJOUINIT	journalized init file was 'string', recovering with 'string'	ERROR
BADJOUINPUT	journalized input file was 'string', recovering with 'string'	ERROR
BADJOUKEY	expected key in journal file	WARNING
BADJOUTLINE	journalized line editing was 'string', recovering with 'string'	ERROR
BADJOUTLPOS	journalized starting line was 'integer', recovering with 'integer'	ERROR
BADJOUTPAGE	journalized page length was 'integer', recovering with 'integer'	ERROR
BADJOUSEC	journalized section file was 'string', recovering with 'string'	ERROR
BADJOUSTR	expected string in journal file	WARNING
BADJOUTERM	journalized terminal type was 'string', recovering with 'string'	ERROR
BADJOUWIDTH	journalized width was 'integer', recovering with 'integer'	ERROR
BADKEY	'keyword' is an invalid keyword	WARNING
BADLENGTHCHANGE	terminal will not support change of length	WARNING
BADLOGIC	internal logic error detected	FATAL
BADMARGINS	margins specified incorrectly	WARNING
BADPROCNAME	variable used as a procedure	ERROR
BADPROMPTLEN	prompt area length = 'integer', must be between 'integer' and 'integer'	WARNING
BADREFCNT	ref count: 'ccc', zap count: 'zzz', address: 'xxxxxxx'	FATAL
BADREQUEST	request "'name'" of 'name' is not understood	WARNING
BADSCREENWIDTH	terminal must be wider than widest window, 'integer' columns	WARNING
BADSECTION	bad section file	ERROR
BADSTATUS	return status 'xxxxxxx' different from last signal 'xxxxxxx'	FATAL
BADSTRCNT	invalid string count found in journal file	WARNING
BADSYMTAB	bad symbol table	ERROR
BADUSERDESC	descriptor from user routine invalid or memory inaccessible	ERROR
BADVALUE	integer value 'integer' is outside specified limits	ERROR
BADWIDTHCHANGE	terminal will not support change of width	WARNING

BADWINDADJUST	attempt to make window less than 1 line long, no adjustment	WARNING
BADWINDLEN	window length = 'integer', must be between 'integer' and 'integer'	WARNING
BEGOFBUF	attempt to move past the beginning of buffer 'buffer name'	WARNING
BINARYOPER	operand combination 'type' 'oper' 'type' unsupported	WARNING
BITMAPREADERR	could not read bitmap file 'name'	ERROR
BOTLINETRUNC	calculated new last line 'integer', changed to 'integer'	INFORMATIONAL
BUILTININV	built-in is invalid at this time	ERROR
BUILTOBSOLETE	built-in no longer supported	INFORMATIONAL
CALLUSERFAIL	CALL_USER routine failed with status %X'status'	WARNING
CANCELQUIT	QUIT canceled by request	WARNING
CANNOTUNSEL	cannot unselect item from unselect action routine	ERROR
CAPTIVE	unable to create a subprocess in a captive account	WARNING
CLIPBOARDFAIL	unexpected clipboard failure	WARNING
CLIPBOARDLOCKED	clipboard is locked by another process	WARNING
CLIPBOARDNODATA	clipboard does not contain the requested data	WARNING
CLIPBOARDZERO	clipboard data has 0 length	WARNING
CLOSEDIC	error closing the dictionary file	ERROR
CLOSEIN	error closing input file 'file-spec'	ERROR
CLOSEOUT	error closing output file 'file-spec'	ERROR
CNVERR	error occurred in the conversion routine	ERROR
COMPILED	compilation completed without errors	SUCCESS
COMPILEFAIL	compilation aborted	WARNING
CONSTRTOOLARGE	constant string too large	ERROR
CONTRADEF	contradictory definition for variable or constant 'name'	ERROR
CONTROLC	operation aborted by Ctrl/C	ERROR
CREATED	file 'file-spec' created	SUCCESS
CREATEFAIL	unable to activate subprocess	WARNING
DEBUG	breakpoint at line 'integer'	SUCCESS
DELETEFAIL	unable to terminate subprocess	WARNING
DICADD	'word' has been added to a dictionary as 'word'	SUCCESS

Appendix B. DECTPU Messages

DICDEL	'word' has been removed from 'word' of a dictionary	SUCCESS
DICUPDERR	error updating dictionary file	ERROR
DIVBYZERO	divide by zero	ERROR
DUPBUFNAME	buffer 'name' already exists	WARNING
DUPKEYMAPLIST	attempt to define a duplicate key-map list 'key-map-list-name'	WARNING
DUPKEYMAP	attempt to define a duplicate key map 'key-map-name'	WARNING
EMPTYKMLIST	key-map list 'key-map-list-name' does not contain any key maps	WARNING
ENDOFBUF	attempt to move past the end of buffer 'buffer name'	WARNING
ERRSYMACTIVE	special error symbol already active	WARNING
EXECUTEFAIL	execution aborted	WARNING
EXITFAIL	attempt to EXIT was unsuccessful	WARNING
EXITING	editor exiting	SUCCESS
EXPCOMPLEX	expression too complex	ERROR
EXPECTED	one of the following symbols was expected:	INFORMATIONAL
EXTNOTFOUND	extension 'name' not found	ERROR
EXTRANEOUSARGS	one or more extraneous arguments specified	ERROR
FACTOOLONG	facility name, 'name', exceeds maximum length 'integer'	WARNING
FAILURE	internal DECTPU failure detected at PC 'number'	FATAL
FAILURE_STATUS	facility 'name' returned failure status of 'xxxxxxx'	ERROR
FENCEPOST	no visible record found in specified range	WARNING
FILECONVERTED	file format is being converted to a supported type	ERROR
FILEIN	'count' line(s) read from file 'filename'	SUCCESS
FILEOUT	'count' line(s) written to file 'filename'	SUCCESS
FLAGTRUNC	value of message flags exceeds maximum value 15, truncated	WARNING
FREEMEM	memory deallocation failure	FATAL
FROMBUILTIN	called from built-in 'name'	INFORMATIONAL
FROMLINE	called from line 'integer'	INFORMATIONAL
FROMPROCLINE	called from line 'integer' of procedure 'name'	INFORMATIONAL
GBLSELOWNER	you are the global selection owner	WARNING
GETMEM	memory allocation failure (Insufficient virtual memory)	ERROR
HIDDEN	global variable 'name' by declaration	INFORMATIONAL

ICONNAMERDERR	could not read the application icon name	ERROR
IDMISMATCH	section NOT restored, section file must be rebuilt	FATAL
ILLCONDIT	illegal compilation conditional	ERROR
ILLEGALTYPE	illegal data type	ERROR
ILLPATAS	pattern assignment target only valid in procedure 'name'	ERROR
ILLREQUEST	request "'name'" is invalid	WARNING
ILLSEVERITY	illegal severity of 'value' specified, error severity used	WARNING
ILLSYSRECMODE	invalid default system record mode	ERROR
INBULTIN	occurred in built-in 'name'	INFORMATIONAL
INCKWDCOM	inconsistent keyword combination	WARNING
INDEXTYPE	array index data type, 'type', unsupported	WARNING
INPUT_CANCELED	input request canceled	WARNING
INSVIRMEM	insufficient virtual memory	FATAL
INVACCESS	invalid file access specified	FATAL
INVBUFDELETE	cannot delete a permanent buffer	WARNING
INVCHARSET	unrecognized character set; using default character set DEC_MCS	WARNING
INVDEVTYPE	invalid device type	FATAL
INVFAOPARAM	FAO parameter 'integer' must be string or integer	WARNING
INVGBLSELDATA	the selected data cannot be processed	WARNING
INVINTERFACE	unrecognized interface	ERROR
INVIOCODE	invalid Operation Code passed to an I/O operation	ERROR
INVITEMCODE	invalid item code specified in list	FATAL
INVNUMSTR	invalid numeric string	WARNING
INVPARAM	parameter 'number's data type, 'type', illegal; expected 'type'	ERROR
INVRANGE	invalid range enclosure specified	WARNING
INVTABSPEC	tabs specification incorrect, not changed	WARNING
INVTIME	invalid time interval	WARNING
INWIDGETCLASS	the widget class cannot be defined	ERROR
JNLACTIVE	asynchronous actions prohibited when journal file open	WARNING
JNLNOTOPEN	journal file not open, recovery aborted	ERROR
JNLOPEN	journal file already open	ERROR
JOURNALBEG	journal of edit session started	INFORMATIONAL

JOURNALCLOSE	journal file successfully closed, journaling stopped	SUCCESS
JOURNALEOF	end of journal file found unexpectedly	WARNING
JRNLBUBFEG	journaling started for buffer 'buffer name'	INFORMATIONAL
JRNLSAFE	buffer 'buffer name' is not safe for journaling	WARNING
JRNLOPENERR	error opening or locking journal file 'journal-file-spec'	ERROR
JRNLOPEN	journal file already open for buffer 'buffer name'	WARNING
KEYMAPNOTFND	key map 'key-map-name' not found in key-map list 'key-map-listname'	WARNING
KEYSUPERSEDED	definition of key 'name' superseded	INFORMATIONAL
KEYWORDPARAM	keyword 'name' used as procedure/variable/constant	ERROR
LINENOTMOD	attempt to change unmodifiable line(s)	WARNING
LINETOOLONG	line is maximum length, cannot add text to it	WARNING
MAXMAPPEDBUF	a single buffer can be mapped to at most 'count' window(s)	WARNING
MAXVALUE	maximum value is 'integer'	WARNING
MINVALUE	minimum value is 'integer'	WARNING
MISSINGQUOTE	missing quote	ERROR
MISSYMTAB	missing symbol table	ERROR
MIXEDTYPES	operator with mixed or unsupported data types	ERROR
MODRANGEMARKS	MODIFY_RANGE requires either two marks or none	ERROR
MOUSEINV	mouse location information is invalid	WARNING
MOVETOCOPYTEXT	moving unmodifiable line(s) from buffer 'string' changed to copy	WARNING
MOVETOCOPY	move from unmodifiable buffer 'string' changed to copy	WARNING
MSGBUFSET	attempt to change modifiable setting of message buffer	WARNING
MSGNOTFND	message was not found; the default message has been returned	WARNING
MULTIDEF	parameter/local/constant 'name' multiply defined	ERROR
MULTIPLenames	there is more than one name matching, all are returned	WARNING
MULTISELECT	multiple identical CASE selectors	ERROR
MUSTBECONST	expression must be a compile-time constant	ERROR

MUSTBEONE	string must be 1 character long	WARNING
NEEDFILENAME	type file name for buffer 'name' (press RETURN to not write it):	SUCCESS
NEEDTOASSIGN	built-in must return a value	ERROR
NOASSIGNMENT	expression without assignment	ERROR
NOBREAKPOINT	no breakpoint is active	WARNING
NOCACHE	insufficient virtual memory to allocate a new cache	ERROR
NOCALLUSER	could not find a routine for CALL_USER to invoke	ERROR
NOCHARREAD	no character was read by the READ_CHAR built-in	WARNING
NOCLA	no conversion source has been specified yet	WARNING
NOCOPYBUF	cannot COPY a buffer to itself	WARNING
NOCURRENTBUF	no buffer has been selected as default	WARNING
NODEFINITION	key 'keyname' currently has no definition	WARNING
NODICENT	no entry found in a dictionary	WARNING
NODICUPD	the dictionary is restricting updating	WARNING
NODIC	no dictionary available in this editing session	WARNING
NOENDOFLINE	returning a range of text with no end of line	SUCCESS
NOEOBSTR	cannot return a string at end of buffer	WARNING
NOFILEACCESS	unable to access file 'name'	ERROR
NOFILEROUTINE	no routine specified to perform file I/O	FATAL
NOFOCUSOWNER	no input focus owner	WARNING
NOGBLSELDATA	no global selection data	WARNING
NOGBLSELOWNER	there is no global selection owner	WARNING
NOJOURNAL	editing session is not being journaled	WARNING
NOKEYMAPLIST	attempt to access an undefined key-map list 'key-map-list-name'	WARNING
NOKEYMAP	attempt to access an undefined key map 'key-map-name'	WARNING
NOLICENSE	DECTPU license validation failed	FATAL
NONAMES	there are no names matching the one requested	WARNING
NONANSICRT	SYS\$INPUT must be supported CRT	
NOPARENT	there is no parent process to attach to	WARNING
NOPROCESS	no subprocess to interact with	WARNING
NOREDEFINE	built-in procedure 'name' cannot be redefined	
NORETURNVALUE	built-in does not return a value	
NOSELECT	no select active	WARNING

NOSHOWBUF	variable SHOW_BUFFER does not exist or is not a buffer	WARNING
NOTARRAY	indexed variable is not an array	WARNING
NOTDEFINABLE	that key is not definable	WARNING
NOTERRORKEYWORD	error handler selector is not an error keyword	ERROR
NOTIMPLEMENTED	built-in compiled by 'name' is not implemented by 'name'	ERROR
NOTJOURNAL	file 'file' is not a journal file	ERROR
NOTLEARNING	you have not begun a learn sequence	WARNING
NOTMODIFIABLE	attempt to change unmodifiable buffer 'string'	WARNING
NOTSAMEBUF	the markers are not in the same buffer	WARNING
NOTSUBCLASS	object is not a subclass of WindowObjClass	WARNING
NOTYET	not yet implemented	WARNING
NOWINDOW	attempt to position the cursor outside all of the mapped windows	WARNING
NO	NO	INFORMATIONAL
NULLSTRING	null string used	WARNING
OCCLUDED	built-in/keyword 'name' occluded by declaration	INFORMATIONAL
ONDELRECLIST	attempt to access a record on the deleted list	FATAL
ONELEARN	cannot start a learn sequence while one is active	WARNING
ONESELECT	select already active, maximum 1 per buffer	WARNING
ONOROFF	parameter 'number' must be ON, OFF, true, or false	ERROR
OPENDIC	error opening the dictionary file	ERROR
OPENIN	error opening 'input-file' as input	ERROR
OPENOUT	error opening 'output-file' as output	ERROR
OVERLAPRANGE	overlapping ranges, operation terminated	WARNING
PARSEFAIL	error parsing 'file-spec'	WARNING
PARSEOVER	parser stack overflow	ERROR
PREMATUREEOF	premature end-of-file detected	ERROR
PRESSRET	press RETURN to continue...	SUCCESS
PROCESSBEG	subprocess activated	SUCCESS
PROCESSEND	subprocess terminated	SUCCESS
PROCSUPERSEDED	definition of procedure 'name' superseded	INFORMATIONAL
QUITTING	editor quitting	SUCCESS
READABORTED	READ_CHAR, READ_KEY, or READ_LINE built-in was aborted	WARNING

READERR	error reading 'input-file-spec'	ERROR
READLINEHELP	DECTPU is executing the READ_LINE built-in function, enter text or cancel	INFORMATIONAL
REALLYQUIT	buffer modifications will not be saved, continue quitting (Y or N)?	SUCCESS
REALLYRECOVER	continue recovering (Y or N)?	SUCCESS
RECJNLOPEN	journal file open, recovery status unchanged	ERROR
RECOVERABORT	recovery aborted by journal file inconsistency, journal file closed	WARNING
RECOVERBEG	recovery started	SUCCESS
RECOVERBUFBEG	recovery started for buffer 'buffer name'	SUCCESS
RECOVERBUFBEND	recovery complete for buffer 'buffer name'	SUCCESS
RECOVERBUFFILE1	can not recover from file 'file name'	SUCCESS
RECOVERBUFFILE2	type new specification for original input file:	SUCCESS
RECOVERBUFFILE3	type new specification for inserted file:	SUCCESS
RECOVERBUFFILE4	can not find inserted file 'file name'	SUCCESS
RECOVEREND	recovery complete	SUCCESS
RECOVERFAIL	recovery terminated abnormally, journal file inconsistency	ERROR
RECOVERQUIT	no file name specified, nothing recovered	WARNING
RECURLEARN	learn sequence replay halted due to recursion	WARNING
REFRESH_NEEDED	screen refresh needed	WARNING
REGWIDDUP	registration string already associated with a different widget	WARNING
REPLAYFAIL	an inconsistency has been discovered, halting execution	WARNING
REPLAYWARNING	an inconsistency has been discovered, continuing execution	WARNING
REQARGSMISSING	one or more required arguments missing	ERROR
REQUIRESDECW	feature requires the DECTPU DECwindows screen updater	ERROR
REQUIRESMOTIF	feature requires the DECTPU Motif screen updater	ERROR
REQUIRESTERM	feature requires a terminal	ERROR
REQUIRESULTRIX	feature not available on this operating system	ERROR
REQUIRESVMS	feature not available on this operating system	ERROR
RESTOREFAIL	error during RESTORE operation	ERROR
REVERSECASE	CASE limits were reversed	INFORMATIONAL
ROUND	FORWARD was rounded to the top	INFORMATIONAL

SAVEAMBIGSYM	saving ambiguous symbol 'name'	INFORMATIONAL
SAVEERROR	error during SAVE operation	ERROR
SAVEUNDEFPROC	saving undefined procedure 'name'	INFORMATIONAL
SCANADVANCE	*** Scanner advanced to "'name'" ***	ERROR
SEARCHFAIL	error searching for 'file-spec'	
SECTRESTORED	'count' procedure(s), 'count' variable(s), 'count' key(s) restored	INFORMATIONAL
SECTSAVED	'count' procedure(s), 'count' variable(s), 'count' key(s) saved	SUCCESS
SECTUNDEFPROC	saved 'count' undefined procedure(s), 'count' ambiguous symbol(s)	WARNING
SELRANGEZERO	select range has 0 length	WARNING
SENDFAIL	unable to send to subprocess	WARNING
SOURCELINE	at source line 'integer'	INFORMATIONAL
STACKOVER	stack overflow during compilation	ERROR
STATOOLONG	truncating status line to 'count' characters	INFORMATIONAL
STRNOTFOUND	string not found	WARNING
STRTOOLARGE	string greater than 65535 characters	ERROR
STRTOOLNG	string is too long for a conversion source	ERROR
SUCCESS	successful completion	SUCCESS
SYMDELETE	*** Error symbol deleted ***	ERROR
SYMINSERT	*** "'name'" inserted before error symbol ***	ERROR
SYMREPLACE	*** Error symbol replaced by "'name'" ***	ERROR
SYMTBLFUL	all symbol tables are full	ERROR
SYNTAXERROR	syntax error	ERROR
SYSERROR	system service error	ERROR
TEXT	'message'	INFORMATIONAL
TIMEOUT	built-in timed out	WARNING
TOOFEW	too few arguments	ERROR
TOOMANYPARAM	too many formal parameters/local variables	ERROR
TOOMANYRECS	too many records	ERROR
TOOMANY	too many arguments	ERROR
TOPLINETRUNC	calculated new first line 'integer', changed to 1	INFORMATIONAL
TRUNCATE	line truncated to 'count' characters	WARNING
TYPEREDEFINE	widget resource type 'name' has been redefined to data type 'name'	WARNING
UIDICONERR	could not read icon 'name' from UID file	ERROR
UIDOPENERR	could not open the specified UID file(s)	ERROR

UIDWIDGETERR	could not read widget 'name' from UID file	ERROR
UKNFACILITY	unknown facility code specified	WARNING
UNARYOPER	operand combination 'oper' 'type' unsupported	WARNING
UNDEFINEDPROC	undefined procedure call 'name'	ERROR
UNWIDCLA	undefined widget class specified	WARNING
UNKCSCOMP	ignoring unknown compound string component	WARNING
UNKESCAPE	unknown escape sequence read	WARNING
UNKKEYWORD	an unknown keyword has been used as an argument	ERROR
UNKLEXICAL	unknown lexical element	ERROR
UNKOPCODE	unknown opcode 'value'	ERROR
UNKTYPE	unknown data type 'value'	ERROR
UNKWNDESC	unknown descriptor type	ERROR
UNLINKWORKERR	error unlinking work file	ERROR
UNREACHABLE	unreachable code	INFORMATIONAL
WIDMISMATCH	parameter 'number's class, 'class', unsupported	ERROR
WINDNOTMAPPED	the window is not mapped to a buffer	WARNING
WINDNOTVIS	built-in cannot operate on an invisible window	WARNING
WORKFILEFAIL	error opening the work file	ERROR
WRITEERR	error writing 'output-file-spec'	ERROR
YES	YES	INFORMATIONAL

Appendix C. DECTPU Cursor Behavior

This appendix describes cursor behavior in DECTPU applications.

C.1. Cursor Position Compared to Editing Point

The 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.

DECTPU 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.

You can 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 you add 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, the screen manager may update the window automatically if you simply exit from your procedure.

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 maps the current buffer to the current window and moves the editing point to the cursor position. Updating the screen at this point has no effect, because the positions are already synchronized.

C.2. Built-In Padding

The cursor position is not necessarily bound to text. You can move the cursor position 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 previously listed, 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 the 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 deleted; 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. To insert these blank lines, DECTPU uses 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");
```

